App Update Patterns: How Developers Act on User Reviews in Mobile App Stores

Shance Wang[1], Zhongjie Wang[1(✉)], Xiaofei Xu[1], and Quan Z. Sheng[2]

[1] Harbin Institute of Technology, Harbin 150001, Heilongjiang, China
{shance.wang,rainy,xiaofei}@hit.edu.cn
[2] Macquarie University, Sydney, NSW 2109, Australia
michael.sheng@mq.edu.au

Abstract. Mobile app stores receive numerous reviews that contain valuable feedbacks raised by users. Incorporating user reviews into iterative delivery of new App versions would improve the quality and ratings of Apps. To date, there is no explicit answer on whether and to what degree App developers make use of user reviews sufficiently and timely. In this paper, we extract requested features in user reviews and updated features in new versions, identify the latent relation between them, and discover 7 types of Update Patterns (UPs) by grouping similar Atomic Update Units (AUs). UPs delineate common behavioral characteristics of acting on user reviews from perspectives of feature intensity trend, sufficiency and responsiveness. Statistics are conducted to explore the similarity/difference between exhibited update patterns w.r.t. Apps, features, and time. Results would help developers get a clear understanding on their own habits on how to act on user reviews, and thus offer suggestions on utilizing user reviews more efficiently in App development.

Keywords: Mobile App · App store · User review · Atomic Update Unit (AU) · Update Pattern (UP) · Empirical study

1 Introduction

With the flourish of mobile Internet and service-dominant industries, the number of available mobile Apps grows drastically with the rate 4–7% per month. Mobile Apps dominate people's daily lives and have been a major channel through which people access cloud services anytime and anywhere to fulfill personal demands [2].

App store is a centralized platform for users to acquire Apps. In an App store, each App has a homepage showing its descriptions along with its developer, size, versions, etc. Users can submit reviews through its homepage. There are valuable information lurking in enormous amount of user reviews, such as bug reports, feature requests, complaints or appraisal, and the rating [10]. Potential App users could gain a first impression on an App from reviews of its previous users.

User reviews are important to App developers, too. As today's Internet-based services usually adopt agile development approaches such as rapid iterations and

© Springer International Publishing AG 2017
M. Maximilien et al. (Eds.): ICSOC 2017, LNCS 10601, pp. 125–141, 2017.
https://doi.org/10.1007/978-3-319-69035-3_9

continuous delivery, App developers would like to acquire feature requests from reviews, and then to consider fulfilling them in subsequent releases [17]. There has reached a consensus that user reviews bring positive effects to every phase of App development, especially requirement engineering and testing [14].

Because the number of user reviews is enormous and high-quality reviews are often intermingled with much more low-quality ones, App developers have to filter out those informative reviews [3], read them, extract useful requested features and perceive customer preferences [23], and fulfill these feature in next iteration. These actions are regarded as "response to customer voices". However, it is not sure whether all developers follow the same behavior pattern, i.e., do they have the same opinion on the value of user reviews and therefore utilize them timely and sufficiently? As far as we know, until now there are not enough research investigating this issue, which is still a great challenge.

The first work of this paper is to reveal the potential correlation between the features requested in user reviews and the behaviors performed by App developers on updating App into a new version. Since most of Apps are proprietary but open source Apps are in the minority, it is difficult to obtain fined-grained update descriptions by tracking changes of source code or documents. To achieve this goal, we can make use of update logs (i.e., "What's New") written by developers and publicized on the App homepage. An update log of a specific version is a text list describing what features have been added or updated in this version (compared with the previous version) [8]. As there have been many approaches for extracting *requested features* from review texts [5,7], it is possible to apply similar methods to extract *updated features* mentioned in "What's New", too. If a frequently-requested feature in a specific period appears strikingly in the update log of a subsequent version, it is indicated that there is latent correlation between user reviews and update behaviors during this period, thus we may think developers do elaborately consider user reviews when the App is updated. Detailed analysis is conducted from the following four perspectives:

- Changing trend of the intensity of a feature requested in user reviews, i.e., how does the intensity of a feature mentioned in users reviews fluctuate in the time interval between two neighboring versions of an App?
- Intensity of a feature updated in a new version, i.e., to what extent is a feature mentioned in "What's New" of the version?
- Update sufficiency for a specific feature, i.e., to what extent do App developers act on specific user expectations sufficiently in a new version?
- Update responsiveness for a specific feature, i.e., how long does it take for App developers to update an App in terms of the requested feature, or, to what degree do App developers respond to user requests timely?

Next task is to identify common behavior patterns of how App developers act on requested features. They are denoted as *Update Pattern* (UP). The second work is to present an update pattern mining method by splitting App update history into a set of Atomic Update Units (AU) and clustering them into groups. Obtaining UPs would further help us identify their pros and cons, respectively, so as to utilize user reviews more effectively.

