This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

# STRE: An Automated Approach to Suggesting App Developers When to Stop Reading Reviews

Youshuai Tan, Jinfu Chen, Weiyi Shang, Tao Zhang, Sen Fang, Xiapu Luo, Zijie Chen, and Shuhao Qi

**Abstract**—It is well known that user feedback (*i.e.*, reviews) plays an essential role in mobile app maintenance. Users upload their troubles, app issues, or praises, to help developers refine their apps. However, reading tremendous amounts of reviews to retrieve useful information is a challenging job. According to our manual studies, reviews are full of repetitive opinions, thus developers could stop reading reviews when no more new helpful information appears. Developers can extract useful information from partial reviews to ameliorate their app and then develop a new version. However, it is tough to have a good trade-off between getting enough useful feedback and saving more time. In this paper, we propose a novel approach, named STRE, which utilizes historical reviews to suggest the time when most of the useful information appears in reviews of a certain version. We evaluate STRE on 62 recent versions of five apps from Apple's App Store. Study results demonstrate that our approach can help developers save their time by up to 98.33% and reserve enough useful reviews before stopping to read reviews such that developers do not spend additional time in reading redundant reviews over the suggested stopping time. At the same time, STRE can complement existing review categorization approaches that categorize reviews to further assist developers. In addition, we find that the missed top-word-related reviews appearing after the suggested stopping time contain limited useful information for developers. Finally, we find that 12 out of 13 of the emerging bugs from the studied versions appear before the suggested stopping time. Our approach demonstrates the value of automatically refining information from reviews.

**Index Terms**—Mobile applications, maintenance, app reviews.

## 1 Introduction

Mobile devices have become increasingly popular in recent years. People use Mobile Applications (*i.e.*, apps) to chat, shop, and kill time online, especially during the very challenging pandemic period. In 2020, people have downloaded apps about 218 billion times around the whole world[1].

App markets, *e.g.*, Apple's App Store, allow users to rate apps and submit reviews to express their opinions such as complaints and suggestions for new features, which is very useful for developers to improve the quality of their apps. Noted that massive users may encounter bugs which are hardly found by developers during app development [1]. Failures in such apps can hurt the user experience and cause financial loss. Fortunately, developers could find these bugs from reviews. Many prior studies [2–9] have proposed to use reviews to maintain apps. By using the questionnaire method, AlSubaihin *et al.* [10] found 51% of developers frequently/very frequently utilize reviews to fix bugs and find new features.

However, it is infeasible to examine and analyze a plethora of reviews manually in practice. Pagano and Maalej [5] found that an app receives up to 22 reviews every day on average. Especially, several popular apps contain tremendous reviews, *e.g.*, WeChat gets about 60,000 reviews per day [11]. Actually, developers face the dilemma between ample information from reviews and their costs. On one hand, the more reviews are, the more valuable feedback returns. On the other hand, reading too many reviews may delay the release schedule of the software system, especially in a fast-paced release cycle. Maintenance is the final phase of app development lifecycle, which is a continuous process [3]. Developers combine reviews and their own findings to develop a new version and repeat this process. Intuitively, repeated information is contained in reviews over a long period of time [12], which is confirmed in our manual studies. Therefore, to improve developing efficiency, developers could stop reading reviews to develop a better version when most useful information has been submitted for the current version. However, it is impossible to guess this special time.

To bridge this gap, we design an algorithm named STRE, *i.e.*, deciding when to **ST**op reading user **RE**views. In particular, we use a topic modeling technique, *i.e.*, Latent Dirichlet Allocation (LDA) [13] to extract the topics appearing in the historical reviews of the app. We leverage the topics as a vehicle to measure the information that is conveyed in the reviews. If developers do not see new words in each topic from the reviews in a new version for a long period of time, we consider that developers may be unlikely to observe new information by reading more reviews anymore. The length of the waiting time period for new information appearing in each topic is empirically learned from historical reviews in a conservative manner (see details in Section 3.3.2), in order to avoid missing important reviews.

- *Y. Tan, T. Zhang, S. Fang, and Z. Chen are with the School of Computer Science and Engineering, Macau University of Science and Technology, Macau, China.*
- *J. Chen is with Centre for Software Excellence of Huawei Technologies Canada, Kingston, Canada.*
- *W. Shang is with the Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada.*
- *X. Luo is with the Department of Computing, the Hong Kong Polytechnic University, Hong Kong, China.*
- *S. Qi is with the Department of Computer Science, the University of Manchester, Manchester, UK.*
- *T. Zhang is the corresponding author.*

1. https://www.statista.com/

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

We evaluate the effectiveness and efficiency of STRE by suggesting the time to stop reading reviews for versions in five popular apps from iOS. Our results show that our approach can reduce the time needed to read reviews by a large amount. In particular, with covering most of the information (to see details, refer to RQ1 and RQ2 in Section 5) our approach can effectively suggest stopping time that reduces the total time for reading reviews by up to 98.33%. What's more, as STRE could complement existing review categorization approaches, our approach can save effort of 83.58% and 84.64% on average for a classification-based approach [14] and a cluster-based approach [15], respectively. In addition, by manually examining the missed topic-related reviews that appear after the suggested stopping time, we find seven categories of information, where 89% of the reviews are not useful for developers. Finally, we find that the early stopping time does not prevent the developer from identifying emerging bugs from the reviews. To help researchers reproduce our work, we make the dataset, code, and study results available in our replication package [2]. The contributions of our paper are summarized as below:

- To the best of our knowledge, our approach is the *first* to design a tool which could help developers decide when to stop reading reviews. STRE could improve the efficiency of app maintenance.
- We show the effectiveness of STRE via presenting our encouraging results on five popular apps with more than 60 different versions.

The remainder of the paper is organized as follows. In Section 2, we describe the background and motivation of our work. The detailed methodology of STRE is described in Section 3. Section 4 introduces the study setup. Study results are presented in Section 5. Section 6 explains our interviews with five developers. We discuss limitations and present related work in Section 7 and Section 8, respectively. We conclude our paper in Section 9.

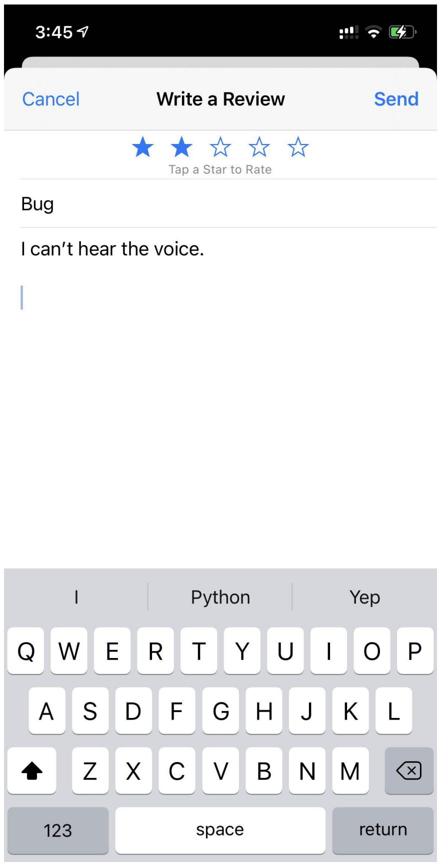
## 2 Background and Motivation

### 2.1 Reviews

Apps provide a channel for users to express their comments concerning bugs, new features, praise for apps, etc. Figure 1 shows the interface that users rate and comment on for Zoom. In this case, the user complains a bug that she cannot hear the voice in Zoom meeting.

Reviews from different apps have diverse characteristics. Due to apps' own functions, users may comment more about several parts of one app. For example, Zoom users usually have some troubles with the microphone and network connection. Moreover, several apps' users may tend to give their feedback quickly after a new version for their beloved apps while some do not. For instance, many people can watch YouTube videos at any time, therefore, they always write the reviews quickly once the version is updated even at midnight.

Reviews all too often are used to improve mobile apps. About 51% of developers frequently/very frequently use reviews to enhance and maintain apps [10]. To listen to their users, some developers try to read all reviews[3].



Bug

I can't hear the voice.

Fig. 1: An example of review window in Zoom.

### 2.2 App lifecycle

The maintenance phase is a continuous process of app development [3]. Developers read reviews to fix bugs or perform improvements at regular intervals in the form of updates to the app [3]. Meanwhile, users post their reviews after the release of a new version.

Since app stores have review procedures to raise the qualities of apps, developers have to delay the release of a new version [10]. For example, a developer who participates in AlSubaihin *et al.*'s [10] research complains that: *"So you try to avoid this horrible situation, which we've been in a few times, where you release something and then it breaks and then you have 11 days of letting your users down and getting negative reviews. You can't do anything about it because Apple takes a long time."*

### 2.3 Motivation

Let us consider a hypothetical scenario. As an app developer, James attaches importance to reviews. Since one version of James's app was released yesterday, thus he analyzes the new reviews tonight. After reading existing reviews, he extracts two bugs and one feature request. Because one of the bugs is very serious and the app store's review

2. https://shorturl.at/apM68

3. https://www.quora.com/Do-app-developers-actually-read-the-reviews-people-give-them

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

mechanism will cost much time, he has to fix bugs and add this feature immediately. Hence, he faces an awkward dilemma. On one hand, he does not want to read the following reviews to save time. On the other hand, he can not ensure that no more useful information will appear in reviews. Therefore, James needs a tool to suggest him when to stop reading reviews.

Due to the above-mentioned motivation, we design a novel approach named STRE. Since reviews from a certain app have their own characteristics, STRE utilizes historical reviews from this app to suggest when to stop reading reviews.

## 3 Methodology

In this section, we present our approach, *i.e.*, STRE, which automatically suggests when to stop reading reviews on a new version of an app based on the historical app reviews. In general, our approach would suggest the stopping time if no new information appears in the app reviews for a certain period of time. An overview of our approach is shown in Figure 2. To sum up, our approach STRE consists of four steps: (1) preprocessing app reviews; (2) identifying review topics (*i.e.*, extracting old information); (3) learning when to stop (*i.e.*, studying the distribution of old information); and (4) suggesting when to stop.

### 3.1 Preprocessing app reviews

In the first phase, we preprocess app reviews. The real-world reviews are highly susceptible to noisy, misspelled, and inconsistent contents due to their huge size and being written by the app users who have different background [16]. Low-quality reviews would lead to low-quality experimental results. In particular, similar to the prior study [17], we adopt the rule-based approach to preprocess our data by the following several steps.

