Automatic Mining of Opinions Expressed About APIs in Stack Overflow

Gias Uddin and Foutse Khomh

Abstract—With the proliferation of online developer forums, developers share their opinions about the APIs they use. The plethora of such information can present challenges to the developers to get quick but informed insights about the APIs. To understand the potential benefits of such API reviews, we conducted a case study of opinions in Stack Overflow using a benchmark dataset of 4522 sentences. We observed that opinions about diverse API aspects (e.g., usability) are prevalent and offer insights that can shape developers' perception and decisions related to software development. Motivated by the finding, we built a suite of techniques to automatically mine and categorize opinions about APIs from forum posts. First, we detect opinionated sentences in the forum posts. Second, we associate the opinionated sentences to the API mentions. Third, we detect API aspects (e.g., performance, usability) in the reviews. We developed and deployed a tool called Opiner, supporting the above techniques. Opiner is available online as a search engine, where developers can search for APIs by their names to see all the aggregated opinions about the APIs that are automatically mined and summarized from developer forums.

Index Terms—API, Opinion, Categorization, Review, API Aspect, API Review Mining.

1 Introduction

APIs (Application Programming Interfaces) offer interfaces to reusable software components. Modern-day rapid software development is often facilitated by the plethora of open-source APIs available for any given development task. The online development portal GitHub [34] now hosts more than 67 million repositories. We can observe a radical increase from the 10 million active repositories hosted in GitHub in 2013. Developers share their projects and APIs via other portals as well, e.g., Ohloh [72], Sourceforge [98], Freecode [31], Maven Central [96], and so on. For example, Ohloh currently indexes more than 650,000 projects, demonstrating an increase of 100,000 projects from 2015[1].

Opinions are key determinants to many of the activities related to software development. The success and adoption of mobile apps largely depend on the reviews about the apps [32]. With the advent and proliferation of online developer forums, we observed a plethora of such reviews about APIs in those forums [108], [110]. The reviews in forum posts can be different from mobile app reviews. For example, in mobile app, every *post* is a review. Contrary to the mobile app reviews, developers in forum posts are not necessarily required to share their positive or negative opinions about an API. Therefore, such API reviews (i.e., positive and negative opinions) can be found in a subset of sentences of a forum post (see below). The participants in the discussions can also be different. For example, in developer forums, software developers are the major participants, i.e., they are the users, authors, as well as reviewers of the APIs. For mobile apps, the mobile users dominate in the reviews, i.e., they do not necessarily have to be developers of mobile apps [24]. The primary focus of both types of reviews (i.e., mobile and API reviews), nevertheless, remains the same, i.e., offering opinion about app/API that may assist others to select and use the app/API [97], [23], [108]. In this paper, we focus on the API reviews extracted from developer forum posts. Specifically, we analyze API reviews posted in Stack Overflow.

The perceptions of developers about an API, and the choices they make about *whether* and *how* they should use it, may, to a considerable degree, be conditioned upon how other developers *see* and *evaluate* the API. As an example, in Figure 1, we present the screenshot of one Stack Overflow conversation containing one answer (①) and two comments(② and ③). These posts express developers' opinions about three Java APIs (Jackson [30], Gson [36], and XStream) offering JSON parsing features for Java. Each opinionated sentence in the posts contains either positive or negative sentiments towards the APIs. None of the posts contain any code snippet. The answer contains discussion about Gson with negative opinions on its usability aspect (hard to use) and positive opinions on the performance aspect (reliable and faster) of two APIs, XStream and Jackson, respectively. The first comment as a reply to the answer contains another positive opinion about Jackson, but this time it is about the usability of the API (easy to use). Later, the developer 'Daniel Winterstein' developed a new version of Gson fixing existing issues and shared his API (③). This example illustrates how developers share their experiences and insights about different aspects (e.g., performance, usability) of an API, as well as how they influence and are influenced by the opinions of others. An aspect can be an attribute (e.g., usability) or a contributing factor towards the usage or development of an API (e.g., licensing or community support). A developer looking for only code examples for Gson would have missed such important insights, i.e., sentiments expressed about different API aspects.

In fact, we observed that more than 66% of Stack Overflow posts that are tagged as "Java" and "Json" contained at least one positive or negative sentiment (see Table 1). Most of these (46%) posts do not contain any code examples. Moreover, more than 80% of the posts with a code example also contained at least one

1. We used the GitHub and Ohloh APIs to generate the statistics. Ohloh was renamed to Black Duck Software in 2014.

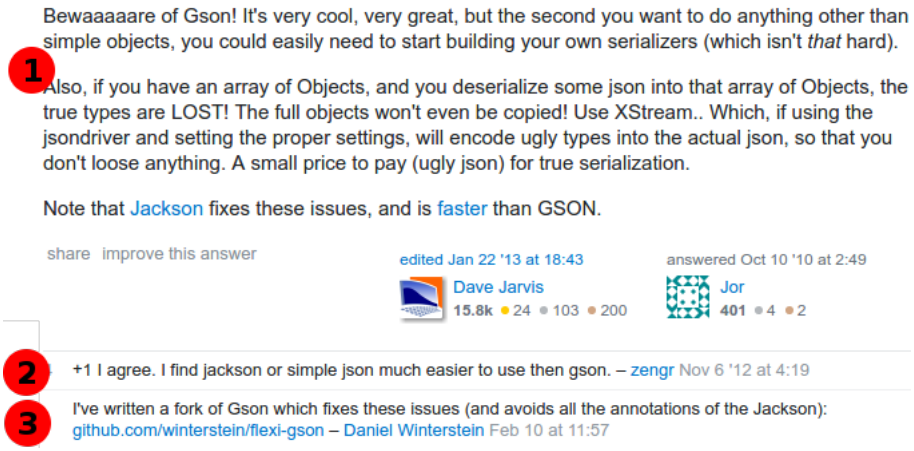

Fig. 1: Example of opinions about multiple APIs in Stack Overflow posts

TABLE 1: Sentiment coverage in Stack Overflow posts tagged as "Java" and "Json"

Posts					Sentences		
Total	**Opinionated**	**With Code**	**Code+Opinion**	**Opinion Only**	**Total**	**Positive**	**Negative**
22,733	15,206	5,918	4782	10,424	87,033	15,845	11,761

opinionated sentence. A sentence is opinionated if it consists of positive or negative sentiment[2]. Opinions are thus more prevalent than code examples in the forums, offering interesting insights about APIs, not trivially obtainable by the mere usage of the code examples. An analysis of the reviews about APIs along the two dimensions (sentiments and aspects) will offer us insights into how APIs are discussed and how such insights can be leveraged to mine and summarize API reviews. For example, an overview of the positive and negative opinions about the API 'gson' in Figure 1 can be that the criticisms against the API are mainly towards two aspects of the API: (1) **Usability:** *"I find jackson or simple json much easier to use than gson"*, (2) **Performance:** *"is faster than GSON"*. Therefore, it would be interesting to learn how developers express sentiments about API aspects in developer forums.

The diverse nature of such information available in the developer forums has motivated the development of techniques to automatically determine the underlying contexts of the forum posts, such as, automatic categorization of posts in Oracle Swing forum [43], [123], detecting similar technological terms (e.g., techniques similar to 'JSON') [19], [18], or automatically detecting problematic API features [122], and so on. For example, the automatic categorization dataset used by Hou and Mo [43] to label Swing forum posts was specific to the Swing APIs (e.g., titleBar, layout are among the labels in the dataset). Therefore, the dataset and their developed technique cannot be applied to detect aspects of more generic nature in API reviews. Intuitively, developers could be interested to learn about the performance aspect of an API, irrespective of its domain. Indeed, we found very little research that focuses on the analysis of API aspects discussed in the API reviews and how such opinions affect their decisions related to the APIs.

With a view to understand how developer express sentiments about different API aspects in the discussions of APIs in the forum posts and to facilitate automated support for such analysis, we proceeded with the following five steps in sequence:

Step 1. A Benchmark for API Reviews

To explore the impact of API aspects in the API reviews, we need a dataset containing the presence of both API aspects and reviews. The contents in the forum posts do not readily provide us such information. There is no dataset available with such information as well. Therefore, we created a benchmark dataset by manually labeling the presence of API aspects and polarity of provided opinions in all the 4,522 sentences from 1338 Stack Overflow posts. Polarity denotes whether the sentence was positive, negative or neutral. A total of 11 different human coders participated in the coding of the sentences. We leverage the benchmark dataset to understand the role of aspects in API reviews.

Step 2. Analysis of Aspects in API Reviews

We conducted a case study using the benchmark dataset to understand how API aspects are discussed in API discussions by the different stakeholders (e.g., API authors and users). We found that developers offer insights about diverse API aspects in their discussions. We found that such insights are often associated with positive and negative opinions about the APIs. Both API authors and users can be active in such opinionated discussions to suggest or promote APIs for different development needs.

Step 3. Automatic Detection of API Aspects

While the analysis of the benchmark dataset offered us interesting insights into the role of API aspects in the API reviews, such analysis is not scalable without the presence of automated techniques to categorize those reviews by aspects. We investigated the feasibility of several rule-based and supervised aspect detection

2. We identified the sentiment words by matching more than 6,000 sentiment lexicons developed by Liu et al. [44] against the words in the posts.

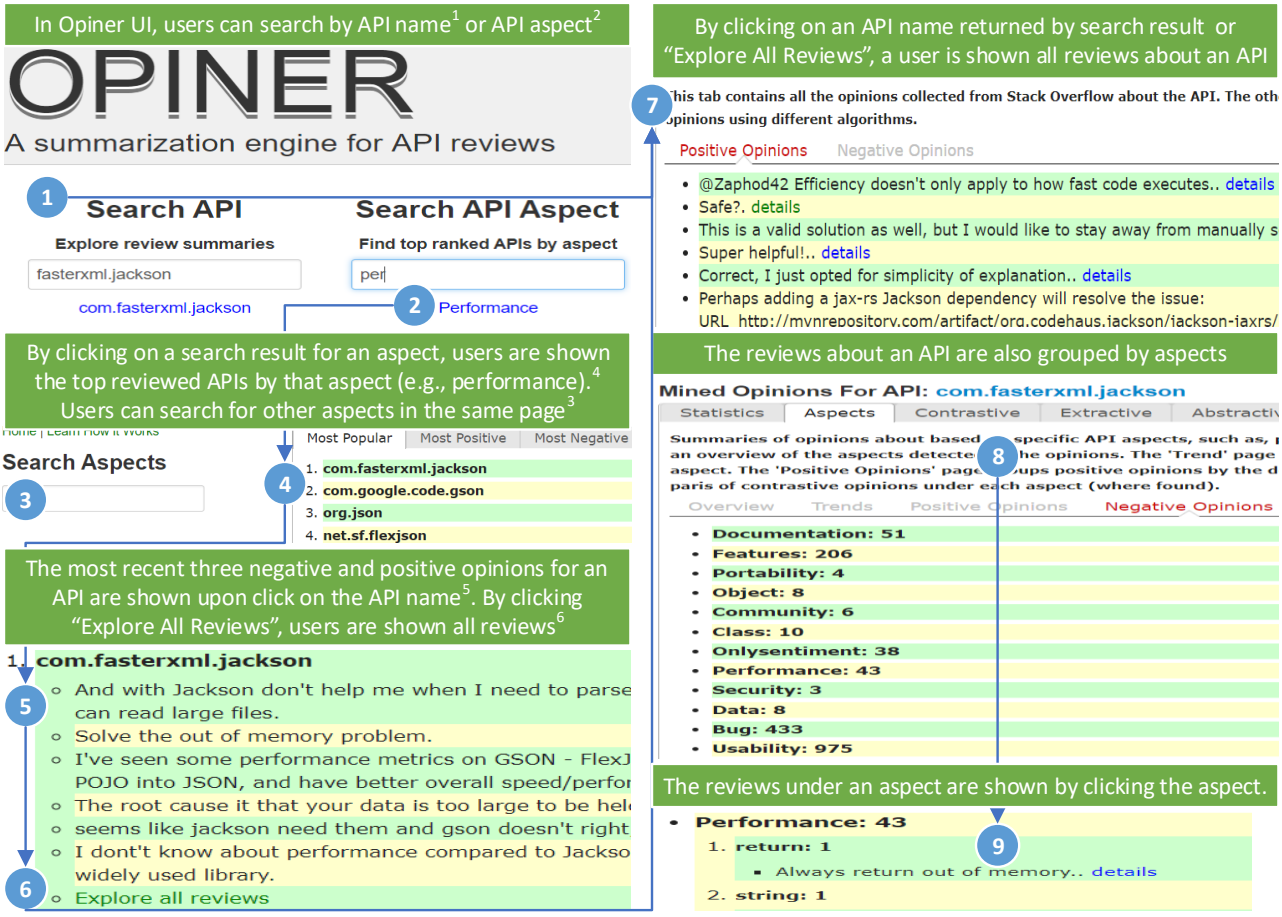

Fig. 2: Screenshots of Opiner opinion search engine for APIs.

techniques. Our supervised aspect detection techniques outperformed the rule-based techniques. We observed that we can detect aspects in the forum posts using the supervised techniques with up to 0.778 precision.

Step 4. Automatic Mining of Opinions from API Reviews

Motivated by the prevalence of API reviews in forum contents, we developed a suite of techniques to automatically mine API reviews from forum posts. First, we detect API mentions in the textual contents of the forum posts with 0.988 precision. Second, we detect opinionated sentences with upto 0.502 precision. Third, we associate opinionated sentences to the APIs about which the opinions were provided with upto 0.959 precision.

Step 5. Opiner - A Search Engine for API Reviews

We developed a tool named Opiner that supports the development infrastructure for each of the above techniques. Using Opiner's infrastructure, we can automatically mine opinions about APIs and detect potential API aspects in those opinions by automatically crawling Stack Overflow. The modular architecture of Opiner allows the easy addition of other online developer forums into the system. We have deployed Opiner as an online search engine for APIs. Using the web-based interface of Opiner, developers can search for APIs and view all the mined opinions grouped by the API aspects (e.g., performance). Since the online deployment in 30 October 2017, Opiner has so far been accessed and used by developers from 55 countries from all the continents (except Antarctica) (as of July 15, 2018 by Google Analytics).