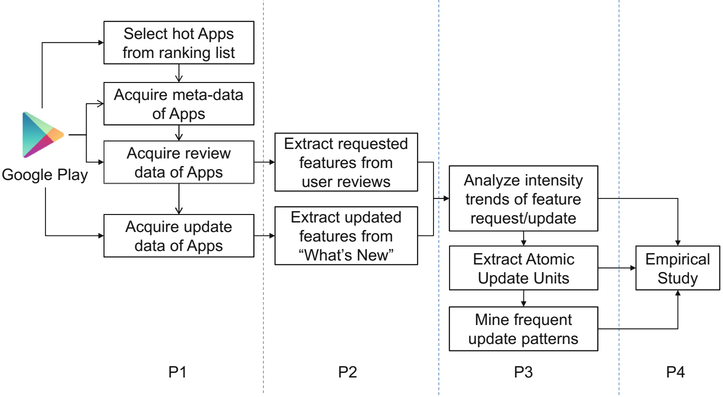


Fig. 1. Overall research framework with four phases

As Apps belong to different categories, offering diversified functionality to diversified target users, and developed by different developers having different working habits, we conjecture that update patterns in different Apps might not be the same. Even in one App, as it contains multiple features such as UI aesthetics, security, performance, we guess that the update patterns for different features might be different. Furthermore, as each feature is updated multiple times in multiple versions, the update patterns exhibited in different time might also be different.

The third work is to conduct an empirical study to observe the similarity/difference among update patterns exhibited in different Apps, on different/similar features, and at different times, respectively.

To sum up, we address the following research questions (RQs) in this study:

- RQ1: Does the correlation between user reviews and App updates really exist? If yes, are there common update patterns? And how to identify them?
- RQ2: In terms of different features of the same App, are there significant difference/similarity between their update patterns?
- RQ3: In terms of similar features of different Apps, are there significant similarity between their update patterns?
- RQ4: For a single feature of an App, are there significant changes of its update patterns exhibited in different time?

Figure 1 demonstrates our research approach consisting of four phases: (P1) *Data Collection*: user reviews and update logs (between March 2016 and March 2017) of 120 representative Apps are collected from Google Play; (P2) *Feature Extraction*: topic-based features are extracted from collected texts by NLP approaches so that raw corpus are transformed into numerical vectors; (P3) *Update Pattern Mining*: intensity trend of each feature along with time is calculated and decomposed into "Atomic Update Units (AUs)", along with the measurement of sufficiency and responsiveness of each AU, and update patterns are

identified by clustering AUs; (P4) *Empirical Study*. P1, P2 and P3 are together for addressing RQ1, and P4 is for RQ2, RQ3 and RQ4.

The main conclusions from this study are briefly summarized as follows:

– Intensity of a feature requested in user reviews fluctuates along with time: it is comparatively higher before and after an update (release of a new version) than at other times. Update sufficiency of features follows a power-law distribution, but the distribution of update responsiveness shows an obvious right-skewed shape. Comparatively speaking, App developers tend to act on user requests in a preferably timely but not quite sufficient manner;
– There are 7 types of update patterns commonly exhibited in developers' update behaviors. They are differentiated from three perspectives: intensity trend of a requested feature, update sufficiency, and update responsiveness.
– App developers tend to adopt more similar update patterns for different features in their own App. Similar features across multiple Apps more tend to exhibit diversified update patterns. Therefore the adoption of update patterns depends largely on "App developers" rather than on "features". However, developers' behaviors lack enough coherence/continuity when they update the same feature at different times.

The remainder of the paper is organized as follows. Section 2 introduces the related work. Section 3 describes data collection and processing. Section 4 presents the methods on feature extraction and update pattern mining. Finally, Sect. 5 reports experimental results and Sect. 6 offers some concluding remarks.

2 Related Work

Extracting Features from Reviews and Update Logs. Feature extraction from "big data" in App stores is a fundamental problem in the App Store Analysis [14]. Various NLP techniques have been developed for this purpose. Finkelstein et al. [5] defined a feature as "a claimed functionality offered by an App, captured by a set of collocated words in App description and shared by a set of Apps in the same category"; a tool `N-gramCollocationFinder` in NLTK was used to extract *featurelets* from reviews. Guzman et al. [7] also used collocation finding approach, but added sentiment analysis for extracting sentiments and opinions associated to features, and topic modeling for grouping related features. Differently, Iacob et al. [9] used syntax templates (keywords + syntax rules) to identify features automatically from reviews. The third dominant approach is statistics-based models such as Latent Dirichlet Allocation (LDA), e.g., [1,6,21]. Since user reviews are numerous, informative reviews should be selected and recommended to App developers for elaborate considerations. Chen et al. [3] used a machine learning approach, EMNB, to classify reviews into "informative" and "non-informative" ones, then used LDA/ASUM to group reviews into clusters and prioritize them. Vu et al. [22] analyzed user reviews for keywords of potential interest which developers can use to search for useful opinions, including keyword extracting, keyword grouping, and keyword ranking.

Feature Classification. It is necessary to classify extracted features into categories so that they are handled by different strategies. Khalid et al. [11] presented a 12-category classification, e.g., `App Crashing`, `Compatibility`, `Feature Removal`, `Hidden Cost`, `Functional Error`, etc. Pagano and Maalej [18] presented a classification with four coarse-grained categories and 17 fine-grained ones, and they used manual annotation and frequent item mining approaches to identify the frequent co-occurrence of features in reviews. Maalej and Nabil [12] utilized several probabilistic techniques to automatically classify reviews into `bug reports`, `feature requests`, `user experiences`, and `ratings`. McIlroy et al. [16] distinguished 14 types of features and introduced a supervised multilabel classification method. In terms of features in "What's New", McIlroy et al. [15] introduced a classification with 5 types, i.e., `new content`, `new features`, `improvement`, `bug fix`, `permission`, and the frequency of their co-occurrence is analyzed to empirically study frequently-updated mobile Apps.

