The Journal of Systems & Software 170 (2020) 110724

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Studying the Relationship Between the Usage of APIs Discussed in the Crowd and Post-Release Defects

Hamed Tahmooresi, Abbas Heydarnoori *, Reza Nadri [1]

Department of Computer Engineering, Sharif University of Technology, Azadi Ave., Tehran, Iran

ARTICLE INFO

Article history:
Received 2 December 2019
Received in revised form 29 June 2020
Accepted 4 July 2020
Available online 13 July 2020

Keywords:
Defect prediction
Crowd knowledge
Stack overflow
API

ABSTRACT

Software development nowadays is heavily based on libraries, frameworks and their proposed Application Programming Interfaces (APIs). However, due to challenges such as the complexity and the lack of documentation, these APIs may introduce various obstacles for developers and common defects in software systems. To resolve these issues, developers usually utilize Question and Answer (Q&A) websites such as Stack Overflow by asking their questions and finding proper solutions for their problems on APIs. Therefore, these websites have become inevitable sources of knowledge for developers, which is also known as the *crowd knowledge*.

However, the relation of this knowledge to the software quality has never been adequately explored before. In this paper, we study whether using APIs which are challenging according to the discussions of the Stack Overflow is related to code quality defined in terms of post-release defects. To this purpose, we define the concept of *challenge of an API*, which denotes how much the API is discussed in high-quality posts on Stack Overflow. Then, using this concept, we propose a set of products and process metrics. We empirically study the statistical correlation between our metrics and post-release defects as well as added explanatory and predictive power to traditional models through a case study on five open source projects including Spring, Elastic Search, Jenkins, K-8 Mail Android Client, and OwnCloud Android client.

Our findings reveal that our metrics have a positive correlation with post-release defects which is comparable to known high-performance traditional process metrics, such as *code churn* and *number of pre-release defects*. Furthermore, our proposed metrics can provide additional explanatory and predictive power for software quality when added to the models based on existing products and process metrics. Our results suggest that software developers should consider allocating more resources on reviewing and improving external API usages to prevent further defects.

1. Introduction

Today software development is heavily based on libraries, frameworks and their offered APIs. However, these APIs may introduce various common defects in software systems. The causes of these defects may be the lack of proper API documentation (Souza et al., 2019), complexity, and poor structure of the API which leads to misunderstandings (Robillard and Deline, 2011; Campos et al., 2016a), backward compatibility issues (Wang et al., 2015), API correctness (e.g., unexpected behavior of the API) (Wang et al., 2015), and the change-proneness of API (Linares-Vásquez et al., 2014). Usually, such defects occur rapidly independent of the application domain (Campos et al., 2016a).

Upon encountering such errors, defects, and even conceptual questions, developers may ask for help by explaining the issue and sometimes attaching their code in Q&A websites such as Stack Overflow (Uddin and Khomh, 2019). Usually, they find their questions answered very quickly (with a median answer time of 11 min) (Mamykina et al., 2011). Both questions and answers may be validated and rated by other developers through *voting* and *comments*. Users who posted *up-voted* questions or answers, receive *reputation* scores which motivates individuals to contribute (Wang et al., 2015). Thus, Q&A websites such as Stack Overflow have become an inevitable source for developers to find solutions to their questions and issues (Wang et al., 2015).

At the time of writing this paper, more than 17.8 million questions, 27.2 million answers and 70 million comments have been submitted to Stack Overflow. According to the Stack Overflow developers' survey, each month, about 50 million people visit the website to learn, share, and build their careers (Anon, 2018). Consequently, the information available on this website constitute an enormous *crowd knowledge* about common errors,

* Corresponding author.
E-mail addresses: tahmooresi@ce.sharif.edu (H. Tahmooresi), heydarnoori@sharif.edu (A. Heydarnoori), rnadri@ce.sharif.edu (R. Nadri).
[1] Present Address: School of Computer Science, University of Waterloo, 200 University Ave. W., Waterloo, ON, Canada, N2L 3G1.

https://doi.org/10.1016/j.jss.2020.110724
0164-1212/

defects, and concepts trusted by millions of developers (de Souza et al., 2014; Mao et al., 2017).

The knowledge obtainable from Stack Overflow covers a wide range of aspects such as security and performance issues (Mao et al., 2017), programming styles (Barua et al., 2014), and API usage obstacles (Wang and Godfrey, 2013). Souza et al. (2019) state that many of the posts on Stack Overflow are primarily about API usage challenges. Moreover, API-related issues inferred from mining Q&A websites, hold particular promise as they contain discussions of the real-world issues encountered by millions of developers (Wang et al., 2015). For example, the method `cos(double angle)` (in the Java programming language) which is offered by the class `java.lang.Math` has made a large number of developers confused. This method returns the trigonometric cosine of an angle and the only argument *angle* should be given in radians. However, many developers do not comply with this requirement. This misunderstanding yields plenty of highly up-voted questions related to this method.[2] Additionally, focusing on more challenging APIs may prevent developers from concentrating on the application itself and new errors related to the application logic may take place.

Prior studies have shown the relation between API changes and the quantity of questions submitted to Stack Overflow (Linares-Vásquez et al., 2014), and mined developers' obstacles (Wang and Godfrey, 2013) and opinions (Uddin and Khomh, 2019) on APIs according to this website. Nevertheless, none of the prior research has focused on analyzing the effect of this knowledge on defect prediction models. We conjecture that if we extract the knowledge about APIs from Stack Overflow, we can make use of this knowledge in better explaining and predicting software defects.

In this paper, we investigate the relationship between the usage of APIs discussed in the crowd and software quality. To this end, we define the concept of *challenge of an API*, i.e., how much an API is discussed in high quality discussions on Stack Overflow. To better study the quality of Stack Overflow discussions, we statistically investigate Stack Overflow quality descriptors mentioned in prior studies (e.g., up votes, view count, favorite count, questioner reputation, and etc.), and employ *Explanatory Factor Analysis* (EFA) (Fabrigar et al., 1999) to identify the underlying relationship between these quality descriptors. Using EFA, we found out that three underlying factors can explain the interrelationships among all quality descriptors. Furthermore, using the concept of the *challenge of an API*, we propose a set of metrics that are based on crowd knowledge to study software quality. We investigate how our proposed metrics can help in explaining software defects, that is, whether adding our metrics to the models built with traditional metrics, increases the proportion of variation that the prediction model accounts for. We also investigate whether our metrics can improve the predictive power of the baseline models.

To measure the code quality, we employ post-release defects since it is widely used by prior studies and researches in the software quality area (Shang et al., 2015; Shihab et al., 2012). Post-release defects are the defects found up to six months after the release of a given version (de Pádua and Shang, 2018). We perform our detailed case study over 17 million discussions of Stack Overflow on five open source projects including Spring, Elastic Search, Jenkins, K-8 Mail client, and OwnCloud client with a focus on the following research questions:

RQ1: Are source code files using more challenging APIs more likely to be defect-prone?

We find positive correlations between crowd-related metrics and post-release defects. Our results show that in 4 out of 10 releases, there exist at least one crowd-related metric which gained a higher correlation with post-release defects compared to *pre-release defects* which is shown to has the highest correlation to post-release defects among traditional metrics (Moser et al., 2008). Given a version, pre-release defects are the defects found up to six months before the release of that version (Chen et al., 2017).

RQ2: Can crowd knowledge help in explaining post-release defects?

We find out that our crowd-related metrics can provide additional statistically significant explanatory power for software quality over traditional baseline metrics. More specifically, we achieve 11% – 51% improvement when we add our metrics to the models based on traditional metrics. Further, we found out our metrics have a positive effect on prediction models.

RQ3: Can crowd knowledge help in predicting post-release defects?

When our crowd-related metrics are added to the model based on traditional metrics, the predictive power of the model increases by 4% – 18% in terms of F1 measure. More specifically, we find out that our metrics provide a better improvement in the projects that use more external APIs.

Our findings could be leveraged by software developers to allocate more reviewing and testing efforts on source codes using more challenging APIs to prevent further defects. However, this does not imply that they should avoid using more challenging APIs. Instead, we just complement prior research on identifying high risk source codes to optimize the process of testing and reviewing.

To the extent of our knowledge, this paper is the first attempt to establish an empirical link between the crowd knowledge obtained from Q&A websites and post-release defects.

In summary, the contributions of this paper include:

- We propose new metrics based on the crowd knowledge which can be used to better explain and predict software defects.
- We perform an empirical study and quantify the statistical relation between our metrics and software quality in terms of post-release defects.

The rest of the paper is organized as follows. Section 2 comes with a few motivating examples. Section 3 describes how we model the crowd knowledge. Section 4 covers how our study is setup. Section 5 presents the results of our case study. Section 6 discusses overall points about our study. Section 7 mentions the threats to the validity of our findings. Section 8 discusses related prior research to this work. Finally, Section 9 concludes this paper and provides future research directions.

2. Motivating examples

This section presents a few examples to motivate investigating the relation between using more challenging APIs discussed in the crowd and the source code quality.

In revision `1e7a75c042` of the OwnCloud Android client[3] the developer uses the `WeakReference` class, which provides a reference that does not protect a referenced object from collection by the Java garbage collector. This revision introduces a bug, i.e., misusing `WeakReference` yielded some build time issues. The bug is later fixed in revision `f75ddcb97e` by passing proper arguments to `WeakReference`. Similar issues have been

[2] E.g., https://stackoverflow.com/questions/12975924/math-cos-gives-wrong-result.

[3] https://github.com/owncloud/android.

already discussed on Stack Overflow in up-voted questions.[4] Further, there exist other discussions about `WeakReference` which suggests that this class may have a high challenge.

The situation can be even more complicated. For example, in the buggy revision `610e1b410c` of Jenkins project,[5] nine external APIs are churned (added, changed, or deleted). By searching Stack Overflow, we found out six of these APIs where mentioned in the discussions on this web site. When, investigated the fixing commit, we observed that four of those APIs where churned again.

The above examples suggest that developers may reproduce common bugs that are already discussed in the crowd. More generally, APIs discussed in the crowd may contain complexities and obstacles which increase the chance of occurring bugs. Accordingly, developers may introduce the exact bugs which are already discussed, or other bugs (which may not be exactly mentioned in the crowd) because of the complexity, vagueness, or other challenges of the used APIs. For example, when using a complex API, the developer may concentrate on that API and this may introduce a bug which is due to the lack of concentration on the application logic.

Prior studies usually focus on the characteristics of source code or the process of development. However, developers face various errors and defects caused by the factors outside the development environment such as complexity and the lack of documentation of the external APIs. Since many of these challenges are discussed on the Stack Overflow, we study the relation between this crowd knowledge and post-release defects. We are going to leverage a source of knowledge about a massive amount of common challenges and errors about APIs which is not available through existing defect prediction approaches.

3. Modeling the crowd knowledge

In this section, we describe how we model the crowd knowledge available on Stack Overflow in order to propose crowd-related metrics.

The high level process of calculating crowd-related metrics is depicted in Fig. 1. Our approach is based on APIs discussed in the crowd. Thus, as the first step, we parse the heterogeneous data of Stack Overflow **(step 1)** to extract the code elements and then APIs from discussions. Next, we identify APIs, which are the main concerns of the discussions, **(step 2)** by using a Random Forrest classifier. Given an API and a discussion, the classifier determines if the API is one of the main concerns of that discussion or not. Then, leveraging quality descriptors associated with discussions on Stack Overflow, we calculate the challenge of APIs, i.e., how much the API is discussed in high quality discussions on Stack Overflow **(step 3)**. Next, using the source code and the history of changes (revisions), we extract APIs used in files along with APIs churned (added, edited, or deleted) in files over time **(step 4)**. Using this data along with the challenge of each API calculated in **step 3**, we calculate our proposed crowd-related metrics **(step 5)**. In the following we further discuss each step in more details.

3.1. Parsing discussions

To extract the APIs, we first need to detect code elements available in discussions. Next, we should parse code elements to obtain APIs. However, analyzing and parsing Stack Overflow discussions have two main challenges:

1. Posts may contain both unstructured fragments (natural language) and structured snippets (code, XML, exception stack trace, etc.). Consequently, intrinsic heterogeneity of the data may prevent us from extracting code constructs (e.g., APIs) properly (Ponzanelli et al., 2015).
2. Stack Overflow treats the structured code snippets as text. Besides, runnability of the code inside the discussion is of no importance for the authors of the posts. As a result, snippets may contain syntax errors and hence, not parsable (Subramanian and Holmes, 2013; Dagenais and Robillard, 2012).

To overcome the mentioned challenges, we use *island grammars*. These robust grammars can parse snippets of code into islands (recognizable constructs of interest) and water (remaining parts) by leveraging detailed rules (Moonen, 2001). Thus, an island grammar comprises two sets of rules: rules for islands describing the language constructs we are interested in, and rules for water describing the rest of the text with which we are not concerned (Afroozeh et al., 2012).