In Figure 2, we show screenshots of the Opiner's user interface. The UI of Opiner is a search engine, where users can search for APIs by their names to look for the mined and categorized opinions about the API. There are two search options in the front page of Opiner. A user can search for an API by its name using ①. A user can also investigate the APIs by their aspects using ②. Both of the search options provide auto-completion and provide live recommendation while doing the search. When a user searches for an API by name, such as, 'jackson' in ①, the resulting query produces a link to all the mined opinions of the API. When the user clicks the links (i.e., the link with the same name of the API as 'com.fasterxml.jackson'), all of the opinions about the API are shown. The opinions are grouped as positive and negative (see ⑦). The opinions are further categorized into the API aspects by applying the aspect detectors on the detected opinions ⑧. By clicking on each aspect, a user can see the opinions about the API (see ⑨). Each opinion is linked to the corresponding post from

where the opinion was mined (using 'details' link in ⑨). When a user searches by an API aspect (e.g., performance as in ②), the user is shown the top ranked APIs for the aspect (e.g., most reviewed, most negatively reviewed in ④). For each API in the Opiner page where the top ranked APIs are shown, we show the most recent three positive and negative opinions about the API ⑤. If the user is interested to further explore the reviews of an API from ⑤, he can click the 'Explore All reviews' which will take him to the page in ⑦.

A detailed evaluation on the effectiveness of Opiner to assist developers in their development tasks was the subject of our recent papers [109], [110]. In this paper, we provide empirical evidence on the role of aspects in API reviews, and report the design and performance of the different components that we developed in Opiner to automatically mine opinions about APIs from forum posts. Specifically, we make the following contributions:

Case Study. We have produced a dataset with manual labels from a total of 11 human coders. The benchmark contains a total of 4522 sentences from 1338 posts from Stack Overflow. Each sentence is labelled as (1) the API aspect that is discussed in the sentence and (2) the polarity (positive, negative, neutral) of the sentence. We conduct a case study using the dataset to demonstrate the presence of API aspects in API reviews and the potential cause and impact of those aspects on the opinions. The dataset is available in our online appendix [111]

Algorithms. We describe the development and evaluation of a suite of **aspect detection** and **opinion mining** techniques to automatically mine and aggregate opinions about APIs from Stack Overflow.

Opiner. We develop a framework and an accompanying tool to automatically mine and categorize opinions about APIs. The Opiner website with the opinion search engine is available online for usage at: http://opiner.polymtl.ca.

Paper Organization. We report our work following the guidelines for the reporting of empirical results as recommended by Singer [94]. In Section 2, we discuss the concepts necessary to understand our research. We summarize the related work in Section 3 by comparing the difference and similarity of the study and techniques presented in this paper against these previous works. In Section 4, we present the methodology of our empirical study by introducing the research questions that we answered in the paper. In Section 5, we introduce the benchmark that we created to investigate the presence of API aspects in the opinions of forum posts. We examine the nature and diversity of API aspects in forum posts by analyzing the benchmark in Section 6. We then describe the techniques we developed to facilitate the automatic mining and categorization of opinions from API reviews in Sections 7, 8. We discuss threats to the validity of our work in Section 9. We conclude the paper in Section 10.

2 Background and Definitions

This research borrows concepts from opinion analysis and software engineering. In this section, we present the major concepts upon which our study was founded.

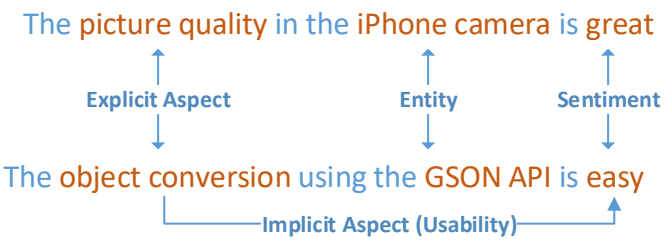

Fig. 3: The presence of aspect and sentiment in reviews about entities (Camera and API)

2.1 API

In this paper, we investigated our API review mining techniques for both open-source and official Java APIs. As such, we analyzed Stack Overflow posts tagged as "Java" where Java APIs are mostly discussed. However, the analysis and the techniques developed can be applicable for any API. In particular, we adopt the definition of an API pioneered by Martin Fowler. An API is a "set of rules and specifications that a software program can follow to access and make use of the services and resources provided by its one or more modules" [118]. An API is identified by a name. An API consists of one or more modules. Each module can have one or more source code packages. Each package can have one or more code elements, such as classes, methods, etc. For the Java official APIs available through the Java SDKs, we consider an official Java package as an API. Similar format is adopted in the Java official documentation (e.g., the `java.time` package is denoted as the Java date APIs in the new JavaSE official tutorial [73]).

2.2 Opinion

Bing Liu, in his book [57], defined opinion as:

Opinion. "An opinion is a quintuple $<e_i, a_{ij}, s_{ijkl}, h_k, t_l>$, where e_i is the name of the entity, a_{ij} is an aspect of e_i, s_{ijkl} is the sentiment on aspect a_{ij} of entity e_i, h_k is the opinion holder, and t_l is the time when the opinion is expressed by h_k".

The sentiment s_{ijkl} is positive or negative. Both entity (e_i) and aspect (a_{ij}) represent the opinion target. An aspect about an entity in an opinion can be about a property or a feature supported by the entity. Aspects can be of two types [57]:

Explicit Aspect.
When the aspect is about a specific feature of the entity. Consider the sentence: "The picture quality of the iPhone camera is good". Here 'picture quality' is an explicit aspect of the entity 'iPhone camera'. Explicit aspects are more domain specific. For example, 'picture quality' can be an aspect of a camera, but not car or a house.

Implicit Aspect.
When the aspect is about a specific property of the entity. Consider the sentence: "The new iPhone does not easily fit in one hand." Here the opinion is about the implicit aspect, *size* of the entity 'iPhone'. Implicit aspects can be generalizable

across domains. For example, 'size' can be an aspect of a camera, car, or a house, and so on.

We can define opinions expressed about APIs similarly. An aspect in an opinion about an API can be about a property (e.g., performance) or a feature of the API. Consider the two example reviews in Figure 3. The review at the top is about an entity 'iPhone camera'. The review at the bottom is about another entity, the 'GSON API'. Both reviews are positive. Both reviews have explicit aspects, the top praising the *picture quality* aspect of the entity *iPhone camera* and the bottom praising the *object conversion* feature for JSON parsing of the *Google GSON* API. In addition, the bottom review about the GSON API also praises the 'usability' property of the GSON API ('easy to use'), which is an implicit aspect.

• **Opinion Unit of Analysis.** Opinions about an entity can be analyzed in three units: document, sentence, clause. A document can have more than one sentence. A sentence can have more than one clause. Consider the following opinion: " I use the Google GSON API to convert Java objects to JSON, it is easy to use." This sentence has two clauses: (a) 'I use the Google GSON API to convert Java objects to JSON', and (b) 'it is easy to use'.

A document in a developer forum post can be the entire post. For example, in Figure 1, there are three posts (one answer and two comments). Each of the posts can be considered as a separate document. In this paper, we analyze the reviews about APIs at the sentence level. For example, in Figure 1, the first post (①) contains six sentences. The three posts in Figure 1 contain a total of eight sentences. Our choice of analyzing opinions at the sentence level rather than at the post level is based on the observation that a post can contain both positive and negative opinions and information about more than one API. For example, the first post in Figure 1 contain opinions about three different APIs. The opinions contain both positive and negative sentiments. This granularity also allows us to investigate the presence of the API aspects in the reviews in a more precise manner. For example, by analyzing each sentence, we can find that the opinions in the first post of Figure 1 are about two different API aspects (usability and performance).

Our choice of analyzing opinions at the sentence level rather than at the clause-level is based on the observation that the identification of clauses in a sentence from developer forums can be challenging. Clause-level sentiment detection relies on proper division of clauses in a sentence, which depends on the grammatical structure in the sentence [56]. Sentences posted in Stack Overflow can be grammatically incorrect. The presence of non-English vocabulary in the sentences can also confuse a linguistic dependency parser that is used to identify the clauses [93], [104]. The analysis of sentiments in developer forums at the clause-level is our future work.

2.3 Sentiment Analysis

Sentiment analysis focuses on the detection of subjectivity and polarity in a given input. According to Bing Liu [57], polarity detection in a given input (e.g, a sentence) focuses on the identification of three types of polarity: (1) Positive, (2) Negative, and (3) Neutral. In contrast, a subjective sentence expresses personal beliefs, feelings or views. *Subjectivity* detection in a sentence refers to the labeling of a sentence to one of two categories: (1) Subjective, and (2) Objective. An objective sentence is considered as a *neutral* sentence, i.e., it does not exhibit any kind of sentiments. Subjectivity detection can thus be considered as a first step towards the detection of polarity (e.g., to discard the objective sentences) [56]. In this paper, we focus on the detection of polarity in sentences. In the rest of paper, we denote sentiment labeling/detection as the labeling/detection of the three types of polarity (i.e., positive, negative, and neutral).

• **Sentiments vs Emotions in Opinion Analysis**. Emotion detection focuses on a finer-grained detection of the underlying expressions carried over by the sentiments, such as, anger, frustration. Gerrod Parrott identified six prominent emotions in social psychology [80]: (1) Joy, (2) Love, (3) Surprise, (4) Anger, (5) Sadness, and (6) Fear. This paper focuses on the analysis of sentiments in API reviews, because sentiment detection is predominantly used in other domains (e.g, cars, movies) to mine and summarize opinions about entities [57], [78]. The analysis of emotions in API reviews is our immediate future work.

2.4 API Aspect

We conducted two surveys to understand the API aspects about which developers prefer to explore opinions in the forums posts. A total of 178 professional software developers from Github and Stack Overflow participated in the surveys [105], [106]. In a series of open and closed questions, we asked developers of their preference of API aspects in the opinions about APIs. We analyzed the responses to the open-ended question using open coding. We created all of the "cards", splitting the responses for eight open-ended questions. This resulted into total 1121 individual quotes; each generally corresponded to individual cohesive statements. In further analysis, two authors and a senior software engineer in the Industry acted as coders to group cards into themes, merging those into categories. By analyzing the responses to the open and closed questions, we identified the following 11 major aspect categories.

Performance:
The opinion is about the performance of an API. For example, "How well does the API perform?". Related themes emerged in the open codes were scalability, faster or slower processing, and so on.

Usability:
The opinion is about how usable an API is. Related themes emerged in the responses were the design principles of an API and and their impacts on the usage of the API. As Stylos et al. [100] observed, an API can produce the right output given an input, but it still can be difficult to use.

Security:
The opinion is about the security principles, the impact of security vulnerabilities and the understanding of the security best practices about an API. For example, "How secure is the API in a given networked setting?". Related themes emerged from the open codes were authentication, authorization, encryption, and so on.

Documentation:
The opinion is about how the API is documented in dif-

ferent development resources. For example, "How good is the official documentation of the API?" Developers cited the necessity of good official documentation, as well as diverse informal documentation sources, such as, informal documents (e.g., developer forums), blog posts, and so on.

Compatibility:
The opinion is about how an API works within a given framework. Compatibility issues were also raised during the versioning of an API with regards to the availability of specific API features. For example, "Is the API compatible with the Java Spring framework?".

Portability:
The opinion is about how an API can be used across different operating systems, e.g., Linux, Windows, Mac OS, OSX, Android, and so on. For example, "Can the API be used in the different Linux distros?". The themes emerged during the open coding were related to the specificity of an API towards a target operating system environment.

Community:
The opinion is about the presence and responsiveness of the people using and supporting an API. For example, "How is the support around the community?" Related community sources discussed in the responses were mailing lists, developer forums, API authors and users responding to queries related to the usage of the API.

Legal:
The opinion is about the licensing and pricing issues of an API. For example, "What are licensing requirements?". Related legal terms raised in the responses were open vs closed source versions of an API, the free vs paid versions of an API, the legal implication of the different open source licenses within an enterprise setting.

Bug:
The opinion is about the presence, proliferation, fixing, or absence of specific bugs/issues of an API. For example, "Does the API lose information while parsing?".

Other General API Features:
Opinions about any feature or usage of an API. While the above implicit aspects about an API are discussed with regards to the usage of specific API features, developers can also offer opinions about an API feature in general. For example, "The object conversion in the GSON API is great".

Only Sentiment:
Opinions about an API without specifying any aspect or feature. For example, "GSON API is great."

A manuscript discussing the results of the two surveys is currently undergoing a minor revision as a journal article at the TSE [107].

As we note above, nine out of 11 categories are implicit aspects. We observed that the respondents in our survey were interested in the analysis of implicit aspects in API reviews. The reason is that such aspects can be generalizable across APIs from diverse domains. For example, we can expect to see opinions about the 'usability' aspect of APIs offering diverse features (e.g., graphics processing vs JSON parsing). Therefore, such aspects can be used to compare competing APIs. For example, any two competing APIs can be compared against each other based on how usable they are. For the 'Other General API Features' category, developers were mostly interested in reviews about the specific 'usage' of the API (e.g., usage of an API in a particular way). The category 'Only Sentiment' is used as the default labeling of a sentence that expresses mere sentiments towards an API without discussing specific attributes/features of the API. In this paper, we focus on the analysis and detection of the above 11 categories: Nine implicit aspects, Explicit aspects (i.e., Other General API Features), and Only Sentiment.

3 Related Work

We previously published two papers describing the algorithms and architecture used in Opiner to *summarize* API reviews [108], [110]. This paper discusses the techniques developed in Opiner to *mine* API reviews. In the Opiner processing engine, the summarization of API reviews takes place only *after* the reviews are mined. In addition to the mining techniques, this paper differs from [108], [110] by offering following two contributions:

1) **Benchmark (Section 5)**. We describe the creation of a benchmark dataset. We leverage the benchmark to design and evaluate algorithms to mine opinions about APIs (Section 8).
2) **Case study (Section 6)**. We leverage the benchmark to report the prevalence of aspects and sentiments in API reviews.

• **Improvements over [108].** This paper improves the analysis of static aspect detection process in [108] as follows (see Section 7):

1) We investigate both rule-based and supervised detection techniques. We observed that the supervised techniques outperformed the rule-based techniques.
2) We extend the analysis of the supervised classifiers in [108] by investigating their performance on both *balanced* and *imbalanced* datasets. We conduct this to understand whether and how we could improve the accuracy of the classifiers, e.g., by adding more data for each aspect into the benchmark.

• **Improvements over [110].** We only briefly outline the Opiner mining engine architecture in [110], because it was a tool demo paper (Steps 1-5 in Section III-A, Page 4). In this paper, we describe in detail each algorithm in the Opiner API review mining engine. In addition, we report the detailed evaluation of each mining algorithm (see Section 8).