Incorporating User Reviews in App Development. Apps as typical Internet-based services, evolve fast in both external interfaces and internal functionalities [8]. To cope with unpredictable changes and failures, Apps need to be adaptive, too [4]. Syer et al. [20] found that the evolution of Apps are quite different from the evolution of traditional software, and direct feedback contained in user reviews facilitates design and testing respectively [10,14]. Nayebi et al. [17] studied how developers organize release strategy of Apps, and found that the majority of developers are willing to bend their time-based strategies to accommodate users' feedback, and believe that the rationale for release decisions affects user feedback. Martin et al. [13] studied App update logs and analyzed the causality between continuous updates of an App and its influence/rating, i.e., what types of updates would upgrade Apps' rating more easily. Palomba et al. [19] analyzed how developers utilize user reviews to improve App rating: they traced user reviews onto source code changes for monitoring the extent to which developers accommodate crowd requests and follow-up user reactions as reflected in their ratings, and results indicate that developers implementing user reviews are rewarded in terms of ratings.

3 Data Collection and Preprocessing

3.1 App Selection

We choose Google Play as the source to collect mobile Apps and their update logs and user reviews. Firstly, Android dominates mobile OS marketplace with the market share above 74%. Secondly, the amount of Apps and users in Google Play are both the biggest among Android App stores in the world, and the amount and quality of user reviews are the highest, too. Thirdly, its users are distributed across the world, thus the user reviews are extensively representative. To specify the candidate Apps in our study, we choose the top-100 free Apps and top-20 paid Apps from the hot ranking list of Google Play (date: March 1, 2016). User reviews and update logs of the 120 Apps between March 31, 2016 and March

30, 2017 are collected. In terms of user reviews, we collect nickname of users, review text, review date, and ratings. In terms of update logs, we collect App descriptions, distribution of user ratings, and the text of "What's new".

3.2 Dataset Preparation

User reviews and update logs of Apps are both dynamically updated, and Google Play only shows the latest version in the homepage of an App (previous versions cannot be seen), and shows only a set of latest user reviews (i.e., not complete historical reviews). Therefore, it is impossible to crawl required data all at once. What we adopt is to constantly monitor the homepage of each App and identify changes of update logs and reviews and record them instantly. We develop our own Google Play crawler which regularly crawls updated information from Apps' homepages and store them in an incremental manner. The tool constructs a virtual `HTTP` request and sends it to Google Play, then collects the returned review data (in `JSON` format), compares with local files, extracts updated reviews and saves to local files. Similarly, after an App update log is crawled, the tool compares it with previous update logs and check if it has been changed or not; if yes, a new version has been released and "What's New" is stored in local files.

As update frequency of user reviews is higher, we use five servers for the crawling task. Tasks for collecting reviews of 120 Apps are allocated onto these servers (each is responsible for 15–25 Apps). The sixth server is responsible for collecting update logs of 120 Apps one time per day (because the update frequency of Apps is comparatively lower). All the six servers are virtual machines on `Aliyun` (the biggest PaaS cloud service provider in China), with the configuration as: 1 GHz CPU, 1 GB memory, 2 Mpbs bandwidth, 40 GB harddisk, OS `Windows Server 2008 R2`. The servers are physically located in Silicon Valley, USA.

After 12-month collection, 17,557,170 reviews are collected, with 1,296 times of App updates. There are totally 5,923,379 distinct users each having contributed at least one review. Majority of the Apps have less than 500 daily reviews in most of days, while a few Apps have above 4,000 reviews in one day.

Preprocessing of the crawled data includes two main steps as the following:

(1) *Remove non-English texts*. Although our crawler access Apps only from English zone of Google Play, there are still some non-English review texts. We use *table lookup* to filter them out. English vocabulary table is offered by `aspell`[1]. If a word belongs to this table, it is considered as an English word. If the ratio of non-English words in a review is higher than a threshold (we use 0.3), this review is considered as a non-English one and is to be discarded. Finally, 51.16% reviews are discarded and 8,575,276 reviews remain in the dataset.

[1] http://wordlist.aspell.net/dicts.

(2) *Remove stopwords and stemming.* This is a common step in natural language processing. Besides the `Lextek`'s stopword list[2], when the reviews of a specific App are handled, the full and abbreviated names of this App and the names of its common operations are added to the stopword list.

4 Mining App Update Patterns

4.1 Extracting Features from User Reviews and "What's New"

BTM (Biterm Topic Model) is employed to extract features from texts of user reviews and "What's New". BTM is an optimized unsupervised LDA model especially for short texts (e.g., tweets, short reviews) by explicitly modeling the word co-occurrence patterns to enhance the topic learning and using aggregated patterns in the whole corpus for learning topics to solve the problem of sparse word co-occurrence patterns at document-level [24].

Reviews and "What's New" of each App are the input of BTM and a topic distribution matrix A is generated. If the total number of reviews is m, total number of update logs is l, and the number of topics generated by BTM is n, then A is $(m+l)\times n$ dimensions, and the value A_{ij} is the probability with which the i-th review or update log covers the j-th topic. Each topic is represented by a set of keywords and is used as a feature covered by the App.