Using this type of grammars, partial code even with syntax errors can be parsed. We employ island grammars implemented by Ponzanelli et al. (2015) which constructs a heterogeneous abstract syntax tree (H-AST) from discussions of Stack Overflow. Thus, for each discussion, the AST of code snippets along with mentioned code elements inline with natural language is available. Thanks to island grammars, the code elements also include incomplete fragments like method and class declarations lacking the body.

3.2. Identifying mentioned APIs

Each discussion may contain long snippets of code with a huge amount of APIs. Inspired by Wang and Godfrey (2013), we consider method invocations along with class usages such as extending a class and annotations as API usage. However, in some cases an API mentioned in the code snippets or natural language texts of a discussion, is not considered the main concern of that discussion. For example, in Fig. 2, the class `String` is not as important as `doInBackground` and `getJSONFromUrl` methods. Ignoring this challenge would introduce serious noises to our final results since APIs such as `String` and `Boolean` are frequently used and would be diagnosed as highly challenging.

On Stack Overflow, APIs may emerge in seven main locations in a discussion: (1) tags, (2) title, (3) inline with text inside the question, (4) code snippet inside the question, (5) inline with text in the answers, (6) code snippets inside the answers, and (7) inline with comments. According to prior studies, mentioning an API in title, tags (Wang and Godfrey, 2013) or inline with text (Uddin and Robillard, 2017) shows the importance of that API for the questioner.

To setup a rule for our study, we considered presence of APIs in each location as a binary feature (e.g., occurrence in tags, title, etc.) and generated a classification model. We employed random forest as it shows promise when the features are meaningful (Bishop, 2006). To train the model, we randomly selected 600 discussions among all Java posts with at least one API. That is, we do not consider discussions without any API in title, body, or code snippets. Sampling 600 items among these discussions, ensures a confidence interval of 4% with a confidence level of 95%. Next we asked a group of three developers with at least four years of industrial experience to highlight the APIs that were the main concern of each discussion. That is, for each discussion we asked them to determine the APIs truly involved in the challenge of that discussion. We automatically extracted the locations of the APIs for each discussion. Finally, we trained the model in order to

[4] E.g., https://stackoverflow.com/questions/299659/whats-the-difference-between-softreference-and-weakreference-in-java.

[5] https://github.com/jenkinsci/jenkins.

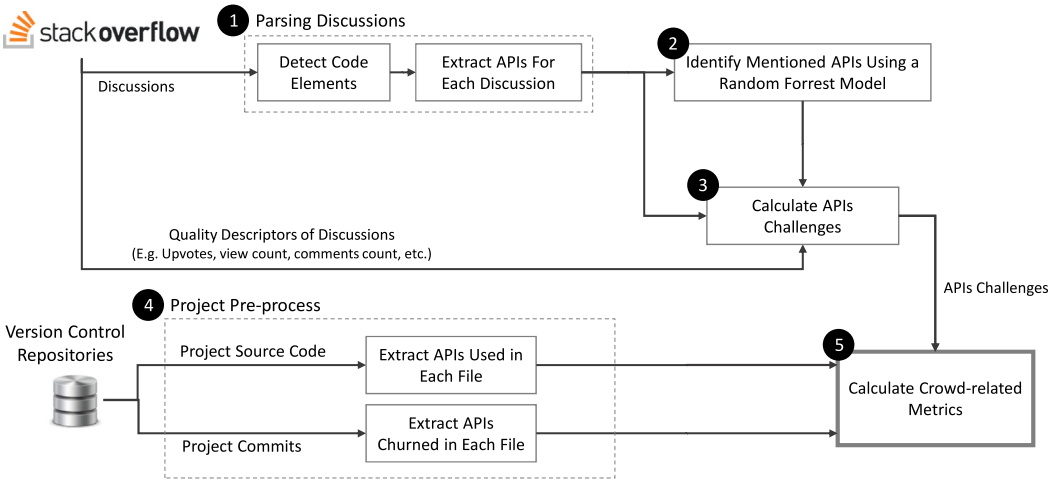

Fig. 1. The steps of using Stack Overflow to calculate our proposed metrics.

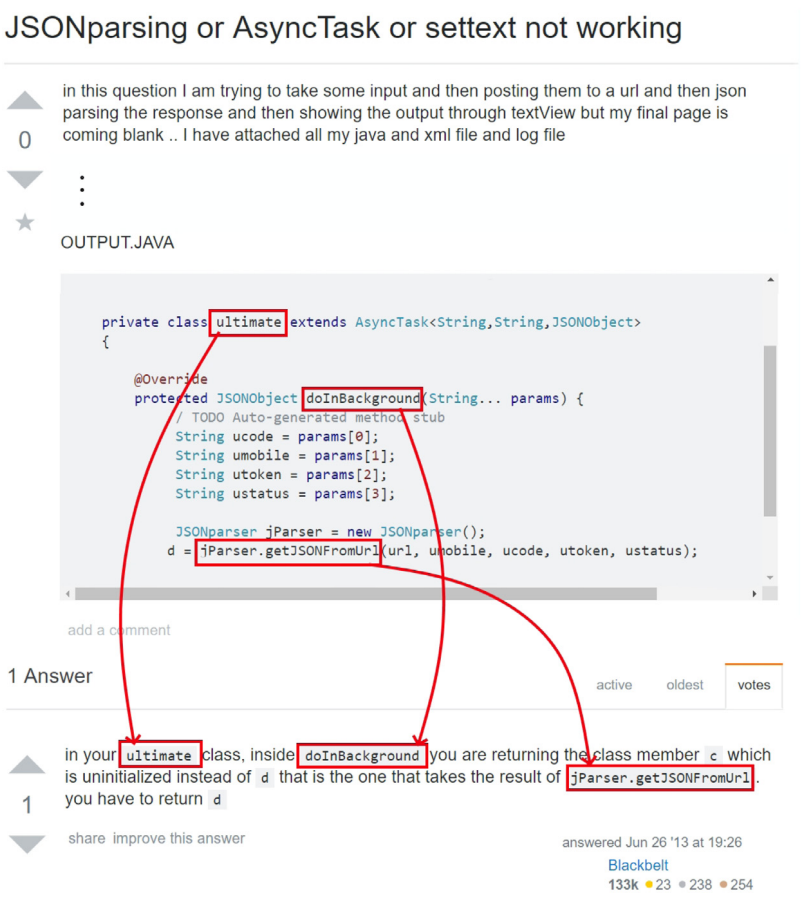

Fig. 2. Mentioning code elements inline in text.

classify the instances. Specifically, given the presence of an API in each location mentioned for a discussion (title, tags, etc.), it indicates whether that API is actually mentioned in the discussion as the main concern or not. We employed Python's *scikit-learn* package for this experiments using the 10-fold cross-validation technique, i.e, using 90% of the data set to train the classification model, and the remaining 10% to evaluate the accuracy of the model. We create 10 random 90%–10% splits of data and repeated the model evaluation for each pair. The trained model gained the mean precision and recall of 91% and 87%, respectively, which denotes that our approach would eliminate much of the noises we were concerned about.

Now, for the discussion d, we use our model to classify all of its APIs and denote the set of mentioned APIs as $Mentioned(d)$.

3.3. Measuring an API's challenge

To formulate the challenge of an API, a naive approach is to calculate the frequency of each API in all discussions and just consider more frequent APIs to be more challenging:

$$NaiveChallenge(api) = \#\ of\ discussions\ mentioning\ api \quad (1)$$

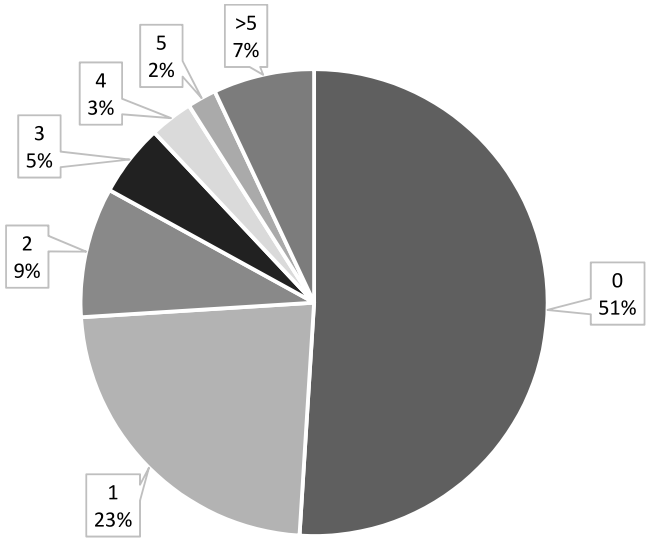

Fig. 3. Distribution of number of up votes on Stack Overflow.

The calculation of this formula is simple. However, this approach may be strongly affected by poor quality, incorrect questions, and common APIs. To address this issue, we may involve the quality of discussions in calculating the API challenge as follows.

3.3.1. Discussion values

Wang et al. (2015) suggest that a higher number of people who dislike a question by down-voting implies *incorrectness* and *poor quality* of that question. In contrast, the score, number of up-votes, comments, answers, views, and etc., denote the *importance*, *challenge*, and *popularity* of the questions. Inspired by Bajaj et al. (2014), we employed eight measures which describe the quality of discussions. These measures are listed in Table 1 along with their definitions and the rationale behind them. Since each of these measures may represent important aspects of the quality of a discussion, in this step we do not remove or merge any of them and postpone all decisions for further statistical analysis as discussed in Section 5.1. After normalizing each measure, we denote *Discussion Value* as $Val(d)$, the vector of eight measures associated with discussion d:

$$Val(d) = \langle UV(d), A(d), C(d), F(d), V(d), AUV(d), TTFA(d), QR(d)\rangle \tag{2}$$

3.3.2. The API challenge

Besides the value of discussions, there exist other issues when it comes to calculating an API's challenge. Fig. 3 shows the distribution of up votes on Stack Overflow. 51% of all discussions have up votes of zero, and just 7% have up votes more than five. Similar distribution holds for other quality descriptors (e.g., answers count and views count). This issue puts obstacles in the way of formulating the challenge of an API. As an example, the class `IOUtils` is mentioned in 1153 discussions with an up vote of 1, and 112 discussions with up votes of more than 1 but less than 10, and only 23 discussions with up votes of higher than 10. Therefore, we need to moderate the effect of the large number of discussions with low quality. Hence, simply counting the number of times an API is mentioned on Stack Overflow may not represent how problematic it is Wang and Godfrey (2013). To overcome this issue, we use a solution similar to the ad hoc approach offered by Wang and Godfrey (2013). The algorithm of calculating the challenge of an API is shown in Algorithm 1.

Algorithm 1: Calculating the challenge of *api* with respect to quality descriptor *qd*

Input : *api*, *qd*
Output: Challenge of *api*

for $d \in Discussions(api)$ **do**
  $quartile \leftarrow GetQuartile(Val_{qd}(d))$;
  $count_{quartile} \leftarrow count_{quartile} + 1$;
$weightedMean \leftarrow \sum_{i=1}^{4} mean_{quartile_i} \times \log(count_{quartile_i} + 1)$;
return $\frac{weightedMean}{Entropy(api)}$;

This algorithm takes an API denoted as *api* and quality descriptor *qd* (e.g., up vote) as input and returns the challenge of *api* with respect to *qd*. $Discussions(a)$ is the set of all discussions in which *api* is mentioned. In this algorithm, values of Val_{qd} are divided into quartiles. For each discussion in $Discussions(a)$ the quartile to which $Val_{qd}(d)$ belongs is extracted (using *GetQuartile*). Then, the frequency of each quartile is calculated. Next, the weighted mean of these frequencies is calculated. The weights are the mean of each quartile in order to consider the value of quality descriptors. Note that, we actually consider $log(count_{quartile_i} + 1)$ instead of $count_{quartile_i}$ since the count of lower quartiles is usually much more than higher ones. As a result, the weights may not be properly reflected in the final weighted mean. Furthermore, using *log* allows important APIs that are used in specific areas (less frequently used while are mentioned in high quality discussions) to be noticed.

Another issue is that, widely used APIs that have relatively few challenges such as `String`, `System`, `toString`, `List`, `Map`, and etc., are frequently employed in any context. Hence, they may have more mentions on Stack Overflow. Although considering the definition of *Mentioned API* alleviates the noise of such APIs, we further need to mitigate the effect of such frequent non-informative APIs. Following Shannon (1948), we employ the *Shannon entropy* to alleviate the effect of too common terms and APIs. Other prior studies also employ entropy when it comes to identifying important topics (Linares-Vásquez et al., 2013b) and important terms on Stack Overflow (Ponzanelli et al., 2014b). Thus, we divide the *weightedMean* by the Shannon entropy of the API which is calculated as follows:

$$Entropy(api) = -\sum_{d \in D} p_d \log_2 p_d \tag{3}$$

where, $p_d \geq 0$ ($\sum_{d \in D} p_d = 1$) is the probability that *api* is mentioned in discussion *d*. Here, the entropy aims to measure the distribution and usage inequality of the API across different discussions (Matei et al., 2018). For example, `toString` has a high entropy (14.7) while the entropy of `WeakReference` is 1.2. This means `toString` is uniformly distributed in a large number of discussions. Thus, observing this API is not surprising, and hence, not much informative.