Other related work can broadly be divided into four categories: (1) Analysis of Developer forums, (2) Automatic traceability recovery of APIs in developer forums, (3) Sentiment analysis in software engineering, and (4) Content categorization in software engineering artifacts. We describe the related work below.

3.1 Analysis of Developer Forums

Online developer forums have been studied to find dominant discussion topics [9], [88], to analyze the quality of posts and their roles in the Q&A process [14], [8], [52], [26], [113], [50], to analyze developer profiles (e.g., personality traits of the most and low reputed users) [10], [33], or to determine the influence of badges in Stack Overflow [2]. Several tools have been developed utilizing the knowledge from the forums, such as autocomment assistance [120], collaborative problem solving [16], [102], and

tag prediction [99]. Parnin et al. [79] and Kavaler et al. [50] analyzed the relationship between API usage and their related Stack Overflow discussions. Both studies found a positive relationship between API class usage and the volume of Stack Overflow discussions. More recent research [55], [37] investigated the relationship between API changes and developer discussions.

As we noted in Section 1, we are aware of no research that investigated the prevalence and impact of sentiments and API aspects in the API reviews of online forum discussions. As a first step towards filling this gap, in this paper, we present the results of a case study in Section 6. The study uses our benchmark dataset that we created to conduct the study (see Section 5). We observed that developers frequently offered opinions about diverse API aspects in those discussions. Our findings corroborate the findings in mobile app reviews by Pagano and Maalej [77]. They observed that the download frequency of an app has a positive correlation with the positive feedback provided to the app. Intuitively, the development of automated crowd-source review analysis tools to assist developers by mining such reviews can be useful. In a recent paper, Gómez et al. [35] proposed a new version of mobile app store (App store 2.0) that can recommend actionable insights to the app developers by exploiting the app reviews. The findings and our study offer insights and motivations into the development of our tool Opiner, by automatically mining and categorizing API reviews from Stack Overflow.

3.2 Sentiment Analysis in Software Artifacts

The development of Opiner was motivated by the findings of the impact of sentiments in software teams and repositories. Novielli et al. [69] analyzed the sentiment scores from the StentiStrength in Stack Overflow posts. They observed that developers express emotions towards the technology, not towards other developers. Ortu et al. [74] observed that bullies are not more productive than others in a software development team. Mäntylä et al. [62] correlated VAD (Valence, Arousal, Dominance) scores [116] in Jira issues with the loss of productivity and burn-out in software engineering teams. Pletea et al. [81] observed that security-related discussions in GitHub contained more negative comments. Guzman et al. [38] found that GitHub projects written in Java as well as the comments posted on Mondays have more negative comments, while the developers in a distributed team are more positive. Guzman and Bruegge [39] summarized emotions expressed across collaboration artifacts in a software team (bug reports, etc.) using LDA [12] and sentiment analysis. Murgia et al. [66] labelled comments from Jira issues using Parrot's framework [80]. Jongeling et al. [48] compared four sentiment tools on comments posted in Jira (SentiStrength [103], Alchemy [45], Stanford NLP [95], and NLTK [68]). While NLTK and SentiStrength showed better accuracy, there were little agreements between the two tools. These findings indicate that the mere application of an off-the-shelf tool may be insufficient to capture the complex sentiment found in the software artifacts.

The observed shortcomings in cross-domain sentiment detection tools motivated us to investigate domain-specific sentiment detection algorithms for API reviews (see Section 8). In Opiner, we experimented sentiment detection in our benchmark dataset using two algorithms. The first algorithm is the Dominant Sentiment Orientation (DSO) as originally proposed by Hu and Liu [44]. We adapted DSO for software engineering by including sentiment words specific to the domain of software engineering. We refer to this adaption as OpinerDSO. Our second algorithm is a fusion of Sentistrength [103] and our adaptation of DSO. The purpose was to improve the overall sentiment detection accuracy in Opiner using the fusion. We refer to this fusion as OpinerDSOSenti.

Sentistrength [103] has been used to detect sentiments in various software engineering projects [74]. Given as input a sentence, Sentistrength produces a numeric value by analyzing the presence of sentiment words in the input sentence. A positive value corresponds to an overall positive opinionated sentence and a negative value corresponds to a negative opinion. The algorithm DSO was originally proposed by Hu and Liu [44] to mine and summarize reviews of computer products. It was later adopted to mine customer reviews about local services, e.g., food, restaurants, etc. [11]. Similar to Sentistrength, DSO also relies on sentiment lexicons in the input sentence. Sentistrength is semi-supervised, while DSO is rule-based. The flexibility in DSO allowed us to incorporate domain-specific sentiment vocabularies into its sentiment detection process.

We developed the Opiner sentiment detection algorithms in January 2017. Since then three more sentiment detection tools have been developed and published online to address the needs of software engineering: SentistrengthSE [47], Senti4SD [13], and SentiCR [1]. SentiCR is supervised polarity classifier that is trained on a dataset of 1600 reviews from Gerrit. SentiStrengthSE was developed on top of SentiStrength by introducing rules and sentiment words specific to the domain of software engineering [47]. Senti4SD [13] introduces a supervised polarity classifier that is trained on a dataset of 4000 posts (questions, answers, and comments) from Stack Overflow. Each post was manually annotated for sentiment polarity. The classifier is trained to detect three types of polarity: positive, negative and neutral. Senti4SD uses linguistic features, such as ngrams, sentiment lexicons and semantic features based on word embeddings. In a recent study by Novieli et al. [71], Senti4SD was found to be a better performer than SentistrengthSE and SentiCR in three benchmark datasets.

3.3 Traceability Recovery of APIs in Developer Forums

As we noted in Sections 1,2, we detect opinionated sentences about APIs. In a forum post, not all the opinions can be associated to an API. Therefore, we need to detect an API name in the textual contents of a forum post where the opinions are found. In software engineering research, traceability recovery often involves two distinct steps [28], [101]: 1) detection of an entity (e.g., a code term), and 2) linking the entity to a software artifact (e.g., an API). The traceability recovery of an API name from the texts can be different from the traceability recovery of code terms. Consider the following text: "I like the Jackson API. It offers more improved performance than the JsonArray of GSON". Here, JsonArray is a code term, but Jackson and GSON are two API names. We observed that opinions in forum texts are mostly provided by indicating the API name, not the code terms. We thus developed an algorithm to automatically detect API names in the forum texts and then, to automatically link each such name to the right API. For example, for the above example, our algorithm

would associate Jackson to the com.fasterxml.jackson API and GSON to the com.google.code.gson API.

Related work in API traceability recovery mostly focused on the detection and linking of code terms. Recodoc [28] resolves code terms in the formal documentation of a project to its exact corresponding element in the code of the project. Baker [101] links code terms in the code snippets of Stack Overflow posts to an API/framework whose name was used to tag the corresponding thread of the post. ACE [85] resolves Java code terms in forum textual contents of posts using island grammars [65]. Bacchelli et al. [5] developed Miler to determine whether an email posted in a developers' mailing lists contains the mention of a given source code element. They compared information retrieval (IR) techniques (LSI [63] and VSM [61]) against lightweight techniques based on regular expressions. Prior to Miler, LSI was also used by Marcus et al. [63], and VSM was compared with a probabilistic IR model by Antoniol et al. [3]. Tools and techniques have been proposed to leverage code traceability techniques, e.g., linking software artifacts where a code term is found [46], associating development emails with the source code in developers' IDE [4], recommending posts in Stack Overflow relevant to a given code context in the IDE [82]. Opiner extends this line of research by proposing a heuristic-based technique to detect API names in the textual contents of the forum posts and to associate those mentions to the opinions provided about the APIs.

3.4 Content Categorization in Software Engineering

In Opiner, we categorize the opinionated sentences about an API into the different API aspects (e.g., performance, usability, etc.) (see Sections 2, 7). The categorization of various software engineering artifacts has been studied in numerous research approaches, such as identifying knowledge patterns and valuable and indispensable text fragments in API official documentation [59], [86], finding similar technological terms in Stack Overflow [20], developing a thesaurus of software-specific terms by mining Stack Overflow [21], leveraging the Stack Overflow contents to automatically translate Stack Overflow posts from English to Chinese [22], clustering Stack overflow posts with similar topics using LDA [124], categorizing the contents of development emails [6], identifying features of mobile app reviews [40], automatically categorizing mobile app reviews into categories, such as bug reports, feature requests, user experiences, and text ratings [58], for examples.

As we noted in Section 1, Hou and Mo [43] proposed techniques to automatically categorize API discussions in the online Oracle Java Swing forums. First, they manually labelled 1035 API discussions in the forum. They then applied the Naïve Bayes classifier to automatically categorize those discussions. They achieved a maximum accuracy of 94.1%. In a subsequent work, Zhou et al. [123] proposed an algorithm to improve accuracy of the automatic categorization. We categorize the opinions about APIs by 11 API aspects, such as, performance, usability, other general features, and so on. We also identify *dynamic aspects* in the sentences labelled as 'other general features'. Unlike Hou and Mo [43], we focused on the usage of the categorization to facilitate the summarization of opinions about APIs and to offer comparisons of the APIs based on the aspects. Thus, our API aspects can be considered as more *generic* than the labels of Hou and Mo [43].

Zhang and Hou [122] identified problematic API features in the discussion Stack Overflow posts using natural language processing and sentiment analysis. To detect problematic API features, they detected sentences with negative sentiments. They then identified the *close* neighbors of the sentences and consider those as units of problematic API features. They applied their technique on online Swing forum posts, i.e., all of those posts were supposed to be related to the Java Swing API. Rupakheti and Hou [89] investigated the design of an automated engine that can recommend the use of an API based on the formal semantics of the API. The negative opinions we mine about APIs can be considered as denoting to *problematic API features*. However, unlike Zhang and Hou [122], we do not empirically validate the cause and impact of the problems. Instead, we present all the positive and negative opinions about an API to the developers, so that they can make a decision by analyzing the causes by themselves.

4 Research Method

In Figure 4, we show the five steps we followed in this study that involves the creation of a benchmark with manual labels of 4522 sentences from a total of 11 human coders (step 1), a case study that investigates the prevalence of API aspects and sentiments in the benchmark (step 2), the development of a suite of algorithms to automatically mine and categorize API reviews from forum posts (steps 3, 4), and the development and deployment of our proof of concept tool, Opiner (step 5).

We analyze and report the performance of the each of the techniques developed and experimented as part of the algorithms development steps. We report the performance of the techniques using three standard measures of information retrieval: precision (P), recall (R) and F1-measure ($F1$)(Equations 1 - 3).

$$P = \frac{TP}{TP + FP} \tag{1}$$

$$R = \frac{TP}{TP + FN} \tag{2}$$

$$F1 = 2 * \frac{P * R}{P + R} \tag{3}$$

TP = Number of true positives, FN = Number of false negatives, FP = Number of false positives, TN = Number of true negatives.

We describe the details of the benchmark creation process in Section 5. We approach and report the case study and the algorithms development (steps 2 - 4) by answering six research questions (three for the case study and three for the algorithms development). We present the research questions below.

4.1 Case Study (Step 2)

We answer the following three questions in Section 6.

RQ1. How are the API aspects discussed in API discussions?

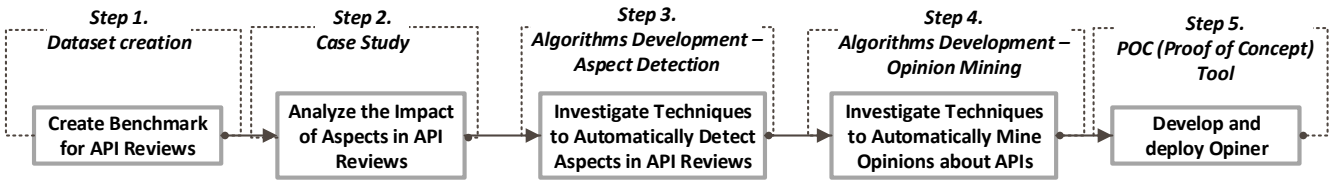

Fig. 4: The five steps followed in this study. First step is data collection. Last step is the development of our POC tool (Opiner).

• **Motivation.** By analyzing the presence of API aspects across the diverse domains, we can understand whether and how all such API aspects are prevalent across domains. Blair-Goldensohn et al. [11] of Google research observed that the opinions in a domain mostly followed the Zip's law [83] in that certain aspects are much more discussed than others. Zip's law is an empirical law that states that given a corpus of natural language texts, the frequency of a given word is close or inversely proportional to its rank in the word frequency table. Depending on the type of domain, a lower ranking can denote either a lower importance (e.g., stopwords, such as 'the' in texts) or higher importance (e.g., most prevalent topic in a discussion). Blair-Goldensohn et al. [11] categorized large corpus of reviews about local services (e.g., hotels, restaurants, barber shops, etc.). They found that discussions about food are much more prevalent in reviews about restaurants than other aspects (e.g., quality of service). If we observe similar phenomenon for API reviews, we can prioritize the analysis of more *important* (e.g., more frequent) API aspects over other aspects.

• **Approach.** For each aspect, we first count the total number of sentences labelled as aspects in the benchmark. This count shows the overall distribution of the aspects and helps us to compare whether some aspects are more prevalent than others. We further compare how an aspect is distributed across the domains. Such analysis help us determine whether the aspects are generic enough to be representative across the domains.

For each aspect (except OnlySentiment), we further identified the themes associated with each aspect in the opinions (e.g., design, usage and other themes in the usability aspect). The opinions labelled as 'OnlySentiment' contained simply sentiments (e.g., thank you, etc.), and thus were not included in our theme-based analysis. For each aspect, we identified the major themes that contributed to the labeling of the sentences to the aspect. We identified the themes related to a given aspect using a card sorting approach as follows: (1) We labelled each sentence based on the themes related to the aspect that was discussed in the sentence. (2) Once all the sentences of a given aspect are labelled, we revisited the themes and grouped those into hierarchical themes. (3) We did step 2 until no further themes emerged. Our card sorting technique was adopted from similar qualitative analyses in software engineering [27].

RQ2. How do the API aspects relate to the opinions provided by the developers?

• **Motivation.** Given that the usage of an API can be influenced by factors related to the specific attributes of an API, we believe that by tracing the positive and negative opinions to specific API aspects discussed in the opinions, we can gain deeper insight into the contextual nature of the provided opinions. Tools can be developed to automatically detect those aspects in the opinions to provide actionable insights about the overall usage and improvement of the API.