4.2 Intensity Trend Chart of Feature Request/Update

Next step is to measure the daily intensity of each feature requested in reviews and updated in new versions, and to get intensity trend of each feature. In terms of one App, suppose $Day(i)$ is the day when the i-th review is published to App store, then the daily intensity of the j-th feature f_j in the k-th day is $I^R(f_j, k) = \sum_{Day(i)=k} A_{ij}$ (abbr. I^R_{jk}). Intensity of feature update in the update log of a new version $I^U(f_j, k)$ (abbr. I^U_{jk}) is measured similarly.

Changing trend of feature intensity along with time (*intensity trend* in short) is obtained based on a feature's daily intensity. Taking the data of `Instagram` App as an example, the intensity trend of a feature f related to *feed* with keywords `feed`, `chronological`, `posts`, etc., is shown in Fig. 2. X-axis is timeline, y-axis is the intensity of feature request/update, the curve is the intensity trend of the feature requested in reviews, and the vertical lines represent the intensity of the feature updated in new versions. This chart is described by $TC(f, t_s, t_e) = \{(k_1, I^R_{f,k_1}, I^U_{f,k_1}), ..., (k_N, I^R_{f,k_N}, I^U_{f,k_N})\}$ where N is the total number of days between date t_s and t_e, and $k_1, ..., k_N$ are N consecutive days. Such chart combines intensity trends of a specific feature in both user reviews and update logs, thus is helpful to observe the global correlation between them.

[2] http://www.lextek.com/manuals/onix/stopwords1.html.

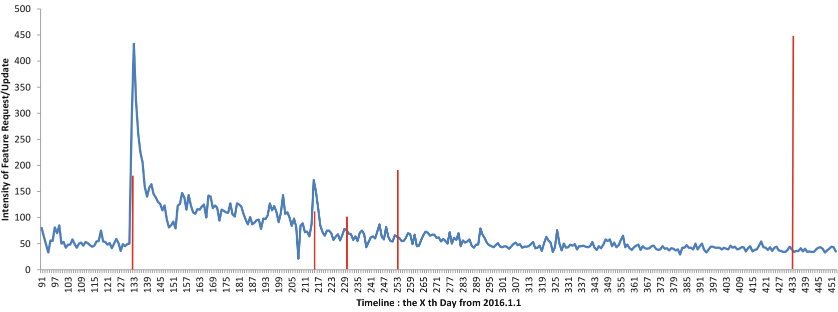


Fig. 2. Feature trend of a feature of Instagram App (Color figure online)

4.3 Atomic Update Units (AU)

The intensity trend for one feature is to be decomposed into a set of "Atomic Update Units (AU)" in terms of a set of App updates which are considered as the actions developers take to respond to user requests in reviews. $TC(f, t_s, t_e) \rightarrow \{AU(f, T_1), AU(f, T_2), ...\}$, and $\forall i$, $AU(f, T_i) =< \{(k_{i1}, I^R_{f,k_{i1}}), (k_{i2}, I^R_{f,k_{i2}}), ...\}, I^U_{f,T_i} >$, where $k_{i1}, k_{i2}, ...$ are consecutive days during the time interval T_i, and I^U_{f,T_i} is the intensity of f updated in the last day of T_i. $AU(f, T_i)$ represents an interval T_i in which users make continuous requests on f in their reviews, and what the developers act on these requests is to make an update on f in the last day of T_i but there are no other updates during T_i.

Easily to imagine, lengths of different AUs are different. To help identify update patterns from AUs, we make normalization on the length of AUs by transforming *absolute dates* into *relative dates* and consequently all AUs are with the same length (e.g., 20 relative time points). A piecewise fitting method is adopted for this transformation. Detailed steps for decomposing intensity trend chart into normalized AUs are listed in Algorithm 1.

For each AU, we calculate the sufficiency and responsiveness to measure the speed and degree describing how developers act on a requested feature in user reviews after the last update (version) on this feature.

Update sufficiency for a feature, i.e., ratio of the update intensity w.r.t. aggregated request intensity in user views, measures to what degree App developers could consider the expectation of users. The higher the value is, the more sufficient the update is. It is measured by $Suf(f, AU_i) = Norm(\frac{I^U_{f,T_i}}{\sum_{k=t^e_i}^{t^s_i} I^R_{f,k}})$ where $Norm(\cdot)$ is a normalization function to make $Suf(f, AU_i) \in (0, 1]$.

Update responsiveness for a feature, i.e., how long it takes for a developer to decide whether to update a requested feature into a new version, is measured by the ratio of the interval in which the features are significantly mentioned w.r.t. the interval between two neighboring versions that both update this feature. It demonstrates to what degree developers act on user expectations timely. To measure responsiveness, in an AU_i, we first calculate a specific time d_i to which