Note that, since the challenge of *api* is calculated with respect to a quality descriptor, $Challenge(api)$ would be a vector composed of the challenges of *api* with respect to all quality descriptors (Eq. (2)). Further we define the challenge of an API as a measure, i.e, the higher the value, the more challenging the API is. Thus, we do not explicitly categorize APIs as challenging and not challenging.

The challenge of an API is calculated from involving millions of discussions and their corresponding quality descriptors. However, according to Algorithm 1, API challenges can be calculated once

Table 1
Variables used as quality descriptor of the Stack Overflow discussions.

Measure	Definition	Motivation and rationale
Up Votes (UV)	Number of users who promoted the question.	Higher UV implies the topic of discussion is important to the community (Bajaj et al., 2014).
Answers Count (A)	Number of answers provided to a question.	Serves as a proxy to the importance of a question. Many answers to a question implies that there may not exist a single direct solution (Wang et al., 2015).
Comments Count (C)	Number of comments provided to the question.	Implies the popularity of a topic. Comments are usually used for requesting more clarification from the author of the question or offering suggestions to improve the post (Wang et al., 2015).
Favorite Count (F)	Number of people who marked the question as favorite	Implies that the question is of interest to Stack Overflow members (Wang et al., 2015).
Views Count (V)	How many times the question is viewed by all visitors (Not necessarily registered users)	Implies the popularity of the question (Wang et al., 2015). Any visit made by either anonymous or authenticated user is counted. Most of the visitors of questions are anonymous, and this is the only publicly available measure capturing the behavior of such users.
Answer Up Votes (AUV)	Number of up votes the highest up voted answer takes	Shows the popularity of the answer (Wang et al., 2015) which implies the popularity of whole discussion.
Time to First Answer (TTFA)	Time (Minutes) it took to get a first answer	Early answers indicate a known bug, confirmation for API usability issues, and a workaround for a buggy API (Wang et al., 2015).
Questioner Reputation (QR)	The reputation score of a questioner given by the Stack Overflow community	Reputation resembles the quality of the users. Thus, high quality users tend to ask high quality questions (Wang et al., 2015).

(can also be updated periodically) and after that, there is no need for Stack Overflow massive data.

As mentioned earlier, complexity, change and fault-proneness, and the lack of documentation of the APIs trigger Stack Overflow questions (Souza et al., 2019). As more developers have common issues associated to an API, the discussions related to that API get more up votes, view count, etc., and as a result the challenge of that API increases.

3.4. Crowd-related metrics

Inspired by prior defect prediction studies (Chen et al., 2017; de Pádua and Shang, 2018), to study the relationship between APIs discussed in the crowd and post-release defects, we propose our crowd-related metrics, covering both aspects of product and process metrics. Product metrics are calculated from static characteristics of the system such as LOC. Process metrics leverage historical project characteristics such as code churn (Nagappan and Ball, 2005).

We propose two metrics *Challenging API Density (CAD)* and *Challenging API Churn (CAC)*. Analogous to prior studies (Shang et al., 2015), each metric is calculated for a file which enables us to study our metrics in the context of post-release defects. *CAD* is a product metric which measures how much a file contains challenging APIs. *CAC* is a process metric which measures how much high challenging APIs are churned in a file over the time.

3.4.1. Challenging API density

Intuition and rationale. When more challenging APIs are used in a software artifact, it may introduce defects similar to the ones reported and discussed by the crowd. Moreover, upon awareness of the threats, developers may focus on these challenges and introduce other defects due to the lack of concentration on the application logic.

Calculation. Considering *CAPIs* as the set of all APIs obtained from the crowd, we calculate the *CAD* of a file (denoted as f) as follows:

$$CAD(f) = \frac{\sum_{api \in f \cap CAPIs} Challenge(api)}{LOC(f)} \tag{4}$$

where $LOC(f)$ denotes the number of total lines of code in f. In fact, $CAD(a)$ is the sum of the challenges of the APIs employed in the f. These are APIs that are not declared in the project, but in the external libraries. Finally, the metric is divided by $LOC(a)$ to factor out the influence of code size, because, prior research have shown that the code size is related to the number of defects in files (Zimmermann et al., 2007).

CAD is a product metric, since it is obtained from a single snapshot of the system. Note that, similar to *Challenge*, $CAD(f)$ is also a vector. For example, CAD_{UV} denotes the value of *CAD* with respect to up vote as the measure of discussion value.

3.4.2. Challenging API churn

Intuition and rationale. Over the time, when developers add or edit more APIs inside a file, they may introduce more bugs in that file. Similar issues hold for API deletion. Deleting an API usage needs considerations such as replacing the functionality offered by the API or ensuring graceful removal of that functionality. Further, editing an API such as changing its parameters needs enough knowledge about the API and its signature. We consider adding, editing, and deleting of APIs as *API churn*.

Calculation. Considering prior commits in which file f is involved as $PC(f)$, and the set of all APIs churned in file f in revision c as $Churned(f, c)$, we calculate *CAC* of the file f as follows:

$$CAC(f) = \frac{\sum_{c \in PC(f)} \sum_{api \in Churned(f,c) \wedge api \in CAPIs} Challenge(api)}{|PC(f)|} \tag{5}$$

In other words, we iterate over all prior commits in which file f is involved and sum up all the challenges of APIs churned in f for each commit. In order to normalize the result, the overall summation is then factored by the length of $PC(f)$. Similar to *CAD*, the metric *CAC* is also a compound metric.

Although we proposed *Challenge* (Algorithm 1) as a measure for the API challenge, in this step we also consider *NaiveChallenge* (Eq. (1)) as a simpler measure. If we obtain similar results when considering *NaiveChallenge* compared to *Challenge*, there would be no need for such a complicated formulation for the challenge of APIs. Hence, we also propose CAD_N and CAC_N, as two crowd-related metrics, which are the results of considering *NaiveChallenge* as the measure of challenge.

4. Case study design

In this section, we introduce the systems that we employ as our case study and other data processing steps.

4.1. Studied systems

When it comes to choosing a project as our case study, various parameters must be taken into account. For example, the programming language, maturity, project size (in terms of LOC), project sufficient history (number of commits), are some of the most mentioned criteria in the literature (Shang et al., 2015; Fukushima et al., 2014). We considered Java as the programming language of the systems because it is popular and is widely studied in prior research (de Pádua and Shang, 2018; Chen et al., 2017). We opt to study open-source Java projects that are highly rated on GitHub (one of the most popular development platforms) to facilitate the replication of our study. Table 2 summarizes the characteristics and considered versions of the selected projects.

We selected Spring,[6] Elastic Search,[7] Jenkins, K-9 Mail Android client application,[8] and OwnCloud Android client application. Spring is an application framework and inversion of control container for Java. Elastic Search is a search engine based on Lucene library. It provides a distributed database of documents along with a full-text search engine. We choose Spring and Elastic Search since they are two large mature and popular projects and are already employed as case study projects in the defect prediction area (Mnkandla and Mpofu, 2016).

Jenkins is the leading open-source development workflow automation server which is known as a prominent continuous integration tool (Ahasanuzzaman et al., 2018). We choose Jenkins because it is a popular, mature and active open source project. Further, it has been recently employed when studying API issues on Stack Overflow (Ahasanuzzaman et al., 2018) and also in the defect prediction area (Borg et al., 2019).

K-9 Mail client is an advanced Android mail client. OwnCloud is the leading open source cloud collaboration platform which provides a straightforward way to sync files and share data. Users can access their files and data through their desktop computers and smart phones. In our case study, we considered the OwnCloud Android client. Note that, we opt to consider two mature Android projects for two main reasons: First, Android applications massively use Android APIs and external libraries (Wang and Godfrey, 2013). Second, the popularity of Android applications has increased in recent years and there has been a large body of research about these applications (Xia et al., 2016; Mojica et al., 2014), thus, we aimed to study the effect of crowd knowledge in explaining and predicting defects in such projects. Most Android applications have small size (LOC) and few history. Thus, we choose two of the largest and most mature projects with more than 74*K* lines of code developed during more than 8 years which is considered enough when studying defects according to prior studies (Chen et al., 2017; de Pádua and Shang, 2018).

4.2. Data processing

In our study, we processed discussions of Stack Overflow from the emergence of this website (Sep. 2008) until Sep. 2019. Prior studies filter low quality discussions to prevent further noises (Bajaj et al., 2014). Hence, we filter questions with minus score. Additionally, we filter questions with score equal to zero while having no accepted answer. By doing so, 18% of all discussions are pruned. We extract Stack Overflow measures (e.g., up votes, down votes, etc.) using Stack Exchange Data Explorer (Anon, 2019).

To traverse Git repositories, we leverage Eclipse jGit, a pure Java library implementing the Git version control system. Hence, we could programmatically perform operations like iterating over commits, reading commit messages, and extracting changes in each commit. For parsing the source code, modeling the Abstract Syntax Tree (AST), and extracting APIs, we employ Eclipse JDT. To extract buggy commits we use the online tool.[9] offered by Rosen et al. (2015) Using this tool, we can easily calculate post-release/pre-release defects for each file by counting the number of defects found up to six months after/before a given version in which that file is involved in those defects.

To ensure high reproducibility of our study, we release the source code of the implementations such as data processing, metrics calculation, tools, and statistical analysis which is publicly available.[10]

5. Case study results

In this section we present and discuss the results of our case study. First we statistically analyze the quality descriptors listed in Table 1. Then, for each research question provided in Section 1, we discuss the underlying motivation, our approach toward answering that question, and finally the obtained experimental results. At last, we conduct a qualitative analysis about the challenge of APIs.

5.1. Analyzing stack overflow quality descriptors

Recall that the pipeline of calculating crowd-related metrics has two main steps: First, we used the quality descriptors listed in Table 1 to calculate the *Challenge* of APIs. Second, we calculated compound metrics *CAD* and *CAC* as discussed in Section 3.4. However, the collinearity of the quality descriptors yields problems of overfitting, multicollinearity, and an unstable model across different data sets (Chen et al., 2017). Thus, we leverage Spearman's rank correlation (Zar, 1972) to analyze the correlation between quality descriptors listed in Table 1. In our analysis, $\rho > 0.7$ is considered strongly correlated (Jiarpakdee et al., 2019) and $\rho < 0.4$ is supposed to be weak (Khomh et al., 2011).

The results of correlation analysis of quality descriptors are available in Fig. 4. The results described in this figure suggest that there may be a smaller set of latent factors describing the association among observed quality descriptors. For instance, UV has a significant correlation with most of the other variables.

To better extract the underlying factors that explain the interrelationships among those variables, we leverage EFA (Fabrigar et al., 1999). This statistical method identifies the underlying relationships between measured variables. Similar to Principle Component Analysis (PCA), EFA is a linear-Gaussian model generally used to reduce the dimensionality of data (Bishop, 2006; DeCoster, 1998). Prior studies employed EFA in order to analyze Stack Overflow discussions (Fu et al., 2015; Matei et al., 2018). Further, underlying factors obtained from EFA are labelable and interpretable (DeCoster, 1998). Since our study is about the crowd knowledge, we are much more interested in gaining an insight and making statements about the underlying factors.

Before diving into EFA, we perform Kaiser–Meyer–Olkin (KMO) test for sampling adequacy (Cerny and Kaiser, 1977). KMO is a measure of how suited the data is for EFA. The overall KMO for our data is 0.88 which indicates that we are allowed to leverage EFA. We apply *varimax* rotation technique as it increases the interpretability of the factors (DeCoster, 1998). For choosing the proper number of factors we take into account the *Kaiser criterion*

6 https://github.com/spring-projects/spring-framework.
7 https://github.com/elastic/elasticsearch.
8 https://github.com/k9mail/k-9.
9 http://commit.guru.
10 http://bit.ly/expdfct.

Table 2
Studied projects, versions considered, GitHub stars (resembles the popularity), LOC, number of pre-release defects and post-release defects for each version of the projects.