• **Approach.** We associated the aspects in the benchmark to the opinionated sentences and analyzed the correlation between those based on three distributions: (a) The overall number of positive and negative opinionated sentences associated with each aspect, and (b) For each aspect with opinions, the distribution of those opinions across the different domains. (c) For each theme associated to a given aspect, the distribution of the opinions across the aspects. We answer the following questions: (1) How are the opinions about APIs distributed across aspects? (2) How are the opinions about aspects distributed across the themes?

RQ3. How do the various stakeholders of an API relate to the provided opinions?

• **Motivation.** An API can be associated with different stakeholders (e.g., API authors, users, organizations developing the API, etc.). Depending on the usage needs, the interaction between the stakeholders and the analysis of the opinions they provide can offer useful insights into how positively or negatively the suggestions are received among the stakeholders. Such insights thus can help in the selection of the API (e.g., the promotion of an API by the API author can persuade developers to use it).

• **Approach.** We analyze the interactions among the stakeholders participating in the discussions about APIs in our benchmark dataset around three dimensions:

(1) **Aspect**: What API aspects are discussed by the different types of stakeholders?
(2) **Signal**: What factors motivated the stakeholder to provide the opinions?
(3) **Action**: What were the actions of the stakeholders in response to an API suggestion or promotion?

We identify the stakeholders, signals and actions in the benchmark as follows.

Stakeholder: We manually examined each sentence in the benchmark and labelled it based on the type of person who originally wrote the sentence. For example, we labelled the sentence as coming from the author of an API, if the sentence contained a disclosure ("I am the author").

Signal: We identified the following signals in the opinions: (1) **Suggestion** about an API, e.g., "check out gson". We used the following keywords as cues: 'check out', 'recommend',

'suggest', 'look for'. (2) **Promotion:** a form of suggestion but from the author/developer of an API. To qualify as an author, the opinion-provider had to disclose himself as an author in the same sentence or same post.

Action: We identified the following actions in the opinions: (1) **Used**: the opinion was from a user who explicitly mentioned that he had used the API as suggested. (2) **Liked**: the opinion does not show any explicit mention of API usage, but showed support towards a provided suggestion or promotion. We only considered opinionated sentences from the opinion provider who was different from the suggestion or promotion provider. For cues, we considered all of the positive comments to an answer (when the suggestion/promotion was provided in an answer), or all the positive comments following a comment where the suggestion/promotion was given. (3) **Unliked**: the opinion-provider did not agree with the provided suggestion or promotion. For cues, we considered all of the negative comments to an answer (when the suggestion/promotion was provided in an answer), or all the negative comments following a comment where the suggestion/promotion was given.

We report the findings using the following metrics:

$$\text{Suggestion Rate} = \frac{\#\text{Suggestions (S)}}{\#\text{Posts (T)}}, \tag{4}$$

$$\text{Promotion Rate} = \frac{\#\text{Promotion (P)}}{\#\text{Posts (T)}}, \tag{5}$$

$$\text{Liked Rate} = \frac{\#\text{Liked}}{\#\text{S} + \#\text{P}} \tag{6}$$

$$\text{Unliked Rate} = \frac{\#\text{Unliked}}{\#\text{S} + \#\text{P}} \tag{7}$$

4.2 Algorithm Development - Aspect Detection (Step 3)

We answer the following question in Section 7.

RQ4. How can we automatically detect aspects in API reviews?

• **Motivation.** While our manually labelled dataset of API aspects can be used to analyze the role of API aspects in API reviews, we need automated aspect detectors to facilitate large scale detection and analysis of API aspects in the forum contents.

• **Approach.** We leverage the benchmark dataset to investigate the feasibility of both rule-based and supervised aspect detection classifiers. We develop one classifier for each aspect. Each classifier takes as input a sentence and labels it as 1 (i.e., the aspect is present in the sentence) or 0 (i.e., otherwise).

4.3 Algorithms Development - Opinion Mining (Step 4)

We answer the following questions in Section 8.

RQ5. How can opinions be detected in API reviews?

• **Motivation.** The second step to mining opinions about APIs is to detect opinionated sentences in the forum posts. An opinionated sentence about an API exhibits positive or negative sentiments towards the API. It is observed that sentiment detection techniques from other domains do not perform well for the domain of software engineering [49]. In 2018, Lin et al [53] observed that sentiment detection tools developed and used in software repositories show inconsistent performance. For our project to mine opinionated sentences about APIs, we need a technique that can show good precision and recall to detect opinionated sentences from API reviews.

• **Approach.** In 2017, we started developing the Opiner sentiment detection engine based on a similar observation as Lin et al [53] that cross-domain sentiment detection tools do not perform satisfactorily for the domain of software engineering. The algorithm incorporates the output of two cross domain sentiment detection techniques with cues taken from sentiment vocabularies specific to the domain of API reviews (e.g., thread-safety). In Section 8.1, we describe the algorithm and compare its performance against other techniques.

RQ6. How can opinions be associated to a APIs mentioned in the forum posts?

• **Motivation.** Once opinionated sentences are found in forum posts, the next step is to associate those opinionated sentences to the *correct* API mentioned in the forum posts. A satisfactory accuracy in the association is required, because otherwise we end up associating wrong opinions to an API.

• **Approach.** We collect all the API information listed in the online Maven Central and Ohloh. We detect potential API mentions (name and url) in the *textual contents* of the forum posts using both exact and fuzzy name matching. We link each such potential API mention to an API in our database or label it as false when it is not a true API mention. We then associate an opinionated sentence to a detected API mention using a set of heuristics. We report the performance of the heuristics using our benchmark dataset.

5 A Benchmark For API Reviews

To investigate the potential values of opinions about an API and the above aspects, we need opinionated sentences from forum posts with information about the API aspects that are discussed in the opinions. In the absence of any such dataset available, we created a benchmark with sentences from Stack Overflow posts, each manually labelled based on the presence of the above aspects in the sentence. In addition, to investigate the performance of sentiment detection on API reviews, we further labelled each sentence based on their polarity (i.e., whether a sentence is neutral or it contains positive/negative sentiment). In total, eight different coders participated in the labeling of the sentences. Therefore, each sentence in our benchmark dataset has two types of labels:

Aspect. One or more API aspects as discussed in the sentence.

Polarity. The sentiment expressed in the sentence. A sentiment can be positive or negative. When no sentiment is expressed, the sentence is labelled as neutral. A sentence can be only one of the three polarity types.

TABLE 2: The Benchmark (Post = Answer + Comment + Question)

Domain	Tags	Posts	Answers	Comments	Sentences	Sentences/Threads
serialize	xml, json	189	68	113	531	66.38
security	authentication, cryptography	105	34	63	399	49.88
utility	io, file	297	85	204	1000	125
protocol	http, rest	126	49	69	426	53.25
debug	loggging, debugging	138	41	89	471	58.88
database	sql, nosql	128	38	82	362	45.25
widgets	awt, swing	179	67	104	674	84.25
text	nlp, string	83	40	36	265	33.13
framework	spring, eclipse	93	36	49	394	49.25
Total		1338	458	809	4522	Average = 62.81

In Table 2, we show the summary statistics of the benchmark. The benchmark consisted of 4,522 sentences from 1,338 Stack Overflow posts (question/answer/comment). Each sentence was manually labelled by at least two coders. The 1,338 posts were collected from 71 different Stack Overflow threads. The threads were selected from 18 tags. We applied the following steps in the benchmark creation process:

1) **Data Collection**: We collected posts from Stack Overflow using a two-step sampling strategy (see Section 5.1)
2) **Data Preprocessing**: We preprocessed the sampled posts and developed a benchmarking application to label each sentence in the post (see Section 5.2)
3) **Manual Labeling of the API aspects**: Total four coders labelled the aspects in the sentences (see Section 5.3)
4) **Manual Labeling of the Polarity**: Total seven coders labelled the polarity of the sentences (see Section 5.4)

5.1 Data Collection

Each tag was also accompanied by another tag 'java'. The 18 tags used in our benchmark sampling corresponded to more than 18% of all the Java threads (out of the 554,917 threads tagged as 'java', 100,884 were tagged as at least one of the tags). In total, 16,996 distinct tags co-occurred with the tag 'java' in those threads. The tags that co-occurred for a given tag (e.g., 'authentication') came mostly from tags with similar themes (e.g, security, authorization). Stack Overflow has a curated official list tag synonyms, maintained and updated by the Stack Overflow community [76]. The 554,917 threads tagged as 'java' contained a total of 4,244,195 posts (= question+answer+comment). With a 99% confidence interval, a statistically significant dataset would require at least 666 posts from the entire list of posts tagged as 'java' in Stack Overflow. Our benchmark contains all the sentences from 1,338 posts.

The 18 tags in our dataset can be broadly grouped under nine domains, each domain containing two tags. For example, the APIs discussed in the threads labelled as the two tags 'json' and 'xml' most likely offer serialization features, e.g., conversion of a Java object to a JSON object, and so on. Similarly, the APIs discussed in the threads labelled as the two tags 'authentication' and 'cryptography' most likely offer the security features. The domain information of each tag was inferred from the description of the tag from Stack Overflow. For example, the information about the 'json' tag in Stack Overflow can be found here: https://stackoverflow.com/tags/json/info. For each domain in Table 2, we have two tags, which were found as related in the Stack Overflow 'Related Tags' suggestions with similar description about their underlying domain. In the rest of paper, we discuss the results on the benchmark dataset by referring to the tags based on their domains. This is to provide concise presentation of the phenomenon of API aspects in API reviews based on the dataset.

To select the 18 tags, we adopted a purposeful maximal sampling approach [121]. In purposeful maximal sampling, data points are selected *opportunistically* with a hope to get diverse viewpoints within the context of the project. For example, while the tag 'authentication' offers us information about the security aspects, the tag 'swing' offers us information about the development of widgets in Java. Determining the representativeness of the tags and the dataset is the not subject of this paper. We do not claim the selected tags to be representative of all the Stack Overflow tags or all the tags related to the Java APIs. While the findings in this paper can offer insights into the phenomenon of the prevalence of API aspects and sentiments in API reviews, such findings may not be observed across all the data in Stack Overflow.

For each tag, we selected four threads as follows: **(1)** Each thread has a score (= #upvotes - #downvotes). We collected the scores, removed the duplicates, then sorted the distinct scores. For example, there were 30 distinct scores (min -7, max 141) in the 778 threads for the tag 'cryptography'. **(2)** We divided the distinct scores into four quartiles. In Table 2, we show the total number of threads under each tag for each quartile, e.g., the first quartile had a range $[-7, 1]$ for the above tag. **(3)** For each quartile, we randomly selected one thread **(4)** Thus, for each tag, we have four threads. One thread was overlapped between two tags. We adopted this strategy based on the observation that more than 75% of the threads for any given tag contained a score close to 0. In Table 3, we show the distribution of the threads in the tags based on the quartiles. The first quartile for the 18 tags corresponded to 92.6% of all the threads. The scores of the threads ranged from minimum -16 to maximum 2,302. Therefore, a random sampling would have more likely picked threads from the scores close to 0. We thus randomly picked threads from each quartile (after sorting the threads based on the scores), to ensure that our analysis can cover API aspects and sentiments expressed in threads that have high scores (i.e., very popular) as well threads with low scores. On average, each thread in our benchmark contained 64.7 sentences. The domain 'utility' had the maximum number of sentences followed by domains 'widgets' and 'serialize'. The domain 'text' had the minimum number of sentences, followed by the domains 'database' and 'framework'.

TABLE 3: The quartiles used in the benchmark for each tag

Tag	Domain	Q1		Q2		Q3		Q4	
		R	T	R	T	R	T	R	T
json	serialize	-11; 6	6,500	7; 20	196	21;36	31	37; 613	16
xml	serialize	-8; 7	11,283	8; 22	212	23; 45	35	48; 225	18
authentication	security	-5;2	919	3; 8	157	9;15	16	16;32	9
cryptography	security	-7; 1	495	2; 8	257	9; 16	18	18; 141	8
io	utility	-12; 4	2,555	5; 20	261	21; 52	31	56; 2302	18
file	utility	-8; 6	4,443	7; 22	141	23; 42	28	44; 324	19
http	protocol	-7; 4	2,658	5; 15	149	16; 33	27	34; 691	13
rest	protocol	-6; 6	3,279	7; 17	106	18; 33	17	36; 93	12
debug	debug	-7; 4	1,448	5; 14	196	15; 26	31	27; 63	13
log	debug	-4; 8	2,220	9; 20	93	21; 36	28	37; 89	15
nosql	db	-3;1	168	2;6	75	7;11	11	12;52	7
sql	db	-8; 3	5,154	4; 14	342	15; 37	27	38; 153	13
swing	widgets	-7; 2	72	-1; 4	1,989	5; 10	59	11; 230	11
awt	widgets	-9;0	1,499	1;8	1,482	9;16	33	17; 230	9
nlp	text	-7; 0	179	1; 6	215	7; 14	27	15; 45	7
string	text	-16; 17	9,649	18; 46	154	47; 85	45	91; 995	33
spring	framework	-9;11	19,607	12;30	266	31;59	44	63; 165	22
eclipse	framework	-14; 15	21,308	16; 41	313	42; 72	63	73; 720	30
Total/Overall Range		-16; 17	93,436	-1; 46	6,604	5; 72	571	12; 2302	273

Q = quartile, Q1 = first quartile, R = Range of distinct scores, T = total threads for a tag

5.2 Data Preprocessing

To assist in the labeling process in the most seamless way possible, we created two web-based survey applications[3]: one to label the aspects in each sentence and another to label the polarity of each sentence. The web-based data collection tool with the coding guide to label aspects is hosted at: http://sentimin.soccerlab.polymtl.ca:28080/. The web-based data collection tool with the coding guide to label polarity is hosted at: http://sentimin.soccerlab.polymtl.ca:48080/.

Aspect Coding App. In Figure 5, we show screen shots of the user interface of the application. The circled numbers show the progression of actions. The leading page ① shows the coding guide. Upon clicking the 'Take the Survey', the user is taken to the 'todos' page ②, where the list of threads is provided. By clicking on a given thread in the 'todos' page, the user is taken to the corresponding page of the thread in the app ③, where he can label each sentence in the thread ④. Once the user labels all the sentences of a given thread, he needs to hit a 'Submit' button (not shown) to store his labels. After that, he will be returned to the 'todos' page ②. The data are stored in a PostgreSQL database. Each thread is assigned a separate web page in the survey. Each such page is identified by the ID of the thread, which is the ID of the thread in Stack Overflow. The welcome page of the survey contains a very short coding guide and a link to a `todo` page. The `todo` page shows the coder a summary of how many threads he has completed and how many are still needed to be coded for each tag. Each remaining thread in the `todo` page is linked to its corresponding survey page in the application. Each thread page in the application contains all the sentences of the thread.

