Noname manuscript No.
(will be inserted by the editor)

An Extended Study On Classifying Stack Overflow Posts on API Issues

Md Ahasanuzzaman · Muhammad Asaduzzaman · Chanchal K. Roy · Kevin A. Schneider

Received: date / Accepted: date

Abstract The design and maintenance of APIs (Application Programming Interfaces) are complex tasks due to the constantly changing requirements of their users. Despite the efforts of their designers, APIs may suffer from a number of issues (such as incomplete or erroneous documentation, poor performance, and backward incompatibility). To maintain a healthy client base, API designers must learn these issues to fix them. Question answering sites, such as Stack Overflow (SO), have become a popular place for discussing API issues. These posts about API issues are invaluable to API designers, not only because they can help to learn more about the problem but also because they can facilitate learning the requirements of API users. However, the unstructured nature of posts and the abundance of non-issue posts make the task of detecting SO posts concerning API issues difficult and challenging.

In this paper, we first develop a supervised learning approach using a Conditional Random Field (CRF), a statistical modeling method, to identify API issue-related sentences. We use the above information together with different features collected from posts, the experience of users, readability metrics and centrality measures of collaboration network to build a technique, called *CAPS*, that can classify SO posts concerning API issues. In total, we consider 34 features along eight different dimensions.

Evaluation of *CAPS* using carefully curated SO posts on three popular API types reveals that the technique outperforms all three baseline approaches we consider in this study. We then conduct studies to find important features and also evaluate the performance of the CRF-based technique for classifying

Md Ahasanuzzaman · Muhammad Asaduzzaman
Software Analysis and Intelligence Lab (SAIL)
Queens University, Kingston, Ontario, Canada
E-mail: : {md.ahasanuzzaman, muhammad.asaduzzaman}@queensu.ca

Chanchal K. Roy · Kevin A. Schneider
University of Saskatchewan, Canada
E-mail: :{chanchal.roy, kevin.schneider}@usask.ca

issue sentences. Comparison with two other baseline approaches shows that the technique has high potential. We also test the generalizability of *CAPS* results, evaluate the effectiveness of different classifiers, and identify the impact of different feature sets.

1 Introduction

Developers depend on frameworks and libraries for effective delivery of software in a timely manner. This is supported through Application Programming Interfaces (APIs) of those frameworks and libraries that provide access to the implemented functionality. For example, the Java Software Development Kit comes with thousands of components that the developers can reuse in their projects. This saves both development time and effort [52]. API designers must work hard to make their APIs accessible to its users. This not only ensures the business success of API providers/designers but also enables them to maintain a healthy satisfied user-base. Towards this goal, API designers need to provide development tools, documentation, and tutorials to support working with their APIs. Despite all these efforts, APIs may suffer from several issues. These include but are not limited to documentation error (including outdated or incomplete documentation), poor memory management, breaking changes that lead to backward incompatibility, and incompatibility of the APIs with underlying operating systems or other external libraries [68]. All these may lead to incorrect use of APIs, introduce bugs and security problems. The rapid changes in APIs do not give the designers much time to validate various changes and thus create confusion among its users [37]. It may also introduce faults in designing APIs, introduce usability issues, and ultimately leads to incorrect behavior in applications using those APIs. API designers need to learn about these issues of using APIs to fix the problems and to find effective ways of informing developers about various API changes.

API related issues can be learned by mining bug repositories [1], newsgroups [32] and emails of developers [6]. However, they are not the only places to look for API issues. Nowadays, developers rely on online forums and question-answering sites to discuss issues of APIs, ask questions and seek help from others. While many question-answering sites (such as Yahoo Answers[1] and Quora[2]) allow users to ask questions on any topics they are interested in, Stack Overflow (SO) particularly focuses on programming related questions. Thus, API issues discussed in SO are of great interest to API designers. However, extracting SO posts concerning API issues is a non-trivial task. This is mostly due to the presence of millions of questions, many of which are not related to API issues, and also due to the unstructured nature of the posts. Keyword searching is not an efficient solution to the problem because of the presence of a large number of *how-to* and *newbie* questions [60] that introduce a lot of noise. This motivates us to investigate the problem further.

[1] https://answers.yahoo.com/
[2] https://www.quora.com/

We model the problem of identifying posts discussing API issues as a binary classification problem. Our goal is to separate these issue-related posts from the others. Towards this goal, we develop a supervised learning technique using Conditional Random Field (CRF) [58], that identifies API issue-sentences in a post. We not only collect features from the output of CRF but also combine that with different features of posts, user experience, and centrality measures to build a logistic regression model, called *CAPS*. In total, we consider 34 features along eight different dimensions. These include features collected from title, body, CRF, answer, the experience of questioners and answerers, readability metrics, centrality measures of collaboration network and additional features collected from the question part of posts). Table 8 summarizes collected features of our model.

We perform a study to determine the most important features. For all three of our datasets, issueSentenceCount, bodyLength, and questionerMedianUpVote features appear as the top-3 important features. We then conduct a study to evaluate the effectiveness of our CRF-based issue sentence classification technique. Evaluation using the largest of our dataset and two other baseline approaches shows that the technique has high potential. Finally, we conduct studies to test the generalizability of *CAPS* results, evaluate the effectiveness of our logistic regression classifier against four other classification algorithms and identify the impact of different sets of features.

To the best of our knowledge, the study most relevant to ours is that of Wang et al. [65]. They develop a mechanism to distill and rank SO posts that are likely to concern API related issues. They select those posts which are asked by the expert users as the candidate issue-related posts. While the technique is useful, it suffers from the problem of missing API issue-related posts that are not asked by experts. During our manual analysis, we found examples of several API issue-related questions that were asked by SO users with low reputation. For example, a low reputed user posted a question[3] in August 2014 that is related to the unexpected behavior in *JUnit* API. After six months, this was opened as a potential issue in the *JUnit* issue tracker labeled as *bug*[4].

Thus, our paper makes the following contributions.

(1) A supervised approach using Conditional Random Field (CRF) that can be used to identify API issue-related sentences in a post.
(2) A classifier that is created by combining the output of a CFR-based supervised learning technique, a diverse set of features from SO posts and the experience of SO users.
(3) An evaluation of the classifier against three other baseline approaches that consider different sources of information.
(4) An empirical study to understand which features are more important for differentiating API issue posts from the non-issue ones.

[3] http://stackoverflow.com/questions/25436505/
[4] https://github.com/junit-team/junit4/issues/1083

(5) An evaluation of our CRF-based supervised learning approach for classifying API issue-related sentences.
(6) A set of studies to understand different aspects of our proposed technique.

The remainder of the paper is organized as follows. Section 2 summarizes previous work related to our study. Section 3 provides the background of our work. We characterize SO posts concerning API issues in Section 4. We describe our proposed technique in Section 5. Section 6 presents the evaluation results. We identify which features are most important for the classification task in Section 7 and Section 8 presents evaluation results for the CRF-based technique that classifies API issue-related sentences. Key issues related to our study are discussed in Section 9. Section 10 summarizes threats to the validity of our work and Section 11 concludes the paper.

2 Related Work

Stack Overflow: A number of studies have been performed to characterize different facets of Stack Overflow. This includes question quality analysis [10], modeling difficulties of questions [31], low-quality post detection [49], topic distribution [11], patterns of asking and answering questions [60] and the personality trait of users [12]. To facilitate developers, a number of recommendation systems are developed using SO data. For example, Bacchelli et al. [7] integrated crowd knowledge in the IDE by developing an Eclipse plugin, called *Seahawk*, that links relevant discussions to the source code. Ponzanelli et al. developed an Eclipse plugin, called *Prompter*, that can automatically retrieve relevant SO discussions by giving a context in the IDE [48]. Asaduzzaman et al. [5] conducted a qualitative study to categorize the unanswered questions. Correa and Sureka conducted an experimental study to analyze and predict the closed questions of SO [21]. In another study, they characterized the deleted questions in SO and build a predictive model to detect deleted questions at their creation time [22]. Chen et al. developed a technique to identify analogical libraries [18]. The technique combines word embedding with relational and categorical knowledge mined from tags of SO questions and tag wikis of those tags. However, none focuses on the classification of API issue posts in SO. A study on SO addressed the detection of user issues and request types [55]. Their primary goal was to categorize the sentences in anomaly, how to, property and explanation categories using discourse analysis. While they focus on user issues, we focus on API issues in our work.

API Analysis: API learning difficulties and other issues have been investigated in many studies and the prime reasons are problematic features, API evolution and learning obstacles. Robillard [52] conducted a study of the obstacles faced by Microsoft developers when learning how to use APIs. Robillard and Deline [53] identified that inadequate API documentation and API structure are the top two obstacles in using APIs. Robbes et al. [51] conducted an empirical study on the actual incidence of the API changes and API deprecations, causing a ripple effect in practice. The study shows that deprecation

messages are not always helpful because of the absence of the guidelines and unclear instructions. Ho and Li [32] analyzed 172 discussions collected from a forum and identified a set of API learning obstacles. Zibran et al. [68] identified 22 factors as the API usability issues. Wang and Godfrey [64] analyzed Android and IOS developer questions on SO to detect API usage obstacles. However, the objectives of these studies are different from ours. While they focus on problems that cause API issues, our study focuses on detecting SO posts concerning API issues.

Wang et al. [65] developed a methodology to recommend API design-related issues combining expert identification, topic mining and selection of late answered questions. However, their methodology only considers questions which are answered late and submitted by the expert users having more participation in SO. While the technique is useful, it may miss API issue-related posts that are not asked by expert users.

Uddin and Khomh developed an API review summarization technique leveraging Stack Overflow posts, called Opiner [61]. Given a set of sentences expressing opinions about an API the technique generates different kinds of summaries of those reviews. A statistical summary of an API represents a rating, sentiment trends and other APIs that are reviewed in the same post. Aspect-based summaries provide opinions regarding specific attributes of an API. Opiner also adopted off-the-shelf algorithms to generate extractive, abstractive, contrastive, and topic-based summaries of API reviews. However, the goal of Opiner is different than ours. While Opiner focuses on generating summaries from API reviews, containing both positive and negative opinionated sentences, *CAPS* focus on separating posts concerning API issues. Questions that ask how to solve a task using an API or collect opinions from other developers can contain several opinionated sentences about that API. However, *CAPS* removes how-to questions or questions asking opinions from developers to identify only those posts concerning API issues.

Conditional Random Fields: Conditional Random Fields (CRFs) [58] have been used in many natural language applications including parts-of-speech tagging and entity linking [56]. CRFs have also been used in extracting contexts and answers from online forums [23]. For example, Wang et al. [63] proposed a probabilistic model in the CRFs framework to predict the replying structure for a threaded online discussion. Raghavan et al. [50] extracted problem and resolution information from online forum discussions by formulating the problem as a sequence labeling task and proposed a method using CRFs. Instead of considering online forums, we consider the question answering site Stack Overflow in our study. The problem we address in this paper is also different from their studies.

Sentiment Analysis: Techniques related to sentiment analysis are also related to our study. The term sentiment analysis refers to the study of classifying subjectivity (neutral or emotionally loaded) and polarity (positive, negative or neutral) of a given text at the document or sentence level. Sentiment analysis techniques have been adopted in software engineering research to solve various problems and existing studies can be divided into three different categories.

The first category focuses on understanding the impact of sentiment on software development activities. This includes but not limited to determining the relation between emotions and activity of developers in the open source Gentoo project [26], determining the impact of emotions expressed on commit comments [29], understand the relation between emotions, sentiment and politeness [44], analyzing emotion in commit logs [57], and security-related discussions in GitHub [47].

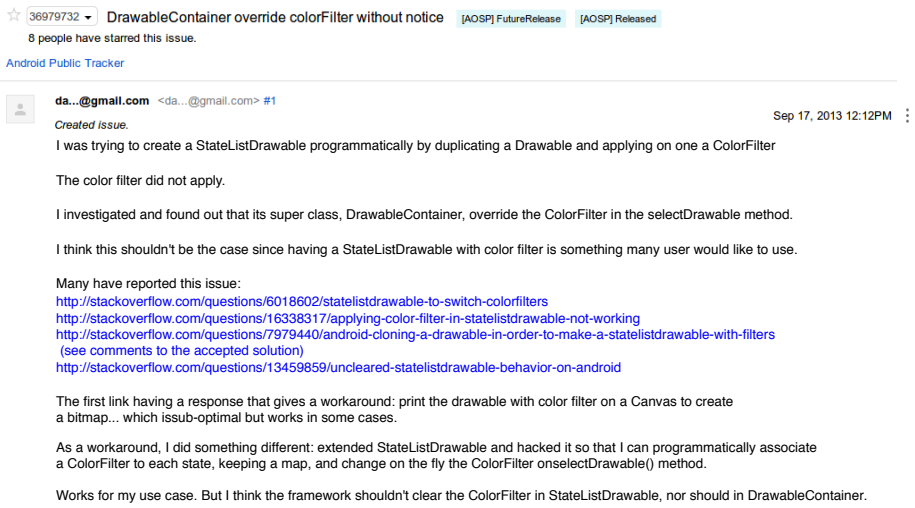
The second category focuses on understanding the challenges in sentiment analysis. Most studies on sentiment analysis use off-the-shelf sentiment analysis tools because of their availability. However, those tools are either trained in movie or product reviews. This raises the concern that those tools may not be applicable in the software engineering domain. To validate this, Jongeling et al. applied four open source tools on manually labeled data of developer emotions [35]. Results from the study show that the output of these tools differ considerably from human evaluators and there exist disagreement between sentiment analysis tools. They observed that the choice of sentiment analysis tools can affect the conclusion of a study. They also failed to replicate two previous studies using different sentiment analysis tools. Thus, they pointed to the need for sentiment analysis tools that specifically target the software engineering domain. Novielli et al. found that due to the domain-specific meaning of sentiment lexicons in technical documents (such as SO posts), SentiStrength may produce inaccurate result [42]. In a separate study, Novielli et al. compared software engineering domain-specific sentiment analysis tools with SentiStrength. They found that SentiStrength misclassifies many neutral texts as either positive and negative [43]. This is corrected by SE specific sentiment analysis tools and a substantial agreement exists among them.