Algorithm 1. Decomposing intensity trend chart into normalized AUs

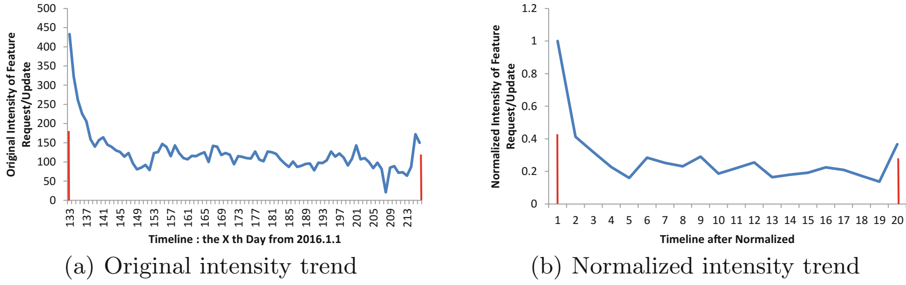
Input: Intensity trend $TC(f, t_s, t_e)$ of a specific feature f in a time interval (t_s, t_e), Number of relative time points in one normalized AU: N
Output: A set of normalized atomic update patterns AU_Set

```
1: size ← number of updates during (t_s, t_e) − 1, AU_Set ← ∅
2: for each i ∈ [1, size] do
3:     NIS ← ∅, T_i = (t_i^s, t_i^e) is the time interval of the i-th AU, N_i = t_i^e − t_i^s + 1
4:     OIS = getFeatureIntensityByDay(TC(f, t_s, t_e), T_i)
5:     max_int = getMaxIntensity(OIS)
6:     for each k ∈ [t_i^s, t_i^e] do
7:         OIS[k] ← OIS[k] / max_int
8:     end for
9:     NIS[1] ← OIS[t_i^s], NIS[N] ← OIS[t_i^e]
10:    for each k ∈ [2, N − 1] do
11:        index ← k / ((N−1) × (t_i^e − t_i^s))
12:        if index is an Integer then
13:            NIS[k] ← OIS[index]
14:        else
15:            NIS[k] ← 1/2 (OIS[⌊index⌋] + OIS[⌈index⌉])
16:        end if
17:    end for
18:    AU(f, T_i) ←< NIS, I^U_{f,T_i} > and add AU(f, T_i) into AU_Set
19: end for
20: return AU_Set
```

the interval from the last update is (t_i^s, d_i); in this interval, aggregated intensity of the requested feature is high enough (above a threshold τ), i.e., $\sum_{k=t_i^s}^{d_i} I_{f,k}^R \geq \tau$, thus $Resp(f, AU_i) = \frac{d_i - t_i^s}{t_i^e - t_i^s}$. A special case is $\sum_{k=t_i^s}^{t_i^e} I_{f,k}^R \leq \tau$, indicating that before the aggregated intensity of f in reviews has not yet reached the threshold, the developers make update on it, thus $Resp(f, AU_i) = 1$. This is the best case indicating developers always take actions on feature requests as quickly as possible. In the study we set $\tau = 4$. It should be noted that we have checked that choosing different *threshold* results in similar CDF of the responsiveness of all AUs.

Sufficiency and responsiveness are both relative measures, i.e., we can compare $Suf(f, AU_i)$ and $Suf(f, AU_j)$ to infer in which AU the feature f is updated more sufficiently, and compare $Resp(f, AU_i)$ and $Resp(f, AU_j)$ to infer in which AU the feature f is updated more timely. Solely observing the values of $Suf(f, AU)$ or $Resp(f, AU)$ does not help to draw meaningful conclusions.

Figure 3(a) shows an example AU extracted from the history of `Instagram`, for a feature `Feed`, between May 21, 2016 and August 3, 2016 (i.e., the part between the first and second red lines in Fig. 2). It is normalized by Algorithm 1 and the result is shown in Fig. 3(b). The responsiveness is 1 and the sufficiency is 0.0249, indicating that this update is comparatively timely, but the developers did not address the requested feature adequately.



(a) Original intensity trend (b) Normalized intensity trend

Fig. 3. An example Atomic Update Unit for a feature in *Instagram*

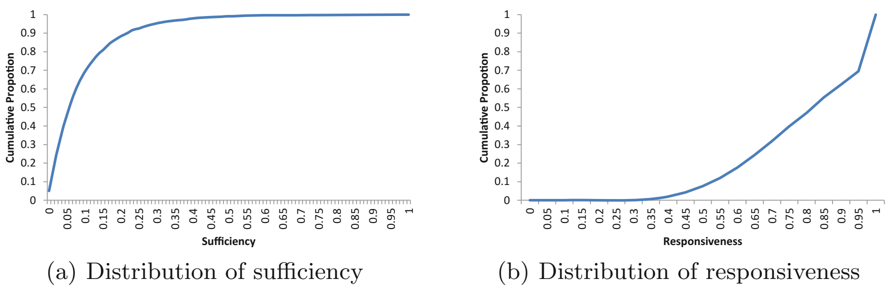
Figure 4(a) and (b) give the distribution of responsiveness and sufficiency of all 2,429 AUs of 120 Apps, respectively. Sufficiency follows power-law distribution, while responsiveness is a right-skewed distribution. To note that, we have checked and found that changing the threshold in the two measurement does not drastically change the shape of their distribution.

4.4 Mining Frequent Update Patterns

Now the "intensity trend" for each feature is decomposed into a set of AUs, and one $AU(f, T_i)$ is with four parts: normalized requested intensity in each day $\{(k_{i1}, I^R_{f,k_{i1}}), ...\}$, normalized update intensity in the last day of this AU (I^U_{f,T_i}), and sufficiency (Suf) and responsiveness ($Resp$) of this update.

We try to find out the common patterns among all AUs of all Apps in their history. A clustering approach is employed to partition AUs into groups: AUs in the same group share comparatively more similar characteristics (requested feature intensity trend, update intensity, sufficiency and responsiveness), while distances between AUs in different groups are larger. We take the `centroid` of each group to be a representative "Update Pattern". `k-Means` algorithm is employed for clustering and Euclidean distance is used for similarity measurement.