Polarity Coding App. In Figure 6, we show screen shots of the user interface of the application. The circled numbers show the progression of actions in sequential order. The application has similar interfaces as the aspect labeling application, with two exceptions: 1) the coding guide in the front page denotes the different polarity categories ①, and 2) each sentence is labelled as the polarity as shown in ④.

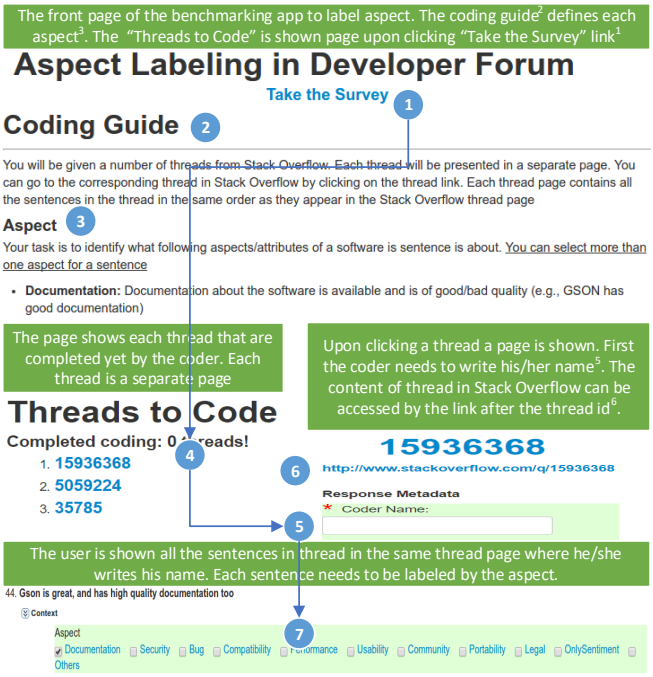

Fig. 5: Screenshots of the benchmarking app to label aspects in API reviews.

3. The app is developed using Python on top of the Django web framework.

TABLE 4: The definitions and examples used for each aspect in the benchmark coding guide

Aspect	Definition	Example
Performance	How the software performs in terms of speed or other performance issues	The object conversion in GSON is fast
Usability	Is the software easy to use? How well is the software designed to meet specific development requirements?	GSON is easy to use
Security	Does the usage of the software pose any security threat	The network communication using the HTTPClient API is not secure
Bug	The opinion is about a bug related to the software	GSON crashed when converting large JSON objects
Community	How supportive/active the community (e.g., mailing list) related to the software?	The GSON mailing list is active
Compatibility	Whether the usage of the software require the specific adoption of another software or the underlying development/deployment environment	Spring uses Jackson to provide JSON parsing
Documentation	Documentation about the software is available and is of good/bad quality	GSON has good documentation
Legal	The usage of the software does/does not require any legal considerations	GSON has an open-source license.
Portability	The opinion about the usage of the software across different platforms	GSON can be used in windows, linux and mobile.
OnlySentiment	Opinions about the software without specifying any particular aspect/feature of the software.	I like GSON.
Others	The opinion about the aspect cannot be labelled using any of the above categories	N/A

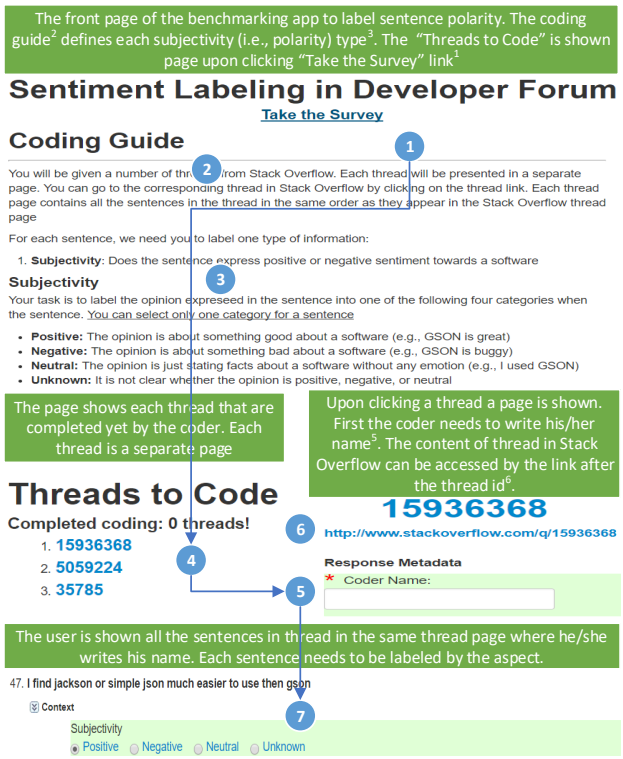

Fig. 6: Screenshots of the benchmarking app to label sentiment polarity in API reviews. In the app, *subjectivity* refers to *polarity*, i.e., the labeling of a sentence as one of three polarity types, i.e., positive, negative, or neutral. As we noted in Section 2, we analyze polarity in sentences.

5.3 Labeling of API Aspects in Sentences

The coders followed a coding guide to label each sentence. In Figure 5 we show this coding guide, where each aspect was formally defined with an example (see circle ① in Figure 5). In Table 4, we present the definition and example used for each aspect in the coding guide. Besides, during the labeling, each checkbox corresponding to the aspect showed the definition as a tooltip text (see circle ④ in Figure 5).

5.3.1 Coders

The benchmark was created based on inputs from four different coders. Except the first coder, none of the other coders were authors of the paper, nor where they associated with the work or the project at all.

C1. The first author of this paper was the first coder. The first author coded all 71 threads.
C2. The second coder was a post-doc from Ecole polytechnique. He coded 14 of the 71 threads. The second coder was a Post-doc with 27 years of experience as a professional software engineer (ranging from software developer to manager).
C3. The third coder was a graduate student from the University of Saskatchewan. He coded three out of the remaining 57 threads.
C4. The fourth coder was another graduate student from the University of Saskatchewan. He coded six out of the remaining 54 threads.

5.3.2 Labeling Process

The coding approach is closely adapted from Kononenko et al. [51]. The coding approach consisted of three distinct phases:

- **P1. Collaborative Coding.** The first two coders collaborated on multiple sessions on 14 different threads from the benchmark to develop a detailed guidance on how each API aspect can be labelled.
- **P2. Code Completion.** The first coder then labelled the rest of the 57 threads.

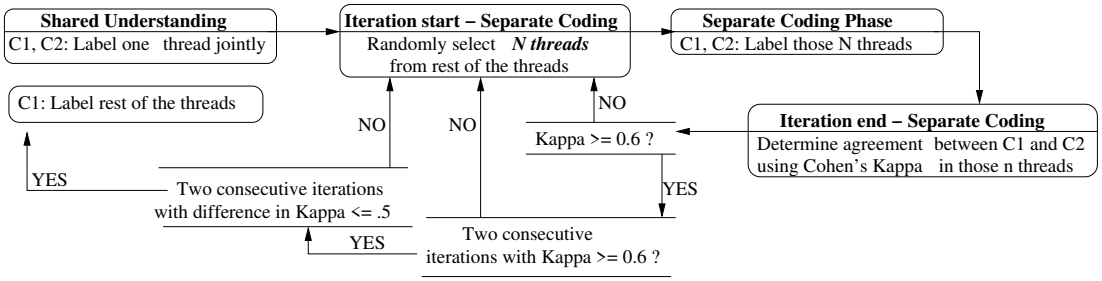

Fig. 7: The steps used in the labeling process of the benchmark (C1 = First Coder, C2 = Second Coder)

- **P3. Code Validation.** Two statistically significant subsets of the sentences from the 57 threads (that were coded by C1) were selected and coded separately by two other coders C3 and C4. Their codings were compared to validate the quality of labels of C1.

We describe the phases below.

P1. Collaborative Coding.
There were two steps:

1) **Shared Understanding.** The first two coders jointly labelled one thread by discussing each sentence. The purpose was to develop a shared understanding of the underlying problem and to remove individual bias as much as possible. To assist in the labeling process in the most seamless way possible, we created a web-based survey application. In Figure 5, we show screen shots of the user interface of the application.
2) **Separate Coding.** The two coders separately labelled a number of threads. A Cohen kappa value was calculated and the two coders discussed the sentences where disagreements occurred over Skype and numerous emails. The purpose was to find any problems in the labels and to be able to converge to a common ground, if possible. The two coders repeated this step five times until the agreement between the two coders reached the *substantial* level of Cohen Kappa value (see Table 5) and the change in agreements between *subsequent* iterations were below 5%, i.e., further improvements might not be possible. In Table 6 we show the number of sentences labelled in each iteration by the two coders and the level of agreement (Cohen Kappa value) for the iteration. The coders met over Skype after each separate coding session to discuss about the disagreements. An agreement was reached for each such disagreed labels. The separate coding sessions took a total of around seven hours for each coder. The agreement reached the substantial level at iteration 3 and remained there in subsequent iterations.

The multiple sessions of coding between the first two coders were conducted to ensure that a clear understanding of each of the aspects can be established. The second coder was selected because of his extensive experience software engineering (both in the Academia and Industry), such that a clear guideline is formed based on the collaborative sessions between them and that can be used as the basis for the rest of the coding.

The manual labels and the discussions were extremely helpful to understand the diverse ways API aspects can be interpreted even when a clear guideline was provided in the coding guide. One clear message was that a sentence can be labelled as an aspect if it contains clear indicators (e.g., specific vocabularies) of the aspects and the focus of the message contained in the sentence is indeed on the aspect. Both of these two constraints are solvable, but clearly can be challenging for any automated machine learning classifier that aims to classify the sentences automatically.

P2. Code Completion.
The first coder then completed the coding of the rest of 57 threads. The first coder labelled each sentence based on knowledge obtained from the collaborating coding.

P3. Code Validation.
We asked the two other coders (C3 and C4) to analyze the quality of the coding of the first coder in the 57 threads. For each coder C3 and C4, we randomly selected a statistically significant subset of the 57 threads. With a 95% confidence interval, a statistically significant subset of the sentences from the 57 threads should have at least 341 sentences. C3 assessed 345 sentences (from three threads), while C4 assessed 420 sentences (from another three threads). Each of the coders C3 and C4 completed their coding task as follows.

1) The first coder explained to C3 and C4 the coding guide as it was finalized between the first two coders.
2) The two coders C3 and C4 then completed the coding of their subsets separately. Once they were done, they communicated their findings with the first coder.
3) The agreement level was Cohen κ = 0.72 (percent agreement 96.84%) between C1 and C3, and Cohen κ = 0.804 (percent agreement 95.91%) between C1 and C4.

The agreement levels between C1 and C3, C4 remained substantial and even reached to the almost perfect level between C1 and C4. Therefore, the individual coding of C1 had a substantial agreement level with each of the other three coders. While the threat to potential bias can always present in individual coding, the agreements between C1 and other coders showed that this threat was largely mitigated in our coding session (i.e., we achieved a substantial agreement level between the coders).

5.3.3 Analysis of the Disagreements

In this section, we discuss the major themes that emerged during the disagreements between the coders. In Table 7, we show the number of total labels for each coder and for each aspect.

The major source of disagreement occurred for three aspects: security, documentation, and compatibility. For example, the sec-

TABLE 5: Interpretation of Cohen κ values [114]

κ value	interpretation
< 0	poor
0 – 0.20	slight
0.21 – 0.40	fair
0.41 – 0.60	moderate
0.61 – 0.80	substantial
0.81 – 0.99	perfect

TABLE 6: Progression of agreements between the first two coders to label aspects

Iteration	1	2	3	4	5
Sentence	83	117	52	24	116
Kappa κ	0.28	0.34	0.62	0.71	0.72

ond coder labelled the following sentence as 'Security' initially: "The J2ME version is not thread safe". However, he agreed with the first coder that it should be labelled as 'Performance', because 'thread safety' is normally associated with the performance-based features of an API. The first coder initially labelled the following sentence as 'Documentation', assuming that the URLs referred to API documentation: "URL[Atmosphere] and URL[DWR] are both open source frameworks that can make Comet easy in Java". The first coder agreed with the second coder that it should be labelled as 'Usability', because it shows how the frameworks can make the usage of the API Comet easy in Java.

The agreement levels between the coders were lower for sentences where an aspect was discussed implicitly. For example, the second coder labelled the following sentence as 'Community': "I would recommend asking this in it's own question if you need more help with it". The first coder labelled it as 'Other General Features'. After a discussion over Skype on the labels where the first coder shared the screen with the second coder, it was agreed to label as 'Community' because the suggestion was to seek help from the Stack Overflow community by posting another question. The following sentence was finally labelled as 'Other General Features': "I have my own logging engine which writes the logs on a separate thread with a blocking queue". The first coder labelled it as 'Performance', assuming that it is about the development of a logging framework in a multi-threaded environment. However, the second coder pointed out that the focus of the sentence was about what the logging engine does, *not* about the performance implication that may come due to usage of threads. The category 'Other General Features' was useful to ensure that when the labeling of an aspect in a sentence was not directly connected to any of the pre-defined aspects, the quality of the labeling of those aspects can be ensured by putting all such spurious sentences into the 'Other General Features' category.

5.4 Labeling of Polarity in Sentences

In this section, we describe the labeling process that we followed to code the polarity of each sentence in our benchmark dataset. Each sentence in our dataset was judged for the following four types with regards to the sentiment exhibited in the sentence:

Positive. The opinion is about something good about a software (e.g., GSON is great),

TABLE 7: Distribution of labels per coder per aspect

↓Aspect | Coder →	Coder1	Coder2	Coder3	Coder4
Security	1	54	38	1
Bug	1	56	9	43
Document	15	39	3	15
Portability	4	13	3	8
Usability	42	317	97	143
Community	3	16	5	7
Compatibility	4	19	8	5
Performance	15	36	2	81
General Features	60	335	150	146
OnlySentiment	6	60	26	32

Negative. The opinion is about something bad about a software (e.g., GSON is buggy),

Neutral. The opinion is just stating facts about a software without any emotion (e.g., I used GSON), and

Unknown. The polarity of the opinion is not clear.

The labeling of each sentence was followed by a coding guide as shown in Figure 6, where, each polarity category was formally defined with an example (see circle ③ in Figure 6). We used the same description for each polarity category as provided above. Besides, during the labeling, each checkbox corresponding to the polarity category showed the definition as a tooltip text.