#### *3.1.1 App reviews cleaning*

As reviews may contain noisy and inconsistent contents, in this step, we wish to clean noise data and make reviews consistent. In particular, we first convert capital letters in reviews into lowercase letters. Second, we remove the non-English words and punctuations. Generally, the number of consecutive repeated letters is less than three [17]. Therefore, next, for each word in a sentence, we remove the redundant consecutive repeated letters if the number of a consecutive repeated letter is more than two by using regular expression. For example, the word "niceeee" is converted into "nice". In addition to repeated letters, there exist consecutive duplicated words. Therefore, we also remove consecutive duplicate words in a sentence of each review with regular expression, *e.g.*, "very very very good" is changed to "very good". Finally, we remove the stop words that are common words (*e.g.*, "the", "a", "in") which present meaningless information.

#### *3.1.2 Reviews tokenization*

In the second step, we transform reviews into tokens. In particular, each sentence of the review is divided into tokens. Tokens of all sentences are used to form token sequence(s). In this work, we use the popular library Natural Language Toolkit (NLTK)[4] [18] to perform tokenization.

#### *3.1.3 Reviews transformation*

To make our analysis more understandable and efficient, in the last step, we transform reviews into consolidated forms which are appropriate for analysis. First, we extract the numbers from each review. In particular, we replace all numbers with a unified word, *i.e.*, "$\langle digit \rangle$". End users might type a review that has spelling errors. Therefore, second, we correct the misspelled words in each review. In our approach, we use a generic spell checking library Enchant[5] to correct reviews that have spelling mistakes. Reviews often use different forms of a word, such as love, loves, and loved. To make reviews consolidated, we lemmatize words of each review to reduce inflectional forms of a word to a common base form. The lemmatization phrase can be divided into two steps. First, we lemmatize all verbs. Then we conduct the lemmatization for nouns without "ss" ends, since it is not suitable for words ending with 'ss' (*e.g.*, "pass" to "pas"). In our approach, we use library NLTK to lemmatize the reviews.

### 3.2 Identifying review topics

In the second phase, we extract old information from historical reviews. Reviews from different versions often shared similar topics. For Amazon Shopping, users tend to comment on the conditions of this app. For example, a user of Amazon Shopping added a review "The Amazon app no longer works on my iPad". Thus we could take the word *app* as a representative word of old information.

Inspired by prior study [19–30], we use statistical topic model to identify review topics. Designed from the domain of NLP and information retrieval, topic modeling could help us organize and summarize documents at a scale that would be impossible by human annotation [31]. They are statistical methods that could conduct the unsupervised analysis for the words of the original texts to discover the themes that run through them. Thus, in our work, we use a topic model to identify topics of reviews due to the following reasons: (1) topic model requires no training data [19]; (2) topic model performs well in unstructured data such as app reviews [32]; (3) topic model is scalable to large-scale documents in practice [19]. To make our description more specific, we present an illustrative example, whose historical data are reviews of eight versions from an app named Amazon Shopping.

#### *3.2.1 Choosing a topic model*

We apply Latent Dirichlet Allocation (LDA) to identify statistical topics of reviews. LDA is a soft clustering algorithm which is ideal for text [13]. In LDA, a topic can be represented by a collection of words that co-occurred frequently in the reviews. For example, the word *app* affiliates to a topic of Amazon Shopping. Many prior studies [12, 33, 34] used LDA-based algorithms to capture the topics of reviews.

In the application of LDA for given $N$ reviews $R_1, R_2, ..., R_N$, LDA aims to automatically discover a list of

4. https://www.nltk.org
5. http://www.abisource.com/projects/enchant/

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

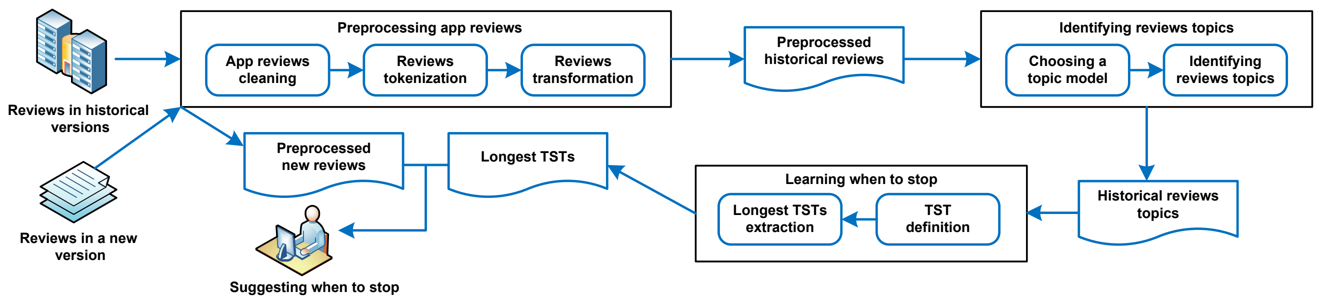


Fig. 2: An overview of our approach STRE.

TABLE 1: Our illustrative running example of identifying review topics

| Topics | Word 1 | Word 2 | Word 3 | Word 4 | Word 5 | Word 6 | Word 7 | Word 8 | Word 9 | Word 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Topic 1 | amazon 0.057 | app 0.038 | love 0.029 | work 0.019 | shop 0.018 | iPad 0.013 | update 0.009 | make 0.009 | need 0.009 | great 0.009 |
| Topic 2 | amazon 0.053 | service 0.022 | customer 0.019 | delivery 0.014 | order 0.013 | get 0.013 | package 0.012 | time 0.012 | great 0.012 | deliver 0.012 |
| Topic 3 | app 0.024 | use 0.020 | search 0.017 | get 0.014 | amazon 0.013 | item 0.013 | page 0.012 | bottom 0.012 | find 0.012 | easy 0.011 |

$(M, M <= N)$ topics, *i.e.*, $T = \{T_1, T_1, ..., T_M\}$. Each topic $T_i$ is defined by its top words, *i.e.*, top $K$ words. In other words, each topic refers to a probability distribution over the top $K$ words in the reviews.

### 3.2.2 *Identifying reviews topics*

To extract old information from historical reviews, we first set the number of topics. The number of topics, *i.e.*, $M$, is an input to control the granularity of the topics. Prior study [12] defines the number of topics ranging from six to 12. Their results show that $M$ ranging from six to 12 achieves a good trade-off between precision and recall performance metrics adopted by their approach. Similar to the prior study, we set the number of topics $M$ from five to 20 in a conservative strategy according to the coherence [35] metric. Coherence measures the relative distance between words within a topic. The value of coherence ranges from 0 to 1, and a larger value for coherence indicates a higher quality of the topic model. Thus we choose to use the number of topics with the largest coherence score as the number of topics.

For the threshold of top $K$ words in each topic, prior studies [36, 37] found that top ten words can capture most of the semantics and be used to represent the topic itself. Therefore, we take the top ten words in each review topic to stand for the topic. The output of the LDA model is $M$ topics and top ten semantically related words for each topic. Each top word has a contribution score for its corresponding topic. After this step, we identify three topics with the corresponding top ten words in each topic in our illustrative running example (cf., Table 1). The value below each top word is its contribution score. For example, the word "delivery" contributes a 0.014 score in the second topic.

## 3.3 Learning when to stop

In the third phase, we aim to use the identified topics from subsection 3.2 to study the distribution of old information. Since we take topics of historical reviews as old information, we could suggest stopping reading reviews when all topics appear in reviews of new app version. Note from last subsection 3.2, a topic consists of top words. Therefore, we speculate that a topic appears in reviews of the new app version if most of its top words have appeared.

For the above-mentioned purpose, we first define a metric, *i.e.*, Topic Stability Time ($TST$), to measure the review topic stability. Then we utilize historical reviews to learn the longest $TST$s, which helps us determine the stop point.

### 3.3.1 $TST$ *definition*

Since each topic appears in the form of top words in each version, we consider a topic in a stable state if no top word appears at that point. Therefore, we define $TST$ as the time interval between the appearance of two top words from the same topic. In our approach, each topic consists of ten top words. Therefore, there are a total of nine instances of $TST$ for each topic.

### 3.3.2 *Longest* $TST$*s extraction*

As STRE is a conservative approach, we extract the Longest $TST$s of all topics from historical reviews to set the stopping metric in reviews of a new app version. Gao *et al.* [12] proposed an approach, named IDEA, which considers that the topics of reviews from the same app is stable across different releases. Based on this intuition, the distributions of information in different versions from the same app are similar. Thus we consider that the characteristics of $TST$ in a new version would be similar to the trend of $TST$ in historical versions. We split the historical reviews by version and extract the longest $TST$s for all topics. Some app versions may have a very long $TST$. To filter out these outliers, we remove the $TST$s that are at least one standard deviation away from mean $TST$s of the topic. For example, for one study of Amazon Shopping, we extract ten top words of topic 3. The top word *easy* of topic 3 appears after 0.78 day, which is the maximum interval time between two top words. According to our method, we take 0.78 as the longest $TST$s for topic 3.

To be more clear, the red solid lines in Figure 3 show the examples of the extracted longest $TST$s. Since the distribution of topics are different, the longest $TST$s vary greatly. Several topics could be in a stable state for nearly one day, while some ones become unstable quickly (*i.e.*, a top word appears). In Figure 3, the top and bottom topics have a relatively higher longest $TST$; while the middle topic has a much lower longest $TST$.

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

### 3.4 Suggesting when to stop

In the final phase, we utilize the extracted longest $TST$s from historical reviews to suggest the stop reading time in reviews of a new app version. Given the reviews of a new version, we consider that a topic is stable if the reviews at a time point satisfy the following condition: **the stable time of such a topic is longer than the longest historical $TST$ of the same topic**. When all of the historical topics are stable in the new version, we suggest to stop reading reviews.

To explain the process of suggesting when to stop more clearly, we present a hypothetical example in Figure 3. In the first topic (*i.e.,* the first sub-figure), the longest $TST$ is 0.54 days. When stable time of the seventh top word nearly reached 0.54, the eighth top word appeared. Thus topic 1 becomes stable during the eighth top word at last. In the second topic, its longest historical $TST$ is relatively short. It becomes stable just before the tenth top word appears. In the third topic, all ten top words appear quickly, thus it becomes stable quickly. Overall, we could only stop reading reviews if all the topics exist in their stable state, which happens at the black dot in the first sub-figure.

## 4 Study Setup

In this section, we present the setup of our study which evaluates the performance of our approach.

### 4.1 Subject apps

We choose five popular open-source apps including Zoom, YouTube, Amazon Shopping, Twitter, and Starbucks as our subject systems. We choose these five apps due to the following reasons: 1) these apps contain many reviews across different versions (*e.g.*, YouTube contains an average of 1,427 reviews per version); 2) version update cycles of these apps are different. The overview of the five subject systems is shown in Table 2.