The third category focuses on developing sentiment analysis for SE domains. For example, Islam and Zibran developed SentiStrength-SE that adopted several changes to prevent the misclassification of SentiStrength [33]. Ahmed et al. developed a sentiment analysis technique specifically designed for code review comments leveraging a supervised learning algorithm [3]. Calefato et al. developed a classifier, called Senti4SD, that can perform sentiment analysis of developer communications (such as Stack Overflow posts) using Support Vector Machine algorithm [17]. However, none focus on classifying SO posts concerning API issues. In our study, we use the Senti4SD for sentiment analysis that is specifically designed for SO posts.

3 Background

3.1 Motivating Example

This section presents an example that shows the benefits of classifying SO posts concerning API issue. Although there are many other examples, due to space limitation we cannot discuss many others.



36979732 DrawableContainer override colorFilter without notice [AOSP] FutureRelease [AOSP] Released

8 people have starred this issue.

Android Public Tracker

da...@gmail.com <da...@gmail.com> #1 Sep 17, 2013 12:12PM

Created issue.

I was trying to create a StateListDrawable programmatically by duplicating a Drawable and applying on one a ColorFilter

The color filter did not apply.

I investigated and found out that its super class, DrawableContainer, override the ColorFilter in the selectDrawable method.

I think this shouldn't be the case since having a StateListDrawable with color filter is something many user would like to use.

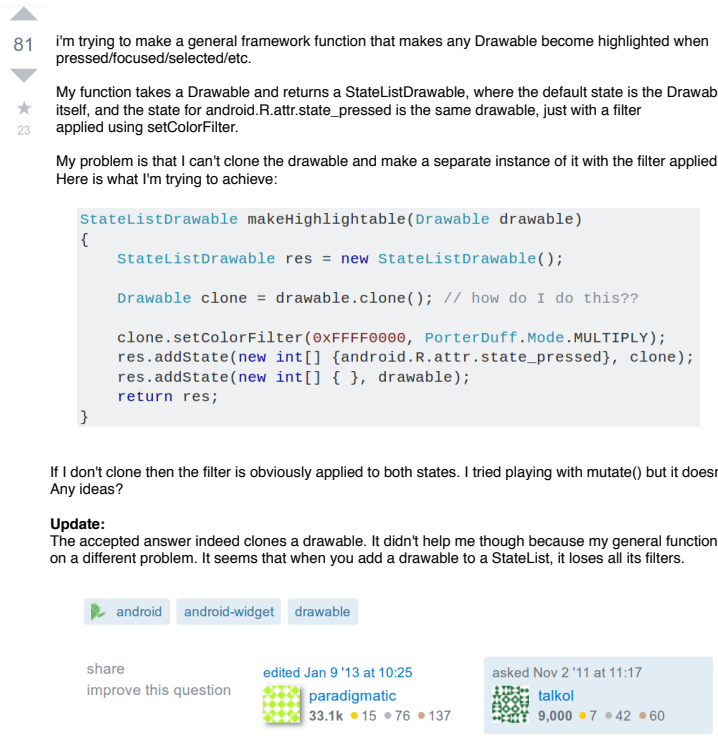
Many have reported this issue:
http://stackoverflow.com/questions/6018602/statelistdrawable-to-switch-colorfilters
http://stackoverflow.com/questions/16338317/applying-color-filter-in-statelistdrawable-not-working
http://stackoverflow.com/questions/7979440/android-cloning-a-drawable-in-order-to-make-a-statelistdrawable-with-filters
(see comments to the accepted solution)
http://stackoverflow.com/questions/13459859/uncleared-statelistdrawable-behavior-on-android

The first link having a response that gives a workaround: print the drawable with color filter on a Canvas to create a bitmap... which issub-optimal but works in some cases.

As a workaround, I did something different: extended StateListDrawable and hacked it so that I can programmatically associate a ColorFilter to each state, keeping a map, and change on the fly the ColorFilter onselectDrawable() method.

Works for my use case. But I think the framework shouldn't clear the ColorFilter in StateListDrawable, nor should in DrawableContainer.

(a) Issue report



Android: Cloning a drawable in order to make a StateListDrawable with filters

81

23

i'm trying to make a general framework function that makes any Drawable become highlighted when pressed/focused/selected/etc.

My function takes a Drawable and returns a StateListDrawable, where the default state is the Drawable itself, and the state for android.R.attr.state_pressed is the same drawable, just with a filter applied using setColorFilter.

My problem is that I can't clone the drawable and make a separate instance of it with the filter applied. Here is what I'm trying to achieve:

```
StateListDrawable makeHighlightable(Drawable drawable)
{
    StateListDrawable res = new StateListDrawable();

    Drawable clone = drawable.clone(); // how do I do this??

    clone.setColorFilter(0xFFFF0000, PorterDuff.Mode.MULTIPLY);
    res.addState(new int[] {android.R.attr.state_pressed}, clone);
    res.addState(new int[] { }, drawable);
    return res;
}
```

If I don't clone then the filter is obviously applied to both states. I tried playing with mutate() but it doesn't help.. Any ideas?

Update:
The accepted answer indeed clones a drawable. It didn't help me though because my general function fails on a different problem. It seems that when you add a drawable to a StateList, it loses all its filters.

android android-widget drawable

share improve this question

edited Jan 9 '13 at 10:25 paradigmatic 33.1k ● 15 ● 76 ● 137

asked Nov 2 '11 at 11:17 talkol 9,000 ● 7 ● 42 ● 60

(b) Stack Overflow Post

Fig. 1: A Stack Overflow post added in the Android issue tracker

Table 1: Overview of datasets

API	Date	Questions	Answers	Sample Issues	Sample Non-Issues
Android	2008 - 2017	9,94,237	14,20,973	2,000	2,000
Jenkins	2008 - 2017	22,782	26,464	250	250
Neo4j	2008 - 2017	13,434	16,215	250	250
Total		10,30,453	14,63,652	2,500	2,500

The example is about the issue of Android APIs. One of the Android developers filed an issue[5] about the design problem of ControlFilter and DrawableContainer class (check Figure 1). However, SO users start discussing this issue almost two years before filing this issue by the developer in the issue tracker. The Android developer detected this hidden issue with the help of SO discussions and therefore, they mentioned five different SO questions related to this issue. Other Android developers analyzed the discussions of those SO posts and using that knowledge they found a generalized solution just after two weeks of submitting the issue report. Eventually, this issue gets fixed almost two and a half year after the initial discussion in SO. This example indicates that SO posts concerning API issues can not only help API designers/developers to learn about API issues faster but can also help them to solve the problem. However, in the myriad of SO posts, it is very difficult for the API designers to find these issue posts. Therefore, a machine learning approach that can automatically classify issue-related posts will be useful for API designers.

3.2 Dataset Creation

This section describes the construction of our dataset. To determine what constitutes API-related issues, we use API usability factors discussed by Zibran et al. [68]. Although they presented a number of usability factors, in this paper we focus on only five of them. These are missing features, documentation, memory management, correctness, and backward compatibility. We consider any post related to these factors as API-related issues.

API Selection: We selected SO posts covering three different types of APIs based on several criteria. First, we chose those APIs that are popular, diverse in nature and have active user bases. Second, we need to identify API issue-related posts to train a classifier. We can make our decision with confidence for only those APIs that we are familiar with. Finally, we wanted APIs that have different sizes, and have a varying number of SO posts. Table 1 shows the types of APIs we considered in this study. Among these API types, the largest is the Android[6]. It allows developers to create applications and games for mobile devices. Jenkins[7] is a continuous integration and continuous delivery application. Neo4j[8] APIs provide access to scalable graph databases.

[5] https://issuetracker.google.com/issues/36979732
[6] https://developer.android.com
[7] https://jenkins.io
[8] https://neo4j.com

Posts Collection: We collected the September 2017 SO data dump from the Stack Exchange Data Dump[9]. This data includes the publicly available history of question and answer posts, tags, votes on the posts, and the reputation of the users from August 2008 to September 2017. We downloaded the four files (i.e. posts, users, votes, and tags) which were more than 80GB in total size. We considered three types of APIs (Android, Neo4j, and Jenkins) in this study. To collect SO posts related to those APIs, we leveraged the tags of a SO post. Each tag is a keyword or label that helps to categorize a post. Similar to the previous study [18], we used the tag to select SO posts related to a particular API. For the Android API, we selected posts that are tagged as "Android". For the Jenkins and Neo4j APIs, we also selected posts that are tagged as "Jenkins" and "Neo4j" respectively.

Sampling and Categorization: We need to create a label dataset of issue and non-issue posts to classify them in a supervised approach. Finding issue-related posts was not easy because SO does not support identifying API issue-related posts. We found that in the SO community, active users provide links to issue trackers (we consider Android issue tracker[10], Neo4j issue tracker[11] and Jenkins issue tracker[12]) in the question or answer sections of a post. These are candidate posts concerning API issues that need to be manually validated. We traversed each of the SO posts and extracted the link part with the tag "$\langle a \rangle .. \langle /a \rangle$". Then we checked whether the link contained particular issue tracker address or not. We only considered those posts for investigation where the links pointed to issue trackers or the issue tracker pointed SO posts in their issue description. We identified 2,237 posts for Android, 290 posts for Neo4j, and 275 posts for Jenkins that either contains a link to the issue tracker or the issue tracker contains links of SO posts.

Next, we conducted a manual study on these selected posts. The first two authors of the paper performed the manual analysis independent to each other. If we found a post related to any of the five selected API usability factors (such as missing features, documentation, memory management, correctness, and backward compatibility) we marked the post as an issue post.

To ensure that we selected posts based on the same criteria, we discussed before any labeling was done. In case there was any confusion, we discussed with each other to resolve the confusion. Otherwise, we removed the post from our analysis. Finally, we identified 2000 posts for Android, 250 posts for Neo4j and 250 posts for Jenkins as issue-related posts. We manually analyzed the remaining posts and identified a total number of 100 posts as non-issue posts. To balance the number of issue and non-issue related posts for each of our studied APIs, we needed to identify more non-issue related posts. Thus, for each API, we randomly selected posts two times greater than the required number of posts to identify non-issue posts manually. The above selected questions were

[9] https://archive.org/details/stackexchange

[10] https://issuetracker.google.com/issues?q=componentid:190923%2B

[11] https://github.com/neo4j/neo4j/issues

[12] https://issues.jenkins-ci.org/browse/JENKINS-56165?jql=project%20%3D%20JENKINS

divided into a set of chunks and one chunk was selected for each week for manual analysis by the first two authors. The process continued until the required number of non-issue posts were identified. We calculated the agreement between both coders using Cohens Kappa inter-rater agreement which ranges from -1 to +1 [19]. A Cohens Kappa value of +1 means that both coders identified the same labels for all analyzed posts. In our study, the Cohens Kappa value is 0.91.

Finally, we selected an equal number of issue and non-issue posts. In total, we identified 4,000 Android, 500 Neo4j and 500 Jenkins issue and non-issue related posts. To check whether the number of labeled posts is a statistically representative set of the selected posts, we followed a similar approach to the previous study [45]. To achieve a confidence level of 95% with a margin of error of 5%, the ideal sample size of Android, Neo4j, and Jenkins would be 384, 384 and 374 respectively. Thus, the size of our labeled dataset for each API is higher than the ideal sample size. We further analyze these posts for characterizing issue and non-issue posts in the next section.

4 Characteristic of Issue-Related SO Posts

This section analyzes the characteristics of SO posts concerning API issues. We were interested in learning the impact of reputation in asking issue-related questions or providing answers. We also investigated the time requires to receive an answer or an accepted answer. This is to validate whether issue-related questions take more time to get an answer or not. We then compared the result against non-issue related posts. We also performed a topic model analysis of issue posts to understand the frequently discussed topics.

4.1 Reputation

In this section, we study the reputation of questioners, answerers and accepted answerers of issue and non-issue posts. Stack Overflow only provides the latest reputation scores of users. Thus, we calculate a proxy reputation score of a user (either a question asker or an answerer) at time t by considering all questions asked, all answers posted, and the number of votes casted on those posts before the time t. This is done by following the official guideline of reputation calculation[13].

We use the Mann-Whitney U test, also known as the Wilcoxon unpaired signed-rank test [41], for our study. The Wilcoxon unpaired signed-rank test is a non-parametric test of which the null hypothesis is that the two input distributions are identical [41]. The p-value computed by the Wilcoxon test determines whether the difference between the two distributions are statistically significant. If the p-value is less than 0.05, we conclude that the difference between the two input distributions is significant, otherwise the difference is

[13] https://stackoverflow.com/help/whats-reputation

Table 2: P-values and Cliff's delta for reputation analysis comparing issue and non-issue posts.