	Spring		Elastic search		Jenkins		K-9 Mail		OwnCloud	
Start year	2008		2010		2006		2008		2011	
GitHub stars	30 k		41.8 k		13.1 k		4.4 k		2.8 k	
Release number	v4.0	v5.0	v2.2	v6.1	v1.62	v2.89	v5.0	v5.2	v2.0	v2.7
LOC	980 k	1107 k	1158 k	1322 k	190 k	219 k	105 k	111 k	74 k	94 k
Pre. defects	6768	4709	7241	4121	3563	3290	763	582	649	538
Post. defects	3910	2550	4852	2581	1361	652	673	511	453	401

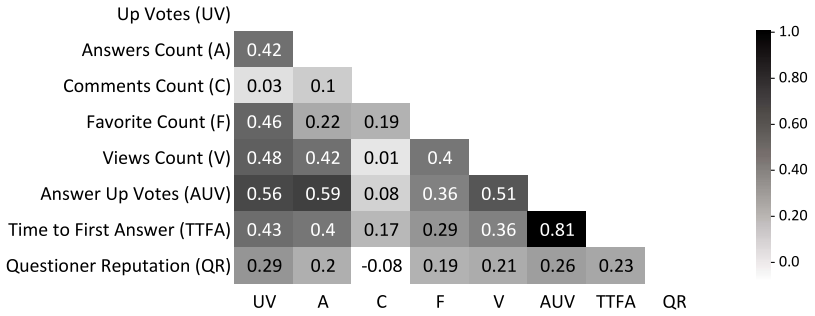

Fig. 4. Spearman rank correlation between quality descriptors of the Stack Overflow discussions (p-value < 0.05).

which states that the number of factors is equal to the number of the eigenvalues of the correlation matrix that are greater than 1.0 (DeCoster, 1998). Additionally, if we plot the eigenvalues of the correlation matrix in descending order, the factors before the major drop in eigenvalue magnitude should be retained which is known as *Scree test* (Cattell, 1966). Eigenvalues demonstrated in Fig. 5 suggest that we may go further with just three factors, i.e., $F1$, $F2$, and $F3$. To elaborate more, Table 3 depicts the factor loadings which shows the relationship of each variable to the underlying factor. Specifically, factor loading is the percentage of variance in that indicator variable is explained by the factor. Table 3 shows that $F1$ explains all variables except C (comment count), and QR (questioner reputation). This suggests that, a single latent factor is behind most of the quality descriptors of Stack Overflow. This confirms prior studies saying the *popularity* is a key rationale behind up votes, view count, favorite count, and many other quality descriptors (Wang et al., 2015; Bajaj et al., 2014) (refer to Table 1 for rationale behind each metric). Therefore, from now on, we will refer to $F1$ as *Pop* which stands for popularity.

According to the factor loadings presented in Table 3, $F2$ and $F3$ complement *Pop* (the factor $F1$) by explaining QR (0.81), and C (0.65) respectively. We manually investigated questions which have high QR or C but low UV. We observed that a highly reputed user may ask questions which are not significantly up voted due to obscurity and complexity of the challenge the questioner has faced.[11] For C, we found that questions with more comments are not necessarily important or popular (in terms of up votes), but more ambiguous and need further clarification and improvement. To increase the understandability of our analysis, we also refer to $F2$ and $F3$ as QR and C respectively.

As obtained factors can explain 88% of the variances (Table 3), from now on, we consider just these three factors as the dimensions of *Val*. Consequently, we will have CAD_{Pop}, CAD_{QR}, CAD_{C}, CAC_{Pop}, CAC_{QR}, and CAC_{C} as our crowd-related metrics. Note that,

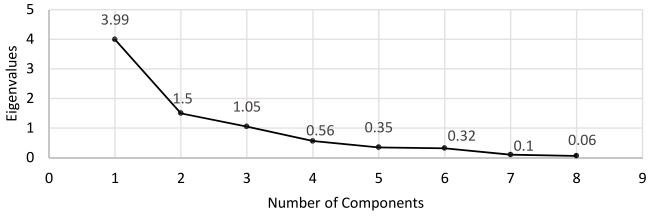

Fig. 5. The Scree plot demonstrating eigenvalues of the correlation matrix in descending order.

recalling *NaiveChallenge*, we also consider CAD_N and CAC_N as two other crowd-related metrics.

Three factors can describe all quality descriptors, i.e., popularity (Pop), questioner reputation (QR) and comments count (C).

5.2. RQ1: Are source code files using more challenging APIs more likely to be defect-prone?

Motivation. Before investigating the effect of crowd-related metrics on explaining defects, we simply want to study whether source code files that use more challenging APIs are more defect-prone.

Approach. Our metrics are not normally distributed. Thus, we calculate Spearman's rank-order correlation between our crowd-related metrics and post-release defects. Since this non-parametric coefficient measures the strength and direction of the association between two ranked variables (Zar, 1972), we can measure to what extent our metrics prioritize files similar to ranking them based on post-release defects.

Results and discussion. Fig. 6 shows the Spearman rank correlation between crowd-related metrics and post-release defects. To better analyze the results, the correlations between post-release defects and two traditional metrics, i.e., pre-release defects (PRE), and code churn (as two of the mostly used process metrics in defect explanation studies (de Pádua and Shang, 2018)) are also included. Most of our metrics have a positive correlation with post-release defects. In every case except both

[11] E.g., https://stackoverflow.com/questions/42759240/calculating-row-wise-delta-values-in-a-dataframe.

Table 3
Factor loadings and variances of each factor.

	Factor loadings								Var	Proportion var	Cumulative var
	UV	A	C	F	V	AUV	TTFA	QR			
F1 (Pop)	0.82	0.73	0.01	0.79	0.70	0.93	0.85	0.05	3.50	0.62	0.62
F2 (QR)	−0.24	−0.18	0.07	0.10	0.04	0.24	0.30	0.81	1.16	0.21	0.83
F3 (C)	0.18	0.32	0.65	0.15	0.23	0.21	0.19	0.07	0.25	0.05	0.88
Commonalities	0.76	0.66	0.42	0.65	0.54	0.96	0.84	0.66			

versions of Spring and Elastic Search v6.1, there exists at least one crowd-related metric with higher correlation than code churn. In four releases, the maximum correlation is obtained by *Pop*-based crowd-related metrics.

Generally, *Pop*-based metrics are more correlated to post-release defects than *QR*-based metrics. Therefore, considering the popularity as the quality descriptor of discussions results in more correlated metrics to post-release defects than the questioner reputation. In contrast, the metrics CAD_C and CAC_C have a weak correlation with post-release defects. This observation shows that comments count is not a proper quality descriptor for a discussion. As mentioned earlier, questions with more comments are more ambiguous and need further clarification and improvement (Wang et al., 2015). Thus, we will drop CAD_C and CAC_C from further analysis.

Similar to metrics based on comments count, CAD_N and CAC_N have a weak correlation with post-release defects. This is due to the fact that *NaiveChallenge* does not take into account any of the mentioned complexities of Stack Overflow discussions. For example, by using *NavieChallenge*, APIs such as `java.lang.Override` and `java.io.PrintStream.println` get a high challenge since they are commonly used in any context. Furthermore, since such common APIs are employed frequently in all source code files, CAC_N and CAD_N get large values for most of the source code files. Hence, we also drop these two measures from further analysis.

The process metrics (CAC_{Pop} and CAC_{QR}) performed slightly better compared to product metrics. We will further discuss our metrics when answering *RQ*2.

Note that, in this step, we are not willing to show that the crowd-related metrics have a better correlation with post-release defects compared to traditional metrics. We just show that, the crowd-related metrics have a positive correlation with post-release defects comparable to traditional metrics.

> *Most of our metrics have positive correlation with post-release defects. Further, in 7 out of 10 releases there exists at least one crowd-related metric with higher correlation than code churn. In four releases, the maximum correlation is obtained by crowd-related metrics.*

5.3. RQ2: Can crowd knowledge help in explaining post-release defects?

Motivation. Analyzing just the correlation between crowd-related metrics and post-release defects does not necessarily convey that these metrics help explaining defects. For example, it may be due to other factors such as LOC, since this factor is correlated to both crowd-related metrics and post-release defects. Therefore, we would like to investigate whether our crowd-related metrics can complement prior traditional metrics in explaining post-release defects.

Approach. Following prior studies (Chen et al., 2017; Shang et al., 2015), we are not going to directly predict defects in this step. Instead, we aim to investigate how much our crowd-related metrics can improve the deviance explained by the traditional baseline models. Considering crowd-related metrics will involve the knowledge and experience of millions of developers on APIs in explaining defects.

Table 4
Traditional product and process metrics that we take into account during our analysis.

	Metric	Description
Product metrics	LOC	Lines of code
	PAR	Number of Parameters
	NOF	Number of Attributes
	NOM	Number of Methods
	CC	Cyclomatic Complexity
	WMC	Weighted Method Count
	NOT	Number of Classes
	DIT	Depth of Inheritance Tree
	RFC	Response For Class
	NOC	Number of Children
	CBO	Coupling Between Objects
	LCOM	Lack of Cohesion in Methods
Process metrics	PRE	Number of Pre-release Defects
	Churn	Number of lines of code added, modified, or deleted
	Entropy	Entropy of changes
	NAUTH	Number of authors who committed

We employ *Logistic Regression* to study the explanatory power of our crowd-related metrics on code quality. Logistic Regression is one of the most popular machine learning techniques in defect prediction area (Jiarpakdee et al., 2019). Palomba et al. (2017) have recently investigated various machine learning techniques, namely, ADTree, Naive Bayes Logistic Regression, Decision Table Majority, and Simple Logistic in defect prediction context. They confirmed that Logistic Regression obtained the best results. We choose post-release defects as the dependent variable of our statistical analysis, since this metric is widely used by prior studies as a software quality indicator (Chen et al., 2017; Shihab et al., 2012).

For generating baseline models, we use the metrics listed in Table 4. Note that, LOC, PAR, NOF, NOM, and CC are recommended by prior studies to evaluate defect prediction models (Taba et al., 2013). Additionally, many approaches in the literature use CK metrics, i.e., WMC, NOT, DIT, RFC, NOC, CBO, and LCOM which describe important object-oriented characteristics of the code (Gyimothy et al., 2005). For process metrics, we take into account PRE, Churn, Entropy, and NAUTH, as recommended by D'Ambros et al. (2010). Note that, LOC, PRE, and code churn have been shown to have the best explanatory power for defects (Gyimothy et al., 2005) and widely used in prior studies when it comes to proposing new metrics (Bird et al., 2011; Shihab et al., 2011).

Using the metrics listed in Table 4, we generate three baseline models: (1) based on traditional product metrics (referred as TPROD) (2) based on process metrics (referred as TPROC), and (3) based on both product and process metrics (TPROD + TPROC). Then, different combinations of our metrics are added to the baseline models to obtain combined models. However, analyzing all possible combinations of our metrics may not make much sense. Thus, for the sake of rationality and understandability of our analysis, we generate the models listed and described in Table 5.

Note that many traditional product and process metrics are highly correlated with each other (Shihab et al., 2010). Since we

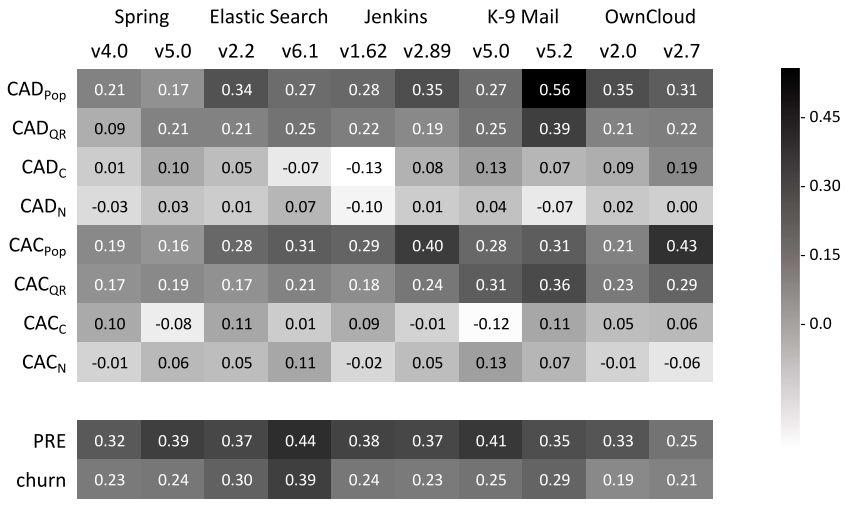

Fig. 6. Spearman rank correlation between crowd-related metrics and post-release defects.

use various metrics in this step, for each model, we employ PCA to avoid the problem of multicollinearity among the independent variables. Thus, instead of the variables, we use a set of principle components (PC) which are independent and therefore do not suffer from multicollinearity (Shihab et al., 2010). We select only the minimum number of PCs that account for at least 95% of the variance in observations.

Note that, for each version of the projects under study, we only used the Stack Overflow discussions before that version to calculate the *Challenge* metric. Finally, we examine the added explanatory power after the addition of our metrics. We employed D^2 to measure the deviance explained in order to examine the explanatory power of each model. D^2 is analogous to R^2 in linear regression and is a proper metric for measuring the deviance explained by logistic regression models (Guisan and Zimmermann, 2000; Chen et al., 2017). Thus, if the base models earn a higher D^2 value after the inclusion of our metrics this shows that our proposed metrics can give extra information when it comes to understanding defects. For calculating D^2 we employed **modEvA** R package.[12]

Before creating models, we test the normality of each metric by using the approach offered by d'Agostino (1971) which combines skewness and kurtosis to produce an omnibus test of normality. The test suggests that none of the metrics come from a normal distribution (p-value < 0.05). Additionally, we calculated the skewness of the metrics in all versions of our case study projects. Values greater than one indicates that most of the values are on the low scale (Chen et al., 2017). We observed that our metrics are skewed. Therefore, similar to prior studies (Chen et al., 2017; Shang et al., 2015) we log transformed all of the metrics.