TABLE 2: Overview of Our Subject Apps

| App Name | #Reviews | #Versions |
|---|---|---|
| Zoom | 14,184 | 22 |
| YouTube | 48,517 | 34 |
| Amazon Shopping | 12,853 | 16 |
| Twitter | 31,372 | 43 |
| Starbucks | 4,943 | 18 |

### 4.2 Data collection

In this subsection, we describe the details of our data collection. We collect reviews from the App Store platform. In particular, we download reviews from Qimai Data [6]. Since we need a reasonable amount of reviews to perform our analysis, we utilize reviews that are posted from June 1st, 2020 to March 1st, 2021. In total, we collected 111,869 reviews for the five apps. Each review contains a review title, a review text, and a review rating score. Second, we extract each review's contents, title, post timestamp, and the rating score. Finally, we assign each review into the version that the review belongs to. In particular, we first extract all versions and their release timestamps. We then split the reviews into a corresponding release by comparing their timestamps. The data scale of collected reviews and versions is shown in Table 2.

6. https://www.qimai.cn/

## 5 Study Results

In this section, we present the evaluation results to verify the effectiveness of our approach.

We first build our data set. Specifically, we split all the reviews into two sets as their versions with the chronological order. The first set contains historical versions of the given reviews (training set) and the second set includes their new versions (testing set). Apps may not have readily available historical versions of reviews to learn their topics. Therefore, we adopt an incremental validation method [38] to learn the information from reviews in the historical versions. For an app with a total of n versions, we start our study with the (n/2+1)th version and use its prior n/2 versions to learn historical information. For example, Amazon Shopping has a total of 16 versions in our study. Therefore, we use the first eight versions to learn historical information in order to suggest stopping time for the 9th version. Afterward, we would use the information from the previous nine versions, to suggest the stopping time for the 10th version. The rest versions can be done in the same manner until the stopping reading time of the 15th version is suggested. Note that the last version is discarded because the last version is not a full version so that it is difficult to analyze the last version for all apps. Then we apply STRE to the data set.

As our tool misses some top words after the stopping reading time, to evaluate whether the related reviews are significant, we propose the research question RQ1: *Do reviews that appear after the suggested stopping time provide useful information?* We manually analyze these missed reviews. To gauge whether STRE would not cover several emerging bugs, we bring up RQ2: *Do emerging bugs appear after the suggested stopping time of reading reviews?* We utilize an external tool to conduct the studies in RQ2. RQ1 and RQ2 could confirm that STRE would not miss important information. To gauge whether STRE could help developers save much time and improve the developing efficiency, we formulate RQ3: *How much time could developers save by using our algorithm?* We empirically show that developers could save much effort when using other existing review categorization tools [14, 15] together with the help of STRE. We use an internal metric to evaluate whether the cut reviews contain enough information in RQ3. In RQ3, we also try another way of applying STRE, which first classifies reviews into several categories [15, 39] then applies STRE to reviews of the same category afterwards. We discuss the corresponding results in RQ3.

To further evaluate STRE, we propose a simple baseline approach that is based on similarity score. We elaborate the baseline approach in RQ2. Since RQ1 assesses STRE from the perspective of its construction (*i.e.*, top words), we just compare the baseline approach with STRE in RQ2 and RQ3.

**RQ1: Do reviews that appear after the suggested stopping time provide useful information?**

**Motivation:** Reviews appeared before the suggested stopping time may not cover all the information. We call a

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

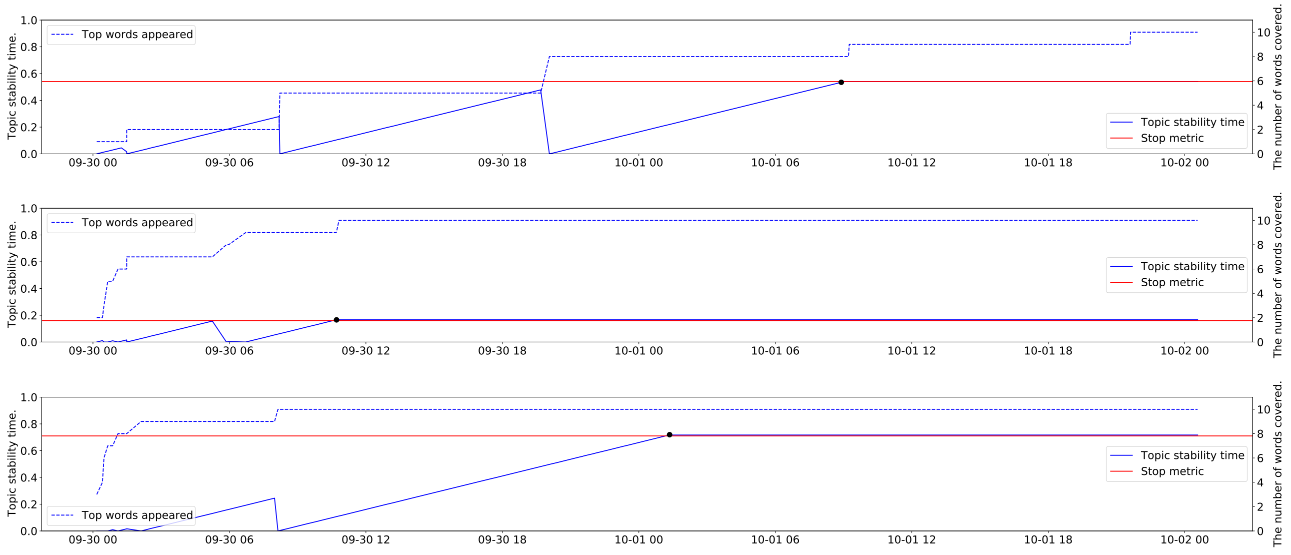


Fig. 3: An illustrative example of suggesting when to stop by using the longest $TST$s. Each sub-figure represents a certain topic of historical reviews. The red solid line is the stop metric (*i.e.*, the longest $TST$). The blue solid line is the $TST$. The blue dotted line is the number of top words that have appeared. The dark spot is the time that this topic becomes stable.

review that appears after the suggested stopping time and covers the top words in the review topics as a missed review. Such missed reviews might play an important role for developers. Therefore, in this research question, we would like to know whether the missed reviews in each topic are significant for developers.

**Approach:** To identify the usefulness and informativeness of reviews, the prior study [14] classifies app reviews into different categories, *e.g.*, bug and feature request. Such categories are helpful for developers to develop and maintain apps. Similar to the prior study, we classify the missed reviews into different categories and discuss the usefulness for developers of each category. To achieve the above-mentioned goal, we follow a systematic process to classify missed reviews. The process is described as follows:

*Step 1: Extracting reviews.* In this step, we extract the missed reviews based on the missed top words. For each new version in each studied app, we extract the missed top words, *i.e.*, the top word appearing after the suggested stop reading time. Then we extract the reviews that contain the missed top words after the suggested stop reading time.

*Step 2: Generating initial categories.* The manual examination of the categories of reviews may be subjective. To mitigate subjectivity, two authors of this paper first read all the missed reviews, then put reviews into several categories individually. They are majoring in computer science and have at least five years experience of software development.

*Step 3: Generating final categories.* With the initial categories generated in the last step, the two authors discuss their categories to reach a consensus on a final category, which is used in the next step to classify missed reviews.

*Step 4: Classifying missed reviews.* Two authors classify each missed review into a specific category that is generated from the last step, independently. An agreement ratio is calculated based on the independent classification. If there exists any disagreement, a discussion is held with another author in order to come to a consensus.

*Step 5: Discussing the value of the missed reviews.* To understand whether these missed reviews contain much useful information, the two authors discuss each review one by one. The discussion by the two authors considers both the information in the missed review themselves and the information that is already provided in the reviews before the suggested stopping time. Reviews containing pure noise may not be very useful to the developers, while reviews about reporting an unseen *bug* can be critical for developers [14].

**Result: Missed reviews are classified into seven categories.** Table 3 shows the results of the collected missed reviews and their corresponding categories. We find only 170 missed reviews based on the missed top words, while the number of reviews appearing before the suggested stopping time is 1,480. The ratio between relevant reviews before and after the stopping reading time is 8.71. Such a result also reflects that our approach achieves a high information coverage rate. In addition, such a small scale number makes our manual study on classifying missed reviews feasible. The missed reviews are classified into seven categories: bug, special feature request, general feature request, praise, dispraise, pure noise, and information sharing (cf., Table 3). The Cohen's and Fleiss' kappa agreements of the two authors' results are 0.8745 and 0.874 (almost perfect agreements [40, 41]), respectively.

**The missed reviews provide very limited help for developers.** Table 3 shows the number of missed reviews in each category, where we assume that only bug and general feature request are vital categories for developers [10]. The total number of them is 45 (4+15+26). Also, two authors manually find the identical/ similar one of a missed review, which appears before our suggested stopping time. To mitigate the subjective bias, We only consider multiple reviews being about the same bug/ general feature request if both authors agree. These missed reviews are useless for developers, as they are reviews about the same bug

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

or feature request appeared before the suggested stopping time. The result shows that 27 of 45 reviews (*i.e.*, vital reviews) are belong to these reviews, and the ratio is 0.6. Such a result implies that only 18 reviews may contain important information for developers. The rate between these 18 reviews and all 170 comments is 0.11. We discuss the usefulness of each category of reviews in detail below.

**Bugs.** Reporting bugs is one of the major reasons of reviewing an app [42]. Missing reading a review that reports a critical bug may be detrimental for an app. We find a total of 19 missed reviews that are related to bugs. These bugs can be divided into update-related and update-unrelated. Out of the 19 reviews, we find that eight of the associated bugs are already reported by other users before our suggested stopping time. For example, one of the missed reviews in app Starbucks mentioned that *"Upon using the app, it crashes multiple times."*; while a similar bug was reported five days ago with different wording, mentioning *"mobile ordering is broken. Customer support just says try reinstalling the app."*.

However, the similar reviews may contain supplementary information for bug reproduction, which is the first step to fix bugs from reviews [43]. Therefore, to gauge whether we would omit valuable information for bug reproduction, we analyze the eight bugs whose reviews appear both before and after the stopping time suggested by STRE. Li *et al.* [43] found that the descriptions of actions before bugs, device, Android OS, and OS version are significant for crash reproductions according to their manual analysis. We would like to see if the review for each bug after the suggested stopping time would provide additional aforementioned information to reproduce the bug. However, we find that none of the reviews provide any additional information. Overall, we find that we could leave out the eight missed reviews that appeared after the stopping time for bug reproduction, since they do not contain any supplementary information.