API	Category of users	P-value (adjusted)	Cliff's delta
Android	Questioners	$2.8e^{-15} < 0.05$	0.24 (small)
	Answerers	$2.8e^{-16} < 0.05$	-0.21 (small)
	Accepted Answerers	$4.2e^{-3} < 0.05$	-0.07 (small)
Neo4j	Questioners	$0.356 > 0.05$	-0.98 (negligible)
	Answerers	$5.5e^{-4} < 0.05$	-0.21 (small)
	Accepted Answerers	$0.018 < 0.05$	0.16 (small)
Jenkins	Questioners	$0.004 < 0.05$	-0.17 (small)
	Answerers	$1.000 > 0.05$	-0.03 (negligible)
	Accepted Answerers	$0.004 < 0.05$	-0.21 (small)

Table 3: Result of reputation analysis of Android dataset (IQ: Issue Questioner, IA: Issue Answerer, IAA: Issue Accepted Answerer, NIQ: Non-issue Questioner, NIA: Non-issue Answerer, NIAA: Non-issue Accepted Answerer)

Reputation	IQ (%)	IA (%)	IAA (%)	NIQ (%)	NIA (%)	NIAA (%)
<100	**26.3**	35.5	18.5	55.4	22.1	13.3
100 - 1000	27.5	29.5	26.5	24.7	28.9	26.0
1000 - 10000	27.3	25.9	35.9	16.8	32.4	37.5
10000 - 50000	6.3	6.1	12.2	1.67	11.4	15.7
>50000	**2.4**	2.5	6.4	1.26	4.9	7.2

Table 4: Result of reputation analysis of Neo4j dataset (IQ: Issue Questioner, IA: Issue Answerer, IAA: Issue Accepted Answerer, NIQ: Non-issue Questioner, NIA: Non-issue Answerer, NIAA: Non-issue Accepted Answerer)

Reputation	IQ (%)	IA (%)	IAA (%)	NIQ (%)	NIA (%)	NIAA (%)
<100	49.2	21.2	10.8	39.8	22.1	7.0
100 - 1000	28.8	26.2	23.4	32.3	26.5	26.3
1000 - 10000	18.2	37.9	45.1	21.2	37.4	45.6
10000 - 50000	3.2	13.9	19.8	3.9	10.5	17.5
>50000	1.0	0.6	0.9	1.7	3.3	3.5

not significant. To quantify the difference in the distributions of the metrics, we also calculate the cliff's delta *(d)* effect size. We interpret d using the thresholds that are provided by Romano et al. [54]. Since we are performing multiple comparisons (e.g., comparisons of questioners, answerers and accepted answerers) our result can be affected by the type I error in null hypothesis testing. To overcome this problem, we control the false discovery rate (FDR) by adjusting the p-values based on the method proposed by Benjamini and Yekutieli [13]. We use the stats package of the R (p.adjust) to adjust the p-values.

Table 2 presents the adjusted P-values and cliff's delta of the reputation of questioners, answerers and accepted answerers for both issue and non-issue posts of the studied APIs. As shown in the Table 2, we observe significant differences between the issue and non-issue posts of Android for each category

Table 5: Result of reputation analysis of Jenkins dataset (IQ: Issue Questioner, IA: Issue Answerer, IAA: Issue Accepted Answerer, NIQ: Non-issue Questioner, NIA: Non-issue Answerer, NIAA: Non-issue Accepted Answerer)

Reputation	**IQ (%)**	**IA (%)**	**IAA (%)**	**NIQ (%)**	**NIA (%)**	**NIAA (%)**
<100	44.9	30.9	24.1	29.7	28.4	11.8
100 - 1000	23.5	25.5	20.4	28.1	26.4	18.9
1000 - 10000	22.7	30.9	39.4	32.8	32.8	43.7
10000 - 50000	6.9	7.4	8.7	8.2	8.2	17.2
>50000	2.4	5.0	7.3	1.3	3.8	8.3

of users. However, for the questioners in Neo4j and for the answerers in Jenkins dataset we do not find a significant difference between issue and non-issue posts.

We also categorize users into five different groups based on their reputation level. Table 3, 4 and 5 show the percentage of users that ask and answer API issue and non-issue posts for each dataset (Android, Neo4j and Jenkins). We observe that each category of users participate in issue and non-issue posts. For the Android dataset (see Table 3), we observe that users with high reputation (>50,000) ask more issue questions compared to non-issue questions (2.4% and 1.26% respectively). On the contrary, users with low reputation (>100) ask more non-issues questions compared to issue questions (55.4% and 26.3% respectively). However, such findings cannot be generalized to Neo4j and Jenkins dataset. In fact, low reputation users ask more in non-issue questions compared to issue questions for both Neo4j and Jenkins dataset. We also observe that high reputation users post more answers and accepted answers in non-issue posts compared to issue posts for each of the studied APIs.

4.2 Time Duration

We investigate two kinds of time duration: 1) duration between the post creation and submission of the accepted answer and 2) duration between the post creation and submission of the first answer. Figure 2 shows the results of our analysis for all studied APIs. We observe that more non-issue questions receive first answers and accepted answers within one hour of posting questions compared to the issue category for all studied APIs. However, we observe that the percentage of issue questions that require more than seven days to receive the first answer or the accepted answer is higher compared to the non-issue category for all three datasets. In general, we observe that more than 60% of all issue questions take more than seven days to receive the accepted answer. Furthermore, more than 31% of all issue questions take more than seven days to receive the first answer.

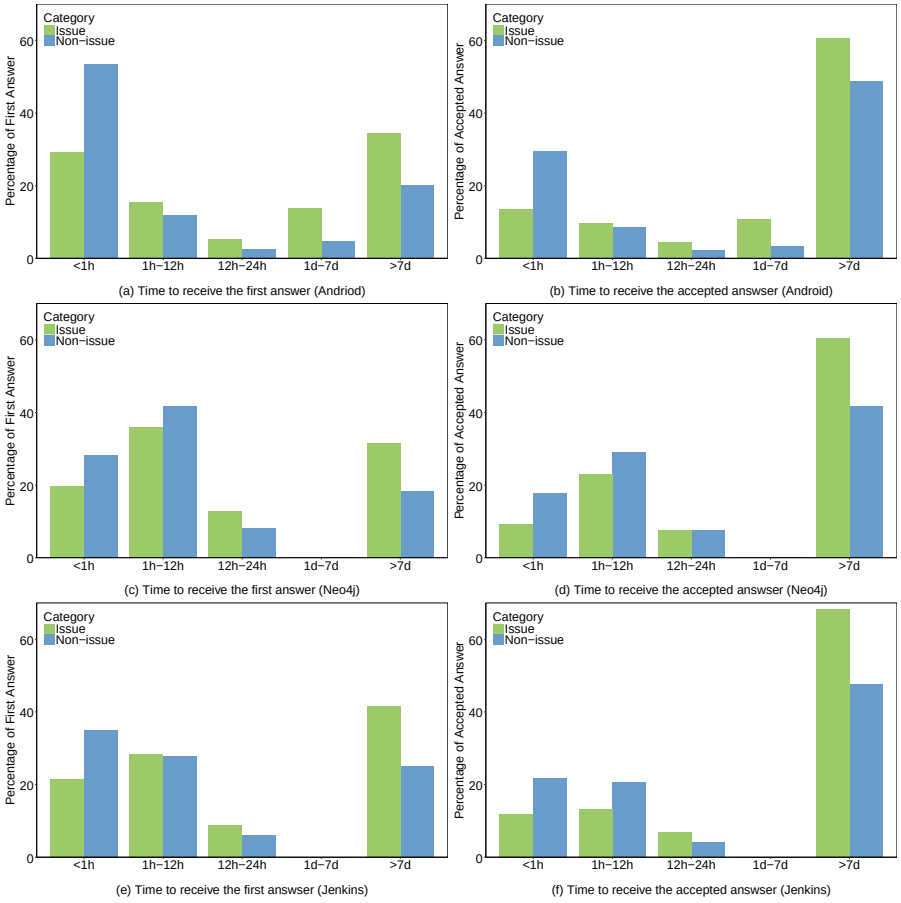


Fig. 2: Time requires to receive the first answer and the accepted answer for both issue and non-issue posts. We did not find any values corresponding to the period between 1 and 7 days for the lower four plots.

4.3 Topic Distribution Analysis

We ran the LDA model [15], an unsupervised learning method to generate topic word distribution for our datasets of API issue posts using the tool MALLET [39]. In this model, we need to set a user-defined parameter such as the number of topic K that controls the granularity of the discovered topics. Prior research shows that there is no single value of K that is appropriate in all situations and all datasets [27, 62]. In this analysis, we want to know what are the topics that are discussed in the issue posts in the studied APIs. We aim for topics of medium granularity so that the discovered topics cover the board trends in our dataset. We followed an approach similar to Barua et.al. [11] to identify the number of topics. We experimented with different values of K.

Table 6: Topic distribution analysis

Topic	Top Topic Words
T1	intent, image, bitmap, imageview, bitmapfactory, media, findviewbyid, bundle, onactivityresult, audio
T2	response, json, jsonobject, request, url, session, jsonarray, httppost, asynctask, progressdialog
T3	layout_width, wrap_cont, textview, linearlayout, fill_par, relativelayout, gravition, edittext, layout_grav
T4	ndk, device, debug, platform, fail, command, source, target, emul, version
T5	android, item, style, drawable, color, anim, tabhost, parent, res, layoutparameter
T6	intent, context, void, android, class, string, notif, overrid, log, message
T7	com, dalvikvm, android, debug, app, thread, freed, method, error, activitymanagement
T8	com, android, compile, org, gradle, class, support, google, app, error
T9	android, mediaplay, video, player, image, screen, drawable, videoview, text, png
T10	string, null, connect, log, ioexception, printstacktrace, return, inputstream, buffer, fileinputstream

Next, we analyzed topic words for each of those values of K. Finally, we set K to 40 which provided the characterization that we desired. Table 6 shows the top ten most important topics based on the topic probability distribution. The top words of T1 are “image”, “bitmap”, “imageview”, “audio”, etc., which are related to the image and media. T4 discusses with the emulator or version related problem. T7 and T8 discuss with the error in debugging, thread or compilation. Topic 9 discusses the layout and design of the Android and the last topic is about string related discussion.

5 Proposed Technique

This section presents our proposed technique for **c**lassifying **A**PI issue **p**osts in **S**tack Overflow, called *CAPS*. We consider the problem as a binary classification task that requires to generate a model consisting of features of API issue-related posts. The model is used to train a classification method that classifies any SO posts into two classes, issue, and non-issue. To avoid bias, we need to train the method using an equal number of issue and non-issue posts.

Our proposed technique consists of the following steps. The first step is the **Sentence Extraction and Text Transformation**. The textual content of a post is interleaved with HTML tags. Thus, we need to parse those tags to get the textual content of the post. Besides, we also apply a text transformation mechanism for successfully extracting sentences. The second step is the **Issue Sentence Identification**. We argue that issue-related sentences are valuable for our classification task because they provide important hints for deciding whether a post is API issue-related or not. Thus, we develop a supervised learning approach using Conditional Random Field (CRF), a statistical modeling method, to classify API issue-related sentences. The third step is the **Discriminative Classifier Generation**. In this step, we generate a set of

```
<p>I am not using Fragments, still there is a reference of
FragmentManager. If any body can throw some light on some
hidden facts to avoid this type of issue:</p>
<pre><code>
java.lang.IllegalStateException: . . .
at android.app.FragmentManagerImpl.checkStateLoss . . .
at android.app.FragmentManagerImpl . . .(FragmentManager.java:399)
. . . . . </code></pre>
```

```
<p>I already tried a </p>
<pre><code>webView.destroy();
webView = null;
</code></pre>
<p>in onDestroy() of my activity, but that doesn't help much.</p>
```

Fig. 3: Text before removing code snippet

feature values considering textual content, structural properties, user experience, and issue sentences (if any) of posts. This leads to the development of a classification model. The model is used to train our machine learning method. We also describe how the trained machine learning method can be used for classifying issue posts.

5.1 Sentence Extraction and Text Transformation

This step is responsible for extracting textual content, code examples, and stack traces from each SO post that are interleaved with HTML tags. We also extract sentences from SO posts which are essential to train CRFs. We use Stanford Parser [38] to extract sentences. This parser relies on sentence ending characters to find the boundary of a sentence. When we analyze the HTML data, we find that many text units are not terminated by an ending symbol or are split by structural elements (i.e., code snippets). To overcome these, we remove the code snippets and inject punctuation as sentence ending symbol. We extract each code examples and map them with a unique id. We find that SO users add code examples using $\langle pre\rangle\langle code\rangle \ldots \langle /code\rangle\langle /pre\rangle$ tag. Whenever the $\langle pre\rangle\langle code\rangle$ part of the above tag is not ended with a sentence ending symbol and the next word starts with an upper case letter, we inject a period. This solves the problem of split text units due to the insertion of the code snippet. Figure 3 shows above scenarios that are taken from SO posts. The result is shown in Figure 4, after applying code collapsing and removing HTML tags. Here, a period is added to indicate the end of the sentence where it meets the above criteria.

We also separate exception code (i.e., having stack traces or log information) from the normal code. If we find match of the content in $\langle code\rangle\langle /code\rangle$ with a regular expression (see Figure 5), we consider them as problematic source code (i.e., stack traces).

We remove the content and place the single word **PROBLEM_CODE**. Otherwise, if the content does not match the regular expression, we place the

```
I am not using Fragments, still there is a reference of
FragmentManager. If any body can throw some light on some
hidden facts to avoid this type of issue: PROBLEM_CODE.
```

```
I already tried a NORMAL_CODE in onDestroy() of my activity, but
that doesn't help much.
```

Fig. 4: Text after removing code snippet

```
at (.)*?\\([A-Z][a-z0-9A-Z]*.java:[0-9]*
(Exception:|Error:)(.)*?\\([A-Z][a-z0-9A-Z]*\\.java:[0-9]*\\)
(V|D|I|W|E|F|INFO|DEBUG|WARN|FATAL|VERBOSE|ERROR)\\/(.)*?\\(
```

Fig. 5: Regular expression for extracting problematic code

word **NORMAL_CODE** after removing the code snippet. We resort to manual analysis to determine the accuracy of regular expression for extracting the problematic code. Using a 95% confidence and a 5% confidence interval, we observe that the statistical sample size for 3,212 code examples is 343. The first author of the paper manually examines 350 code examples and associated posts. We find that the automatic analysis result matches the result of the manual analysis for all test cases. This ensures that the process is very accurate.