Finally, to complement our analysis we study the effect of each metric on the model's outcome. To this end, we follow a similar approach used by prior research (Shihab et al., 2012; Mockus, 2010). First, we set all metrics to their mean value and calculate the model's outcome. Next, to measure the effect of the metric M, we increase the value of M by 10% of its mean value while keeping all other metrics to their mean value and recalculate the model's outcome. The difference between outcomes represents the effect of the metric M on the model. If the effect becomes positive it means that increasing the value of metric results in an increase in the likelihood of the model's outcome. Conversely, if the effect becomes negative this means that increasing the metric results in a decrease in likelihood of the model's outcome. In this step, we consider the composition of all metrics (Model 10 in Table 5) since we intent to analyze the effect of all of our metrics.

Results and discussion. Table 6 shows the deviance explained of three baseline models (highlighted rows) compared to the combined models (using our crowd-related metrics) and the resulted improvement (percentage). Model 1 is made up of just TPROD. Model 2 is the result of adding product metrics CAD_{QR} and CAD_{Pop} to the baseline Model 1. Comparing models 1 and 2 shows that adding our crowd-related product metrics improves D^2 at least 6% (Spring v5.0) and up to 44% (OwnCloud v2.7). Model 3 shows the base model generated by baseline process metrics (TPROC). When compared to Model 4 which is the result of adding our crowd-related process metrics CAC_{QR} and CAC_{Pop} to the Model 3, at least 7% improvement is obtained in all releases. Our process metrics improved Model 3 up to 78%. The last baseline Model 5 is composed of both TPROD and TPROC metrics. When adding metrics based on *QR* (Model 6), in 8 out of 10 studied versions improvement is noticeable (3% – 18%). When it comes to adding *Pop*-based crowd-related metrics to the baseline Model 5 results (Model 7) get better than Model 6 which is based on *QR* and the improvement is visible in all studied versions (5% – 41%). Similar to Model 2, we added our product metrics to base Model 5 (Model 8) which increases D^2 in all versions except Spring v5.0 (5%–25%). Adding our process metrics to the Model 5 (Model 9) improves D^2 slightly better than our product metrics (5% – 40%).

The last model is made up of all crowd-related metrics (Model 10) which shows an improvement in all versions (11% – 51%). Thus, when we added all of our crowd-related metrics to the best baseline model (Model 5), the explanatory power of the overall model increases at least 12% in all releases.

> *Our proposed metrics can provide additional statistically significant explanatory power for software quality over existing product and process metrics.*

Generally, our crowd-related metrics perform better in mobile applications (i.e., K-9 Mail and OwnCloud) than Jenkins and the results for Jenkins are also better than Elastic Search. The poorest results belong to Spring. Similar results hold for our correlation analysis (Section 5.2). We further analyzed the source code of these projects and found that the amount of external API usage differs between these projects. Table 7 shows two measures about each studied version, i.e., the percentage of files with external APIs and external API ratio. In both measures, external APIs means

12 https://rdrr.io/cran/modEvA/man/Dsquared.html.

Table 5
List of combined models built to answer *RQ2* along with the rationale behind each model. We will compare the deviance explained by each of these models to its corresponding baseline model.

	Baseline model	Included crowd-related metrics	Rationale
Model 1	TPROD	CAD_{QR}+CAD_{Pop}	How much our **product** based metrics can improve the base model of **product metrics**?
Model 2	TPROC	CAC_{QR}+CAC_{Pop}	How much our **process** based metrics can improve the base model of **process** metrics?
Model 3	TPROD+TPROC	CAD_{QR}+CAC_{QR}	How much our metrics based on *QR* can improve the base model of all baseline metrics?
Model 4		CAD_{Pop}+CAC_{Pop}	How much our metrics based on *Pop* can improve the base model of all baseline metrics?
Model 5		CAD_{QR}+CAD_{Pop}	How much our **product** based metrics can improve the base model of all baseline metrics?
Model 6		CAC_{QR}+CAC_{Pop}	How much our **process** based metrics can improve the base model of all baseline metrics?
Model 7		CAD_{QR}+CAD_{Pop} +CAC_{QR}+CAC_{Pop}	When adding **all** of our metrics to the base model of all baseline metrics, how much improvement can be achieved?

Table 6
Deviance explained (D^2) of three baseline models (highlighted rows) compared to the combined models (using our crowd-related metrics) and the resulted improvement (percentage). The "*" indicates that the metrics are not statistically significant when added to the baseline model. In all other models the crowd-related metrics are statistically significant (p-value < 0.05).

	Baseline Model	Included Metrics	Spring		Elastic Search		Jenkins		K-9 Mail		OwnCloud	
			v4.0	v5.0	v2.2	v6.1	v1.62	v2.89	v5.0	v5.2	v2.0	v2.7
Model 1	TPROD	-	0.14	0.17	0.15	0.12	0.17	0.14	0.12	0.10	0.11	0.09
Model 2		CAC_{QR}+CAC_{Pop}	0.15 (11%)	0.18 (6%)	0.16 (10%)	0.13 (8%)	0.20 (18%)	0.17 (21%)	0.15 (25%)	0.14 (40%)	0.15 (36%)	0.13 (44%)
Model 3	TPROC	-	0.25	0.21	0.23	0.27	0.24	0.19	0.17	0.16	0.11	0.09
Model 4		CAD_{QR}+CAD_{Pop}	0.27 (9%)	0.23 (10%)	0.27 (17%)	0.29 (7%)	0.29 (21%)	0.23 (21%)	0.21 (24%)	0.25 (56%)	0.17 (55%)	0.16 (78%)
Model 5	*TPROD+ TPROC*	-	0.28	0.33	0.32	0.27	0.26	0.23	0.19	0.18	0.12	0.14
Model 6		CAC_{QR}+CAD_{QR}	0.28* (0%)	0.34 (3%)	0.34 (6%)	0.29 (9%)	0.26 (0%)	0.26 (14%)	0.21 (8%)	0.20 (10%)	0.13 (12%)	0.17 (18%)
Model 7		CAC_{Pop}+CAD_{Pop}	0.29 (5%)	0.36 (8%)	0.38 (20%)	0.31 (15%)	0.35 (33%)	0.28 (20%)	0.27 (40%)	0.24 (31%)	0.16 (35%)	0.20 (41%)
Model 8		CAC_{QR}+CAC_{Pop}	0.29* (5%)	0.33* (0%)	0.36 (11%)	0.29 (7%)	0.33 (25%)	0.25 (9%)	0.23 (23%)	0.20 (10%)	0.14 (20%)	0.17 (18%)
Model 9		CAD_{QR}+CAD_{Pop}	0.30 (7%)	0.35 (5%)	0.37 (16%)	0.30 (10%)	0.35 (33%)	0.27 (16%)	0.23 (22%)	0.23 (25%)	0.13 (11%)	0.20 (40%)
Model 10		CAC_{QR} + CAC_{Pop}+ CAD_{QR} + CAD_{Pop}	0.31 (11%)	0.39 (18%)	0.39 (23%)	0.34 (25%)	0.36 (38%)	0.29 (27%)	0.28 (45%)	0.25 (37%)	0.18 (47%)	0.21 (51%)

the APIs which belongs the to external frameworks or libraries. The external API ratio is measured as follows which denotes the portion of external API calls among all API calls in a project:

$$External\,API\,ratio = \frac{\#\,of\,external\,API\,calls}{\#\,of\,all\,API\,calls} \quad (6)$$

Table 7 shows that mobile applications use more external APIs than Jenkins, Elastic Search, and Spring, and the same holds for external API ratio. We conclude that, although our crowd-related metrics help explaining defects in all projects under study, these metrics perform better when the project uses more external APIs. Obviously, this is because our crowd-related metrics heavily rely on the use of external APIs (Eqs. (4) and (5)). Therefore, the growth in the use of external frameworks and libraries (As it is happening in the software development industry) increases the applicability of our metrics.

Table 6 shows that *Pop*-based metrics (CAC_{Pop} and CAD_{Pop}) generally performed better than *QR*-based metrics (CAC_{QR} and CAD_{QR}). This may be due to the fact that questions with *higher* questioner reputation but *lower* popularity contain advanced challenges that a professional developer asks but is not faced by many developers. Thus, these challenges may be less prevalent. However, comparing Model 10 with Model 7 shows that, adding *QR*-based metrics to the model composed of both traditional metrics and *Pop*-based metrics (Model 7), increases the explanatory power of the overall model.

According to the results, process based metrics performed relatively better than product metrics. However, in spite of the product metrics, process metrics need enough project history and are not available at the beginning of a project.

Finally, Table 8 shows the effect of each of our metrics on the model's outcome. Generally, our metrics have positive effects on the model and none of the metrics have negative effect on the outcome. This implies that, using more challenging APIs has a positive effect on the chance of post-release defects. Because, when *CAC* and *CAD* metrics increase, this means that the source code file is using more challenging APIs which increases the chance of bug. However, *QR*-based metrics have less effect on the outcome compared to *Pop*-based metrics. The results for Spring project shows that our crowd-related metrics have poor effect on the outcome for this project which may be due to the amount of external API usage in this project (Table 7).

Our crowd-related metrics have positive effects on defect proneness.

5.4. RQ3: Can crowd knowledge help in predicting post-release defects?

Motivation. By now, we analyzed the correlation between our crowd-related metrics and post-release defects. Further, we showed that our metrics can improve explanatory power of defect prediction models. However, we have not studied weather our metrics can improve predictive power of the overall model yet.

Approach. In this step, we would like to find out if our crowd-related metrics can improve traditional defect prediction models. Thus, we generate a model based on baseline metrics described in Table 4 (referred as BASE) and another model based on both our and the baseline metrics (referred as BASE+CRD). Similar to previous step, we use logistic regression as the prediction model. The dependent variable of our models is a two-value variable that represents whether or not a file has one or more post-release bugs. Again, we use PCA to avoid the problem of multicollinearity

Table 7
Percentage of files with external API Studied projects along with external API ratio of each studied version.

	Spring		Elastic search		Jenkins		K-9 Mail		OwnCloud	
	v4.0	v5.0	v2.2	v6.1	v1.62	v2.89	v5.0	v5.2	v2.0	v2.7
Files with External API	18%	15%	24%	23%	34%	39%	68%	72%	70%	75%
External API ratio	11%	8%	13%	16%	17%	21%	31%	33%	35%	39%

Table 8
Effect of crowd-related metrics on post-release defects. Effect is measured by setting a metric to 110% of its mean value while keeping all other metrics at their mean values. Generally, hour metrics have a positive effect on post-release defects.

Metric	Spring		Elastic search		Jenkins		K-9 Mail		OwnCloud	
	v4.0	v5.0	v2.2	v6.1	v1.62	v2.89	v5.0	v5.2	v2.0	v2.7
CAC_{QR}	0.1%	0.0%	0.3%	0.5%	0.8%	3.0%	1.1%	1.6%	2.5%	2.9%
CAC_{Pop}	0.3%	0.5%	0.9%	1.6%	2.2%	3.0%	4.1%	4.5%	4.0%	5.3%
CAD_{QR}	0.0%	0.0%	0.2%	0.0%	0.3%	0.5%	1.9%	1.4%	1.5%	0.6%
CAD_{Pop}	0.1%	0.2%	0.2%	0.5%	0.9%	2.1%	2.5%	2.7%	3.0%	2.7%

among the independent variables. Further, similar to RQ2, for each version, we only used the Stack Overflow discussions before that version to calculate the *Challenge* metric.

The classification performance is evaluated using the 10-fold cross-validation. We employ standard classification evaluation measures that are commonly used in defect prediction including accuracy, recall, precision, F1 (the harmonic mean of precision and recall), and area under the ROC curve (AUC) to evaluate the performance of the models. AUC is used toward imbalanced data since is obtained by varying the classification threshold over all possible values (Kamei et al., 2013). The range of AUC is [0, 1] and larger AUC value indicates better prediction performance. AUC is a common measure in defect prediction studies since the data sets of defects are usually imbalanced (Kamei et al., 2013).

Results and discussion. Table 9 shows the averages for precision, recall, accuracy, F1, and AUC measures of two models, i.e, the model based on only traditional metrics described in Table 4, and the model based on both traditional and crowd-related metrics. In all projects, when our crowd-related metrics are added to the baseline model, results improved in terms of precision (5%–28%), recall (2%–15%), accuracy (2%–17%), F1 (4%–18%), and AUC (3%–20%). We focus on F1 since it is a traditional method of assessing both precision and recall at the same time (Bird et al., 2009). The percentage of improvement with respect to F1 for each project is also described in the last row of Table 9. For the mobile projects (i.e., K-9 Mail and OwnCloud), the F1 improvement is greater than Jenkins, Elastic Search, and Spring. the poorest improvement is for Spring v5.0. Similar to the conclusions we drew for *RQ2*, the results show that our crowd-related metrics perform better for the projects that use more external APIs (recall Table 7).