For the rest 11 missed reviews that are related to bugs, we find that most of them are related to very specialized issues, such as incorrect UI operations or network signal issues. For example, a Starbucks user commented that he could not find the nearest location due to a bad internet connection. Although still with value to read, we consider that these reviews can be treated as low-priority issues for developers when the amount of reviews are huge.

**Special feature request.** Users may request special features that may not be useful for all the apps' user base. For example, a user hopes that Starbucks could add an option to pick roast for the misto. Most people may not upload this request. Thus special feature requests may contain a little useful information for developers.

**General feature request.** Developers rely on general feature requests to figure out high-priority ones that should be implemented in the next versions [44]. Thus high-priority general feature requests are vital for developers. Out of the 26 reviews, we find 19 of the general feature requests are commented on by other users before our suggested stopping time. For example, one of the missed general feature request in Starbucks mentioned that *"I love Starbucks and their app. Only thing I wish you could do is customize more."*; while a similar request was reported three days ago, mentioning *"I love this app except I don't have all the choices."*. For the rest seven missed reviews that are related to general feature requests, we find that most of them are not high-priority requests, such as personal preferences. For example, a YouTube user found that this app did not allow keyboard shortcuts to control video playback in a new version. The user of the app wants this feature back. Thus we assume that these reviews contain low-priority information for programmers.

**Praise.** Although it is important for developers to know that their users like the apps, reviews with praises may contain a little useful information for app development. For example, a user commented *"Thank goodness for Zooming. i have knitted many wonderful hours with first class teachers."* on Zoom.

**Dispraise.** Most of the dispraising reviews are not useful to developers, since such reviews typically do not contain useful guidance on how to improve the apps. For example, a user commented, *"So why does a trillionaire have such an annoying app."* on Amazon Shopping.

**Pure noise.** The reviews only containing pure noise could be removed indisputably. For example, a user commented *"Subscribe to dreammmmmmmm poggggg."* on YouTube.

**Information sharing.** Some of the reviews want to provide information to the developers and could be useful for the developers to read. However, such information often contains personalized unpleasant experiences, which are difficult for developers to have actionable guidance. For example, a Twitter user commented that *"Logged in earlier to check things out now later on I try logging in again this afternoon; it says I exceed the number of attempts."*.

Our manual study shows that 89% of the reviews that appear after the suggested stopping time do not contain useful information for developers.

**RQ2: Do emerging bugs appear after the suggested stopping time of reading reviews?**
**Motivation:** STRE just retains a few reviews after the suggested stopping reading time, the missed reviews may report serious bugs, e.g., emerging bugs [12]. An emerging bug is defined as a bug that if the bug rarely appears in a previous time period but is mentioned by a significant proportion of reviews in a more recent time period [12]. Emerging bugs are vital information for developers. Therefore, in this research question, we would like to answer whether emerging bugs appear after the suggested stopping time by our approach. In addition, the emerging bugs may only cover a few bug instances. In order to increase the scale of our study, we aim to study more bugs in the reviews of our subject apps.
**Approach: Emerging bugs.** To answer RQ2, we first collect emerging bugs for our studied apps. To collect emerging bugs of apps, we apply IDEA [12] on the raw reviews of each app. IDEA is a tool that can automatically detect emerging bugs across different versions of apps. IDEA takes the raw reviews as input data, and outputs the specific sentences that may contain emerging bugs. In particular, we reused the implementation of IDEA from Gao *et al.* [12] and ran IDEA on our collected reviews to collect emerging bugs.

IDEA may report false positive emerging bugs. Therefore, in the second step, we manually examine the emerging bugs outputted from IDEA. To rigorously identify the

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

TABLE 3: Summary of the review categories.

| Category | Definition | Number | Summary |
|---|---|---|---|
| Bug (eight same ones appeared before stopping) | Reviews that report problems. | 4 (update-related) | Eight of the associated bugs are already reported by other users before our suggested stopping time. The remaining 11 ones are related to very specialized issues, such as the bad internet connection. |
| | | 15 (update-unrelated) | |
| Special feature request | Feature request reviews that appear once in a version. | 15 | These features may not be useful for all the apps' user base. For example, a user hopes that Starbucks could add an option to pick roast for the misto. |
| General feature request | Feature request reviews appearing more than once in a version. | 26 | 19 of the general feature requests are commented on by other users before our suggested stopping time. Most of the remaining ones are not high-priority requests, such as personal preferences. |
| Praise | Reviews that expresses appreciation. | 26 | Reviews of praises may contain a little useful information for app development. For example, "*Thank goodness for Zooming. i have knitted many wonderful hours with first class teachers.*" |
| Dispraise | Reviews that opposite of praise. | 34 | Most of the dispraising reviews are not useful to developers. For example, "*So why does a trillionaire have such an annoying app.*" |
| Pure noise | Reviews that are meaningless. | 33 | Pure noise could be removed indisputably. For example, "*Subscribe to dreammmmmmmm poggggg.*" |
| Information sharing | Reviews that inform others about some information of the app. | 17 | Such information may not give developers actionable guidance. For example, "*Logged in earlier ... it says I exceed the number of attempts.*" |

specific sentences that are related to the emerging bug, two authors of this paper who have at least five years experience of software development extract emerging bugs from sentences outputted from IDEA independently. Each author removes sentences that are unclear or unrelated to bugs. If there exists any disagreement, a discussion would take place with another author in order to come to a consensus.

Furthermore, we evaluate whether reviews appeared before our suggested stopping time could cover all the emerging bugs. If the sentences (outputs of IDEA) about an emerging bug appear before our suggested stopping time, we think reviews appeared before our suggested stopping time could cover this emerging bug. If we find other sentences (not outputs of IDEA) related to an emerging bug appear before our suggested stopping time, we consider reviews that appeared before our suggested stopping time could cover this emerging bug.

The baseline approach iterates over all reviews of a certain version. When processing a new review, the baseline approach calculates the similarity scores between it and all saved reviews. If all similarity scores are less than a threshold, the review will be saved. Otherwise, the baseline approach removes the review. Note that the first review should be saved. After the iteration, the submitted time of the last saved reviews is the stopping time of the baseline approach. One other possible way is the cluster-based algorithm. However, since the inputs of clustering algorithms are all reviews of a certain version, it becomes impractical to obtain all reviews when beginning work on a new version. Achananuparp *et al.* [45] defined the scoring threshold for similar pairs as 0.5 to evaluate 14 existing text similarity measures. Drawing inspiration from them, we also set the threshold as 0.5 and utilize word2vec [46] to represent reviews.

**Bug-related reviews.** As it is challenging to manually identify all the reviews into bugs or not bugs, we first use labelled data [15] to find all bug-related reviews by using SVM [39]. Second, as some bug-related reviews refer to the same bug, we classify the bug-related reviews into clusters with K-means [47]. To get more satisfactory clustering results, we obtain embeddings of reviews with SimCSE [48], which could deal with the anisotropy problem. A prior research [49] collected a large dataset with 4,416 versions and 21,380 bug-related commits, which means that each version contains about five bugs on average. In order to avoid missing some bugs, conservatively, we set K as 10 for every version in our experiments. If one review of a cluster appears in reviews appeared before our suggested stopping time, it means that reviews appeared before our suggested stopping time could cover this cluster. If one review of a cluster appearing in the rest reviews is the same or similar to one review in reviews appeared before our suggested stopping time, we consider this cluster could be covered by reviews appearing before our suggested stopping time.

**Result: With suggesting when to stop reading reviews, STRE only misses one emerging bug.** By using IDEA, we collected a total of 13 emerging bugs in our studied apps. As the outputs of IDEA are sentences, we obtain 13 groups of sentences, which are corresponding to 13 emerging bugs. We find sentences from seven emerging bugs that appeared before our suggested stopping time. Therefore, following our suggested stopping time would not miss these seven emerging bugs. For the remaining six emerging bugs, we find reviews about five of the six emerging bugs exist before the suggested stopping time, although these reviews are not picked up by IDEA. Yet, these five emerging bugs still will not be missed if following our suggested stopping time and only one emerging bug will be missed. In total, only one emerging bug is not covered before the suggested stopping time. When using the baseline approach, sentences of only eight emerging bugs are found before the stopping time; while the other three bugs are not covered in the reviews before stopping time and we could not find extra sentences about them before the stopping time.

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

**STRE only misses 5.14% of the bug-related reviews.** After clustering, we obtain 584 clusters of bug-related reviews from five apps, where reviews of the 141 clusters of bug-related reviews appear all after the suggested stopping time. In other words, the stopping time suggested by STRE would cause missing these bug-related reviews. We find 21 of the 141 clusters of bug-related reviews are either misclassified or obscure. For the remaining 120 clusters of bug-related reviews, we find same or similar reviews about the 90 clusters of bug-related reviews before the stopping time. Hence, we would only miss 30 clusters of bug-related reviews in total and the missing rate is about 5.14%. As we only obtain 13 emerging bugs, this missing rate could mitigate the occasionality of RQ conclusion.

STRE could cover 92.31% of emerging bugs. STRE only omits one emerging bug while baseline approach leaves out three ones. STRE could cover 94.86% of the clusters of bug-related reviews.

**RQ3: How much time could developers save by using our algorithm?**

**Motivation:** Since RQ1 and RQ2 have demonstrated that STRE could retain almost all the important information, in this research question, we would like to evaluate how much time developers could save with STRE. Our approach could complement existing classification-based and cluster-based review categorization approaches and work in tandem to help developers. Then we explore how much time developers may save when using these methods with the help of STRE. We also gauge whether the tool stops too early from the internal perspective.

**Approach:** First, we calculate time that can be saved by applying our approach.

Second, we select one classification-based approach [14] and one cluster-based method [15] to show effort savings. For the classification-based approach, it first splits reviews into sentences as one review may contain more than one intention. Then it classifies sentences into Information Giving, Information Seeking, Feature Request, Problem Discovery, and Others. Since developers only concern the first four meaningful categories, we take the percentage of numbers of meaningful sentences in the cut reviews provided by STRE and the raw reviews as the tool-rate. We set one minus tool-rate as the effort savings. For the cluster-based approach, *i.e.*, CLAP, it first classifies reviews into seven categories: functional bug report, suggestion for new feature, report of performance problems, report of security issues, report of excessive energy consumption, request for usability improvements, and other. Then CLAP extracts clusters from the functional bug report and suggestion for new feature. For the next four categories, CLAP takes them as cohesive clusters and does not conduct cluster algorithms on them. However, in our studies, we do not find any cluster by using CLAP. Thus, we record the number of reviews from the next four categories (report of performance problems, report of security issues, report of excessive energy consumption, and request for usability improvements). We take the percentage of numbers of reviews belonging to the four categories in the cut reviews provided by STRE and the raw reviews as the tool-rate. We set one minus tool-rate as the effort savings.