Besides, we apply naming convention in Java to detect API elements (i.e. method and class name). If we find a word having the first letter of each internal word capitalized and does not contain the opening and closing brackets, we consider them as a class. However, if the first letter is lowercase and maintains a camel case convention, we consider them as a method. Next, we remove the word in the text and replace with the word CLASS or METHOD. This generalized information help classifiers to learn better.

5.2 Issue Sentence Identification

In this step, we propose a supervised learning approach using Conditional Random Fields (CRFs) [58] for identifying issue-related sentences. We consider the detection of issue sentences as a sequence labeling task because of the availability of the contextual information in a SO post. An issue-related post should contain one or more issue-related sentences. We first introduce the CRF model and later describe its application to classify issue and non-issue sentences.

5.2.1 Conditional Random Field

CRFs are undirected and discriminative graphical models trained to maximize the conditional probability [58]. They are a sequential version of the logistic regression and a log-linear model for sequential labeling. Linear chain is a

Table 7: Examples of issue and non-issue sentences in SO posts

PostId	Sentence	Label
5796611	So, should this really be considered a "bug", since we are officially advised to use Activity.getApplication() and yet it doesn't function as advertised	Issue
12803797	It seems that the Android documentation about layout aliases is incorrect, and certainly appears inconsistent	Issue
12389115	So i added my project an AsyncTask class that i wrote a while ago for quick testing purposes but it is causing memory leak errors	Issue
6218143	If anyone knows of a good Android book that deals with this please let me know	Non-issue
11014953	I want to provide user credentials from an Android application to the API, get the user logged in, and then have all subsequent API calls pre-authenticated	Non-issue
3264383	What is the difference between Service, Async Task & Thread	Non-issue

common special-case graph structure, which corresponds to a finite state machine and is suitable for sequence labeling. A linear chain CRF compute the probability of label sequence give an observation sequence, assuming that the current label depends only on the previous label and observation, as given below:

$$P(Y|X,W) = \frac{1}{Z(X)} \prod_{t=1}^{T} exp\Big\{ \sum_{k} w_k f_k(y_{t-1}, y_t, x_t) \Big\} \tag{1}$$

where, $Y = y_1, y_2, y_3, ..., y_T$ denotes the label sequence and $X = x_1, x_2, x_3, ..., x_T$ denotes the input sequence, $f_k(.)$ denotes the k^{th} feature function which is often binary-valued, but can be real-valued, w_k denotes the weight of the k^{th} feature function. $Z(X)$ is the normalization constant that makes the probability of all state sequences sum to one, defined as follows:

$$Z(X) = \sum_{y} \prod_{t=1}^{T} exp\Big\{ \sum_{k} w_k f_k(y_{t-1}, y_t, x_t) \Big\} \tag{2}$$

Inference to the most probable labeling sequence given the observation sequence, can be efficiently calculated by dynamic programming using the Viterbi Algorithm in the following way:

$$Y^* = argmax_Y P(Y|X,W) \tag{3}$$

CRFs have many advantages over other generative models. One of the important advantages is that a wide variety of the arbitrary number of independent and non-independent features computed from the observation state can be used along with observation for labeling task because there is no constraint that feature components and observation should be independent of each other.

5.2.2 CRF Training

To train the supervised CRF model, we first generate a manually annotated dataset of issue and non-issue sentences. For each API, we selected 20% of the previously identified issue and non-issue posts for sentence level annotation with a stratified sampling approach. The first two authors of the paper manually annotated each sentence of the selected posts in any of the two classes: (1) issue sentence and (2) non-issue sentence. If we find a sentence containing information about the API issue, we label it as an *issue*. Otherwise, we use the *non-issue* to label the sentence. Table 7 shows examples of manual annotations at a sentence-level. We discussed before we started labeling and resolved any conflict through discussion. In total, we manually labeled 3,364 sentences of Android posts, 744 sentences of Neo4j posts and 467 sentences of Jenkins posts. In this annotation process, the Cohen's Kappa agreement value is 0.89. We use manually annotated issue sentences for training CRF. In order to train the CRF, we select textual and structural features of sentences from the posts. We consider the following features:

Words: In order to have better contextual information, we consider the words of a sentence as features for CRF. We do not perform any stop word removal or stemming operations. The reason behind this is that frequent words can be representative of a class [6]. Furthermore, stemming operation can hamper the contextual information. One of the advantages of CRFs is that they easily afford the use of arbitrary features of input. Therefore, the number of features in CRF is not fixed and it varies with the sentence containing a different number of words.

Part-of-Speech (POS): Part-of-speech (POS) tags are extracted from the sentence to include additional information of the grammatical structure and category of words of a sentence. We used Stanford NLP Part-Of-Speech Tagger [59] to extract POS information from sentences.

Sentiment Information: When we annotate the data, we find that most of the API issue-related sentences express a negative sentiment. In our conference version of the paper, we used an open source sentiment analysis tool, called SentiWordNet 3.0 [8] to capture the sentiment information. Although several previous studies use off-the-shelf sentiment analysis tools for analyzing the effects of emotion in collaborative development activities [26, 29, 45, 47, 57], those tools are trained on non-technical domains. The previous study of Novielli et al. warns that sentiment lexicons often have domain-specific meaning [42]. Thus, off-the-shelf sentiment analysis tools have difficulties capturing sentiment in the software engineering domain [35, 43]. Among various software engineering domain-specific sentiment analysis tool, Senti4SD is specifically designed for SO posts [17]. The technique uses a combination of four different features. The lexicon-based feature considers existing sentiment lexicons. Keyword-based features consider the count of different categories of n-grams and words (such as unigram, bigram, uppercase words). Semantic features consider the similarity between stack overflow documents and prototype vectors representing sentiment categories in the distributional semantic model. Evalu-

ation with SentiStrenth using a manually curated dataset collected from Stack Overflow posts shows that Senti4SD reduces the misclassification of neutral documents that are classified as negative documents by the SentiStrength, the most popular general purpose sentiment analysis tool. Thus, in this study, we apply Senti4SD that given a sentence automatically classifies a sentence in positive, negative and neutral sentiment category. We use the category name as the feature value.

Normalized Position in a Post: During the manual investigation of API issue posts, we observe cases where API issue-related sentences are expressed in the beginning and in the middle of a post. Thus, the position of a sentence in a SO post can be useful. We consider the position of a sentence as a heuristic feature for the CRF model to identify problematic sentences. The possible feature values are *BEGIN*, *MIDDLE*, and *BOTTOM*. For a given sentence we determine the normalized position as follows. If the sentence is located within top 20% lines of a post, we put the sentence in the *BEGIN* category. If the sentence is located within bottom 20% lines of a post, we put the sentence in the *END* category. Otherwise, the label is set to the *MIDDLE* category.

5.3 Discriminative Classifier Generation

5.3.1 Feature Collection

In this section, we briefly describe the set of features we collected for each post to train the classifier. We selected these features after analyzing both categories of posts (issue and non-issue). Each SO post contains a title, a question, zero or more answers and comments. Thus, we consider features for the title, the question and the set of answers associated with that question. In SO, novice users typically ask more how-to questions and participate less in question answering. Bajaj et al. [9] found that majority of the accepted answers are provided by users with high reputation. Thus, the reputation of questioners and answerers can help in detecting issue-related posts. Reputation indicates the expertise of a contributor in SO. However, we do not use the reputation score directly to measure the experience of questioners or answerers. This is because the SO data dump provides the latest reputation score instead of specifying the score at each point in time. Thus, to calculate the reputation of a user at the time (t) of asking a question, we consider all the questions and answers that were posted by the user before the time t. We also consider the number of upvotes and downvotes of those questions and answers that were received before the time t. We also consider three other features that can also serve as a proxy for reputation. These features fall under the experience category. We also consider two features whose values are calculated by detecting issue sentences in question body using the CRF.

We also consider three more feature categories: readability metrics, centrality measure and other information of a question. The readability metrics are used to indicate the level of difficulty for understanding a passage in English.

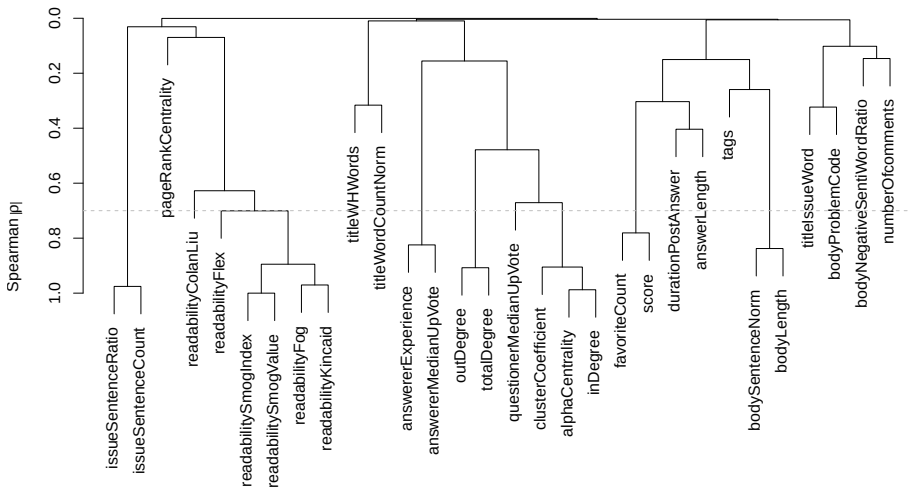


Fig. 6: Hierarchical clustering of variables according to Spearman's $|\rho|$ in Android, Neo4j and Jenkins dataset.

The higher the value of a readability metric, the more difficult the passage to comprehend. Zimmerman et al. leverage readability metrics to quantify the quality of bug reports [69]. Fan et al. found that readability metrics are a good indicator for determining the validity of bug reports [24]. Based on the prior studies, we also expect that readability metrics of a SO post's description can help us to identify API issue posts. We consider six different readability metrics in this study. We use the python package readability to calculate readability features for the question of SO posts. Inspired by the previous study of Zanetti et al. [66], we consider seven features under the centrality measure dimension. These features capture the degree of activity in the question answering community. We expect that such information can help us identifying API issue posts. The activity graph is constructed by considering questioners and answerers of SO posts. Following Zanetti et al. [66], we establish a link between a questioner and an answerer of that post if the answer is submitted within 30 days of the question creation time. Finally, we consider five different features for the question info feature category. These are commentCount, viewCount, score, tags, and favoriteCount. Table 8 summarizes the eight sets of features we selected for the classification model.

5.3.2 Correlation and Redundancy Analysis

The first step is to perform a correlation analysis. Our goal is to remove features that are highly correlated because they could affect negatively in our supervised classification task. We apply a variable clustering approach on each of the datasets to construct a hierarchical structure using an R package, called **misc**. Those features that are correlated appear in sub-hierarchies. For each

Table 8: Summary of features

Dimension	Feature	Description
Title	titleIssueWord	True if contains bug, issue, error or exception
	titleWHwords	True if starts with How,Where or What
	titleWordCountNorm	Normalized number of words in a title
Body	bodyProblemCode	True if contains stacktrace or log information
	bodyNegativeSentiWordRatio	The ratio of negative sentiment word to the total number of word in the post
	bodyNegativeSentiWordRatio	The normalized value of the number of sentences in body
CRF	issueSentenceCount	Total number of issue sentences in the post body
	issueSentenceRatio	The ratio of issue-related sentences to total number of sentences in the body
Answer	answerPresent	True if there is at least one answer
	durationPostAnswer	The time duration between the post created and the first accepted answer posted
	answerIssueWord	True if contains bug, issue exception or error
Experience	questionerMedianUpVote	Median of the upvote count of previous posts that are asked by the questioner
	answererMedianUpVote	Median of the upvote count of previous answers posted by the answerer
	questionerExperience	The total number of previous questions and answers posted by the questioner
	answererExperience	The total number of previously accepted answers posted by the answerer
	questionerQualityPost	The median of differences between the number of upvotes and downvotes of previously asked questions by the questioner
Readability	readabilityFlech	$206.835 - 1.015\frac{Words}{Sentences} - 84.6\frac{Syllables}{Words}$ [25]
	readabilityFog	$0.4\frac{Words}{Sentences} + 40\frac{Complex\ Words}{Words}$ [28]
	readabilitySmogIndex	$1.043 \times \sqrt{Polysyllables} + 3$ [36]
	readabilityKincaid	$0.39\frac{Words}{Sentences} + 11.8\frac{Syllables}{Words} - 15.59$ [34]
	readabilityColanLiu	$5.88\frac{Characters}{Words} + 29.6\frac{Sentences}{Words}$ [20]
	readabilitySmogValue	$3 + \sqrt{Polysyllables}$ [36]
Centrality Measure	alphaCentrality	These features represent questioners answerers activity in question answering community [16, 24, 66]
	clusterCoefficient	
	pageRankCentrality	
	inDegree	
	outDegree	
	totalDegree	
	lccMembership	
Question Info	commentCount	The number of comments added to the question of the SO post
	viewCount	The number of views of the SO post
	score	The score of the SO post
	tags	Number of tags associated to the question
	favoriteCount	The number of times a question is selected as favorite by SO users. A question is marked as a favorite by clicking the star beneath the vote counter

sub-hierarchy where the correlation of features is greater than 0.7, only one feature is selected toward building the classification model [40] (see Figure 6). For example, we find issueSentenceCount and issueSentenceRatio are highly correlated. Therefore, we only consider issueSentenceCount and remove the other. We also remove readabilitySmogIndex, readabilitySmogValue, readabilityKincaid, answererExperience, totalDegree, inDegree, alphaCentrality, favoriteCount, and bodySentenceNorm after performing correlation analysis.