5.4.1 Coders

The benchmark was created based on inputs from seven different coders. Except the first coder, none of the other coders were authors of the paper.

C1. The first author of this paper was the first coder. The first author coded all 71 threads.

C2. The second coder was a post-doc from Ecole polytechnique. He coded 15 of the 71 threads. The second coder was a Post-doc with 27 years of experience as a professional software engineer (ranging from software developer to manager).

C3-C6. The four coders (C3-C6) were all graduate students at the Ecole Polytechnique Montreal. C6 coded 53 of the threads. Each of C3-C5 coded 24 threads.

C7. The seventh coder was a professional software developer. He coded the 653 sentences where a majority agreement was not reached among the first five coders (discussed below). He had a five years of professional software development experience in Java and C#.

5.4.2 Labeling Process

The labeling process of polarity is adapted from the one reported in [15], which also had a group discussion with all the coders prior to the individual annotation sessions. For each sentence, we collected the labels until we reached a majority vote. For a given sentence, there can be two scenarios:

TABLE 8: Weighting scheme to compute weighted Cohen κ [25]

Class	Negative	Neutral	Positive
Negative	0	1	2
Neutral	1	0	1
Positive	2	1	0

1) We find a majority agreement from the first three codes. For example, for the sentence "I used GSON and it works well", suppose two coders label it as positive and the other label it as anything other than positive. We label the sentence as positive, because the majority vote is 'positive'.
2) We do not find a majority vote from the first three coders. For example, for the sentence "I used GSON and it works well", suppose one coder labels it as positive, another as neutral, and the other as unknown. We then ask the fourth coder (C7) to label the sentence. The fourth coder is not given access to any of the labels of the other coders, to ensure that he is not biased by any particular coder. The fourth coder was given access to all the contents of threads where the sentences were found, to ensure that his assessment of the polarity will be based on all the information as it was the case for the other coders. We then assign the sentence the label that is the majority voted.

We report the inter-rater agreements using weighted Cohen kappa (κ) [25] and percent agreement. Within the context of sentiment detection where the detection of positivity vs negativity can be more important than the detection of neutral vs non-neutral sentences, a disagreement is severe, if the sign of the polarity changes from positive to negative and vice versa. In contrast, a disagreement is mild if the sign of polarity changes from neutral to non-neutral, and vice versa. Following state of art [71], [49] in disagreement analysis between sentiment tools in software engineering, we thus distinguish between the *mild disagreements* and *severe disagreements* by assiging a severe disagreement a weight of 2 and a mild disagreement a weight of 1 in our weighted Cohen κ calculations (see Table 8)[4].

In Table 9, we show the agreements between the coders. The agreement was the highest (76.8%) between the coders C1 and C2, followed by 73.8% between C3 and C5. The coder C7 was assigned to label the sentences where a majority was not reached previously. That means, the polarity of those sentences was not easy to understand. This difficulty is reflected in his agreement levels with the other coders (maximum 39.8%).

5.4.3 Analysis of the Disagreements

We discuss the major themes that emerged from the analysis of the disagreements between the coders. Specifically, we focus on the sentences for which we needed to use the fourth coder (C7) to find a majority vote. The major sources of *conflicts* among the first three coders for such sentences were as follows:

- **Sarcasm.** A sentence containing both polarity and sarcasm. For example, *"2005 is not Jurassic period and I think you shouldn't dismiss a library (in Java, at least) because it hasn't been updated for 5 years"*. The three coders labelled it as: C1(n), C3(o), C5(p), where 'p' denotes positive, 'n' denotes negative, and 'o' denotes neutral. The coder C7 labelled it as positive, which we believe is the right label given that the opinion towards the 'library' is indeed positive.
- **Convoluted Polarity.** When the sentence has diverse viewpoints. Consider the sentence *"For folks interested in using JDOM, but afraid that hasn't been updated in a while, there is a fork called CoffeeDOM which exactly addresses these aspects and modernizes the JDOM API. . . "*. The three coders labelled it as: C1(p), C3(n), C5(o). C7 labelled it as positive, which we believe is correct since the sentiment towards the target CoffeeDOM is positive.

4. We used the Metrics library [41] to compute Weighted Cohen κ

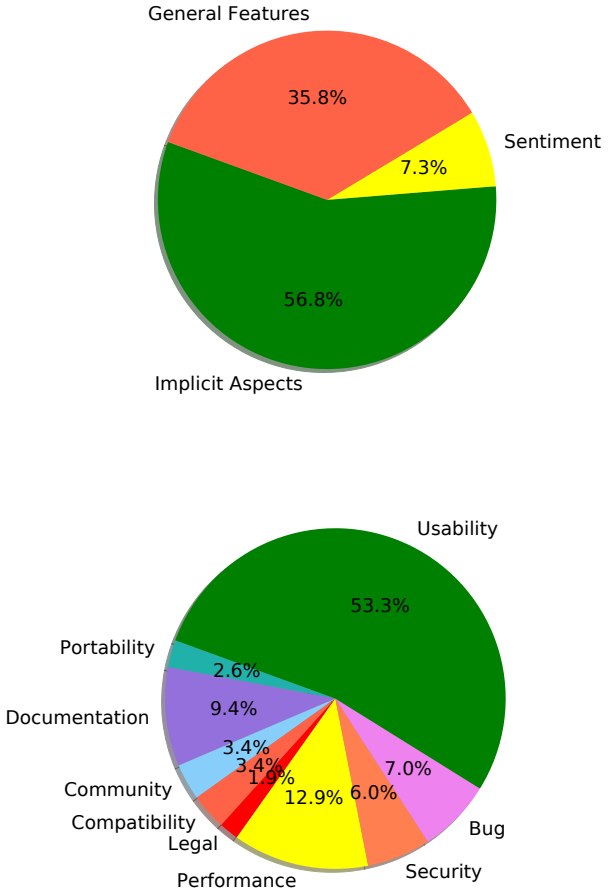

Fig. 8: The distribution of aspects in the benchmark. The top pie shows the distribution of the implicit vs OnlySentiment and Other General Features in the benchmark. The bottom pie shows the distribution of each individual aspect among the implicit aspects.

6 Analysis of Aspects in API Reviews

In this section, we leverage the benchmark dataset to understand the impact of aspects and sentiments in API reviews.

6.1 How are the API aspects discussed in the benchmark dataset of API discussions? (RQ1)

Observation 1. Figure 8 shows the overall distribution of the aspects in the benchmark. The top pie chart shows the overall

TABLE 9: The weighted κ and percent agreement between the coders in their labeling of the polarity of the sentences. Due to the symmetric nature of the table (i.e., same coders in both rows and columns), we put – for some cells (e.g., C3,C2) to avoid the duplicate reporting of the agreement between the coders (e.g., the pair C2, C3 is the same as the pair C3, C2)

↓ Coder →		C2	C3	C4	C5	C6	C7
C1	**Weighted κ**	0.640	0.468	0.205	0.537	0.273	0.033
	Perfect Aggrement	76.8%	66.3%	56.9%	72.0%	54.4%	34.6%
	Disaggrement Severe	3.4%	2.1%	0.8%	2.1%	5.1%	27.4%
	Disaggrement Mild	19.8%	31.6%	42.3%	25.9%	40.6%	38.1%
C2	**Weighted κ**	–	0.468	0.200	0.574	–	-0.187
	Perfect Aggrement		66.8%	63.4%	73.4%	–	36.8%
	Disaggrement Severe		2.0%	0.0%	1.7%	–	31.6%
	Disaggrement Mild		31.2%	36.6%	24.9%	–	31.6%
C3	**Weighted κ**	–	–	–	0.420	0.290	0.086
	Perfect Aggrement			–	73.8%	59.8%	37.3%
	Disaggrement Severe			–	0.9%	3.9%	8.4%
	Disaggrement Mild			–	25.3%	36.3%	54.2%
C4	**Weighted κ**	–	–	–	–	0.127	-0.016
	Perfect Aggrement				–	58.3%	39.8%
	Disaggrement Severe				–	1.6%	2.8%
	Disaggrement Mild				–	40.1%	57.5%
C5	**Weighted κ**	–	–	–	–	0.231	0.078
	Perfect Aggrement					58.3%	31.7%
	Disaggrement Severe					4.0%	5.5%
	Disaggrement Mild					37.7%	62.8%
C6	**Weighted κ**	–	–	–	–	–	-0.016
	Perfect Aggrement						28.1%
	Disaggrement Severe						8.6%
	Disaggrement Mild						63.3%

TABLE 10: The percentage distribution of aspects across the domains

↓Aspect|Domain→	Serialize	Debug	Protocol	Utility	Security	Text	Framework	Database	Widget
Tag→	json, xml	debug,log	http, rest	io, file	authentication, cryptography	nlp,string	spring,eclipse	nosql,sql	swing,awt
Performance	12.1	4.9	8.6	**36.5**	4.3	8.9	6.9	3.7	14.1
Usability	13.2	13.6	9.5	**21.9**	5.9	6.5	7.2	7.9	14.3
Security	0.6	8.6	0.6	0	**71.8**	0	17.8	0.6	0
Bug	6.9	21.2	5.3	**29.1**	3.7	3.7	15.9	5.3	9.0
Community	16.1	4.3	4.3	17.2	2.2	3.2	**20.4**	11.8	**20.4**
Compatibility	6.5	16.1	2.2	14.0	6.5	8.6	**18.3**	15.1	12.9
Documentation	8.7	3.2	19.4	**20.9**	10.3	6.7	15.0	7.1	8.7
Legal	12.0	2.0	4.0	0	20.0	6.0	6.0	**26.0**	24.0
Portability	5.7	2.9	0	24.3	1.4	2.9	2.9	4.3	**55.7**
General Features	12.7	10.5	10.0	**21.8**	8.2	6.1	7.5	7.9	15.2
Sentiment	13.2	6.6	8.9	**21.6**	4.9	3.7	10.6	13.2	17.2

distribution of all the implicit aspects against the two other aspects (Other General Features and Only Sentiment). **The majority (56.8%) of the sentences are labelled as at least one of the implicit aspects.** 7.3% of the sentences simply contain the sentiment towards and API or other entities (e.g., developers). 35.8% of the sentences are labelled as 'Other General Features'. Among those, only 1% also have one or more of the implicit aspects labelled. In total, 95.2% of the sentences are labelled as only one aspect. Among the rest of the sentences, 4.6% are labelled as two aspects and around 0.2% are labelled as three aspects. The major reason for some sentences having three aspects was that they were not properly formatted and were convoluted with multiple potential sentences. For example, the following sentence was labelled as three aspects (performance, usability, bug): "HTML parsers . . . Fast .. Safe .. bug-free . . . Relatively simple . . . . . ".

Observation 2. **The discussion of API aspects in forum posts follow a Zipfian distribution, i.e., some aspects (e.g., Usability, Performance) are discussed much more often than others.** Out of the sentences labelled as at least one of the nine implicit aspects, 53.3% of the sentences are labelled as 'Usability', followed by the aspect 'Performance' (12.9%). Similar Zipfian distributions were observed in the opinions of other domains. For example, in Google local service reviews, Blair-Goldenshon et al. [11] observed that certain aspects (e.g., food, service) were present in a much greater proportion than the other aspects (e.g., decor). This distribution did not change when we applied our aspect classifiers on all the threads tagged as Java+Json [109].

Observation 3. **The sentences labelled as 'Usability' exhibit a multi-faceted nature of usability themes, such as, API design,**

usage, etc. A detailed investigation is required to determine the proper ways to divide this aspect into more focused sub-aspects (e.g., design vs learnability), which we leave as our future work. **The two aspects 'Community' and 'Compatibility' were not reported as frequently as 'Usability' because they contained themes that were not directly related to the basic usage of an API.** Developers discussed about 'Compatibility' when they looked for an API that could be compatible to a framework they are already familiar with. For example, one opinion labelled as 'Compatibility': "I would advise against Spring MVC for REST because it is not JAX-RS compliant." The next sentence was also labelled as 'Compatibility': "Using a JAX-RS compliant interface gives you a little bit better portability if you decide to switch to a different implementation down the road. "

Observation 4. In Table 10, we show the distribution of aspects across the 18 tags. **Four of the implicit aspects (Performance, Usability, Bug, and Documentation) and 'General Features', and 'Only Sentiment' have the largest concentration around the two tags (io, file), which include APIs offering utility features (e.g., file read/write)**. The discussions about community support have the largest concentration in the posts tagged as the framework-related tags (eclipse and spring). The aspect 'Legal' has the most concentration in the 'database' domain. Unlike other domains, the posts under the 'database' domain have more discussions around proprietary software (i.e., database). The discussion about 'Portability' are found mainly in the 'Widget' domain (i.e., Swing and AWT tags), due mainly to the the two popular Java UI APIs (Swing and AWT). The Java AWT API relies on native operating system for look and feel and Swing avoids that.

Observation 5. In Table 11, we show the distribution of the themes in the sentences labelled as an aspect in the benchmark (except for the OnlySentiment category). **In total, 26 major themes emerged from the discussion of the aspects in the benchmark. The 26 major themes contain a total of 222 sub-themes in the dataset.** For example, the theme 'Design' has sub-themes as 'code smells', 'API configuration', to 'expected vs desired behavior', and so on. The following three themes are found as exclusive to the individual aspects: 1) *cost* and *authority* in 'Legal'. The theme *cost* is related to the pricing of an API, and *authority* is about the contact person/organization for a given API. 2) *expertise* in the 'Other General Features', which concerns discussions about an expert for a given API or the domain itself. The following major themes were prevalent in the discussions of the aspects:

Performance. The two major themes are the 'speed' and the 'concurrency' of the provided API features, followed by the scalability and the memory footprint of the features.

Usability. The usage of an API is the major theme, such as, whether it is easy or difficult to use, and so on. The second major theme is the 'Design' of the API.

Security. Similar to the 'Usability' aspect, the discussions revolve around the 'design'. However, the design issues focus on the development of a secure API, e.g., role based access control.

Bug. The usage and design of an API are the major themes (e.g., design and proper usage of Null Pointer Exception API).

Community. The discussion about the engagement and activity of the 'community' of authors/developers/maintainers supporting an API was the major theme.

Compatibility. The major theme is the 'compliance' of an API in a framework (e.g., which API is compatible with the Spring framework to offer JSON parsing) or the support/availability of a feature in different versions of an API.

Documentation. The discussion of both formal (e.g., Javadoc) and informal (e.g., blogs, other forum posts) resources.