Our analysis complements prior findings of this paper that our proposed metrics when added to traditional metrics can improve predictive power of prediction models.

Our crowd-related metrics can improve traditional post-release defect prediction models by 4%–18% in terms of F1 measure.

5.5. Qualitative analysis of the API challenge

By now, we have shown that crowd-related metrics can improve traditional defect prediction models in terms of explanatory and predictive power. Now, we provide some insight about what are the most challenging APIs used by each of the systems under study. We consider $Challenge_{Pop}$ as the measure of challenge, since we showed that Pop-based metrics perform better than QR-based metrics in terms of correlation with post-release defects (Section 5.2) and improving the explanatory power of defect prediction models (Section 5.3).

Due to the large number of APIs, we first study at library (package) level and then drill down to the API level. We extract all external libraries for each project. For obtaining the challenge for each library in a project, we simply calculate the mean of $Challenge_{Pop}$ for all APIs of the library that are used in the project. Fig. 7 shows most challenging libraries for each project under study. For Spring, `java.io`, `java.util`, `javax.servlet`, and `aspectj` are the top most challenging libraries. For Elastic Search, `apache.lucene`, `javax.net`, and `java.lang`, and for Jenkins `javax.servlet`, `springframework`, and `java.beans` are the most challenging libraries. For K-9 Mail and OwnCloud, Android APIs are more challenging than other APIs, i.e., `android.content`, `android.widget`, and `android.view`.

To provide insight at the API level, due to the large number of APIs, for each project we only focused on the most challenging libraries, i.e., `java.io` for Spring, `apache.lucene` for Elastic Search, `javax.servlet` for Jenkins, `android.content` for K-9 Mail, and `android.widget` for OwnCloud. Fig. 8 demonstrates challenges of APIs for each library. For `java.io`, APIs such as `BufferedInputStream.read` which are related to reading input and files are more challenging. In `javax.servlet` APIs such as asynchronous processing, and in `apache.lucene` APIs related to queries and document similarity (`like`) have more challenge. In `android.widget`, APIs related to layout and size of widgets (such as `addFooterview` and `getGlobalVisibleRect`), and in `android.content` library APIs related to Android services and event passing between components such as `bindService` and `sendBroadcast` are more challenging.

To better describe how our approach works, we investigated some of the high challenging APIs. For example, `Context.sendBroadCast` is intended to broadcast the given intent to all interested broadcast receivers in Android. However, many developers have challenges understanding how this API works. For example, in a discussion[13] the developer uses `sendBroadcast` method inside a dialog which causes the application to crash. In the answer, it is mentioned that the developer should first register the receiver with an action in the `onCreate` method and then unregister it in the `onDestroy` callback method. In revision `15a163bfef` of the K-9 Mail project the developer does not unregister the receiver and a similar defect is introduced.

As another example, `getExternalFilesDir` returns the absolute path to the directory on the primary shared/external storage device where the application can place persistent files it owns. In an `Activity` subclass it is enough to call `getExternalFilesDir()`, but in other classes it should be called with a context. This issue is already discussed in Stack Overflow.[14] In revision `d14fad563e` of OwnCloud project, the developer calls this API without providing any context and a similar defect occurs.

As the final example, `onSaveInstanceState` is invoked when an Android activity may be temporarily destroyed. This method is used to save the instance state. However, using this API have common challenges. For example, if no `id` is set for a view, Android will not properly call `onSaveInstanceState`.[15]

13 https://stackoverflow.com/questions/8022999/how-do-i-sendbroadcastintent-from-mydialog-and-receive-in-myactivity/8023310.

14 https://stackoverflow.com/questions/34638931/android-says-cannot-resolve-method-getexternalfilesdirnull.

15 https://stackoverflow.com/questions/28586443/android-view-onsaveinstancestate-not-called.

Table 9
Performance comparison between the model composed of only traditional metrics and the model composed of both traditional and crowd-related metrics. The percentage of the improvements for F1 measure is available in the last row.

		Spring		Elastic search		Jenkins		K-9 Mail		OwnCloud	
		v4.0	v5.0	v2.2	v6.1	v1.62	v2.89	v5.0	v5.2	v2.0	v2.7
Precision	BASE	0.34	0.40	0.46	0.31	0.29	0.30	0.28	0.25	0.34	0.32
	BASE+CRD	0.37	0.42	0.48	0.33	0.32	0.33	0.32	0.28	0.44	0.37
Recall	BASE	0.61	0.66	0.69	0.75	0.62	0.76	0.41	0.50	0.46	0.40
	BASE+CRD	0.66	0.68	0.74	0.77	0.65	0.84	0.44	0.53	0.49	0.46
Accuracy	BASE	0.60	0.71	0.78	0.71	0.75	0.66	0.56	0.47	0.53	0.49
	BASE+CRD	0.64	0.72	0.79	0.74	0.78	0.68	0.60	0.53	0.62	0.54
AUC	BASE	0.65	0.69	0.77	0.63	0.55	0.61	0.57	0.59	0.66	0.60
	BASE+CRD	0.71	0.73	0.79	0.74	0.65	0.69	0.67	0.72	0.79	0.70
F1	BASE	0.43	0.50	0.55	0.44	0.39	0.43	0.33	0.33	0.39	0.35
	BASE+CRD	0.47 (+10%)	0.52 (+4%)	0.58 (+%5)	0.46 (+%5)	0.43 (+10%)	0.47 (+9%)	0.38 (+13%)	0.37 (+11%)	0.46 (+18%)	0.41 (+16%)

In revision `2c32aa6d3a` of OwnCloud Android application, the developer uses this API for a `Fragment` class without setting any `id` which introduced a defect.

Note that, the defects described above are not diagnosed by the traditional defect prediction models (using metrics described in Table 4). Though, when our crowd-related metrics are added to the model, the bug is diagnosed.

6. Discussion

In this section we discuss the overall points about our findings. Today software development is heavily based on external packages and libraries. Although, our results show a relation between using more challenging APIs and defects, developers cannot and should not avoid using APIs with high challenge, because, leveraging libraries makes the code base smaller which increases the maintainability. Further, developers do not worry about further development and improvement of the external libraries since it is outsourced to experienced companies or communities (Visser et al., 2016). Nevertheless, as mentioned earlier, using APIs offered by libraries may introduce new defects because of reasons such as vagueness or lack of documentation (Souza et al., 2019). Thus, according to our results, to improve the quality of software, API designers should better focus on documenting their APIs by covering aspects such as special cases of the API arguments and outputs, as well as common mistakes and unexpected behaviors of the APIs they offer which trigger many discussions on Stack Overflow (Souza et al., 2019). On the other side, the API users should allocate more resources on reviewing source code files containing more APIs with high challenge.

According to Algorithm 1, one may argue that older APIs have more time to be discussed and our measures may diagnose them as more challenging. Further, newer APIs are discussed in fewer discussions, and may be considered less challenging. Thus, is there any correlation between the challenge and the age of APIs? We simply estimate the age of an API as the maximum age of the discussions (days) mentioning that API. The Spearman rank correlation between the age and the challenge of the APIs is 0.07 which suggests that the newer APIs are not necessarily less challenging than the older ones. This may be due to the fact that when an API emerges, the community discusses that API until all common challenges are covered, and after that, new questions mentioning that API are rarely asked on Stack Overflow.

As mentioned earlier our crowd-related metrics are calculated from millions of discussions and their quality descriptors. Thus, it is not possible for developers to identify more challenging APIs without a proper tool. To pave the way for using our approach and further studies on the use of crowd knowledge, we developed a plugin for IntelliJ IDEA – One of the most popular Java Integrated Development Environments (IDE) – that shows the challenge for the external APIs used in source code files. As depicted in Fig. 9, by clicking a button (1), the plugin analyzes the currently open source code file and lists the APIs along with their corresponding challenge at the bottom of the IDE (2). The developer can filter APIs with a low challenge by setting a minimum challenging score (3). To better provide an insight, by clicking each API, the plugin shows a list of related Stack Overflow questions in a new tab (4). Thus, by using this tool, developers can obtain an insight about the challenges of the APIs they use and find files containing APIs with high challenges. Therefore, they would be able to allocate more resources on reviewing them to prevent further defects. Nevertheless, the usefulness of this plugin should be investigated in a separate user study, since the focus of our study is to analyze the relation between APIs discussed in the crowd and post-release defects. This plugin and its usage manual are publicly available through our replication package which is mentioned in Section 4.2.

7. Threats to validity

In this section we discuss the threats to the validity of our study.

External Validity. Our study is based on five popular open source Java projects publicly available on GitHub. However, the results of our study may not necessarily generalize to all software systems and programming languages. Further, the H-AST we used for parsing discussions which is offered by Ponzanelli et al. (2015) is just available for discussions of Java language. For other languages, we need to implement the island grammar of those languages. Thus, more studies on other systems are needed to figure out whether our findings can be generalized. Nevertheless, we considered projects with different sizes and contexts (platform software, search engine, CI tool, and mobile application) in order to increase the generalizability of our research.

Internal Validity. In this paper, we take a step toward studying the relationship between crowd knowledge and post-release defects. To control the scope and the structure of our study we focused on APIs and introduced our metrics based on challenges of the APIs discussed on Stack Overflow. However, there exist other factors that can be considered as crowd knowledge. Reputation of developers of the project on Stack Overflow or other software developers communities such as GitHub, and the similarity of the project's code fragments to buggy code snippets on Stack Overflow, may improve our results, though require a dedicated study.

Our study on the relationship between crowd knowledge and post-release defects cannot claim causal effects. That is, increasing the explanatory or predictive power of a defect prediction

(a) Spring (b) Elastic Search (c) Jenkins

(d) K-9 Mail (e) OwnCloud

Fig. 7. The challenges of libraries for each project under study. For obtaining the challenge for each library in a project, we simply calculate the mean of $Challenge_{Pop}$ for all APIs of the library that are used in the project. Items with bigger font size and bolder color are considered more challenging.

(a) java.io (b) apache.lucene

(c) javax.servlet (d) android.widget

(e) android.content

Fig. 8. The APIs of top challenging libraries in the studied projects. Items with bigger font sizes are considered more challenging.

model using a set of metrics does not mean those metrics cause defects. It just indicates a possibility of a relationship which should be studied in depth using various approaches such as user study. Additionally, our goal was to understand the relationship between crowd-related metrics and post-release defects.

Construct validity. The Stack Overflow community is capable of generating a rich source of both questions and answers about APIs. For instance, it covers more than 87% of the all Android APIs (Wang et al., 2015). However there still exist many other APIs we could not measure their challenges because of not being discussed in the crowd. That is, there may not be any discussion about new APIs for a while or some APIs may not be popular at all to be discussed because of non-popularity.

For example, `tdunning` – a Java library which offers some advance data structures – is used in Elastic Search. However there are only 17 discussions talking about this library of which only

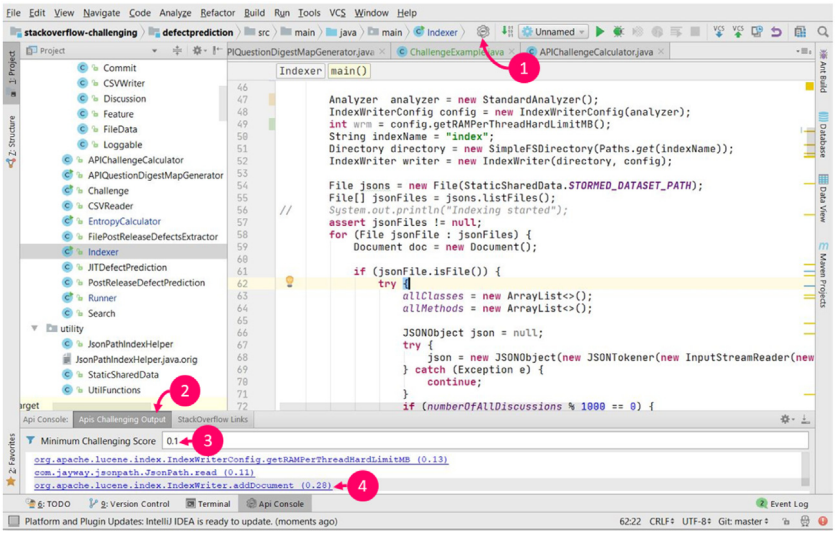

Fig. 9. A snapshot of the proposed IntelliJ IDEA plugin.

six discussions have positive up votes. Moreover, most of these discussions have not properly mentioned APIs of this library. As a result, our approach could not extract any API from this library. This issue may be due to two main reasons; (1) not having any serious challenge, or (2) just not being a popular library on Stack Overflow because of not being popular among developers or being new. The latter reason threats the validity of our study since our approach heavily relies on Stack Overflow. We investigated the popularity of this library and observed that it has more than 1.1*K* stars on GitHub which suggests that this library is not an obscure software package. Hence, the first reason, i.e., not having serious challenge, may be the cause of being non-popular on the Stack overflow. `atinject` – a dependency injection library for Java – used in Spring project, is another example. There are only nine discussions about this library on Stack Overflow. This library has just 93 stars on Google Code, which is considered low popularity. Our observation suggests that due to the obscurity of this library, the community of Stack Overflow does not discuss its APIs, though, `atinject` may have APIs with a high challenge. Nevertheless, we tried to mitigate this issue in our formulation of the challenge for less popular projects by considering entropy and applying a log transformation on the frequency of each quartile. Hence, APIs that are not as frequent as prevalent APIs are allowed to be noticed since they usually have lower entropy and also the log function alleviates their low frequencies. `mindrot.jBCrypt` – a Java password hashing library – is an example of such cases. All of the APIs of this library are totally mentioned in just 105 discussions. However, APIs such as `hashpw` (11 mentions) and `checkpw` (8 mentions) gain the $Challenge_{Pop}$ of 0.37 and 0.32 respectively which is larger than the mean (0.23).