The second step is to perform a redundancy analysis. The objective of this step is to remove features that do not have unique signal compared to other features. To perform the analysis we apply the **redun** function in the **rms** package of R. We apply a similar approach that is presented by McIntosh et al. [40] for redundancy analysis. We use the default threshold value of 0.9 for

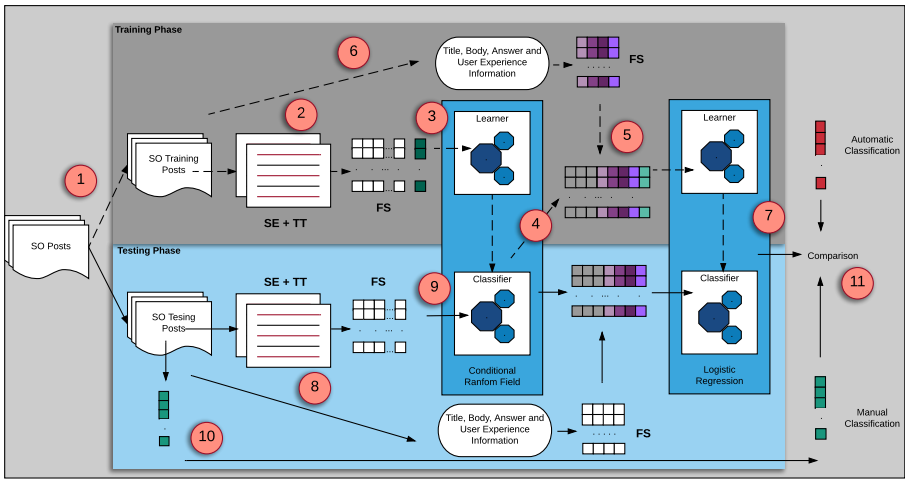


Fig. 7: Overview of our proposed technique (SE = Sentence Extraction, TT = Text Transformation and FS = Feature Selection)

this analysis [40]. After performing the redundancy analysis we do not find any redundant feature.

5.3.3 Training and Testing the Classifier

To train the classifier we need to generate a training dataset. For each post, we collect all the feature values as described in the previous section. These act as predictors or independent variables. The target or response variable indicates whether the post is API issue-related or not. Thus, the response variable has two classes. We use logistic regression to derive our classification model. Logistic regression is a discriminative classification model that operates on the real-valued vector input. It is also a probabilistic classifier that given a test post generates a class probability value. This value indicates the likelihood of the post belonging to that class.

Figure 7 explains the training and the testing phases of *CAPS*. We first create a dataset containing manually classified SO posts. We select these SO posts for the training of *CAPS* (Point 1).

We extract sentences and perform text transformation (Point 2). We then determine the feature values for each post (Point 3) (check Table 8). We also train the CRF (Point 4) classifier. We generate the title, question body, answer and experience level features for each post (Point 6). We use all these and CRF features to train our classification model. Point 7 shows the actual output of the training. To avoid any bias, we use the same number of issue and non-issue-related posts to train our classification model. To test *CAPS*, we also need manually classified SO posts. For each test post, we determine the title, question body, answer, and experience of users (Point 8). We also collect the features for CRF (Point 9). A total of eight sets of feature values act as an input to our classifier build in the training phase. Then, a vector for collected

features fed into the previously trained classification model for testing. The classifier then determines whether the test post is API issue-related post or not. Recall that the posts in our dataset are already manually classified. We collect the results of manual classification (Point 10) and compare the results with that of the classifier (Point 11).

6 Evaluation

We evaluate *CAPS* in two different ways. First, we compare the technique with three baseline approaches. Second, we compare the technique with the work of Wang et al. [65]. The following section describes each of the experiment in detail.

6.1 Comparison with Baseline Approaches

We compare *CAPS* with three baseline approaches that address the same problem using different heuristics and sources of information. These are a text classification technique, a CRF-based technique and a reputation-based technique.

6.1.1 Description

We consider a machine learning-based text classification technique because it has shown great promise in various problems in software engineering [6, 46]. To determine whether issue-related sentences can solely be used for classifying issue-related posts, we include the CRF-based technique in this study. Finally, we consider a reputation-based technique to determine the usefulness of reputation for solving the classification problem. We briefly describe each of the technique as follows.

Text Classification Technique: In our study, we consider a machine learning-based text classification technique that automatically learns from training data. In our case, the data comes from SO posts. Our technique is statistical because we provide manually labeled API issue and non-issue-related posts to learn each class. Automatic text classification has been found effective in various problem areas that deal with a large amount of textual content. Examples include but are not limited to content classification of developer emails [6], separating features from bug reports, discovering tutorial sections that explain a given API type [46]. Typically, a text classification technique generates features using terms appearing in documents. The documents are modeled as vectors of features and these features values are determined from the frequency of those terms in documents. In our case, documents are SO posts and features are the set of words appearing in those posts.

- **Term Selection:** We consider each post as a bag of words. For each post, we collect any words by tokenizing the title, question body, and answers including any code fragments in them. We neither perform any stop word removal nor apply any stemming. This is because frequently appearing words or words that derive from the same root word can representative of a class [6]. Depending on the size of the posts, the set of words can be very large. Instead of considering all terms as features that can lead to overfitting problem, we consider a subset of terms as features using frequency-based feature selection technique. The technique performs often well when many thousands of features are considered.
- **Machine Learning Method:** We select the logistic regression classifier that learns from the training data and performs the classification. Logistic regression is a discriminative classification model that operates on the real-valued vector input. Despite the simplicity, logistic regression has been found effective in text classification tasks. Details of the technique can be found elsewhere [4].

CRF-based Technique: Conditional Random Fields (CRFs) are statistical modeling methods. We use CRF to detect API issue-related sentences. We hypothesize that if a post contains such sentences, it is an API issue post. To validate the hypothesis, we make the following change to allow CRF to classify issue-related posts. We train CRF using manually validated API issue sentences. Given a test post, we apply CRF to its textual content to detect issue sentences. We classify the post as API issue-related if CRF identifies any issue sentences in it.

Reputation-based Technique: We also implement another technique that uses the reputation of SO users to classify a post into issue category. The basic idea is that if a post question is asked or answered by a user with high reputation, it is likely to be an issue post. We include the technique to verify to what extent the claim can be supported by empirical study. However, it is difficult to define the term high reputation. Thus, we determine the quartiles of the reputation of SO users participated in the posts of target APIs and consider anything above the third quartile value as the high reputation. For a test post, if the reputation of the questioner or any of its answerers is greater than the third quartile of the distribution of reputation of users, we classify the posts into issue category.

6.1.2 Evaluation Metrics

We evaluate our results using *precision*, *recall* and *F-measure* which have been used in a number of previous studies [46]. We compare our generated ground truth with automatically generated classification. The correctly classified posts have been considered as *true positive* and the post incorrectly classified as belonging to the class have been considered as *false positive*. The post incorrectly

Table 9: Evaluation results of our proposed technique and three other baseline approaches for classifying API issue posts. P, R and F denote precision, recall and F-measure respectively. The highest precision, recall and F-measures across all techniques are highlighted in boldface.

API	Technique	Issue			Non-issue		
		P	R	F	P	R	F
Android	Reputation	0.63	0.43	0.51	0.27	0.25	0.26
	CRF	0.46	**0.83**	0.59	0.30	0.34	0.32
	Text Classification	0.54	0.56	0.54	0.49	0.51	0.49
	CAPS	**0.95**	0.71	**0.81**	**0.76**	**0.95**	**0.84**
Neo4j	Reputation	0.38	0.25	0.30	0.32	0.31	0.31
	CRF	0.28	0.45	0.34	0.36	0.51	0.42
	Text Classification	0.50	0.59	0.54	0.64	0.55	0.59
	CAPS	**0.95**	**0.75**	**0.83**	**0.83**	**0.95**	**0.88**
Jenkins	Reputation	0.43	0.36	0.39	0.32	0.31	0.31
	CRF	0.42	0.62	0.50	0.39	0.52	0.45
	Text Classification	0.49	0.50	0.49	0.50	0.59	0.54
	CAPS	**0.92**	**0.71**	**0.80**	**0.73**	**0.92**	**0.81**

labeled belonging to other class have been computed as the *false negative*. Thus, *precision*, *recall* and *F-measure* can be computed as the following way:

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

$$F - measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{6}$$

Here, TP denotes as *true positive*, FP denotes as *false positive* and FN denotes as *false negative*.

6.1.3 Experimental Setup

We use our dataset consists of SO posts that are labeled into two classes (issue and non-issue) to perform the evaluation. We apply 10 fold stratified cross-validation to measure the performance of each compared technique. We split the dataset into 10 different folds of equal sizes. We use the 9 folds (90% of data) to train the technique and the remaining fold is used to test the performance of the technique. We repeat the process 10 times by rotating the training and test folds. The MALLET [39] tool is used to train the CRF and we reuse the implementation of the logistic regression available in the Weka [30].

6.1.4 Evaluation Results

Table 9 summarizes the results of our evaluation for both issue and non-issue classes. The highest precision, recall and F-measure values for both issue and

non-issue posts across all techniques are highlighted in boldface. The results clearly suggest that *CAPS* outperforms the other three baseline approaches. For the Android API, the reputation-based technique performs the worst. While the precision and recall values are 0.63 and 0.43 for the issue class, both values are dropped for the non-issue class (0.27 and 0.25 respectively). The CRF-based technique improves the performance on the issue class but it does not work well on the non-issue class. Since the number of non-issue posts is expected to be much higher than the issue posts, the low recall value for the non-issue class makes the technique ineffective. Text classification technique also does not perform well comparing other three baseline approaches. While the precision and recall values are 0.54 and 0.56 for the issue class, for the non-issue class the values are 0.49 and 0.51 respectively. *CAPS* achieves the best precision and recall values for both classes. While the precision and recall values are 0.95 and 0.71 for the issue class, the technique achieves 0.76 and 0.95 for non-issue class. We also observe similar results for the Neo4j and Jenkins API. The reputation-based technique performs the worst again. The text classification technique and the CRF-based technique rank the second and the third positions respectively. *CAPS* performs the best among all four techniques.

6.2 Comparison with the Work of Wang et al. [65]

A related work to our study is that of Wang et al. [65]. Given a collection of SO posts, their technique recommends a ranked list of API issue-related posts. Since the implementation is not publicly available, we re-implement the technique. The technique first detects experts and retains only those posts that are asked by expert users. It then detects dominant SO discussion topics and selects only those posts for recommendation that are related to those topics. Next, the technique filters late answered posts using a statistical quality control technique, called *control charts*. The remaining posts are sorted based on a set of metrics derived from SO data.

Their work does not focus on classifying issue-related posts and does not utilize textual features of SO posts. Thus, it is difficult to compare *CAPS* with their work. However, we follow the following approach for the purpose of comparison. We select an equal number of issue and non-issue-related posts (1,000 in total) of Android API from our dataset for training *CAPS*. The remaining 1,000 issue posts of Android API are used for testing. We would like to find how many of these issue-related posts are detected by the technique of Wang et al.. We fed all the Android-related posts except those we use for training *CAPS* as input to the technique. After filtering, the technique selects 91,234 posts out of the 994,237 Android posts. This selected set of posts only contain 312 issue-related posts out of the 1,000 we selected for testing. Thus, the technique achieves 31.2% accuracy. However, *CAPS* correctly classifies 750 issue-related posts out of the 1,000 and achieves 75% accuracy.

6.3 Comparison with Opiner

Another study related to ours is the work of Uddin and Khomh [61]. They developed a tool, called Opiner, that summarizes API reviews collected from Stack Overflow posts. Although related the goal of Opiner is different from *CAPS*. For example, *CAPS* focuses on removing a large number of how-to and newbie questions that make it difficult to locate posts concerning API issues. However, Opiner focuses on collecting and summarizing opinions about APIs. Those how-to and newbie questions can be useful to Opiner for opinion mining. Despite the differences, we were interested in learning whether Opiner can be used to identify posts concerning API issues. Thus, we make a set of changes in Opiner to make the comparison possible.

Opiner categorizes sentences into a set of aspect categories leveraging two supervised algorithms. To train Opiner, we reuse the dataset developed by Uddin and Khomh that consists of 4,594 manually labeled sentences from 1,338 Stack Overflow posts into aspect categories. Then, we collected sentences from our test dataset and give that to Opiner to classify those sentences into aspect categories. Opiner uses a combination of SentiStrength and the Sentiment Orientation algorithm to determine the polarity of those aspect-oriented sentences. We then apply heuristics to classify posts into issue and non-issue categories leveraging negative sentiment aspect-oriented sentences. We do this in two different way. If a SO post contains at least one negative sentiment aspect-oriented sentence, we categorize the post into issue category. Otherwise, the post is classified into non-issue category. We refer to this as the basic approach. The other approach considers majority voting where the polarity of the post dictates by the polarity of the largest number of sentences. In case of a tie which can happen when a post has an equal number of positive and negative sentiment sentences, we break the tie considering the sum of sentiment polarity scores of sentences.