A key issue is to specify the number of clusters (k) in `k-Means` algorithm. We assign different values (2 to 18) to k and make experiments and calculate



(a) Distribution of sufficiency (b) Distribution of responsiveness

Fig. 4. Distribution of Sufficiency and Responsiveness of AUs of 120 Apps in one year

the average distance between `centroids` of clustering results; along with the increase of k, the average distance gradually decreases, and when $k = 7$, the decreasing speed becomes significantly slow. Thus we choose $k = 7$.

Table 1 gives an overview of the 7 update patterns. It is easy to see that these patterns show significant diversity on intensity trend of features in user reviews (column 5), update sufficiency (column 2), and update responsiveness (column 3), implying they have the ability of differentiating how developers take update behaviors on user reviews. Take Pattern 2 and Pattern 4 as examples, Pattern 2's responsiveness and sufficiency are comparatively lower than the ones

Table 1. Overview of the identified 7 Update Patterns

No.	Sufficiency (ranking)	Responsiveness (ranking)	Ratio	Intensity trend
Pattern 1	0.195(2)	0.803(4)	6.67%	
Pattern 2	0.050(6)	0.455(7)	7.70%	
Pattern 3	0.058(4)	0.606(6)	16.43%	
Pattern 4	0.349(1)	0.996(1)	8.53%	
Pattern 5	0.049(7)	0.745(5)	18.66%	
Pattern 6	0.102(3)	0.995(2)	24.84%	
Pattern 7	0.054(5)	0.891(3)	17.17%	

of Pattern 4, and the decreasing trend of the requested intensity in Pattern 2 is more gentle than the one in Pattern 4. Ratios of these UPs (column 4) vary from 6.67% to 24.84%, indicating that they are widespread in real world.

5 Empirical Study

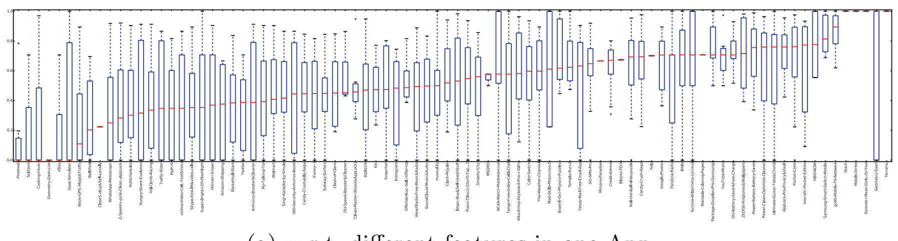
Since we have got 7 diversified UPs, now we focus on how UPs are adopted on different/similar features in the same or different Apps, and at different times.

5.1 Update Patterns w.r.t. Different Features in One App

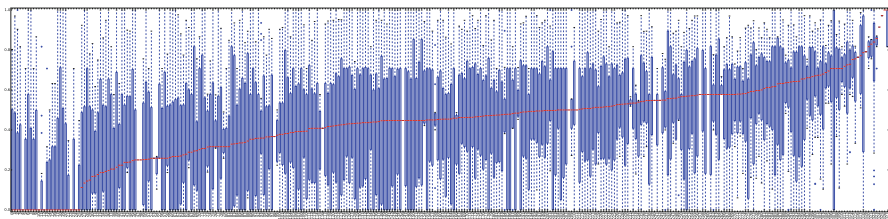
In terms of one App, there are multiple features requested in reviews and updated in new versions that are conducted by the same developers. We check whether different update patterns are adopted for different features in the same App.

For each feature in an App, we construct a vector containing the ratios of 7 Update Patterns adopted in all AUs of this App, i.e., $v(App, f) = (r_1, r_2, ..., r_7)$, and r_i is ratio of the i-th UP. If $v(App, f_j)$ and $v(App, f_k)$ are similar, we can infer that f_j and f_k exhibit similar update pattern distributions in this App. Cosine similarity is used to measure the similarity of two vectors.

For all Apps, the distribution of the similarity of arbitrary feature pairs in the same App is shown in Fig. 5(a), where each column is an App. Most Apps show quite scattered distributions in $[0, 1]$, and there are 73.2% Apps having the variance of feature pair similarity being above 0.2.

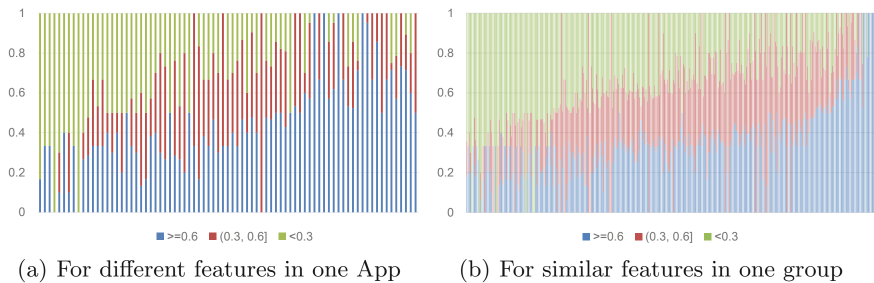


(a) w.r.t. different features in one App



(b) w.r.t. similar features in one group

Fig. 5. Distribution of update patterns similarity



(a) For different features in one App (b) For similar features in one group

Fig. 6. Classification of update pattern similarity (high, medium, low)

We calculate the ratios of feature pairs in the same App with high (≥ 0.7), medium [0.3, 0.7) and low (< 0.3) similarity, respectively. Result is shown in Fig. 6(a) where each column is an App. Different Apps show quite different observations: a small proportion of Apps on the left side tend to adopt different update patterns for different feature pairs, while a larger proportion on the right side tend to adopt similar update patterns. Holistically, developers tend to adopt more similar update patterns for different features in their own Apps.