Third, in order to evaluate whether our approach suggests to stop reading reviews too early, leading to missing information, we measure the covered information from the reviews that appear before the suggested stopping time. We define a metric, *i.e.*, **I**nformation **C**overage **R**ate ($ICR$) as follows:

$$ICR = \frac{\sum SAT}{\sum SET} \tag{1}$$

where $SAT$ means the contributed **S**core of an **A**ppeared **T**op word in a new version before the suggested stopping time. $SET$ is the contributed **S**core of an **E**xisting **T**op word. Specifically, we sum up scores of all top words in this version. The larger value of $ICR$ is, the more top words appear. In other words, a larger value of $ICR$ indicates that more review information is covered by our approach. $ICR$ can reveal whether reviews that appear before the suggested stopping time could contain enough information. Note that, to avoid bias of evaluation, this evaluation metric is based on the topic model trained directly from all of the reviews in the new version, instead of the historical versions. In other words, the metric is used to measure before the suggested stopping time, how much information from all the reviews of the new version is covered.

**Result: Our approach can reduce much time for reading reviews.** Figure 4 shows the results of the suggested stop reading time and the information coverage rate for each given new version in the testing set of all apps. In Figure 4, each sub-figure is the result of a new version. The X-axis is elapsed time of the version and the Y-axis is the information coverage rate. We provide data of information coverage rates in our replication package. Table 6 shows our suggested stopping time and the total duration of each version. The results show that our approach can save an average time of 62.79%, 77.44%, 86.76%, 58.86%, and 56.49%, for Zoom, YouTube, Amazon Shopping, Twitter, and Starbucks, respectively. By looking closer at the data, we find that the time saving rates of long versions are much higher than those of short versions. For the versions which last for more than 15 days, our approach can save an average time of 88.57%, 98.33%, 95.55%, 91.06%, and 88.9%, for Zoom, YouTube, Amazon Shopping, Twitter, and Starbucks, respectively. For example, in the third version of Amazon Shopping, the version lasted for 28 days; while our approach suggests stopping reading reviews slightly after the first day. For the versions which last for less or equal to 15 days, our approach can save an average time of 51.74%, 76.04%, 80.16%, 55.47%, and 45.68%, for Zoom, YouTube, Amazon Shopping, Twitter, and Starbucks, respectively. For example, the fourth version of Twitter only lasted for one day and even our approach suggests stopping reading reviews after one day, the app updated to another version shortly. This feature is satisfactory because it is more necessary for developers to stop early when reading reviews of long versions. For the baseline approach (see details of the approach in RQ2), it could save an average time of 29.78%, 18.66%, 23.53%, 23.61%, and 49.73%, for Zoom, YouTube, Amazon Shopping, Twitter, and Starbucks, respectively. Obviously, the baseline approach could not save as much time as STRE.

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

TABLE 4: The workload comparison of two other tools between using STRE and not utilizing STRE. (Note: we count the number of sentences for the classification-based method while counting the number of reviews for the cluster-based method. We abbreviate "Sentences / Reviews" to "Sen. / Rev.".)

| Approach | App | # Sen. / Rev. (with STRE) | # Sen. / Rev. (without STRE) | Effort savings (%) |
|---|---|---|---|---|
| Classification -based | Zoom | 272 | 1,143 | 76.20 |
| | YouTube | 666 | 5,446 | 87.77 |
| | Amazon Shopping | 253 | 1,637 | 84.54 |
| | Twitter | 560 | 2,888 | 80.61 |
| | Starbucks | 152 | 473 | 67.86 |
| Cluster-based | Zoom | 23 | 98 | 76.53 |
| | YouTube | 204 | 1,527 | 86.64 |
| | Amazon Shopping | 14 | 91 | 84.62 |
| | Twitter | 121 | 642 | 81.15 |
| | Starbucks | 1 | 6 | 83.33 |

**If we classify reviews into several categories first, the suggested stopping time may be too late, which leads to less time saving from the technique.** We utilize the same classifier in RQ2 and focus on bug-related reviews in the experiments. However, it just saves an average time of 54.14%, 62.58%, 78.59%, 33.11%, and 34.25%, for Zoom, YouTube, Amazon Shopping, Twitter, and Starbucks, respectively. As we first extracts categories from reviews, the information in each category would be more sparse. The sparse information would cause TSTs longer, so the suggested stopping time is delayed. In addition, the mistakes from the classification approach may also add noise into the process. The mis-classified reviews may delay the suggested stopping time since the corresponding topic may not appear again in the same category.

**Our approach does not miss much topic-related review information.** The results from Table 5 show that the average of $ICR$ across all the new versions in Zoom is 98.64%. The majority of the $ICR$ across versions of Zoom is over 0.9 and only one version has an $ICR$ less than 0.9 (0.899). Similarly, the average of $ICR$ in other four apps is 97.39%. The results mean that most of the top words in each topic appear before the cut-off line. Such results imply that at the stop reading time, our approach can cover most of the topic-related information. For the baseline approach, the average of $ICR$ across all the new versions are 99.35%, 99.81%, 100%, 95.12%, and 74.38%, for Zoom, YouTube, Amazon Shopping, Twitter, and Starbucks, respectively. For the first three apps, the baseline performs a little better than STRE. However, the average of ICR of Starbucks is extremely low, which reveals the instability. Furthermore, STRE could save much more time. Our approach can save an average time of 62.79%, 77.44%, 86.76%, 58.86%, and 56.49%, for Zoom, YouTube, Amazon Shopping, Twitter, and Starbucks, respectively. While the baseline approach could save an average time of 29.78%, 18.66%, 23.53%, 23.61%, and 49.73%, for Zoom, YouTube, Amazon Shopping, Twitter, and Starbucks, respectively. In short, STRE covers similar or higher information than the baseline, while suggesting a much earlier time. From our perspective, the reason is that STRE makes full use of top words which could be satisfactory metrics for information. By contrast, the baseline approach is only based on the inflexible similarity threshold.

TABLE 5: The Information Coverage Rate (ICR) by the suggested stopping time for each studied app.

| App | Min | Max | Average | Median |
|---|---|---|---|---|
| Zoom | 89.90% | 100% | 98.64% | 100% |
| YouTube | 93.96% | 100% | 99.06% | 100% |
| Amazon Shopping | 95.58% | 100% | 98.90% | 100% |
| Twitter | 89.69% | 100% | 98.67% | 100% |
| Starbucks | 77.53% | 100% | 92.92% | 93.69% |

STRE can save an average time of 62.79%, 77.44%, 86.76%, 58.86%, and 56.49%, for Zoom, YouTube, Amazon Shopping, Twitter, and Starbucks, respectively, without missing topic-related review information. Overall, STRE saves much more time while covering similar or higher information than the baseline.

TABLE 6: Results of the suggested stopping time/total time (by days) of all studied releases of the apps. Values in the brackets show the percentage of saved time.

Zoom

| Ver. 1 | Ver. 2 | Ver. 3 | Ver. 4 | Ver. 5 | Ver. 6 | Ver. 7 | Ver. 8 | Ver. 9 | Ver. 10 |
|---|---|---|---|---|---|---|---|---|---|
| 2.80/4 (-29.92%) | 1.22/12 (-89.87%) | 2.01/7 (-71.24%) | 2.89/24 (-87.98%) | 1.14/12 (-90.51%) | 2.24/20 (-88.82%) | 4.73/7 (-32.46%) | 1.99/18 (-88.92%) | 2.1/4 (-47.62%) | 2.98/3 (-0.60%) |

YouTube

| Ver. 1 | Ver. 2 | Ver. 3 | Ver. 4 | Ver. 5 | Ver. 6 | Ver. 7 | Ver. 8 |
|---|---|---|---|---|---|---|---|
| 0.97/5 (-80.51%) | 0.94/7 (-86.60%) | 0.8/9 (-91.12%) | 2.31/5 (-53.79%) | 0.58/7 (-91.75%) | 0.54/7 (-92.28%) | 2.01/7 (-71.27%) | 0.83/8 (-89.65% |
| Ver. 9 | Ver. 10 | Ver. 11 | Ver. 12 | Ver. 13 | Ver. 14 | Ver. 15 | Ver. 16 |
| 0.67/7 (-90.45%) | 0.73/3 (-75.83%) | 0.93/5 (-81.41%) | 3.1/13 (-76.15%) | 1.14/68 (-98.33%) | 0.56/4 (-85.91%) | 1.06/4 (-73.61%) | 1.99/2 (-0.32%) |

Amazon Shopping

| Ver. 1 | Ver. 2 | Ver. 3 | Ver. 4 | Ver. 5 | Ver. 6 | Ver. 7 |
|---|---|---|---|---|---|---|
| 1.12/26 (-95.70%) | 3.10/14 (-77.86%) | 1.21/28 (-95.68%) | 1.65/35 (-95.28%) | 1.24/14 (-91.14%) | 5.55/15 (-62.99%) | 1.47/13 (-88.66%) |

Twitter

| Ver. 1 | Ver. 2 | Ver. 3 | Ver. 4 | Ver. 5 | Ver. 6 | Ver. 7 |
|---|---|---|---|---|---|---|
| 2.11/5 (-57.81%) | 1.97/2 (-1.66%) | 1.23/7 (-82.43%) | 1/1 (-0.48%) | 0.92/5 (-81.62%) | 0.99/1 (-1.09%) | 1.72/6 (-71.31%) |
| Ver. 8 | Ver. 9 | Ver. 10 | Ver. 11 | Ver. 12 | Ver. 13 | Ver. 14 |
| 1.87/13 (-85.65%) | 1.34/2 (-32.95%) | 4.21/5 (-15.74%) | 2.48/3 (-17.38%) | 1.53/11 (-86.10%) | 1.58/11 (-85.65%) | 2.45/5 (-50.98%) |
| Ver. 15 | Ver. 16 | Ver. 17 | Ver. 18 | Ver. 19 | Ver. 20 | Ver. 21 |
| 1.86d/19d (-90.24%) | 1.62d/20d (-91.88%) | 0.90d/2d (-54.95%) | 2.00d/9d (-77.79%) | 1.02d/12d (-91.52%) | 0.81d/3d (-73.15%) | 0.72d/5d (-85.66%) |