Table 10 compares the evaluation result of the modified Opiner with *CAPS* for all three API types. The highest precision, recall and F-measure values for both issue and non-issue posts across all techniques are highlighted in boldface. Opiner-E:Basic refers to the modified Opiner that implements the basic approach discussed above in classifying SO posts into issue or non-issue categories. Opiner-E:Majority Voting implements the majority voting approach in the classification. Results from the study clearly indicate that *CAPS* performs considerably better than Opiner-E. This is most likely due to the reason that Opiner is not designed for removing how-to and newbie questions. Furthermore, *CAPS* also considers a large number of features that affects the performance considerably.

7 Importance of Features for Classifying API Issue Posts

This section investigates which features are most important for classifying API issue posts. Since different projects have different characteristics, the impor-

Table 10: Comparison with the modified Opiner for classifying API issue posts. The highest precision, recall and F-measures across all techniques are highlighted in boldface.

API	Technique	N-gram	Issue			Non-Issue		
			P	R	F	P	R	F
Android	Opiner-E: Basic	U	0.61	0.43	0.60	0.61	0.77	0.68
		B	0.59	0.48	0.53	0.62	0.72	0.67
		T	0.59	0.48	0.53	0.62	0.72	0.66
	Opiner-E: Majority Voting	U	0.59	0.64	0.61	0.67	0.62	0.64
		B	0.57	0.71	0.64	0.70	0.51	0.60
		T	0.57	**0.73**	0.64	0.70	0.52	0.60
	CAPS		**0.95**	0.71	**0.81**	**0.76**	**0.95**	**0.84**
Neo4j	Oopiner-E: Basic	U	0.57	0.56	0.56	0.64	0.66	0.65
		B	0.62	0.40	0.48	0.62	0.80	0.70
		T	0.55	0.65	0.60	0.66	0.56	0.61
	Oopiner-E: Majority Voting	U	0.57	0.56	0.56	0.64	0.66	0.65
		B	0.56	0.65	0.60	0.66	0.57	0.64
		T	0.64	0.40	0.49	0.62	0.82	0.71
	CAPS		**0.95**	**0.75**	**0.83**	**0.83**	**0.95**	**0.88**
Jenkins	Oopiner-E: Basic	U	61.3	0.40	0.48	0.56	0.75	0.64
		B	0.59	0.50	0.54	0.57	0.67	0.62
		T	0.56	0.68	0.61	0.60	0.47	0.53
	Oopiner-E: Majority Voting	U	0.58	0.57	0.58	0.59	0.59	0.59
		B	0.56	0.68	0.61	0.60	0.47	0.53
		T	0.56	0.68	0.61	0.60	0.47	0.53
	CAPS		**0.92**	**0.71**	**0.80**	**0.73**	**0.92**	**0.81**

tance of features that distinguish API issue posts from the non-issue ones can be quite different. Thus, we conduct the study for each dataset.

The logistic regression algorithm that we used in our *CAPS* model is typically used for modeling linear relationship with the response variable. However, some factors potentially share non-linear relationships with the response variable. Thus, we use restricted cubic splines to add the non-linear terms of factors into the model by following the prior study of McIntosh et al. [40].

Table 11 shows the effect of each factor to classify API issue related posts. We find that issueSentenceCount feature is ranked as the most important factor in each of our studied datasets. As shown in Table 11, we observe that issueSentenceCount has the highest Wald χ^2 value with statistical significance. We also observe that CRF feature shares a non-linear relationship with the response variable as the non-linear term of issueSentenceCount provides a statistical significance and a large explanatory power to the model. However, we observe that in most other cases, the non-linear term does not provide much explanatory power to the model. The second most important feature for our model is bodyLength. The relationship between the bodyLength and the response variable is almost linear since the non-linear term does not provide explanatory power to the model. We also find three features from Experience (questionerMedianUpVote), Answer (durationPostAnswer) and Network Centrality (outDegree) which show a statistical significance in classifying API issue posts.

Table 11: An overview of results of logistic regression models. The overall and non-linear (NL) Wald χ^2 of each factor is shown as the proportion in relation to the total Wald χ^2 of the model. The top factors or features for each dataset are shown in bold and italic. (*) $p < 0.05$; (**) $p < 0.01$; (***) $p < 0.001$. (+) Discarded during factor selection; (-) Non-linear term not allocated.

Factor		**Android**		**Neo4j**		**Jenkins**	
AUC		0.97		0.94		0.93	
AUC optimism		0.0036		0.004		0.04	
Wald χ^2		705		98		87	
Budget of degree of freedom		314		28		30	
Degree of freedom spent		32		31		32	
		Overall	Non-linear	Overall	Non-linear	Overall	Non-linear
Title Feature							
titleWHWords	D.F	1	-	1	-	1	-
	χ^2	28.58***	-	5.43**	-	9.58**	-
titleIssueWord	D.F	1	-	1	-	1	-
	χ^2	3.62*	-	4.72*	-	3.63*	-
titleWordCountNorm	D.F	1	-	1	-	1	-
	χ^2	16.39**	-	3.58	-	5.31*	-
Body Feature							
bodyProblemCode	D.F	1	-	1	-	1	-
	χ^2	0.26	-	1.62	-	3.44	-
bodyNegativeSentiWordRatio	D.F	1	-	1	-	1	-
	χ^2	10.47**	-	6.33*	-	3.90*	-
bodyLength	D.F	4	3	4	3	4	3
	χ^2	167.38***	9.95*	14.27***	0.29*	16.01***	0.41
CRF Feature							
issueSentenceCount	D.F	2	1	2	1	3	2
	χ^2	391.21***	35.06***	40.12***	0.41	42.25***	0.01
Answerer Feature							
durationPostAnswer	D.F	4	3	4	3	4	3
	χ^2	61.16***	11.03	3.08*	0.54	5.11*	0.51
answerPresent	D.F	1	-	1	-	1	-
	χ^2	0	-	0	-	0	-
answerIssueWord	D.F	1	-	1	-	1	-
	χ^2	0	-	0	-	0	-
answerLength	D.F	1	-	1	-	1	-
	χ^2	0.05*	-	5.48*	-	3.25	-
Expertise Feature							
questionerMedianUpVote	D.F	4	3	3	2	3	2
	χ^2	96.97***	3.44*	1.52*	0.8	5.21**	3.22
answererMedianUpVote	D.F	1	-	1	-	1	-
	χ^2	0.73	-	1.25	-	0.08	-
answererExperience	D.F	1	-	1	-	1	-
	χ^2	7.24**	-	1.39*	-	1.11	-
questionerExperience	D.F	1	-	1	-	1	-
	χ^2	5.34*	-	0.44	-	0.24	-
questionerQualityPost	D.F	1	-	1	-	1	-
	χ^2	0.34	-	0.32	-	0.27	-
Readability Feature							
readabilityFog	D.F	1	-	1	-	1	-
	χ^2	10.88*	-	0.56*	-	1.02*	-
readabilityColanLiu	D.F	1	-	1	-	1	-
	χ^2	5.33*	-	0	-	0.21	-
readabilityFlex	D.F	1	-	1	-	1	-
	χ^2	0.94	-	0.05	-	0.28	-
Network Centrality Feature							
clusterCoefficient	D.F	1	-	1	-	1	-
	χ^2	0.75	-	0.33	-	0.11	-
pageRankCentrality	D.F	1	-	1	-	1	-
	χ^2	0.78	-	2.02	-	0.65	-
outDegree	D.F	1	-	1	-	1	-
	χ^2	71.58***	-	3.58*	-	1.21*	-
lccMembership	D.F	1	-	1	-	1	-
	χ^2	0	-	0	-	0	-
Question Feature							
tags	D.F	1	-	1	-	1	-
	χ^2	3.27*	-	3.45	-	3.33	-
numberOfcomments	D.F	1	-	1	-	1	-
	χ^2	5.15	-	0.69	-	0.54	-
Score	D.F	1	-	1	-	1	-
	χ^2	3.75**	-	6.92**		5.67**	-

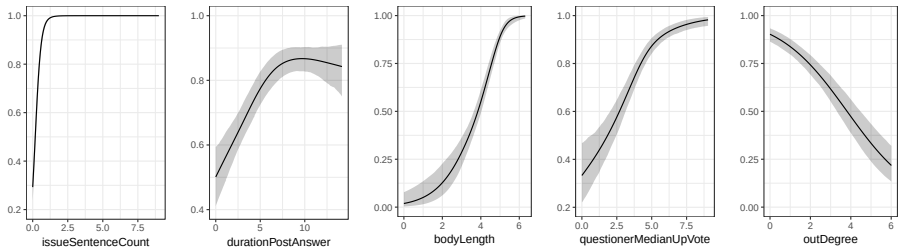


Fig. 8: The estimated probability when the values of issueSentenceCount, durationPostAnswer, bodyLength, questionerMedianUpVote and outDegree change for Android dataset. Y axis is the probability of classifying issue post. X axis is the value of factors except for the durationPostAnswer where we consider the lograthim value of the factor. The gray area shows the 95% confidence interval

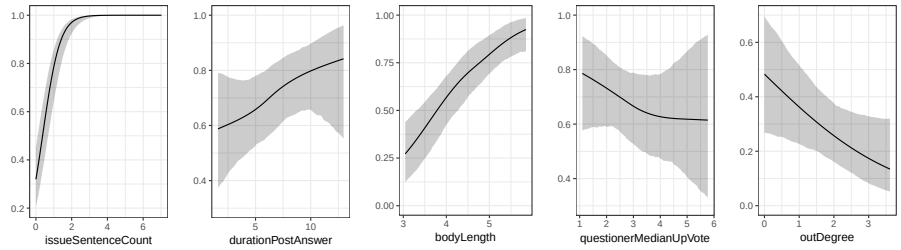


Fig. 9: The estimated probability when the values of issueSentenceCount, durationPostAnswer, bodyLength, questionerMedianUpVote and outDegree change for Neo4j dataset. Y axis is the probability of classifying issue post. X axis is the value of factors except for the durationPostAnswer where we consider the lograthim value of the factor. The gray area shows the 95% confidence interval

To further understand how a factor affects the value of the response variable, we plot the estimated likelihood of classifying API issue posts. We use the **Predict** function in the **rms** R package to plot the estimated likelihood. Figure 8, 9, 10 shows the relationship between important features and the response variable for all three datasets. We observe that the probability of classifying issue posts increases exponentially when the issueSentenceCount is greater than two across all three datasets.

We find that the probability of classifying API issue posts increases with the increase of bodyLength. This result suggests that an API issue post is likely to be more lengthy and descriptive than the non-issue posts. We observe an opposite relationship between the outDegree and the probability of classifying issue post. One possible explanation to this phenomenon is that API issue posts are difficult to answer and questioners receive a small number of answers. We can observe that the probability of classifying API issue posts increases with

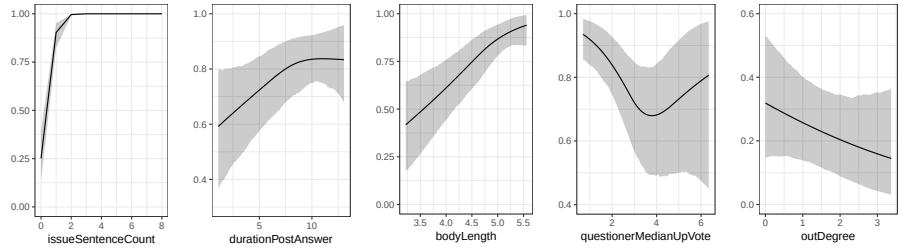


Fig. 10: The estimated probability when the values of issueSentenceCount, durationPostAnswer, bodyLength, questionerMedianUpVote and outDegree change for Jenkins dataset. Y axis is the probability of classifying issue post. X axis is the value of factors except for the durationPostAnswer where we consider the lograthim value of the factor. The gray area shows the 95% confidence interval

the increase of durationPostAnswer. This indicates that the API issue posts require more time to receive the first answer. Although we observe a similar pattern in Android for questionerMedianUpVote, we find opposite pattern in Neo4j and Jenkins.

8 Evaluating CRF-based Supervised Learning Technique

Our discriminative classifier for classifying API issue posts depends on a supervised learning technique for classifying issue-related sentences. During our empirical study, we found that the number of issue sentences plays an important role in classifying API issue posts. Thus, the performance of our classifier also depends on the accuracy of classifying issue-related sentences. However, we have not discussed yet the accuracy of our CRF-based supervised learning technique for issue sentence classification.

To train the CRF-based issue sentence classifier, we consider five different features. These are words that appear in a sentence, part-of-speech tags that are extracted from each sentence to capture grammatical structure and category of words, the level of sentiment expressed by a sentence and the normalized position of a sentence in a post. We run the Senti4SD tool to identify the sentiment of a sentence in one of the three categories (i.e., positive, negative and neutral). The normalized position of a sentence can be one of the following three categories: BEGIN, MIDDLE, and END. We not only report the performance of the issue sentence classifier but also compare the effectiveness of different categories of features. We also compare the classifier with two other baseline approaches. Our goal is to understand whether any of these baseline approaches can replace our CRF-based issue sentence classifier. Finally, we study any impact of pre-processing operation such as stemming and stop word removal on the issue sentence classifier.

Table 12: Comparison of CRF-based issue sentence classifier with two other baseline approaches using Android dataset. P, R and F denote precision, recall and F-measures respectively. The highest precision, recall and F-measures across all techniques are highlighted in boldface.