Legal. The licensing and costing issues of an API are mostly discussed in the sentences labelled as the 'Legal' aspect.

Portability. Our benchmark dataset is comprised of threads tagged as 'Java' and the 18 tags. Therefore, portability is not discussed as a concern of whether an API feature is available across different computing platforms. Rather the issue is how the provided Java Virtual Machine (JVM) supports the specific API feature.

General Features. Contain discussion mostly about general API usage, such as, how to complete task using a provided code example, the problem it can have and whether it works, etc.

How are API aspects discussed in the dataset? (RQ1)

Similar to other domains (e.g., hotels, restaurants), the discussions about API aspects followed a Zipfian distribution, e.g., some API aspects (e.g., Performance and Usability) are more prevalent in the discussions. While themes related to the usage and design are found across the aspects, some themes were more exclusive to some specific aspects (pricing & costing to 'Legal').

6.2 How do the aspects about APIs relate to the provided opinions about APIs? (RQ2)

Observation 6. In Figure 9, we show the distribution of the three polarity classes (i.e., positive, negative, and neutral) in the benchmark. **Among the sentences, 58% are neutral, 23% are positive, and 19% negative. The high concentration of neutral sentences is in line with the findings in previous studies.** For example, the Stack Overflow benchmark produced by Lin et al. [54] has 76% neutral, 9% positive, and 15% negative sentences. In Figure 10, we show the distribution of the positive and negative opinions for each aspect. The percentage number beside each aspect shows the percentages of opinionated sentences that are labelled as the aspect. **Except for two aspects (Security and Documentation) and 'Other General Features', all the other aspects have more than 50% of the sentences labelled as opinionated.** While considering the proportion of positive and negative opinions for a given aspect, developers are more positive in the discussions around 'Security' followed by 'Legal'. The proportion of positivity and negativity in the opinions labelled as the Performance aspect is almost equal, i.e., performance may be a pain point during the usage an API. In contrast, developers expressed more positivity in the opinions related to 'Usability' and more negativity in the opinions related to the aspect 'Portability'. The discussions around portability of an API were related to the availability of the API across different computing platforms, which can be a non-trivial task due to the numerous external factors contributing to the failure of an API in a given platform.

Observation 7. **We observed diverse themes in the positive and negative opinions about aspects**. In Table 12, we show

TABLE 11: The distribution of themes in the aspects

↓Theme|Aspect→	Performance	Usable	Security	Compatibility	Portability	Document	Bug	Legal	Community	Others	Total
Usage	14	805	67	18	3	60	87	–	32	1130	2216
Design	52	551	70	17	29	–	46	–	5	252	1022
Speed	114	–	–	–	–	–	–	–	–	2	116
Informal Docs	–	–	–	–	–	111	–	–	–	46	157
Alternative	2	7	1	3	–	–	–	–	1	90	104
Debugging	–	33	11	–	–	–	25	–	–	34	103
Formal Docs	–	–	–	–	–	88	–	–	–	6	94
Activity	–	27	1	1	–	–	–	2	34	27	92
Concurrency	49	–	–	–	–	–	–	–	–	12	61
Environment	–	18	–	19	–	–	11	–	–	25	73
Compliance	–	–	–	37	–	–	–	8	–	–	45
Ownership	–	–	–	–	–	–	–	–	25	19	44
Resource	32	–	2	–	–	–	9	–	–	–	43
Interoperability	–	–	–	–	41	–	–	–	–	–	41
Reliability	22	7	–	–	–	–	–	–	8	1	38
Footprint	27	1	–	–	–	–	–	–	–	4	32
License	–	2	–	–	–	–	–	29	–	–	31
Scalability	25	–	1	–	–	–	3	–	–	–	29
Benchmarking	1	–	–	–	–	–	–	–	–	5	17
Costing	–	–	–	–	–	–	–	15	–	–	15
Vulnerability	–	–	6	–	–	–	8	–	–	–	14
Extensibility	–	6	–	–	–	–	–	–	–	7	13
Communication	8	–	4	–	–	–	–	–	–	–	12
Building	–	–	–	–	–	–	–	–	–	10	10
Authority	–	–	–	–	–	–	–	4	–	–	4
Expertise	–	–	–	–	–	–	–	–	–	2	2
Total	357	1457	163	95	73	259	189	58	105	1672	4428

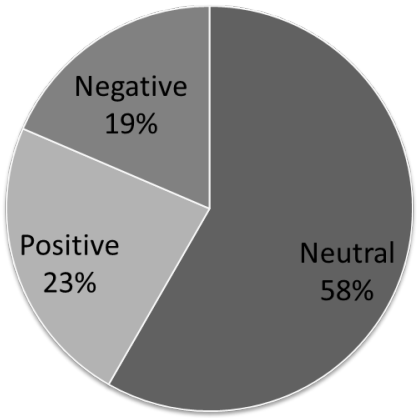

Fig. 9: Distribution of polarity labels in the benchmark

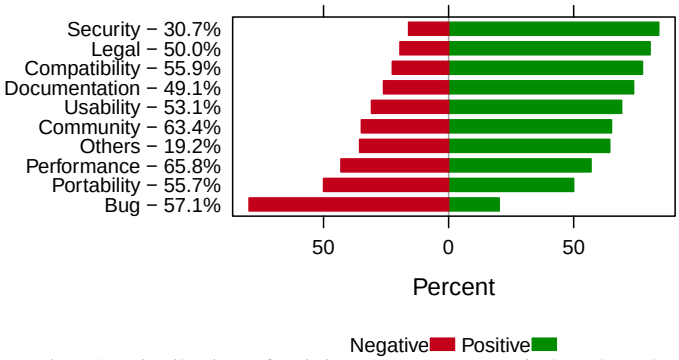

Fig. 10: Distribution of opinions across aspects in benchmark

the distribution of sentiments across the themes for each aspect. For each theme we identified in the benchmark, we show the overall presence of positive and negative opinions across each aspect for the theme in the benchmark opinionated sentences. The bar for each theme shows the relative proportion of positive and negative opinionated sentences for each aspect (white bar for positive opinions and black bar negative opinions). The aspect 'Performance' was discussed favorably across the themes related to usage, speed, reliability and concurrency issues regarding the APIs. For concurrency, their negative comments were related to the multi-threaded execution of the APIs and how the APIs supported communication over a network. The developers were divided in their discussion about the performance-centric design of an API, but were mostly positive about the usability-centric designs. The themes related to the 'Security' were discussed mostly favorably, with relatively minor complaints with regards to the usage of the API. The themes related to the 'Legal' were mainly about the compliance and authorship support of the

TABLE 12: The distribution of polarity for themes across aspects (Black and White bars represent negative and positive opinions, resp.)

↓Theme|Aspect→	Performance	Usable	Security	Bug	Community	Compatibility	Document	Legal	Portability	Features
Usage										
Design										
Speed										
Informal Docs										
Alternative										
Debug										
Formal Docs										
Activity										
Concurrency										
Environment										
Compliance										
Ownership										
Resource										
Interoperability										
Reliability										
Footprint										
License										
Scalability										
Benchmarking										
Vulnerability										
Extensibility										
Communication										
Building										
Authority										

APIs. Certain themes across the aspects were discussed more favorably and in higher volume (design, usage). Depending on the aspects, a given theme was either discussed entirely favorably or unfavorably. For example, while discussing the alternatives to a given API, developers were critical when the performance was a pain point. However, they were happy with regards to the community support of those APIs, in general.

Observation 8. **The concentration of opinions is more for some aspects than other aspects. The opinions are prevalent across the nine domains of the APIs**. In Table 13, we show the distribution of opinions around an aspect across the nine domains. For each aspect and each domain, we show the overall presence of opinions. The percentage beside each aspect name in Table 13 shows the percentage of opinionated sentences labelled as that aspect. The bar for each aspect shows the relative proportion of opinionated sentences for each domain. Performance and Usability both have much more negative opinions than positive opinions for domains 'Protocol' and 'Debug'. The major theme that emerged from the Performance-based negative opinions was related to the analysis and deployment of multi-threaded applications based on the protocols. The aspect 'Legal' mostly contained positive opinions because the APIs discussed in our dataset had open source licenses (e.g., MIT) that promote worry-free usage of the API (except GPL which is also open source but usually avoided by Enterprises [7]). While the diverse distribution of the aspects in our dataset did not affect our current study, the exploration of the distribution in more details is our future work.

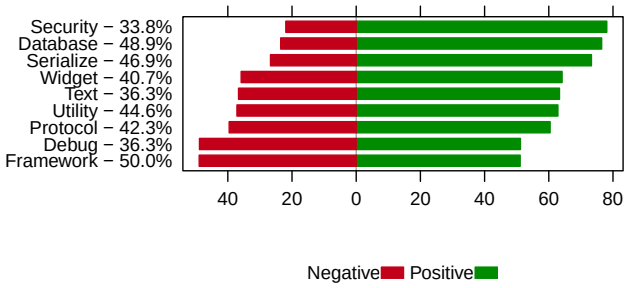

Fig. 11: Distribution of opinionated sentences across domains

TABLE 13: Distribution of opinions across aspects and domains (black and white bars represent negative and positive opinions, resp.)

↓Aspect|Domain→	Security	Database	Serialize	Widget	Text	Utility	Protocol	Debug	Framework
Performance - 65.8%									
Usability - 53.1%									
Security - 30.7%									
Bug - 57.1%									
Community - 63.4%									
Compatibility - 55.9%,									
Documentation - 49.1%									
Legal - 50.0%									
Portability - 55.7%									
Others - 19.2%									

In Figure 11, we show the presence of opinions per domain. The percentages beside each domain show the distribution of opinions in the domain. The bar for each domain further shows the relative proportion of positivity versus negativity in those opinions. Developers expressed more positivity than negativity in all domains except Framework and Debug.

How do the aspects about APIs relate to the provided opinions about APIs? (RQ2)

Developers expressed more opinions towards the discussions of specific API aspects (e.g., Performance). Diverse themes are observed in such opinions (e.g., concurrency). Depending on the domain, the distribution of opinions for a given aspect vary. For example, the opinions about performance are mostly positive for API features related to serialization but mostly negative with regards to the debugging of the performance issues of the APIs.

6.3 How do the various stakeholders of an API relate to the provided opinions? (RQ3)

Observation 9. **We observed three types of stakeholders in our benchmark: User, Author, and User Turned Author. They communicated about API usage and to suggest/promote APIs**.

1) **User:** who planned to use the features of an available API or who responded to queries asked by another user.
2) **Author:** who developed or authored an API.
3) **User Turned Author:** who developed a wrapper around an already available API and shared it with others.

The lower concentration of API authors vs users is because developers only author an API, if there is a need for that. We discuss the communications among the stakeholders in our benchmark below:

(1) Aspect: In Table 14, we show the distribution of aspects in the discussions of three types of stakeholders. We computed the distribution as follows: (a) We identify the posts created by each stakeholder. (b) For each post, we collect the list of labelled aspects and the frequency of each aspect. (c) We count the total number of times an aspect is found in a post created by each stakeholder. While the API users discussed about each of the aspects, we only observed a few cases where the aspects were discussed by the authors. The authors mostly discussed about the usability of the API, followed by specific feature support. The users who decided to develop an API, were interested in the improvement of the features, usability, and fixing of the bugs.

TABLE 14: % distribution of aspects discussed by stakeholders

Aspect	User	Author	User Turned Author
Performance	7.5		
Usability	30.3	21.9	32.2
Bug	3.9		10.2
Portability	1.5	3.1	
Documentation	5.3	25	
Community	2	6.2	
Compatibility	2	3.1	1.7
Legal	1	6.2	
Security	3.3	3.1	13.6
Features	35.8	28.1	42.4
OnlySentiment	7.5	3.1	

(2) Signals: We observed 11 distinct promotions from the authors, e.g.,"Terracotta might be a good fit here (disclosure: I am a developer for Terracotta)". We observed 122 distinct suggestions, e.g., "Give boon a try. It is wicked fast". We observed the following types of interactions between users and authors that were not included in the signals: a) Bug fix: from an author, "DarkSquid's method is vulnerable to password attacks and also doesn't work". From a user, "erickson I'm new here, not sure how this works but would you be interested in bounty points to fix up Dougs code?" b) Support: authors responded to claims that their API was superior to another competing API, e.g., "Performant? . . . While GSON has reasonable feature set, I thought performance was sort of weak spot . . . "

(3) Actions: We observed 967 distinct likes and 585 unlikes around the provided suggestions and promotions. On average each suggestion or promotion was liked 7.27 times and unliked 4.4 times. Two users explicitly mentioned that they used the suggested APIs. There are also some other signals in the post that highlight the preference of users towards the selection of an API based on author reputation: "Jackson sounds promising. The main reason I mention it is that the author, Tatu Saloranta, has done some really great stuff (including Woodstox that I use)."

How do the various stakeholders of an API relate to the provided opinions? (RQ3)

API authors and users provide opinions to suggest and promote APIs. Users show more positivity than negativity towards an API suggestion in the forum post. Polarity towards a suggestion depends on the quality of suggested API, and how well the suggestion answers the question.

6.4 Discussions

In general, we observe greater concentration of opinions than code examples in the benchmark dataset. For example, in Table 15, we show the distribution of opinions, code terms and code snippets in the forum posts. The first row (Percent) shows the percentage of distinct forum posts that contain at least one code terms or snippets or opinions. The second row (Average) shows the average number of code terms, code snippets and opinionated sentences across the forum posts. More than half of the forum posts (66%) contained at least one opinion, while only 11% contained at least one code example. Thus, while developer forums encourage developers to write facts with examples, such facts are not necessarily provided in code examples, rather in the textual contents through positive and negative opinions. Therefore, developers while only looking for code examples to learn the usage of APIs may miss important insights shared by other developers in those opinions.

This observation leads to the question of how we can assist developers to gain insights from the opinions. The prevalence of implicit aspects (Observation 1) in the opinions encouraged us to develop automatic classifiers to detect API aspects in the opinions. The greater prevalence of the two aspects (Usability and performance) than other aspects (Observations 2, 3) shows that such aspects may benefit from a finer grained definition of sub-aspects (e.g., design, usage etc.). Due to the greater concentration of the aspects in some tags, it may be necessary to prioritize aspect detection and analysis in those tags to conform to the needs of the developers (Observation 4). The diverse themes found in the discussions about each aspect offers an interesting question on whether the 11 categories we used are sufficient to categorize the opinions (Observations 5, 7). The greater concentration of opinions in some aspects over others shows that developers may prefer to learn about those aspects over others (Observation 6, 8). This insight can be useful for a number of applications, such as when presenting the most important insights about an API. Finally, the interactive nature of the communications among the API stakeholders as we observed in the opinions offers an interesting opportunity to create opinion analysis tools for APIs that can be useful for all stakeholders (Observation 9). For example, personalized recommendation tools can be developed to recommend improvement opportunities of an API to its author.