Starbucks

| Ver. 1 | Ver. 2 | Ver. 3 | Ver. 4 | Ver. 5 | Ver. 6 | Ver. 7 | Ver. 8 |
|---|---|---|---|---|---|---|---|
| 5.22d/10 (-47.77%) | 2.33/35 (-93.34%) | 2.59/3 (-13.77%) | 2.31/8 (-71.13%) | 1.91/7 (-72.67%) | 9.92/10 (-0.79%) | 2.24/7 (-67.99%) | 2.95/19 (-84.46%) |

## 6 Interviews with Developers

In Section 5, we answer three RQs to evaluate the effectiveness of our tool STRE. However, these studies may

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

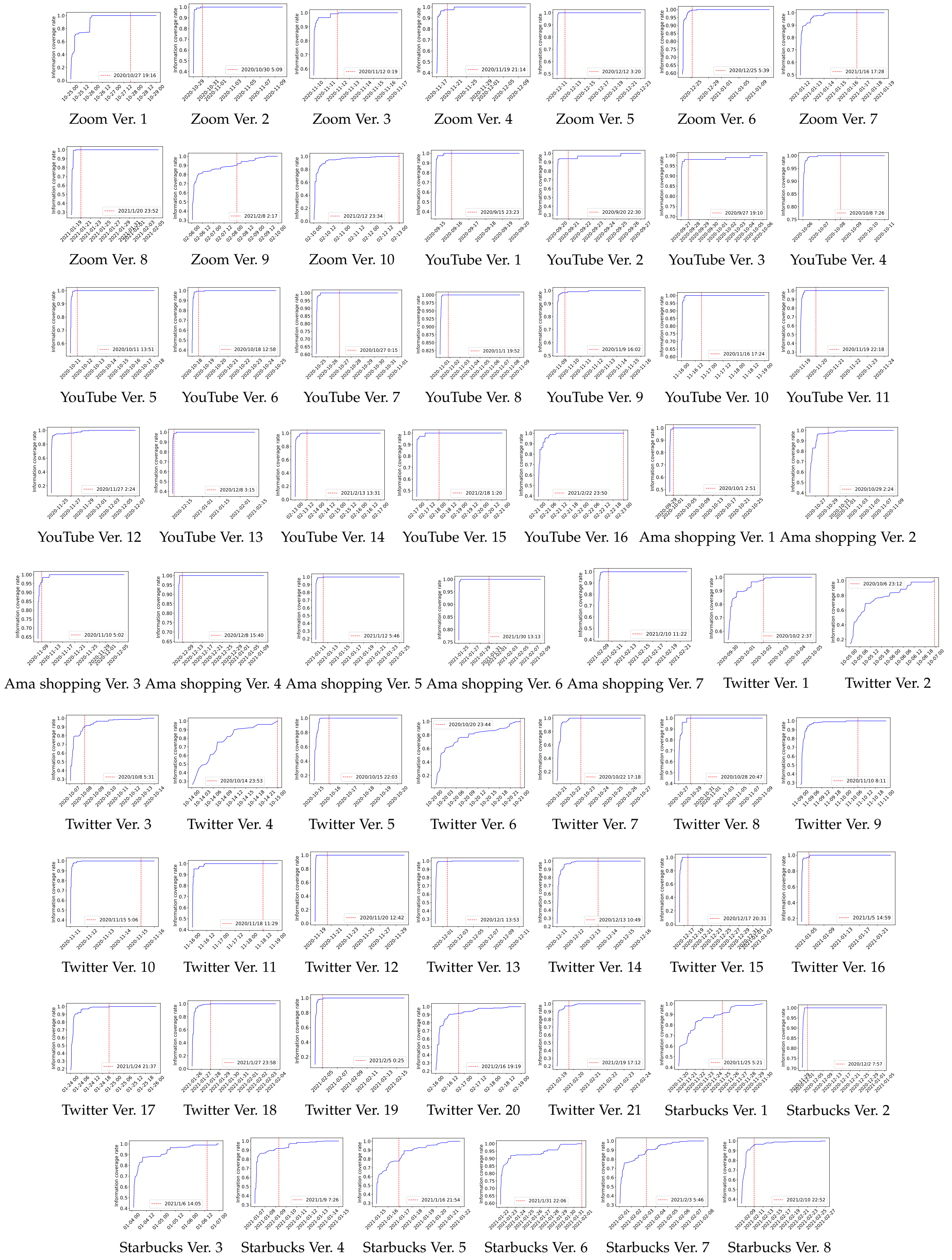


Fig. 4: Illustration of evaluating whether our method is effective or not. The blue broken line is the $ICR$ across the version time. The red dotted line is the time when we stop.

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

not demonstrate how STRE can be used in the practice of mobile app developers. Thus we conduct interviews with five experienced developers. Three of them are professional app developers and two of them are graduate students who had professional app development before graduate studies. All of them worked in different app development teams from different companies with different experiences. Table 7 shows the information of the five interviewees.

To start our interviews, we explain our tool STRE to all participants in detail and show results to them. Afterwards, five questions was asked and answered. The first three questions are about general mobile app development, and aim to understand whether it agrees with requirement of STRE. The questions include: **question 1)** How are app reviews used in your development? **question 2)** Do mobile app developers use time as a criterion for stopping reading reviews? **question 3)** Are there any bugs that are not discovered by users long after the release time?

The last two questions aim to understand practitioners' perspectives particularly on the applicability of using STRE in practice. The last two questions are: **question 4)** To what extent may STRE impact the practice of mobile app developers? **question 5)** Can STRE be integrated into the workflow of the mobile app developers? The interviews last around 12, five, five, six, and four minutes, respectively for each participants.

To analyze the contents of interviews systematically, we conduct coding, which is the technique of qualitative methods. Followed by Yang *et al.* [50], we first perform open coding. We code every part of data with a label. Then two authors conduct axial coding to condense these codes into categories for every question.

**The process and the information of reviews used in the app development process agree with the requirement of STRE.** For question 1, we obtain two ways of using reviews by developers. The Cohen's and Fleiss' kappa agreements of the two authors' results are 0.7142 and 0.709 (substantial agreement [40, 41]), respectively. **Communicate**: Developers utilize reviews to communicate with users:

> *Reviews could help us communicate with our users and collect usage opinions.*

**Collect comments**: They use reviews to collect comments:

> *First, they could reflect the popularity of our app. Second, we could know users' problems in time, and then make corresponding improvements in product design and code development.*

The answers show that developers do take reviews as a significant source of information in their daily development.

For question 2, we obtain the affirmative answer and one factor to affect the time. The Cohen's and Fleiss' kappa agreements of the two authors' results are 0.7142 and 0.709 (substantial agreement [40, 41]), respectively. **Read reviews after a fixed time**: Developers use time as a criterion for stopping reading reviews:

> *We typically read reviews within the first two days after the releases of apps.*

**App popularity**: The popularity of apps affect developers' schedules to read reviews:

> *The stopping time depends on the popularity of the app.*

The confirmation on the use of time as a criterion signifies that the lack of consideration on the factor of time in prior research can be improved and complemented by our approach.

For question 3, we obtain three reasons why some bugs appear long after the release. The Cohen's and Fleiss' kappa agreements of the two authors' results are 0.8625 and 0.862 (almost perfect agreement [40, 41]), respectively. **Rare use**: Some bugs may be posted long after the release time, which are used rarely:

> *Too few users could also conceal the bugs for a long time.*

**Complex**: Complex combinations of features may generate bugs, which appear after the release for a long time:

> *The bugs may be hidden if the testing cases are not complex.*

**User-imperceptible**: Several bugs may be discovered after a long time. However, these bugs can hardly be found by users:

> *Only professional bugs, such as the log4j bug, may happen after a long time. The functional bugs, which may be detected by users, would be posted quickly.*

Some bugs may appear after the release for a long time. However, from the interviewees' responses, the number of these bugs is very small. Thus these bugs may not affect the usability of STRE.

STRE **may be able to help developers improve their development efficiency.** For question 4, the two authors reach a consensus. **Improve development efficiency**: Developers agree that they could arrange their developing schedules or develop the next version early based on the results of STRE:

> *First, it can help us know users' opinions, especially about bugs. Second, the tool can notify us when useful information fully appears. Thus it can help us schedule the next stage of development tasks early.*

Thus STRE could help developers get a more reasonable stop time, rather than rely on their experience.

For question 5, the two authors reach an agreement. **Help read reviews**: All the participants think that STRE could be integrated into the workflow, especially when the app has large amounts of reviews:

> *This tool can be useful when there is a lot of review data.*

However, we do realize that the integration of STRE in real development practices can be further evaluated by user studies from real development, which will be considered in our future research.

STRE could be integrated into app development and help developers improve the development efficiency.

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

TABLE 7: The information of the five interviewees.

| Product type | Role | Whether work with reviews | Years of working experience |
|---|---|---|---|
| professional tool | software engineer | yes | 11 |
| professional tool | software engineer | yes | 12 |
| smart service platform for students | leader | yes | 17 |
| customs declaration and medical devices | algorithm principal | yes | 6 |
| game and video | software engineer | yes | 9 |

## 7 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of this paper.

**External validity.** We only select five subject apps to validate our framework and our findings may not generalize to other apps. However, STRE aims to suggest the stopping time to read reviews based on the historical reviews, it could be applied to other apps easily. In addition, to evaluate the robustness of our approach, we choose five apps in different domains and collect reviews across multiple versions.

We split reviews according to the release time of each app compared to the date of reviews, which is not a precise mapping for some reviews. The app store does not report the version of a review, and we have to adopt this approximate method. We discuss the influence in two scenarios, a review is about an older version and a future version instead of the current version. In the training phase, as our tool utilizes historical reviews to suggest stopping time, and reviews of new versions could not appear in advance. Thus we just discuss the scenarios in the testing phase. For the first scenario, if the review's topics are also covered by other reviews in the old version, results from STRE are still valid. If the review's topics are unique and never covered by other reviews in the old version, they would not affect the results of STRE. Developers never read them, as they just read the reviews before the release time of the next version. If the review's topics are not related to the current version. This means that we should not suggest developers to even read this review. But our current results on the time savings are not compromised. If the review's topics are covered by the reviews of the current version. Then our results from STRE are still valid. For the second scenario, it would never happen. When developers release a version, they can use our approach STRE to check whether it is necessary to stop reading reviews. At this point of time, the future version does not exist, so there are not any reviews for the future version. Thus, this direct segmentation would not affect our tool.