Techniques	**Issue Sentences**			**Non-issue Sentences**		
	P	**R**	**F**	**P**	**R**	**F**
ISC	**0.70**	**0.85**	**0.70**	**0.70**	**0.65**	**0.69**
Text Classification	0.60	0.52	0.55	0.50	0.60	0.54
Sentiment Analysis	0.37	0.57	0.45	0.55	0.59	0.57

8.1 Experimental Setup

We manually annotated sentences of SO posts in all of our three datasets in either issue or non-issue category. We use that annotated data to evaluate the effectiveness of our issue sentence classifier. Here, we report results for the Android dataset because it contains the largest number of SO posts. We use the same 10 fold stratified cross-validation technique to measure the performance. We also use the same metrics that we use for evaluating *CAPS* (i.e., precision, recall, and F-measure). In this experiment, the correctly classified sentences are considered as the true positives and the incorrectly classified sentences are considered as the false negatives.

8.2 Evaluation Results

We consider a text classification technique and a sentiment analysis technique as baseline approaches to compare with our CRF-based issue sentence classifier. Our text classification technique is similar to the one we compared with *CAPS*. Since our objective is to classifying sentences instead of documents, we use the terms appear in sentences for both training and testing. We also use the same logistic regression technique to classify sentences. Recall that we categorize sentences in one of the three sentiment categories using Senti4SD tool. We use that information to build our sentiment analysis technique. We hypothesize that if the sentiment of the sentence is negative, it could be an issue sentence. This is based on the observation that issue-related sentences may contain problem description, difficulties of using APIs and frustration of developers, thus express negative sentiments. Previous studies also use the sentiment information for identifying problematic API features [67]. These motivate us to understand the usefulness of sentiment for solving our classification problem. Thus, we include the sentiment analysis technique in this experiment.

Table 12 summarizes the results of our evaluation for both issue and non-issue sentence categories. The highest precision, recall and F-measure values for both issue and non-issue sentences across all techniques are highlighted in boldface. The results indicate that our CRF-based issue sentence classifier outperforms two other baseline approaches. The sentiment analysis technique

Table 13: Comparison of difference sources of information for issue sentence classification. The highest precision, recall and F-measures across all feature groups are highlighted in boldface.

No	Feature Group	Issue Sentence			Non-issue sentence		
		P	**R**	**F**	**P**	**R**	**F**
1	Words	0.60	0.30	0.40	0.55	0.80	0.65
2	Words + Sentiment	0.53	0.39	0.45	0.56	0.68	0.61
3	Words + Position	0.63	0.41	0.50	0.58	0.85	0.69
4	Words + Sentiment + Position	0.52	0.39	0.44	0.55	0.67	0.60
5	Parts-of-Speech + Sentiment + Position	0.45	0.57	0.51	0.50	0.40	0.44
6	Words + Parts-of-Speech + Sentiment + Position	**0.70**	**0.85**	**0.70**	**0.70**	**0.65**	**0.69**

also does not show good results. This could be due to the fact that off-the-shelf sentiment analysis techniques are not suited to our problem. The text classification technique improves the performance of issue sentences. While the precision and recall values for the issue sentence category are 0.60 and 0.52 respectively, the values are 0.50 and 0.60 for the non-issue sentence category. Our CRF-based issue sentence classifier considerably improves the recall for the issue sentence category (achieves more than 30% improvement) and precision of non-issue sentence category (achieves 20% improvement). This is due to considering different categories of features.

8.3 Impact of Feature Groups

Our CRF-based issue sentence classifier uses four different categories of features. We are interested to learn the impact of different feature groups. Towards this goal, we conduct our issue sentence classification experiment for the Android dataset again. However, this time we run the experiment considering different groups of features. All other settings of the technique remain unchanged. Table 13 shows the performance of CRF-based issue sentence classifier for a different combination of feature groups. The highest precision, recall and F-measure values for both issue and non-issue sentences across all feature groups are highlighted in boldface. When we only consider the Words feature, the precision and recall values for the issue sentence category reach to 0.60 and 0.30 respectively. For the non-issue category, the precision and recall values are 0.55 and 0.80 respectively. One important observation is that the recall values for the issue sentence category are very poor.

Adding the position with Words results in better performance than adding the sentiment with Words. When we combine both position and sentiment features with words, the precision and recall values for the issue sentence category improves (reach to 0.65 and 0.41 respectively). However, for the non-issue precision remains the same but recall drops to 0.67. Adding parts-of-speech information to sentiment and position information does lead to better results. However, when we combine all four feature sets, the technique performs

Table 14: Comparison of CRF-based issue sentence classifier after stemming and stop word removal. P, R and F denote precision, recall and F-measures respectively. The highest precision, recall and F-measures across all techniques are highlighted in boldface.

Techniques	Issue Sentences			Non-issue Sentences		
	P	**R**	**F**	**P**	**R**	**F**
ISC without stemming and stop word removal	**0.70**	**0.85**	**0.70**	**0.70**	0.65	**0.69**
ISC with stemming operation	0.61	0.44	0.51	0.62	**0.78**	0.69
ISC with stop word removal	0.58	0.69	0.63	0.58	0.59	0.66

the best. The precision and recall values for the issue-sentence category reach to 0.70 and 0.85. For the non-issue sentence category, those values reach to 0.70 and 0.65 respectively. This indicates that we need to consider all four information sources for classifying issue sentences.

8.4 Impact of Stemming and Stop Word Removal

Recall that we did not apply stemming and stop-word removal for *CAPS*. In this section, we study whether stemming and stop word removal have any impact on the performance of issue sentence classification. First, we generate the training and the test dataset after performing stemming and removing stop word. Then, we run our issue sentence classifier to evaluate the performance of the newly pre-processed dataset. Table 14 presents the performance of the issue sentence classifier on the dataset after performing stemming and stop word removal operations. The highest precision, recall and F-measure values for both issue and non-issue sentences across all three compared techniques are highlighted in boldface. As shown in Table 14, the performance of issue sentence classifier decreases when we perform stemming and stop word removal on the dataset.

Prior research shows that performing the stemming operation on the posts of question-answering sites may lose semantic information which results in poor performance of text classification [14]. In general, stemming and stop word removal operations negatively impact the performance of the issue sentence classifier in our case too. Thus, we did not incorporate those operations in *CAPS*.

9 Discussion

This section discusses a set of questions related to our study.

9.1 Testing Generalizability of the CAPS Results

SO posts that discuss issues of different APIs may use different words and jargon. This is because these APIs are quite different from each other. We

Table 15: Evaluation results of *CAPS* in classifying unseen APIs

Unseen API	Class	Precision	Recall	F-measure
Android	Issue	0.83	0.74	0.78
	Non-issue	0.77	0.85	0.80
Neo4j	Issue	0.92	0.86	0.89
	Non-issue	0.84	0.91	0.87
Jenkins	Issue	0.90	0.68	0.77
	Non-issue	0.69	0.90	0.78

Table 16: Comparison of different classifiers for classifying SO issue posts. P, R and F denote precision, recall and F-measure values respectively. The highest precision, recall and F-measures across all APIs are highlighted in boldface.

API	Classifier	Issue			Non-Issue		
		P	R	F	P	R	F
Android	Logistic Regression	0.91	**0.94**	0.92	**0.94**	**0.92**	0.92
	Naive Bayes	0.91	0.36	0.52	0.64	0.61	0.74
	SVM	0.87	0.93	0.89	0.93	0.89	0.89
	Decision Tree	0.90	0.92	0.91	0.92	0.91	0.91
	Random Forest	**0.92**	**0.94**	**0.93**	**0.94**	0.91	**0.93**
Neo4j	Logistic Regression	**0.88**	**0.88**	**0.88**	**0.91**	**0.91**	**0.91**
	Naive Bayes	0.72	0.46	0.56	0.67	0.86	0.75
	SVM	0.81	0.68	0.74	0.77	0.88	0.82
	Decision Tree	**0.88**	0.87	0.87	0.90	**0.91**	0.90
	Random Forest	0.87	**0.88**	0.87	**0.91**	0.89	0.90
Jenkins	Logistic Regression	0.85	**0.90**	0.87	**0.96**	0.84	0.88
	Naive Bayes	0.78	0.81	0.79	0.81	0.79	0.80
	SVM	0.81	**0.90**	0.85	**0.96**	0.90	0.84
	Decision Tree	0.86	0.88	0.87	0.89	0.87	0.88
	Random Forest	**0.92**	0.85	**0.88**	0.87	**0.93**	**0.90**

are interested to know whether it is possible to train *CAPS* using SO posts discussing issues of one API and then classify issue posts of another API. To do that we use three-fold cross-validation where the folds are not randomly created. Instead, we create one fold for each API in our datasets. We use any two folds to train the classifier and then test using the third fold. Table 15 reports the results of our experiment. We use the term *unseen API* to refer to the API under testing. The results indicate that the performance of the *CAPS* drops, which is not surprising. However, the results are not affected much which is an indication that *CAPS* can be used to classify issue-related posts of unseen APIs.

9.2 Impact of Different Classifiers

In this section, we are interested to learn the impact of different classifiers. Recall that we use the logistic regression to build the classifier to separate API issue posts from non-issue ones. It may be the case that a different classifier

provides a better result than the current one. Toward this goal, we compare the logistic regression with four other classifiers (i.e., naive Bayes, SVM, random forest and decision tree). Naive Bayes classifiers are based on the Bayes theorem that assumes features are independent of each other. SVMs are supervised learning techniques that construct a hyperplane in a high-dimensional space for classification tasks. A decision tree is a tree-like structure where internal nodes represent possible decisions based on conditions and leaf nodes represent class labels. Finally, random forests use multiple learning algorithms (such as decision trees) to provide better predictive performance.

We apply the default implementation in Weka for these classification algorithms. For the decision tree, we use the J48 class that implements the C4.5 decision tree algorithm. For the SVM, we use the SMO class that implements a sequential minimal optimization algorithm for training a support vector classifier. Table 16 shows the result for all five classifiers for all three datasets. The highest precision, recall and F-measure values for both issue and non-issue posts across all five classifiers are highlighted in boldface. Results from the study suggest that the naive Bayes classifier did not perform well. We observe that both random forest and decision tree classifiers show similar results to the logistic regression. However, both training and testing are faster for the logistic regression. While SVM shows good result for Android and Jenkins, the performance drops for the Neo4j dataset. These justify our selection of logistic regression for the SO issue posts classification.

9.3 Effects of Different Sets of Features

This section investigates how different feature sets impact the performance of *CAPS*. Recall that our classification model considers eight different sets of carefully selected features. Five of them were first proposed in our SANER'2018 paper [2] (title, body, answer, asker/answerer experience, and CRF) and three of them are added in this extended version of the work (readability, network centrality, and question info).

To understand the importance of these feature sets, we run experiments on our largest dataset, called Android. Table 17 shows the overall performance of *CAPS* when we use different sets of features. The highest precision, recall and F-measure values for both issue and non-issue posts across all feature groups are highlighted in boldface. All other settings of our technique remain unchanged. From table 17, we can see that the precision and recall values of *CAPS* reach to 0.60 and 0.56 for the issue class when we only use the title feature set. Among eight feature sets, the precision of issue class reaches to the highest value for the CRF feature set. However, the recall for the CRF is only 0.57 which is much lower than the precision value we obtain. CRF also receives the highest F-measure for both issue and non-issue classes. This indicates the importance of our CRF features.

Next, we add different sets of features to understand their impact. We observe that adding more features improves the result. The precision and recall

Table 17: Impact of different sets of features on the performance of *CAPS*. P, R and F denote precision, recall and F-measure values respectively. The highest precision, recall and F-measures across all feature dimensions are highlighted in boldface.

No.	Feature Dimension	Issue			Non-issue		
		P	**R**	**F**	**P**	**R**	**F**
1	Title	0.60	0.56	0.58	0.59	0.63	0.61
2	Body	0.63	0.58	0.60	0.61	0.66	0.54
3	CRF	0.93	0.57	0.70	0.70	0.95	0.80
4	Answer	0.91	0.31	0.46	0.59	0.95	0.73
5	Asker/Answerer Experience	0.57	0.45	0.51	0.55	0.66	0.60
6	Readability	0.60	0.66	0.63	0.63	0.56	0.59
7	Network Centrality	0.64	0.49	0.56	0.59	0.72	0.65
8	Question Info	0.82	0.60	0.69	0.69	0.87	0.77
9	Title +Body	0.65	0.61	0.63	0.64	0.68	0.66
10	Title + Body+ Answer	0.66	0.63	0.66	0.65	0.69	0.67
11	Title + Body + Answer + Asker/Answerer Experience	0.79	0.68	0.73	0.73	0.83	0.77
12	Title + Body + Answer + Asker/Answerer Experience + CRF	0.95	0.71	0.81	0.76	0.95	0.84
13	TItle + Body + Answer + Asker/Answerer Experience + Readability	0.69	0.68	0.68	0.69	0.70	0.70
14	TItle + Body + Answer + Asker/Answerer Experience + Readability + Network Centrality	0.71	0.68	0.69	0.69	0.71	0.70
15	TItle + Body + Answer + Asker/Answerer Experience + CRF + Readability + Network Centrality	0.87	0.92	0.90	0.92	0.91	0.91
16	TItle + Body + Answer + Readability + Asker/Answerer Experience + CRF + Readability + Network Centrality +QuestionInfo	**0.91**	**0.94**	**0.92**	**0.94**	**0.92**	**0.92**

values reach to 0.95 and 0.71 respectively for the issue class when we add the first five feature sets (no. 12). Those values reach to 0.76 and 0.95 for the non-issue class. Both results are higher than the result of any individual feature set. Although readability, network centrality and question info feature sets improve the result, we obtain the best result combining all feature sets. The precision and recall values reach to 0.91 and 0.94 respectively for the issue class. Thus, we obtain 11% improvement for the F-measure value compared to the result considering only first five feature sets. The precision and recall value for the non-issue class also reach to 0.94 and 0.92 respectively (F-measure value improves by 8%).