5.2 Update Patterns w.r.t. Similar Features Across Multiple Apps

Considering multiple Apps together, there are common features across them. We care about whether similar features in different Apps follow similar update patterns. First we identify similar features among feature sets of different Apps. Keyword matching is used for this purpose, i.e., if two features are described by two similar sets of keywords, there is high probability with which the two features represent the same one. 17,906 pairs of similar features across Apps are identified, and they are grouped into 328 clusters in terms of feature similarity.

Similar as in Sect. 5.1, we construct a vector $v(App, f) = (r_1, r_2, ..., r_7)$ for each f in each App, and then calculate the Cosine similarity between two vectors $v(App_i, f_k)$ and $v(App_j, f_l)$ where $i \neq j$ and f_k, f_l belong to the same feature group. For feature groups, the distribution of similarity of feature pairs in the same group is shown in Fig. 5(b) where each column is a feature group. We can see the degree of dispersion is higher than the one in Fig. 5(a).

Figure 6(b) shows the ratios of feature pairs in the same group with high (≥ 0.7), medium ([0.3, 0.7)) and low (< 0.3) similarity, respectively. Compared with Fig. 6(a), a larger proportion of groups tend to adopt different update behaviors. Statistics show that there are 83.5% groups having the variance of feature pair similarity above 0.2. This implies that similar features across multiple Apps more tend to adopt diversified update patterns compared with different features in the same App, and the adoption of update patterns on App features depends more largely on "App developers" themselves rather than on "features".

5.3 Update Patterns w.r.t. Timeline

Here we consider the time perspective, i.e., along with time, do the update patterns of a specific feature change frequently or remain stable? We pay attention to the "update history" of a single feature in an App.

Suppose in a time interval, a feature f is updated n times in n versions, and the update pattern for the i-th updates is p_i. There is an update pattern sequence $Q(f) = p_1 \rightarrow p_2 \rightarrow ... \rightarrow p_n$. Because $p_1, ..., p_n$ are labels of update patterns rather than numerical values, we cannot measure stability of the sequence from the numerical fluctuation degree's perspective. We use two stability measures:

(1) *local stability*: what proportion of two neighboring updates follow the same update patterns, i.e., $Stab^l(f) = \frac{1}{n-1} \times |\{(p_i, p_{i+1})|p_i = p_{i+1}, i = 1, 2, ..., n-1\}|$;

(2) *global stability*: how many different update patterns appear in the sequence and what the degree of their dispersion is, measured by `negative entropy`, i.e., $Stab^g(f) = \sum_i N(p_i) \log N(p_i)$ where $N(p_i)$ is the times of p_i appearing in $Q(f)$. Then $Stab^l(f)$ and $Stab^g(f)$ are combined with an average to get $Stab(f)$.

Figure 7 shows the distribution of update pattern stability of multiple features in the history of each App. We found that the dispersity of such stability are smaller compared with Fig. 5(a) and (b), with only about 36.0% having the invariance larger than 0.2. The median of most Apps is less than 0.4, indicating that most of them are not stable, i.e., developers tend to lack enough coherence/continuity

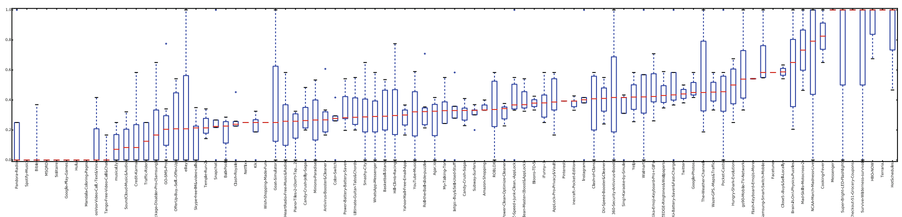


Fig. 7. Distribution of update pattern stability of features w.r.t. Apps

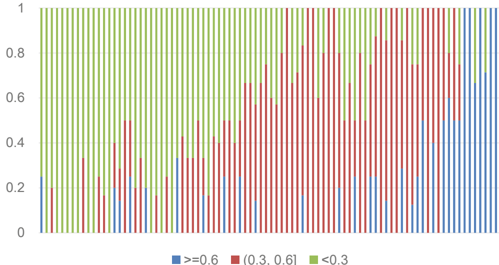


Fig. 8. Classification of update pattern stability w.r.t. Apps (Color figure online)

when they consider to update the same feature at different time. This can be also proved by Fig. 8 where the green (very unstable) and red (medium stable) are dominating, and the blue (very stable) are the minority.

6 Conclusions

In this paper, we identified 7 types of common Update Patterns which generally appear in App developers' behaviors that they perform to act on the requests raised by App users in their reviews. Update patterns are characterized by the feature intensity trend between two neighboring updates, the update sufficiency, and the update responsiveness. Statistics are made to validate a set of hypothesis. This would help developers get a clear understanding on the common habits of how they act on user reviews during delivery and evolution of their Apps.