**Internal validity.** First, the number of topics could influence the results of the LDA algorithm. To reduce this influence, we set an appropriate number according to the value of coherence. Second, the choice of longest topic stability time plays a vital role in the effectiveness of our approach. To get a good trade-off between conservatism and sensibility, we remove values which are deviated by one time the standard deviation in historical topic stability times, then choose the longest one. Third, the number of top words per topic may affect the performance of STRE. On the one hand, if we use too many top words, numerous low-informative words may reduce efficiency of STRE. On the other hand, too few top words could not represent the corresponding topic drastically. To make a good trade-off, we utilize ten top words, as ten words contain most of the useful information [36, 37].

**Construct validity.** First, we directly take the results of LDA (*i.e.*, top words) as the indicator of information in reviews, these words may not be able to represent the whole data. Since many prior studies [12, 33, 34] have utilized the LDA-based method to extract topics in reviews, we assume that these top words could cover the information of reviews. In addition, our study results indicate that the results of STRE (*i.e.*, reviews that appear before the suggested stopping time) contain enough useful information for developers. We take the ICR metric as a proxy to evaluate the condition of information coverage. But RQ1 and RQ2 also indicate that STRE could retain adequate advantageous information before the suggested stopping reading time.

Second, in Section 5, we manually classify 170 reviews and evaluate the reviews. Moreover, we extract emerging bugs by hand. However, we follow the discussional mode to reduce subjectivity. The eight bug-related missed reviews may contain supplementary information for bug reproduction, which is a significant part in bug fixes. However, our manual analysis of RQ1 shows that the eight reviews do not contain supplementary data for crash reproduction. We manually find the similar bugs or feature requests before the suggested stopping time, which may contain some bias. However, reviews are short texts and the two similar reviews may be posted by different rhetorics, so it may be hard to use textual similarity to find the similar bugs or feature requests. For example, "I can't see the screen" and "The background disappears" may comment on the same bug. Furthermore, two authors discuss the results to mitigate the negative effects.

Third, we do not find any cluster when conducting the complementary experiments using CLAP in RQ3. However, this result may not have a significant impact on our study, since we just utilize CLAP to demonstrate that STRE can complement other approaches. However, the outcome of our evaluation may still be affected by the quality of the external tool such as CLAP. We use a classification algorithm to find bug-related reviews in RQ2. The algorithm may cause some mistakes. However, we manually read reviews of 141 bug clusters which all appear after the suggested stopping time. In the future, we will conduct STRE in the real production environment to complement our work.

## 8 RELATED WORK

Cruz *et al.* [51] found automated testing is significant to apps, which could help decrease issues. App reviews can also be incorporated to contribute to the success of apps [6]. Al-Subaihin *et al.* [10] investigated how app stores influence software engineering tasks, which elaborates on

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

the value of reviews for developers. Because of reviews' unstructured nature and varying quality, researchers proposed many methods to help developers analyze reviews more effectively. We consider that these methods can be divided into two sets (1) classification-based and (2) extraction-based ones. Our work is different from these studies, because we aim to help developers decide when to stop reading reviews.

## 8.1 Classification-based methods

To help developers filter out the useless information, several researchers categorize reviews. The intuition is that developers could directly extract the specific information.

Maalej and Nabil [52] proposed a method to classify reviews into bug reports, feature requests, user experiences, and ratings by introducing probabilistic techniques and heuristics. Pagano and Maalej [5] analyzed more than one million reviews from Apple Store. They manually investigated these reviews and identified 17 topics. Based on Pagano and Maalej's research, Panichella *et al.* [14] presented a taxonomy to classify app reviews into categories relevant to software maintenance and evolution using Natural Language Processing, Sentiment Analysis, and Text Analysis. The categories in their taxonomy include *Information Giving*, *Information Seeking*, *Feature Request*, and *Problem Discovery*. They think that this method could be useful in extracting not only sentences which mention specific topics, but in understanding the intentions of users concerning the mentioned topics. Guzman *et al.* [53] found that user feedback from Twitter contains information about requirement engineering and software evolution. They utilized Decision Trees and Support Vector Machines (SVMs) to automatically identify whether tweets are relevant for software companies. Chen *et al.* [16] presented AR-Miner, which used a topic modeling algorithm to help developers prioritize the most informative app reviews.

Beyond simple classification or prioritization, two researches proposed summarising approaches to reduce the number of reviews. Noted that their basis is still classification algorithms. Di Sorbo *et al.* [8] proposed a tool named SURF, which could summarize thousands of reviews and generate an interactive, structured, and condensed agenda of recommended software changes using sophisticated summarization techniques. SURF contains a two-level classification (1) review topic (*e.g.*, UI improvements, security/licensing issues, *etc.*) (2) maintenance task (*e.g.*, bug fixing, feature enhancement, *etc.*). Jha *et al.* [54] presented a lightweight tool named MARC 2.0. This tool first used semantic role labeling to classify reviews into feature requests, bug reports, and otherwise. Then it provided four classic extractive text summarization algorithms for users.

## 8.2 Extraction-based methods

Since developers still need to face large amounts of reviews after applying classification-based methods, several researchers proposed extraction-based methods. These methods could extract development-related information more directly.

To help developers compete with other apps, Assi *et al.* [55] presented FeatCompare to identify high-level features from reviews. Villarroel *et al.* [7] proposed CLAP, which contains three steps. First, CLAP classifies the reviews. Then it divides the reviews into different clusters. At last, it prioritizes the clusters of reviews automatically. Scalabrino *et al.* [15] presented an extension of CLAP. They provided a more fine-grained categorization, expanded their experiments, and gave more details of the tool.

Zheng *et al.* [1] proposed iFeedback, which could perform real-time issue detection based on user feedback texts. iFeedback contained two steps (1) building service runtime indicators and (2) detecting machine learning-based issues. They first applied a specific-tailored natural language processing approach to generate tremendous indicators and then filtered out useless ones with rule-based methods. Finally, they utilized a 2-class classification model to perform anomaly detection and then clustered them.

Guo *et al.* [56] presented an approach named Caspar, which could extract and synthesize user-reported mini stories about app problems from reviews. They assume that users' interactions have two types of events: user actions and associated app behaviors, which each pair can be combined into a mini story. They utilized sophisticated natural language processing techniques to extract ordered events from app reviews. They also train an inference model to automatically predicts possible app problems for different use cases.

Gao *et al.* [12] proposed an LDA-based tool named IDEA, which aimed to identify emerging app issues effectively based on app reviews. They defined the emerging bugs as follows: an issue in a time slice is called an emerging issue if it rarely appears in the previous slice but is mentioned by a significant proportion of reviews in the current slice. However, due to its inherent randomness, IDEA is not stable. Gao *et al.* [11] presented DIVER, an efficient and reliable emerging bugs detecting tool, which had been successfully detected 18 emerging issues of WeChat's Android and iOS apps in one month.

Above approaches could help developers analyze app reviews more effectively. Such as classification-based approaches [14, 52], aim to allow developers to focus on meaningful reviews. Specifically, they split reviews into some categories, such as bugs, feature requests, praises, *etc*. They assume that the classification could help developers get the reviews they want quickly. Furthermore, some cluster-based methods [7, 15] prioritize the clusters of reviews to be implemented. However, they do not consider that developers should stop reading reviews to work for the next version at a certain point. Thus we think that our approach can complement existing approaches and work in tandem to help developers. Specifically, our tool can suggest that developers stop reading reviews, then developers could utilize the other methods (*e.g.* classification-based and cluster-based approaches) to extract valuable information which could help developers maintain and improve their apps.

## 9 CONCLUSION

Reviews are crucial data for developers, as they contain large amounts of useful information for app development. As much repeated information is contained in reviews, developers could just read the reviews which are uploaded around the release time to save time. However, they could

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

not predict when no more new useful information will appear.

To help developers decide when to stop reading reviews, we propose STRE. Study results demonstrate that STRE could help developers save much time and reserve enough useful information for developers. In the future, we plan to propose more effective approaches to help developers analyze reviews so that they can improve the efficiency of software development and maintenance. For popular apps, it may be possible to read reviews periodically even with several automated tools, so developers would split all reviews into huge batches. We will design a tool to suggest developers when no more useful information will appear in a batch.

## ACKNOWLEDGMENTS

We would like to thank Mr. Husheng Yuan, who is the leader of WeMUST development team at Macau University of Science and Technology, for providing some useful suggestions to our tool. This work was supported in part by the Science and Technology Development Fund of Macau, Macau SAR under Grant 0047/2020/A1 and 0014/2022/A, and in part by the China Postdoctoral Science Foundation under Grant 2017M621247.

## REFERENCES

[1] W. Zheng, H. Lu, Y. Zhou, J. Liang, H. Zheng, and Y. Deng, "ifeedback: exploiting user feedback for real-time issue detection in large-scale online service systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 352–363.

[2] S. Krusche and B. Bruegge, "User feedback in mobile development," in *Proceedings of the 2nd International Workshop on Mobile Development Lifecycle*, 2014, pp. 25–26.

[3] T. Vithani and A. Kumar, "Modeling the mobile application development lifecycle," in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, vol. 1, 2014, pp. 596–600.

[4] 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*, vol. 137, pp. 143–162, 2018.

[5] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study," in *2013 21st IEEE international requirements engineering conference*, 2013, pp. 125–134.

[6] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "User reviews matter! tracking crowdsourced reviews to support evolution of successful apps," in *2015 IEEE international conference on software maintenance and evolution*, 2015, pp. 291–300.

[7] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Release planning of mobile apps based on user reviews," in *2016 IEEE/ACM 38th International Conference on Software Engineering*, 2016, pp. 14–24.

[8] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall, "What would users change in my app? summarizing app reviews for recommending software changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 499–510.

[9] L. Yu, J. Chen, H. Zhou, X. Luo, and K. Liu, "Localizing function errors in mobile apps with user reviews," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks*, 2018, pp. 418–429.

[10] A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, and M. Harman, "App store effects on software engineering practices," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 300–319, 2019.

[11] C. Gao, W. Zheng, Y. Deng, D. Lo, J. Zeng, M. R. Lyu, and I. King, "Emerging app issue identification from user feedback: Experience on wechat," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*, 2019, pp. 279–288.

[12] C. Gao, J. Zeng, M. R. Lyu, and I. King, "Online app review analysis for identifying emerging issues," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 48–58.

[13] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[14] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "How can i improve my app? classifying user reviews for software maintenance and evolution," in *2015 IEEE international conference on software maintenance and evolution*, 2015, pp. 281–290.

[15] S. Scalabrino, G. Bavota, B. Russo, M. Di Penta, and R. Oliveto, "Listening to the crowd for the release planning of mobile apps," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 68–86, 2017.

[16] N. Chen, J. Lin, S. C. Hoi, X. Xiao, and B. Zhang, "Ar-miner: mining informative reviews for developers from mobile app marketplace," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 767–778.