9.4 Runtime Performance

To measure the runtime performance of *CAPS*, we calculate the time required to train our model and classify the posts. The majority of the time involves annotating the sentences for training the CRF model. However, this is a one-time operation only. For 5000 SO posts, *CAPS* takes around 7s to train the CRF model and generate the features. It takes around 1.2s on average to train the discriminative classification model for classifying the issue posts. For testing each of the instance posts, it only takes 1ms on average.

10 Threats to Validity

This section summarizes the threats to the validity of our study.

First, we re-implemented the technique developed by Wang et al. [65] since the data and the technique were not publicly available at the time of writing the paper. Although we cannot guarantee that our implementation of the technique does not contain any errors, we spent considerable time in replicating and testing the technique to ensure its correctness.

Second, our dataset consists of posts concerning API issues of three different APIs written in the Java language. One can argue that the results obtained in our study may not hold for other APIs or for different languages. However, we would like to point to the fact that our selection was based on our familiarity with these APIs. To avoid bias, we considered posts from three different APIs. The features we used to develop our technique is also not specific to any particular language.

Third, we reported a set of phenomena while characterizing SO posts concerning API issues. Further study is required to identify reasons that trigger these phenomena, which remains as a future work.

11 Conclusion

Stack Overflow posts concerning API issues become a valuable source of information to API designers. Towards the goal of classifying API issue-related posts, we develop a supervised learning approach using a CRF that can classify issue-related sentences. We combine the features collected from the output of CRF to that of seven other feature categories. This leads to the development of an issue classifier, called *CAPS*. We evaluate *CAPS* using SO posts from three different API types. Results from the study reveal that *CAPS* achieves high precision and recall values for both classes. We also compare *CAPS* with three other baseline approaches and the technique outperforms all of them. We also show that by considering additional sources of information, we can improve the performance of CAPS and we also identify the most important features for the classification tasks. Furthermore, we evaluate the effectiveness of our CRF-based issue sentence classification technique against two other baseline

approaches. In both cases, the technique performs the best. Our approach also enables highlighting problematic issue sentences which can allow developers to (i) filter irrelevant sentences and focus on the API issue-related information, (ii) understand issues more quickly and (iii) be more responsive to issues submitted by users.

References

1. K. Aggarwal, F. Timbers, T. Rutgers, A. Hindle, E. Stroulia, and R. Greiner. Detecting Duplicate Bug Reports with Software Engineering Domain Knowledge. *Journal of Software: Evolution and Process*, 29(3):e1821, 2017.
2. M. Ahasanuzzaman, M. Asaduzzaman, C. K. Roy, and K. A. Schneider. Classifying Stack Overflow Posts on API Issues. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, SANER '18, pages 244–254, 2018.
3. T. Ahmed, A. Bosu, A. Iqbal, and S. Rahimi. SentiCR: A Customized Sentiment Analysis Tool for Code Review Interactions. In *Proceedings of the 32nd International Conference on Automated Software Engineering*, ASE '17, pages 106–111, 2017.
4. P. D. Allison. *Logistic Regression Using SAS: Theory and Application*. 2nd edition, 2012.
5. M. Asaduzzaman, A. S. Mashiyat, C. K. Roy, and K. A. Schneider. Answering Questions About Unanswered Questions of Stack Overflow. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 97–100, 2013.
6. A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza. Content Classification of Development Emails. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 375–385, 2012.
7. A. Bacchelli, L. Ponzanelli, and M. Lanza. Harnessing Stack Overflow for the IDE. In *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering*, RSSE '12, pages 26–30, 2012.
8. S. Baccianella, A. Esuli, and F. Sebastiani. Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining. In *Proceedings of the 7th conference on International Language Resources and Evaluation*, LREC '10, pages 2200–2204, 2010.
9. K. Bajaj, K. Pattabiraman, and A. Mesbah. Mining Questions Asked by Web Developers. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 112–121, 2014.
10. A. Baltadzhieva and G. Chrupala. Predicting the Quality of Questions on Stackoverflow. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing*, RANLP '15, pages 32–40, 2015.
11. A. Barua, S. W. Thomas, and A. E. Hassan. What Are Developers Talking About? An Analysis of Topics and Trends in Stack Overflow. *Empirical Software Engineering*, 19(3):619–654, 2014.
12. B. Bazelli, A. Hindle, and E. Stroulia. On the Personality Traits of StackOverflow Users. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, ICSM '13, pages 460–463, 2013.
13. Y. Benjamini and D. Yekutieli. The Control of the False Discovery Rate in Multiple Testing Under Dependency. *The Annals of Statistics*, 29(4):1165–1188, 2001.
14. M. W. Bilotti, B. Katz, and J. Lin. What Works Better for Question Answering: Stemming or Morphological Query Expansion? In *Proceedings of the Information Retrieval for Question Answering Workshop*, IR4QA ' 04, pages 1 – 3, 2004.
15. D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.
16. P. Bonacich and P. Lloyd. Eigenvector-like Measures of Centrality for Asymmetric Relations. *Social Networks*, 23(3):191 – 201, 2001.
17. F. Calefato, F. Lanubile, F. Maiorano, and N. Novielli. Sentiment Polarity Detection for Software Development. *Empirical Software Engineering*, 23(3):1352–1382, 2018.

18. C. Chen, S. Gao, and Z. Xing. Mining Analogical Libraries in Q&A Discussions – Incorporating Relational and Categorical Knowledge into Word Embedding. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, pages 338–348, 2016.
19. J. Cohen. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.
20. M. Coleman and T. L. Liau. A Computer Readability Formula Designed for Machine Scoring. *Journal of Applied Psychology*, 60:283–284, 1975.
21. D. Correa and A. Sureka. Fit or Unfit: Analysis and Prediction of 'Closed Questions' on Stack Overflow. In *Proceedings of the International Conference on Online Social Networks*, COSN '13, pages 201–212, 2013.
22. D. Correa and A. Sureka. Chaff from the Wheat: Characterization and Modeling of Deleted Questions on Stack Overflow. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 631–642, 2014.
23. S. Ding, G. Cong, C.-Y. Lin, and X. Zhu. Using Conditional Random Fields to Extract Contexts and Answers of Questions from Online Forums. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, ACL '08, pages 710–718, 2008.
24. Y. Fan, X. Xia, D. Lo, and A. E. Hassan. Chaff from the Wheat: Characterizing and Determining Valid Bug Reports. *IEEE Transactions on Software Engineering*, pages 1–30, 2018.
25. R. Flesch. A New Readability Yardstick. *Journal of applied psychology*, 32(3):221, 1948.
26. D. Garcia, M. S. Zanetti, and F. Schweitzer. The Role of Emotions in Contributors Activity: A Case Study on the GENTOO Community. In *Proceedings of the International Conference on Cloud and Green Computing*, CGC '13, pages 410–417, 2014.
27. S. Grant and J. R. Cordy. Estimating the optimal number of latent concepts in source code analysis. In *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation*, SCAM '10, pages 65–74, 2010.
28. R. Gunning. The technique of clear writing. 1952.
29. E. Guzman, D. Azócar, and Y. Li. Sentiment Analysis of Commit Comments in GitHub: An Empirical Study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 352–355, 2014.
30. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.
31. B. V. Hanrahan, G. Convertino, and L. Nelson. Modeling Problem Difficulty and Expertise in Stackoverflow. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work Companion*, CSCW '12, pages 91–94, 2012.
32. D. Hou and L. Li. Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers' Newsgroup Discussions. In *Proceedings of the 19th International Conference on Program Comprehension*, ICPC '11, pages 91–100, 2011.
33. M. R. Islam and M. F. Zibran. Leveraging Automated Sentiment Analysis in Software Engineering. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 203–214, 2017.
34. R. L. R. J. P. Kincaid, R. P. Fishburne Jr and B. S. Chissom. Derivation of New Readability Formulas (Automated Readability Index, Fog Count and Flesch Reading Ease Formula) for Navy Enlisted Personnel. Technical report, 1975.
35. R. Jongeling, P. Sarkar, S. Datta, and A. Serebrenik. On Negative Results When Using Sentiment Analysis Tools for Software Engineering Research. *Empirical Software Engineering*, 22(5):2543–2584, 2017.
36. G. H. M. Laughlin. SMOG Grading-a New Readability Formula. *Journal of Reading*, 12(8):639–646, 1969.
37. M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk. How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC '14, pages 83–94, 2014.
38. C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of 52nd Annual*

Meeting of the Association for Computational Linguistics, ACL '12, pages 55–60, 2014.
39. A. K. McCallum. Mallet: A machine learning for language toolkit. 2002.
40. S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.
41. M. Neuhäuser. Wilcoxon–Mann–Whitney Test. *International encyclopedia of statistical science*, pages 1656–1658, 2011.
42. N. Novielli, F. Calefato, and F. Lanubile. The Challenges of Sentiment Detection in the Social Programmer Ecosystem. In *Proceedings of the 7th International Workshop on Social Software Engineering*, SSE '15, pages 33–40, 2015.
43. N. Novielli, D. Girardi, and F. Lanubile. A Benchmark Study on Sentiment Analysis for Software Engineering Research. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 364–375, 2018.
44. M. Ortu, G. Destefanis, M. Kassab, S. Counsell, M. Marchesi, and R. Tonelli. Would You Mind Fixing this Issue? In *Proceedings of the 15th International Conference on Agile Software Development*, XP '15, pages 129–140, 2015.
45. S. Panichella, A. D. Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. How Can I Improve My App? Classifying User Reviews for Software Maintenance and Evolution. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, pages 281–290, 2015.
46. G. Petrosyan, M. P. Robillard, and R. De Mori. Discovering Information Explaining API Types Using Text Classification. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, pages 869–879, 2015.
47. D. Pletea, B. Vasilescu, and A. Serebrenik. Security and Emotion: Sentiment Analysis of Security Discussions on GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 348–351, 2014.
48. L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 102–111, 2014.
49. L. Ponzanelli, A. Mocci, A. Bacchelli, M. Lanza, and D. Fullerton. Improving Low Quality Stack Overflow Post Detection. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, ICSME '12, pages 541–544, 2014.
50. P. Raghavan, R. Catherine, S. Ikbal, N. Kambhatla, and D. Majumdar. Extracting Problem and Resolution Information from Online Discussion Forums. In *Proceedings of the 16th International Conference on Management of Data*, COMAD '10, page 77, 2010.
51. R. Robbes, M. Lungu, and D. Röthlisberger. How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 56:1–56:11, 2012.
52. M. P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26(6):27–34, 2009.
53. M. P. Robillard and R. Deline. A Field Study of API Learning Obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.
54. J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine. Exploring Methods for Evaluating Group Differences on the NSSE and Other Surveys: Are the T-test and Cohensd Indices the Most Appropriate Choices. In *Proceeding of the Annual Meeting of the Southern Association for Institutional Research*, pages 1 – 51, 2006.
55. A. Sandor, N. Lagos, N.-P.-A. Vo, and C. Brun. Identifying User Issues and Request Types in Forum Question Posts Based on Discourse Analysis. In *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, pages 685–691, 2016.
56. M. Silfverberg, T. Ruokolainen, K. Lindén, and M. Kurimo. Part-of-Speech Tagging using Conditional Random Fields: Exploiting Sub-Label Dependencies for Improved Accuracy. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, ACL '14, pages 259–264, 2014.

57. V. Sinha, A. Lazar, and B. Sharif. Analyzing Developer Sentiment in Commit Logs. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 520–523, 2016.
58. C. Sutton and A. McCallum. An Introduction to Conditional Random Fields. *Foundations and Trends in Machine Learning*, 4(4):267–373, 2012.
59. K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich Part-of-speech Tagging with a Cyclic Dependency Network. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*, NAACL '03, pages 173–180, 2003.
60. C. Treude, O. Barzilay, and M.-A. Storey. How Do Programmers Ask and Answer Questions on the Web? In *Proceedings of 33rd International Conference on Software Engineering*, ICSE '11, pages 804–807, 2011.
61. G. Uddin and F. Khomh. Automatic Summarization of API Reviews. In *Proceedings of the 32nd International Conference on Automated Software Engineering*, ASE '17, pages 159–170, 2017.
62. H. M. Wallach, I. Murray, R. Salakhutdinov, and D. Mimno. Evaluation Methods for Topic Models. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1105–1112.
63. H. Wang, C. Wang, C. Zhai, and J. Han. Learning Online Discussion Structures by Conditional Random Fields. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 435–444, 2011.
64. W. Wang and M. W. Godfrey. Detecting API Usage Obstacles: A Study of iOS and Android Developer Questions. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 61–64, 2013.
65. W. Wang, H. Malik, and M. W. Godfrey. Recommending Posts Concerning API Issues in Developer Q&A Sites. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 224–234, 2015.
66. M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer. Categorizing Bugs with Social Networks: A Case Study on Four Open Source Software Communities. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 1032–1041, 2013.
67. Y. Zhang and D. Hou. Extracting Problematic API Features from Forum Discussions. In *Proceedings of the 21st International Conference on Program Comprehension*, ICPC '13, pages 142–151, 2013.
68. M. F. Zibran, F. Z. Eishita, and C. K. Roy. Useful, But Usable? Factors Affecting the Usability of APIs. In *Proceedings of the 18th Working Conference on Reverse Engineering*, WCRE '11, pages 151–155, 2011.
69. T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010.