To overcome the problem of not sufficiently discussed APIs in Stack Overflow (like `tdunning`), more sources of crowd knowledge are required. For example, we can employ GitHub issues or mine Slack public Q&A chats in which developers discuss about APIs' obstacles (Chatterjee et al., 2019). Nevertheless, as Stack Overflow is rapidly growing, the chance of covering more APIs is increasing as well. Nevertheless, studies about identifying challenging APIs and listing challenges of them are still underway (Uddin and Khomh, 2019; Ahasanuzzaman et al., 2018) and refining the measurement of API challenges is aimed in our future studies.

We choose well-known traditional baseline variables with respect to adding explanatory and predictive power to our model. Our models can be further improved using other variables. However, according to our research questions, we do not aim to obtain the best deviance explained and predictive power. Thus, adding other variables to improve the results should not impact our conclusions.

In this study, we do not take API *versions* into account. However, different versions of the APIs may have different challenges. For example, many of the challenges related to an API may be mitigated in newer versions. On the other hand, some challenges may remain across versions. Taking versioning into account would add major complications to the problem of using crowd knowledge which demands a separate study. Thus, we plan to involve versioning in our future work.

To extract buggy commits, we used the online tool offered by Rosen et al. (2015). This tool assumes that developers mention the type of change in the commit message. For example, when they fix a bug, the commit message may contain terms like *fixed* or *fix*, *bug #543* (in which 543 is the id of the reported bug). This data may contain noises since many developers may not include enough information in the commit messages (Fan et al., 2019). Although this approach is employed by various recent studies in the defect prediction area (Herzig et al., 2013), we will try to leverage other approaches in future.

Finally, we only focused on post-release defects as a software quality indicator. However, other indicators quantifying software quality can be employed.

8. Related work

In this section, we describe related work with respect to the use of crowd knowledge in software engineering and defect prediction.

8.1. Use of crowd knowledge in software engineering

In recent years, many studies have employed crowd knowledge obtained from Q&A websites to solve software engineering problems in diverse areas (Mao et al., 2017). Some studies leverage this kind of websites to automatically generate (Parnin et al., 2012) or augment documentation for the APIs (Treude and Robillard, 2016; Uddin et al., 2019). Some others, analyze and categorize Stack Overflow discussions to help the community better understand programming issues, topics, and trends in various areas such as web development (Barua et al., 2014),

mobile applications (Rosen and Shihab, 2016), and question types across different programming languages (Allamanis and Sutton, 2013). Tian et al. conducted an exploratory study of developers' perception of architecture smells by analyzing related discussions on Stack Overflow (Tian et al., 2019).

More related to our study, Wang and Godfrey (2013) mine API-related posts regarding iOS and Android. They state that API-related posts are primarily about API usage obstacles. Consequently, they extracted API usage challenges which may be used by the API designers to solve the complexity or other problems of their APIs. They use topic modeling to obtain common scenarios behind each challenge.

Additionally, several methods and tools have been proposed to help developers with coding and debugging (Mao et al., 2017; Rahman et al., 2014). Campos et al. (2016b) propose an approach which recommends a ranked list of question–answer pairs from Stack Overflow based on the developer's current development task. Campos et al. (2016a) come up with a tool to find fixes for API-usage-related bugs, which is based on matching code snippets being debugged against related snippets on Stack Overflow. There are also other studies offering recommender systems using Stack Overflow (Ponzanelli et al., 2014a; Cordeiro et al., 2012; Rahman et al., 2016). Usually, these studies leverage APIs being used by the developer as the key element to characterize the current context of their programming status. Chen and Kim (2015) leverage the crowd knowledge to help developers debug their code fragments. They utilize code clone detection to find similar code snippets, available in the question part of discussions, to the developer's code. At last, they recommend code snippets which may be buggy along with crowd-suggested solutions for developers.

However, none of the aforementioned studies investigated the effect of this knowledge on defect prediction models, e.g., explaining and predicting software defects. In our paper, we propose metrics based on crowd knowledge to better explain and predict software defects.

8.2. Software defect prediction

There exists a large body of research trying to model defects using two main categories of software metrics: product and process (Shang et al., 2015). Product metrics are defined based on static characteristics of software which resemble the complexity of source code. LOC (Akiyama, 1971), code dependency metrics (Zimmermann and Nagappan, 2008), object-oriented metrics (Chidamber and Kemerer, 1994) are treated as examples of this category. However, LOC has been shown to typically be the best among metrics in this category (Herraiz and Hassan, 2010).

On the other hand, the process metrics leverage historical project characteristics. This category includes metrics such as prior changes which is also known as *code churn* (Nagappan and Ball, 2005), entropy of changes or number of prior defects (Hassan, 2009), Inter-developer interaction, social network metrics (Meneely et al., 2008) and metrics related to developers' expertise obtained from prior commits and defects (Mockus and Weiss, 2000; Rahman and Devanbu, 2011; Posnett et al., 2013), branching activities and strategies (Shihab et al., 2012), are other examples of the process metrics.

Prior studies also conduct various analyses over existing metrics. For example, Shihab et al. (2010) observe that the PRE and LOC metrics have higher correlation with post-release defects than many change based metrics. They also observe that the LOC metric is highly correlated with the majority of the code metrics. They state that the high correlation values can lead to multicollinearity problems if these independent variables were combined in a single logistic regression model.

Further, researchers proposed metrics quantifying other aspects of software with the aim of improving software defect prediction. Jin et al. (2012) focus on performance bugs and cover a wide spectrum of characteristics of these bugs. Shang et al. (2015) studied the relationship between logging characteristics and the code quality. They introduce a set of log related metrics and statistically investigate the effect of these metrics on explaining post-release defects. Similar to our approach, they propose *log density* metric, i.e., the number of logging statements in each file, and *average number of log lines added or deleted in a commit* which inspired us to propose *CAC* and *CAD* metrics. They measure how their offered metrics can improve explanatory power of the prediction model. de Pádua and Shang (2018) investigate the relationship between exception handling practices and post-release defects by offering new metrics and measuring the improvement over explanatory power of the overall model. Chen et al. (2017) use a statistical topic modeling approach to approximate software concerns as topics and define various topic-based metrics to help explain the file defect-proneness. *Number of Defect-prone Topics (NDT)* and *Topic Membership (TM)* of a file are two examples of their metrics. Using D^2 measure they show that, their metrics provide additional explanatory power for software quality over existing structural and historical metrics.

Linares-Vásquez et al. (2013a) statistically reveal that the instability (change-proneness) and fault-proneness of APIs used by Android applications threats the success of those applications, according to app stores (e.g., Google Play) user ratings. They used the total number of changes and bugs fixed in source code repositories of the used APIs as the independent variable.

In this paper, we proposed *CAD* and *CAC* as two crowd-related metrics which are based on the challenge of the APIs discussed in the crowd. *CAD* is a product metric that measures how much a file is using external APIs which are discussed in high quality posts on Stack Overflow in a single snapshot. *CAC* measures how much such APIs are churned in a file over time (process metric). Our metrics are different from existing defect prediction metrics in the way that they are based on latent crowd knowledge available on Q&A websites. Thus, to the best of our knowledge, this paper is the first attempt to establish an empirical link between the crowd knowledge available in Q&A websites and explaining and predicting software defects.

9. Conclusions and future work

Q&A websites such as Stack Overflow has turned to be an unavoidable tool for developers to ask and find solutions to their questions, issues and errors. However, the effect of the crowd knowledge obtainable from this huge source of information on explaining defects has never been empirically studied before. In this paper, we attempted to model this crowd knowledge by proposing a set of metrics and statistically investigated the relation between these crowd-related metrics and software quality.

Our findings show that; (1) there are a few underlying factors which can explain interrelationship among quality descriptors of Stack Overflow such as up votes, favorite count, views count and so on; (2) our crowd-related metrics have positive correlation with post-release defects; (3) our crowd-related metrics can complement traditional product and process metrics in explaining post-release defects; (4) further, our metrics have a positive effect on the model generated by traditional metrics; and finally, (5) crowd-related metrics can improve the predictive power of traditional models. Nevertheless, using challenging APIs is inevitable in today's software systems and developers cannot avoid using them. Instead, we suggest that they should focus more on files with more challenging APIs to prevent further defects.

In future, we will do more filtering on discussions to prevent noisy items. Further, we will cover quality indicators other than post-release defects to evaluate our approach. We will also improve our formulation of API challenge in order to better characterize and formalize the challenges of the APIs discussed in the crowd. We plan to focus on the inter-relation between APIs which is a valuable source of knowledge. That is, when a set of APIs frequently are being discussed together, there may be challenges that are due to the inter-relation between those APIs. We will strive toward involving other factors other than the API challenge from Q&A websites as well as involving more sources of crowd knowledge such as GitHub and Slack. Taking into account versions of the APIs is another improvement we plan to make. At last, we plan to replicate this study on more case studies to further assess the generality of our results.

CRediT authorship contribution statement

Hamed Tahmooresi: Conceptualization, Methodology, Software, Validation, Formal analysis, Writing - original draft. **Abbas Heydarnoori:** Project administration, Supervision, Methodology, Writing - review & editing. **Reza Nadri:** Software, Validation, Investigation, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Afroozeh, A., Bach, J.-C., Van den Brand, M., Johnstone, A., Manders, M., Moreau, P.-E., Scott, E., 2012. Island grammar-based parsing using gll and tom. In: International Conference on Software Language Engineering. Springer, pp. 224–243.

Ahasanuzzaman, M., Asaduzzaman, M., Roy, C.K., Schneider, K.A., 2018. Classifying Stack Overflow posts on API issues. In: 25th International Conference on Software Analysis, Evolution and Reengineering. IEEE, pp. 244–254.

Akiyama, F., 1971. An example of software system debugging. In: IFIP Congress (1), Vol. 71. pp. 353–359.

Allamanis, M., Sutton, C., 2013. Why, when, and what: Analyzing Stack Overflow questions by topic, type, and code. In: 10th Working Conference on Mining Software Repositories. IEEE, pp. 53–56.

Anon, 2018. Stack Overflow developer survey. URL https://insights.stackoverflow.com/survey/2018/.

Anon, 2019. Stack exchange data explorer. URL https://data.stackexchange.com.

Bajaj, K., Pattabiraman, K., Mesbah, A., 2014. Mining questions asked by web developers. In: 11th Working Conference on Mining Software Repositories. ACM, pp. 112–121.

Barua, A., Thomas, S.W., Hassan, A.E., 2014. What are developers talking about? An analysis of topics and trends in Stack Overflow. Empir. Softw. Eng. 19 (3), 619–654.

Bird, C., Nagappan, N., Gall, H., Murphy, B., Devanbu, P., 2009. Putting it all together: Using socio-technical networks to predict failures. In: 20th International Symposium on Software Reliability Engineering. IEEE, pp. 109–119.

Bird, C., Nagappan, N., Murphy, B., Gall, H., Devanbu, P., 2011. Don't touch my code!: Examining the effects of ownership on software quality. In: 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM, pp. 4–14.

Bishop, C.M., 2006. Pattern Recognition and Machine Learning. springer.

Borg, M., Svensson, O., Berg, K., Hansson, D., 2019. SZZ unleashed: an open implementation of the SZZ algorithm-featuring example usage in a study of just-in-time bug prediction for the Jenkins project. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation. pp. 7–12.

Campos, E.C., Monperrus, M., Maia, M.A., 2016a. Searching Stack Overflow for API-usage-related bug fixes using snippet-based queries. In: 26th Annual International Conference on Computer Science and Software Engineering. IBM Corp., pp. 232–242.

Campos, E.C., de Souza, L.B., Maia, M.d.A., 2016b. Searching crowd knowledge to recommend solutions for API usage tasks. J. Softw.: Evol. Process 28 (10), 863–892.

Cattell, R.B., 1966. The scree test for the number of factors. Multivariate Behav. Res. 1 (2), 245–276.

Cerny, B.A., Kaiser, H.F., 1977. A study of a measure of sampling adequacy for factor-analytic correlation matrices. Multivariate Behav. Res. 12 (1), 43–47.

Chatterjee, P., Damevski, K., Pollock, L., Augustine, V., Kraft, N.A., 2019. Exploratory study of slack q&a chats as a mining source for software engineering tools. In: 16th Working Conference on Mining Software Repositories. IEEE, pp. 490–501.