[17] Y. Man, C. Gao, M. R. Lyu, and J. Jiang, "Experience report: Understanding cross-platform app issues from user reviews," in *2016 IEEE 27th International Symposium on Software Reliability Engineering*, 2016, pp. 138–149.

[18] E. L. Bird, Steven and E. Klein, *Natural Language Processing with Python*, 2009.

[19] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, 2010, pp. 95–104.

[20] D. Binkley, D. Heinz, D. Lawrie, and J. Overfelt, "Understanding lda in source code analysis," in *Proceedings of the 22nd international conference on program comprehension*, 2014, pp. 26–36.

[21] C. Bird, T. Menzies, and T. Zimmermann, *The art and science of analyzing software data*, 2015.

[22] J. C. Campbell, A. Hindle, and E. Stroulia, "Latent dirichlet allocation: extracting topics from software engineering data," in *The art and science of analyzing software data*, 2015, pp. 139–159.

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

[23] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 552–555.

[24] V. Garousi and M. V. Mäntylä, "Citations, research topics and active countries in software engineering: A bibliometrics study," *Computer Science Review*, vol. 19, pp. 56–77, 2016.

[25] H. Hemmati, Z. Fang, M. V. Mäntylä, and B. Adams, "Prioritizing manual test cases in rapid release environments," *Software Testing, Verification and Reliability*, vol. 27, no. 6, p. e1609, 2017.

[26] A. Hindle, C. Bird, T. Zimmermann, and N. Nagappan, "Relating requirements to implementation via topic analysis: Do topics extracted from requirements make sense to managers and developers?" in *2012 28th IEEE International Conference on Software Maintenance*, 2012, pp. 243–252.

[27] L. Layman, A. P. Nikora, J. Meek, and T. Menzies, "Topic modeling of nasa space system problem reports: Research in practice," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 303–314.

[28] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshynanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *2013 35th International Conference on Software Engineering*, 2013, pp. 522–531.

[29] X. Sun, X. Liu, B. Li, Y. Duan, H. Yang, and J. Hu, "Exploring topic models in software engineering data analysis: A survey," in *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2016, pp. 357–362.

[30] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Validating the use of topic models for software evolution," in *2010 10th IEEE working conference on source code analysis and manipulation*, 2010, pp. 55–64.

[31] D. M. Blei, "Probabilistic topic models," *Communications of the ACM*, vol. 55, no. 4, pp. 77–84, 2012.

[32] J. Uys, N. Du Preez, and E. Uys, "Leveraging unstructured information using topic modelling," in *PICMET'08-2008 Portland International Conference on Management of Engineering & Technology*, 2008, pp. 955–961.

[33] E. Noei, F. Zhang, and Y. Zou, "Too many user-reviews, what should app developers look at first?" *IEEE Transactions on Software Engineering*, 2019.

[34] T. Zhang, J. Chen, X. Zhan, X. Luo, D. Lo, and H. Jiang, "Where2change: Change request localization for app reviews," *IEEE Transactions on Software Engineering*, 2019.

[35] M. Röder, A. Both, and A. Hinneburg, "Exploring the space of topic coherence measures," in *Proceedings of the eighth ACM international conference on Web search and data mining*, 2015, pp. 399–408.

[36] M. V. Mantyla, M. Claes, and U. Farooq, "Measuring lda topic stability from clusters of replicated runs," in *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, 2018, pp. 1–4.

[37] D. Newman, J. H. Lau, K. Grieser, and T. Baldwin, "Automatic evaluation of topic coherence," in *Human language technologies: The 2010 annual conference of the North American chapter of the association for computational linguistics*, 2010, pp. 100–108.

[38] N. Bettenburg, R. Premraj, T. Zimmermann, and . Sunghun Kim, "Duplicate bug reports considered harmful ... really?" in *2008 IEEE International Conference on Software Maintenance*, 2008, pp. 337–345.

[39] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[40] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[41] K. L. Gwet, *Handbook of inter-rater reliability: The definitive guide to measuring the extent of agreement among raters*, 2014.

[42] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE software*, vol. 32, no. 3, pp. 70–77, 2014.

[43] S. Li, J. Guo, M. Fan, J.-G. Lou, Q. Zheng, and T. Liu, "Automated bug reproduction from user reviews for android applications," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020, pp. 51–60.

[44] X. Franch and G. Ruhe, "Software release planning," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion*, 2016, pp. 894–895.

[45] P. Achananuparp, X. Hu, and X. Shen, "The evaluation of sentence similarity measures," in *International Conference on data warehousing and knowledge discovery*. Springer, 2008, pp. 305–316.

[46] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[47] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[48] T. Gao, X. Yao, and D. Chen, "Simcse: Simple contrastive learning of sentence embeddings," in *EMNLP*, 2021.

[49] D. E. Krutz, M. Mirakhorli, S. A. Malachowsky, A. Ruiz, J. Peterson, A. Filipski, and J. Smith, "A dataset of open-source android applications," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 522–525.

[50] N. Yang, P. Cuijpers, R. Schiffelers, J. Lukkien, and A. Serebrenik, "An interview study of how developers use execution logs in embedded software engineering," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 61–70.

[51] L. Cruz, R. Abreu, and D. Lo, "To the attention of mobile software developers: guess what, test your app!" *Empirical Software Engineering*, vol. 24, no. 4, pp. 2438–2468, 2019.

[52] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? on automatically classifying app reviews," in *2015 IEEE 23rd international requirements engineering conference*, 2015, pp. 116–125.

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

[53] E. Guzman, R. Alkadhi, and N. Seyff, "A needle in a haystack: What do twitter users say about software?" in *2016 IEEE 24th International Requirements Engineering Conference*, 2016, pp. 96–105.

[54] N. Jha and A. Mahmoud, "Using frame semantics for classifying and summarizing application store reviews," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3734–3767, 2018.

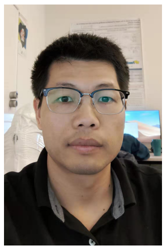
[55] M. Assi, S. Hassan, Y. Tian, and Y. Zou, "Featcompare: Feature comparison for competing mobile apps leveraging user reviews," *Empirical Software Engineering*, vol. 26, no. 5, pp. 1–38, 2021.

[56] H. Guo and M. P. Singh, "Caspar: extracting and synthesizing user stories of problems from app reviews," in *2020 IEEE/ACM 42nd International Conference on Software Engineering*, 2020, pp. 628–640.

**Tao Zhang** Tao Zhang received the BS degree in automation, the MEng degree in software engineering from Northeastern University, China, and the PhD degree in computer science from the University of Seoul, South Korea. After that, he spent one year with the Hong Kong Polytechnic University as a postdoctoral research fellow. Currently, he is an associate professor with the School of Computer Science and Engineering, Macau University of Science and Technology (MUST). Before joining MUST, he was the faculty member of Harbin Engineering University and Nanjing University of Posts and Telecommunications, China. He published more than 70 high-quality papers at renowned software engineering and security journals and concerences such as the IEEE Transactions on Software Engineering, IEEE Transactions on Information Forensics and Security, IEEE Transactions on Dependable and Secure Computing, IEEE Software, ICSE, etc. His current research interests include AI for software engineering and mobile software security. He is a senior member of IEEE and ACM.

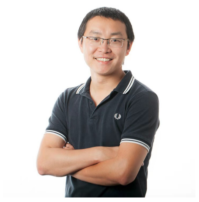
**Youshuai Tan** Youshuai Tan is a postgraduate student at the School of Computer Science and Engineering, Macau University of Science and Technology (MUST), under the supervision of Prof. Tao Zhang. He obtained his B.Eng. from Harbin Engineering University in 2021. His works have been published in JSS and TR. His current research interests include software engineering and natural language processing.

**Sen Fang** Sen Fang is a Ph.D. student in the School of Computer Science and Engineering, Macau University of Science and Technology (MUST), under the supervision of Prof. Tao Zhang. Before joining MUST, he got MSc in Electronics and Communication Engineering from Central China Normal University in 2020. His research interests lie in software engineering and NLP, particularly using NLP technologies to build effective models for representing the source code.



**Jinfu Chen** Jinfu Chen is a senior researcher at Centre for Software Excellence of Huawei Technologies Canada, Kingston. He has received his Ph.D from Concordia University, M.Sc. degree from Chinese Academy of Sciences, and B.Eng. from Harbin Institute of Technology. His research interest lies in empirical software engineering, software performance engineering, performance testing, code clone detection, software security, and software log mining. Contact him at https://jinfuchen.github.io/jinfu

**Xiapu Luo** Xiapu Luo received the PhD degree in computer science from the Hong Kong Polytechnic University and then spent two years with the Georgia Institute of Technology as a postdoctoral research fellow. He is an associate professor with the Department of Computing, the Hong Kong Polytechnic University. His current research interests include mobile/ IoT security and privacy, blockchain, network security and privacy, software engineering, and Internet measurement. He has received seven best paper awards (e.g., INFOCOM'18, ISPEC'17, ISSRE'16, etc.) and one paper received best paper nomination (i.e., ESEM'19).



**Weiyi Shang** Weiyi Shang is a Concordia University Research Chair at the Department of Computer Science. His research interests include AIOps, big bata software engineering, software log analytics and software performance engineering. He serves as a Steering committee member of the SPEC Research Group. He is ranked top worldwide SE research stars in a recent bibliometrics assessment of software engineering scholars. He is a recipient of various premium awards, including the SIGSOFT Distinguished paper award at ICSE 2013 and ICSE 2020, best paper award at WCRE 2011 and the Distinguished reviewer award for the Empirical Software Engineering journal. His research has been adopted by industrial collaborators (e.g., BlackBerry and Ericsson) to improve the quality and performance of their software systems that are used by millions of users worldwide. Contact him at shang@encs.concordia.ca; http://users.encs.concordia.ca/ shang/.

**Zijie Chen** Zijie Chen received the B.S. degree in Software Engineering from Beijing Institute of Technology, Zhuhai, China, in 2020. He is currently working toward the M.S degree in Applied Mathematics and Data Science with the School of Computer Science and Engineering, Macau University of Science and Technology, Macau, China. His research interests include Intelligent Software Engineering, Big Data, and Machine Learning.

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3285743

**Shuhao Qi** Shuhao Qi is a student at the Department of Computer Science at The University of Manchester, Manchester, UK. He obtained his B.Eng. from Harbin Engineering University. His research interests include data mining, social network, and bioinformatics.

Authorized licensed use limited to: University of Waterloo. Downloaded on July 21,2023 at 14:30:38 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.