Future work is to empirically validate whether there is latent causality between update patterns and the popularity/rating of an App, i.e., would some update patterns significantly boost the recognition of Apps, while others not? This helps recognize App developers' bad habits on incorporating user reviews in App updates and give suggestions on adopting more appropriate update patterns in future's App updates. Another work is to exploit a machine learning based predication method which gives App developers advices to update specific features based on user reviews published from the last update to the current date, so that "voices" of App users are to be addressed more timely and adequately.

Acknowledgments. Work in this paper is supported by the Natural Science Foundation of China (No. 61772155, 61472106).

References

1. Carreño, L.V.G., Winbladh, K.: Analysis of user comments: an approach for software requirements evolution. In: International Conference on Software Engineering, pp. 582–591. IEEE (2013)
2. Cerf, V.G.: Apps and the web. Commun. ACM **59**(2), 7–7 (2016)
3. Chen, N., Lin, J., Hoi, S.C., Xiao, X., Zhang, B.: Ar-miner: mining informative reviews for developers from mobile app. marketplace. In: International Conference on Software Engineering, pp. 767–778. ACM (2014)
4. Cugola, G., Ghezzi, C., Pinto, L.S., Tamburrelli, G.: Adaptive service-oriented mobile applications: A declarative approach. In: International Conference on Service-Oriented Computing, pp. 607–614. Springer (2012)
5. Finkelstein, A., Harman, M., Jia, Y., Martin, W., Sarro, F., Zhang, Y.: App. store analysis: Mining app. stores for relationships between customer, business and technical characteristics. Research Note of UCL Department of Computer Science 14, 10 (2014)
6. Gorla, A., Tavecchia, I., Gross, F., Zeller, A.: Checking app. behavior against app. descriptions. In: International Conference on Software Engineering, pp. 1025–1035. ACM (2014)

7. Guzman, E., Maalej, W.: How do users like this feature? a fine grained sentiment analysis of app. reviews. In: International Conference on Requirements Engineering, pp. 153–162. IEEE (2014)
8. Hao, Y., Wang, Z., Xu, X.: Empirical study on the interface and feature evolutions of mobile apps. In: International Conference on Service-Oriented Computing, pp. 657–665. Springer (2016)
9. Iacob, C., Harrison, R.: Retrieving and analyzing mobile apps feature requests from online reviews. In: IEEE Working Conference on Mining Software Repositories, pp. 41–44. IEEE (2013)
10. Iacob, C., Harrison, R., Faily, S.: Online reviews as first class artifacts in mobile app. development. In: International Conference on Mobile Computing, Applications, and Services, pp. 47–53. Springer (2013)
11. Khalid, H., Shihab, E., Nagappan, M., Hassan, A.E.: What do mobile app users complain about? IEEE Softw. **32**(3), 70–77 (2015)
12. Maalej, W., Nabil, H.: Bug report, feature request, or simply praise? on automatically classifying app. reviews. In: IEEE International Conference on Requirements Engineering, pp. 116–125. IEEE (2015)
13. Martin, W., Sarro, F., Harman, M.: Causal impact analysis applied to app. releases in google play and windows phone store. Research Note of UCL Department of Computer Science 15, 07 (2015)
14. Martin, W., Sarro, F., Jia, Y., Zhang, Y., Harman, M.: A survey of app. store analysis for software engineering. IEEE Transactions on Software Engineering, PP(99), 1–32 (2016)
15. McIlroy, S., Ali, N., Hassan, A.E.: Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. Emp. Softw. Eng. **21**(3), 1346–1370 (2016)
16. McIlroy, S., Ali, N., Khalid, H., Hassan, A.E.: Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews. Emp. Softw. Eng. **21**(3), 1067–1106 (2016)
17. Nayebi, M., Adams, B., Ruhe, G.: Release practices for mobile apps-what do users and developers think?. In: IEEE International Conference on Software Analysis, Evolution, and Reengineering, vol. 1, pp. 552–562. IEEE (2016)
18. Pagano, D., Maalej, W.: User feedback in the appstore: An empirical study. In: IEEE International Conference on Requirements Engineering, pp. 125–134. IEEE (2013)
19. Palomba, F., Linares-Vásquez, M., Bavota, G., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A.: User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In: IEEE International Conference on Software Maintenance and Evolution, pp. 291–300. IEEE (2015)
20. Syer, M.D., Nagappan, M., Hassan, A.E., Adams, B.: Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source android apps. In: Conference of the Center for Advanced Studies on Collaborative Research, pp. 283–297. IBM Corp. (2013)
21. Takahashi, H., Nakagawa, H., Tsuchiya, T.: Towards automatic requirements elicitation from feedback comments: Extracting requirements topics using lda. In: International Conference on Software Engineering and Knowledge Engineering, pp. 489–494 (2015)
22. Vu, P.M., Nguyen, T.T., Pham, H.V., Nguyen, T.T.: Mining user opinions in mobile app. reviews: A keyword-based approach. In: International Conference on Automated Software Engineering, pp. 749–759. IEEE (2015)

23. Wang, H., Wang, Z., Xu, X.: Time-aware customer preference sensing and satisfaction prediction in a dynamic service market. In: International Conference on Service-Oriented Computing, pp. 236–251. Springer (2016)
24. Yan, X., Guo, J., Lan, Y., Cheng, X.: A biterm topic model for short texts. In: International World Wide Web Conference, pp. 1445–1456. ACM (2013)