Chen, F., Kim, S., 2015. Crowd debugging. In: 10th Joint Meeting on Foundations of Software Engineering. ACM, pp. 320–332.

Chen, T.-H., Shang, W., Nagappan, M., Hassan, A.E., Thomas, S.W., 2017. Topic-based software defect explanation. J. Syst. Softw. 129, 79–106.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493.

Cordeiro, J., Antunes, B., Gomes, P., 2012. Context-based recommendation to support problem solving in software development. In: 3rd International Workshop on Recommendation Systems for Software Engineering. IEEE, pp. 85–89.

Dagenais, B., Robillard, M.P., 2012. Recovering traceability links between an API and its learning resources. In: 34th International Conference on Software Engineering. IEEE Press, pp. 47–57.

d'Agostino, R.B., 1971. An omnibus test of normality for moderate and large size samples. Biometrika 58 (2), 341–348.

D'Ambros, M., Lanza, M., Robbes, R., 2010. An extensive comparison of bug prediction approaches. In: 7th Working Conference on Mining Software Repositories. IEEE, pp. 31–41.

DeCoster, J., 1998. Overview of Factor Analysis. Tuscaloosa, AL.

Fabrigar, L.R., Wegener, D.T., MacCallum, R.C., Strahan, E.J., 1999. Evaluating the use of exploratory factor analysis in psychological research. Psychol. Methods 4 (3), 272.

Fan, Y., Xia, X., da Costa, D.A., Lo, D., Hassan, A.E., Li, S., 2019. The impact of changes mislabeled by SZZ on just-in-time defect prediction. IEEE Trans. Softw. Eng. in press.

Fu, H., Wu, S., Oh, S., 2015. Evaluating answer quality across knowledge domains: Using textual and non-textual features in social Q&A. In: 78th ASIS&T Annual Meeting: Information Science with Impact: Research in and for the Community. American Society for Information Science, p. 88.

Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., Ubayashi, N., 2014. An empirical study of just-in-time defect prediction using cross-project models. In: 11th Working Conference on Mining Software Repositories. ACM, pp. 172–181.

Guisan, A., Zimmermann, N.E., 2000. Predictive habitat distribution models in ecology. Ecol. Model. 135 (2–3), 147–186.

Gyimothy, T., Ferenc, R., Siket, I., 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Softw. Eng. 31 (10), 897–910.

Hassan, A.E., 2009. Predicting faults using the complexity of code changes. In: 31st International Conference on Software Engineering. IEEE Computer Society, pp. 78–88.

Herraiz, I., Hassan, A.E., 2010. Beyond lines of code: Do we need more complexity metrics. In: Making Software: What Really Works, and Why We Believe It. O'Reilly Media, pp. 125–141.

Herzig, K., Just, S., Zeller, A., 2013. It's not a bug, it's a feature: How misclassification impacts bug prediction. In: International Conference on Software Engineering. IEEE Press, pp. 392–401.

Jiarpakdee, J., Tantithamthavorn, C., Hassan, A.E., 2019. The impact of correlated metrics on the interpretation of defect models. IEEE Trans. Softw. Eng. in press.

Jin, G., Song, L., Shi, X., Scherpelz, J., Lu, S., 2012. Understanding and detecting real-world performance bugs. ACM SIGPLAN Not. 47 (6), 77–88.

Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2013. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. 39 (6), 757–773.

Khomh, F., Chan, B., Zou, Y., Sinha, A., Dietz, D., 2011. Predicting post-release defects using pre-release field testing results. In: 27th IEEE International Conference on Software Maintenance. IEEE, pp. 253–262.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., Poshyvanyk, D., 2013a. API change and fault proneness: A threat to the success of android apps. In: 9th Joint Meeting on Foundations of Software Engineering. ACM, pp. 477–487.

Linares-Vásquez, M., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., 2014. How do API changes trigger Stack Overflow discussions? A study on the android SDK. In: 22nd International Conference on Program Comprehension. ACM, pp. 83–94.

Linares-Vásquez, M., Dit, B., Poshyvanyk, D., 2013b. An exploratory analysis of mobile development issues using stack overflow. In: 10th Working Conference on Mining Software Repositories. IEEE, pp. 93–96.

Mamykina, L., Manoim, B., Mittal, M., Hripcsak, G., Hartmann, B., 2011. Design lessons from the fastest Q&A site in the west. In: SIGCHI Conference on Human Factors in Computing Systems. ACM, pp. 2857–2866.

Mao, K., Capra, L., Harman, M., Jia, Y., 2017. A survey of the use of crowdsourcing in software engineering. J. Syst. Softw. 126, 57–84.
Matei, S.A., Jabal, A.A., Bertino, E., 2018. Social-collaborative determinants of content quality in online knowledge production systems: Comparing Wikipedia and Stack Overflow. Soc. Netw. Anal. Min. 8 (1), 36.
Meneely, A., Williams, L., Snipes, W., Osborne, J., 2008. Predicting failures with developer networks and social network analysis. In: 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 13–23.
Mnkandla, E., Mpofu, B., 2016. Software defect prediction using process metrics elasticsearch engine case study. In: 2016 International Conference on Advances in Computing and Communication Engineering (ICACCE). IEEE, pp. 254–260.
Mockus, A., 2010. Organizational volatility and its effects on software defects. In: 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 117–126.
Mockus, A., Weiss, D.M., 2000. Predicting risk of software changes. Bell Labs Tech. J. 5 (2), 169–180.
Mojica, I.J., Adams, B., Nagappan, M., Dienst, S., Berger, T., Hassan, A.E., 2014. A large-scale empirical study on software reuse in mobile apps. IEEE Softw. 31 (2), 78–86.
Moonen, L., 2001. Generating robust parsers using island grammars. In: 8th Working Conference on Reverse Engineering. IEEE, pp. 13–22.
Moser, R., Pedrycz, W., Succi, G., 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: 30th International Conference on Software Engineering. ACM, pp. 181–190.
Nagappan, N., Ball, T., 2005. Use of relative code churn measures to predict system defect density. In: 27th International Conference on Software Engineering. ACM, pp. 284–292.
de Pádua, G.B., Shang, W., 2018. Studying the relationship between exception handling practices and post-release defects. In: 15th Working Conference on Mining Software Repositories. ACM, pp. 564–575.
Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R., 2017. Toward a smell-aware bug prediction model. IEEE Trans. Softw. Eng. 45 (2), 194–218.
Parnin, C., Treude, C., Grammel, L., Storey, M.-A., 2012. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. Tech. rep., Georgia Institute of Technology.
Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., Lanza, M., 2014a. Mining Stack Overflow to turn the IDE into a self-confident programming prompter. In: 11th Working Conference on Mining Software Repositories. ACM, pp. 102–111.
Ponzanelli, L., Mocci, A., Bacchelli, A., Lanza, M., Fullerton, D., 2014b. Improving low quality Stack Overflow post detection. In: International Conference on Software Maintenance and Evolution. IEEE, pp. 541–544.
Ponzanelli, L., Mocci, A., Lanza, M., 2015. Stormed: Stack Overflow ready made data. In: 12th Working Conference on Mining Software Repositories. IEEE, pp. 474–477.
Posnett, D., D'Souza, R., Devanbu, P., Filkov, V., 2013. Dual ecological measures of focus in software development. In: International Conference on Software Engineering. IEEE Press, pp. 452–461.
Rahman, F., Devanbu, P., 2011. Ownership, experience and defects: A fine-grained study of authorship. In: 33rd International Conference on Software Engineering. ACM, pp. 491–500.
Rahman, M.M., Roy, C.K., Lo, D., 2016. Rack: Automatic API recommendation using crowdsourced knowledge. In: 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, Vol. 1. IEEE, pp. 349–359.
Rahman, M.M., Yeasmin, S., Roy, C.K., 2014. Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions. In: Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering. IEEE, pp. 194–203.
Robillard, M.P., Deline, R., 2011. A field study of API learning obstacles. Empir. Softw. Eng. 16 (6), 703–732.
Rosen, C., Grawi, B., Shihab, E., 2015. Commit guru: Analytics and risk prediction of software commits. In: 10th Joint Meeting on Foundations of Software Engineering. ACM, pp. 966–969.
Rosen, C., Shihab, E., 2016. What are mobile developers asking about? A large scale study using stack overflow. Empir. Softw. Eng. 21 (3), 1192–1223.
Shang, W., Nagappan, M., Hassan, A.E., 2015. Studying the relationship between logging characteristics and the code quality of platform software. Empir. Softw. Eng. 20 (1), 1–27.
Shannon, C.E., 1948. A mathematical theory of communication. Bell Syst. Tech. J. 27 (3), 379–423.
Shihab, E., Bird, C., Zimmermann, T., 2012. The effect of branching strategies on software quality. In: ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ACM, pp. 301–310.
Shihab, E., Jiang, Z.M., Ibrahim, W.M., Adams, B., Hassan, A.E., 2010. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In: ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 1–10.
Shihab, E., Mockus, A., Kamei, Y., Adams, B., Hassan, A.E., 2011. High-impact defects: A study of breakage and surprise defects. In: 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM, pp. 300–310.
Souza, L.B., Campos, E.C., Madeiral, F., Paixão, K., Rocha, A.M., de Almeida Maia, M., 2019. Bootstrapping cookbooks for APIs from crowd knowledge on Stack Overflow. Inf. Softw. Technol. 111, 37–49.
de Souza, L.B., Campos, E.C., Maia, M.d.A., 2014. Ranking crowd knowledge to assist software development. In: 22nd International Conference on Program Comprehension. ACM, pp. 72–82.
Subramanian, S., Holmes, R., 2013. Making sense of online code snippets. In: 10th Working Conference on Mining Software Repositories. IEEE Press, pp. 85–88.
Taba, S.E.S., Khomh, F., Zou, Y., Hassan, A.E., Nagappan, M., 2013. Predicting bugs using antipatterns. In: IEEE International Conference on Software Maintenance. IEEE, pp. 270–279.
Tian, F., Liang, P., Babar, M.A., 2019. How developers discuss architecture smells? An exploratory study on stack overflow. In: IEEE International Conference on Software Architecture. IEEE, pp. 91–100.
Treude, C., Robillard, M.P., 2016. Augmenting API documentation with insights from Stack Overflow. In: 38th IEEE/ACM International Conference on Software Engineering. IEEE, pp. 392–403.
Uddin, G., Khomh, F., 2019. Automatic mining of opinions expressed about APIs in Stack Overflow. IEEE Transactions on Software Engineering.
Uddin, G., Khomh, F., Roy, C.K., 2019. Towards crowd-sourced API documentation. In: 41st International Conference on Software Engineering: Companion Proceedings. IEEE Press, pp. 310–311.
Uddin, G., Robillard, M.P., 2017. Resolving API mentions in informal documents. arXiv preprint arXiv:1709.02396.
Visser, J., Rigal, S., Wijnholds, G., van Eck, P., van der Leek, R., 2016. Building Maintainable Software, C# Edition: Ten Guidelines for Future-Proof Code. " O'Reilly Media, Inc.".
Wang, W., Godfrey, M.W., 2013. Detecting API usage obstacles: A study of ios and android developer questions. In: 10th Working Conference on Mining Software Repositories. IEEE Press, pp. 61–64.
Wang, W., Malik, H., Godfrey, M.W., 2015. Recommending posts concerning API issues in developer Q&A sites. In: 12th Working Conference on Mining Software Repositories. IEEE, pp. 224–234.
Xia, X., Shihab, E., Kamei, Y., Lo, D., Wang, X., 2016. Predicting crashing releases of mobile applications. In: 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ACM, p. 29.
Zar, J.H., 1972. Significance testing of the spearman rank correlation coefficient. J. Amer. Statist. Assoc. 67 (339), 578–580.
Zimmermann, T., Nagappan, N., 2008. Predicting defects using network analysis on dependency graphs. In: 30th International Conference on Software Engineering. ACM, pp. 531–540.
Zimmermann, T., Premraj, R., Zeller, A., 2007. Predicting defects for eclipse. In: 3rd International Workshop on Predictor Models in Software Engineering. IEEE.

Hamed Tahmooresi is a Ph.D. student at the Sharif University of Technology. His research interests include software engineering, software architecture and design, and mining software repositories. Contact him at tahmooresi@ce.sharif.edu

Abbas Heydarnoori is an assistant professor at the Sharif University of Technology. Before, he was a post-doctoral fellow at the University of Lugano, Switzerland. Abbas holds a Ph.D. from the University of Waterloo, Canada. His research interests focus on software evolution, mining software repositories, and recommendation systems in software engineering. Contact him at heydarnoori@sharif.edu

Reza Nadri is currently a Master's student and research assistant at the University of Waterloo, Canada. Before, he got his Bachelor's degree from the Sharif University of Technology. His research interests include mining software repositories, software analytics, recommendation systems in software engineering, and social aspects of software engineering. Contact him at rnadri@uwaterloo.ca