TABLE 15: Distribution of opinions and code in forum posts

	Code Terms	Code Snippets	Opinions
Percent	5.53%	10.99%	66.07%
Average	0.13	0.17	1.31

7 Automatic API Aspect Detection (RQ4)

In the absence of any specific classifiers available to automatically detect the aspects in API reviews, we experimented with both rule-based and supervised classifiers to automatically detect API aspects. Because more than one aspect can be discussed in a sentence, we developed a classifier for each aspect. In total, we have developed 11 classifiers (Nine classifiers for the nine implicit aspects, one for the 'Only Sentiment' category and and the other to detect the 'Other General Features'). This section is divided into two parts:

Rule-based Aspect Detection (Section 7.1).
We present a rule-based technique based on topic modeling to automatically detect the 11 API aspects.

Supervised Aspect Detection (Section 7.2).
We present a suite of supervised classification techniques to automatically detect the 11 API aspects.

We describe the algorithms below.

7.1 Rule-Based Aspect Detection

We investigated the feasibility of adopting topic mining into a rule-based aspect classifier. We first describe the algorithm and then present the performance of the algorithm in our benchmark.

7.1.1 Algorithm

In our exploration into the rule-based detection of the API aspects, we consider each aspect to be representative of specific *topic* in our input documents. A document in our benchmark is a sentence in the dataset. In topic modeling, a topic is a cluster of words that mostly occur together in a given set of input documents. Topic modeling has been used to find topics in news articles (e.g., sports vs politics, and so on.) [12]. We used LDA (Latent Dirichlet Allocation) [12] to produce *representative* topics for each aspect. LDA has been used to mine software repositories, such as, to automatically detect trends in discussions in Stack Overflow [9], and so on.

To determine representativeness of topics for a given aspect, we use the topic *coherence* measure as proposed by Röder et al [87].[5] The coherence measure of a given model quantifies how coherent the topics in the model is, i.e., how well they represent the underlying theme in the data. In Table 16, the third column (C)

5. We used gensim [84] to produce the topics and its coherencemodel to compute c_v coherence.

TABLE 16: Performance of topic-based aspect detection (C = Coherence, N = Iterations, JT, JF = Jaccard, T for positives, F negatives)

Aspect	Topic	C	N	JT	JF	Precision	Recall	F1-score
Performance	faster, memory, thread, concurrent, performance, safe, robust	0.52	100K	0.045	0.007	0.270	0.540	0.360
Usability	easy,works,work,uses,solution,support,example,need,simple	0.33	300K	0.026	0.011	0.501	0.436	0.467
Security	encrypted,decryption,bytes,crypto,security,standard,salt,cipher	0.54	200K	0.054	0.007	0.152	0.669	0.248
Bug	exception,nullpointerexception,throw,thrown,getters,error,bugs	0.55	300K	0.038	0.005	0.151	0.429	0.223
Community	years,explain,downvote,community,reputation,support	0.61	200K	0.045	0.005	0.093	0.523	0.158
Compatibility	android, reflection,equivalent,compatible,engine,backend	0.53	300K	0.040	0.008	0.053	0.452	0.094
Documentation	answer,example,question,article,link,tutorial,post	0.53	300K	0.044	0.012	0.137	0.509	0.215
Legal	free,commercial,license,licensing,open-source,illegal,evaluate	0.50	200K	0.080	0.004	0.093	0.780	0.166
Portability	windows,linux,platform,unix,redis,android,compact	0.53	200K	0.090	0.007	0.058	0.557	0.106
OnlySentiment	thanks,good,right,little,correct,sorry,rant,rhetorical,best	0.61	100K	0.028	0.005	0.219	0.371	0.276
General Features	array,write,need,implementation,service,server,size,lines	0.49	200K	0.014	0.012	0.370	0.203	0.262

shows the final coherence value of the final LDA model produced for each aspect. The higher the value is the more coherent the topics in the model are.

For each aspect, we produce the topics using the following steps:(1) We tokenize each sentence labelled as the aspect and remove the stopwords[6] (2) We apply LDA on the sentences labelled as the aspect repeatedly until the coherence value of the LDA model no longer increases. In Table 16, the fourth column (N) shows the number of iterations we ran for each aspect. For example, for Performance, the maximum coherence value of Performance-based topics was reached after 100,000 iterations of the algorithm, and for Usability it took 300,000 iterations. For each aspect, we produce two topics and take the top 10 words from each topic as a representation of the aspect. Therefore, for each aspect, we have 20 words describing the aspect. For example, for the aspect Performance, some of those words are: fast, memory, performance, etc. In Table 16, we show an example of topic-words for each aspect. Analyzing the optimal number of topic words for an aspect is our future work.

Given as input a sentence, we detect the presence of each of the 11 aspects in the sentence using the topics as follows:

(1) We compute the Jaccard index [61] between the topic-words (T) of a given aspect and the words in the sentence (S) using the following equation:

$$J = \frac{Common(T, S)}{All(T, S)} \quad (8)$$

(2) If the Jaccard index is greater than 0 (i.e., there is at least one topic word present in the sentence for the aspect), we label the sentence as the aspect. If not, we do not label the sentence as the aspect.

7.1.2 Evaluation

In Table 16, the last four columns show the performance using Precision P, Recall R, and F1-score $F1$. The precision values range from maximum 50.1% (for Usability) to 5.3% (for Compatibility). The recall values are the maximum 78% for the aspect Legal and minimum 20.3% for the aspect 'Other General Features. We observe two major reasons for this low performance:

1) **Specificity.** The detection may require contextual information. For example, the keyword 'fast' may denote that the sentence may be about 'Performance'. However, performance can be discussed using diverse themes, e.g., API adoption in a multi-threaded environment. One probable solution to this is to relax the constraints on the number of topics and on the number of words describing a topic (e.g., more than 10 words per topic). However, such constraint relaxation can introduce the following problems, i.e., increase in generic keywords.
2) **Generality.** Some keywords were common across the aspects. The generic keyword 'standard' is found in the topics of both 'Usability' and 'Security'. For the aspect Bug, one such uninformative keyword was 'getters'. One possible solution would be to consider the generic keywords as *stopwords*. Our list of stopwords is provided by the Scikit-Learn library, which is used across different domains. In our future work, we will analyze the techniques to automatically identify domain-specific stopwords [60] and apply those on our rule-based aspect classifiers.

We assigned a sentence to an aspect, if we found at least one keyword from its topic description in the sentence (i.e., Jaccard similarity index greater than 0). Determining the labeling based on a custom threshold from the Jaccard index values can be a non-trivial task. In Table 16, the two columns JT and JF show the average Jaccard index for all the sentences labelled as the aspect (T) and not (F) in the benchmark. While for most of the aspects, the values of JF are smaller than the values of JT, it is higher for 'Other General Features'. Moreover, the values of JT are diverse (range [0.014 – .09]), i.e., finding a common threshold for each aspect may not be possible.

Automatic Detection of API Aspects (RQ4)
Rule-Based Classification

The performance of rule-based aspect detectors suffer from the absence of *specific* and the presence of *generic* keywords in the topic description. The mere presence of keywords to detect the aspects in the API reviews may not produce satisfactory results, due to the contextual nature of the information required to detect the aspects.

7.2 Supervised Aspect Detection

Because the detection of the aspects requires the analysis of textual contents, we investigated the supervised classifiers using

6. We used the NLTK tokenization algorithm and the stopwords provided by Scikit-learn library.

the Bag-of-Words (BoW) model. In a BoW model, each sentence is represented as a bag of words, without considering the inherent grammar or word order. To train and test the performance of the classifiers, we used 10-fold cross-validation. To report the performance of the classifiers, we use precision, recall, and F1-score as introduced in Section 4. In addition, we report two other measures: 1) MCC (Matthews correlation coefficient), and 2) AUC (Area Under ROC Curve).

7.2.1 Algorithm

We selected two supervised algorithms that have shown better performance for text labeling in both software engineering and other domains: SVM and Logistic Regression. We used the Stochastic Gradient Descent (SGD) discriminative learner approach for the two algorithms. For SVM linear kernel, we used the `libsvm` implementation. Both SGD and libsbm offered more flexibility for performance tuning (i.e., hyper-parameters) and both are recommended for large-scale learning.[7]

We applied the SVM-based classification steps as recommended by Hsu et al. [117] who observed an increase in performance based on their reported steps. The steps also included the tuning of hyper parameters. Intuitively, the opinions about API performance issues can be very different from the opinions about legal aspects (e.g., licensing) of APIs. Due to the diversity in such representation of the aspects, we hypothesized each as denoting a sub-domain within the general domain of API usage and tuned the hyper parameters of classifiers for each aspect.[8]

We analyzed the performance of the supervised classifiers on two types of the dataset.

Imbalanced. As recommended by Chawla [17], to train and test classifiers on imbalanced dataset, we set lower weight to classes with over-representation. In our supervised classifiers, to set the class weight for each aspect depending on the relative size of the target values, we used the setting as 'balanced' which automatically adjusts the weights of each class as inversely proportional to class frequencies.

Balanced/Undersampled. We produce a balanced subset of the labels for a given API aspect from the benchmark dataset by following an undersampling strategy [17]. We used the following steps [17]. For each aspect labelled in our benchmark dataset (1) We transform the labels as 1 for the sentences where the aspect is labelled. We transform the labels of all other sentences as 0, i.e., the sentence is not labelled as the aspect. (2) We determine the label with the least number of sentences. For example, for the aspect 'Legal', the minority was the label 1. (3) We create a bucket and put all the sentences with the minority label. (4) From the sentences with the majority label, we randomly select a subset of sentences with a size equal to the number of minority label.

Picking the Best Classifiers. To train and test the performance of the classifiers, we applied 10-fold cross-validation on the benchmark for each aspect as follows:

(1) We put a target value of 1 for a sentence labelled as the aspect and 0 otherwise.
(2) We tokenized and vectorized the dataset into ngrams. We used n = 1,2,3 for ngrams, i.e., unigrams (one word as a feature) to trigrams (three consecutive words). We investigated the ngrams due to the previously reported accuracy improvement of bigram-based classifiers over unigram-based classifiers [115].
(3) As recommended by Hsu et al. [117], we normalized the ngrams by applying standard TF-IDF with the optimal hyper-parameter (e.g., minimum support of an ngram to be considered as a feature).
(4) For each ngram-vectorized dataset, we then did a 10-fold cross-validation of the classifier using the optimal parameter. For the 10-folds we used Stratified sampling[9], which keeps the ratio of target values similar across the folds.
(5) We took the average of the precision, recall, and F1-score of the 10 folds.
(6) Thus for each aspect, we ran our cross-validation 18 times: nine times for the balanced dataset, and nine times for the imbalanced dataset for the aspect. Out of the nine runs for each dataset, we ran three times for each candidate classifier and once for each of the ngrams.
(7) We picked the best performing classifier as the one with the best F1-score among the nine runs for each dataset (i.e., balanced and imbalanced).

7.2.2 Evaluation

In Table 17, we report the performance of the final classifiers for each aspect for both balanced and imbalanced datasets. Except for one aspect (Security), SVM-based classifiers were the best. Except for one aspect (Performance), the precisions of the classifiers developed using the balanced dataset are better. For the aspect performance, the precision using the imbalanced dataset is 0.778 (gain +.098 over the balanced dataset). The performance gain using the balanced dataset is maximum for the aspect Bug (Precision +0.276). The precision of classifier to detect the Community aspect using the imbalanced dataset is the lowest. By using the balanced dataset, the precision of the detector to detect 'Community' has improved on average to 0.641 (i.e., a gain of +0.250). For aspects (Community, Legal, Compatibility), the number of sentences labelled as those is low (less than or around 100). Therefore, the improvement using the balanced dataset is encouraging - with more data, the same classifiers that performed poorly for the imbalanced dataset may perform better.

Unigram-based features were better-suited for most of the aspects (six), followed by bigram-based features (four). The diversity in ngram selection can be attributed to the underlying composition of words that denote the presence of the corresponding aspect. For example, performance-based aspects can be recognized through the use of bigrams (e.g., thread safe, memory footprint). Legal aspects can be recognized through singular words (e.g., free, commercial). In contrast, compatibility-based features require sequences of words to realize the underlying context.

The supervised classifiers outperformed the topic-based algorithm for two reasons: (1) The topics were based on unigrams.

7. We used the SGDClassifier of Scikit [91]
8. We computed hyper parameters using the GridSearchCV algorithm of Scikit-Learn
9. We used the Stratified KFold implementation of sklearn

TABLE 17: Performance of the supervised API aspect detectors. N = Ngram (U = Unigram, B = Bigram, T = Trigram). I = Imbalanced (The number shows the coverage in percentage of the labels for an aspect in the dataset), U/B = Understampled i.e., Balanced)

				Precision		Recall		F1 Score		MCC		AUC	
Aspect	**N**	**Classifier**	**Dataset**	**Avg**	**Stdev**	**Avg**	**Stdev**	**Avg**	**Stdev**	**Avg**	**Stdev**	**Avg**	**Stdev**
Performance	B	SVM	I 7.7	0.778	0.096	0.455	0.162	0.562	0.125	0.565	0.120	0.722	0.080
			U/B	0.680	0.060	0.796	0.103	0.732	0.072	0.433	0.147	0.711	0.070
Usability	B	SVM	I 31.8	0.532	0.045	0.749	0.101	0.620	0.055	0.416	0.092	0.721	0.111
			U/B	0.649	0.033	0.780	0.091	0.707	0.052	0.371	0.108	0.680	0.048
Security	U	Logit	I 3.6	0.775	0.272	0.578	0.174	0.602	0.160	0.624	0.148	0.696	0.082
			U/B	0.876	0.081	0.786	0.111	0.823	0.079	0.678	0.139	0.835	0.070
Community	U	SVM	I 2.1	0.391	0.323	0.239	0.222	0.256	0.204	0.271	0.217	0.614	0.110
			U/B	0.641	0.211	0.534	0.226	0.572	0.198	0.242	0.358	0.615	0.174
Compatibility	T	SVM	I 2.1	0.500	0.500	0.078	0.087	0.133	0.144	0.192	0.199	0.539	0.043
			U/B	0.619	0.086	0.664	0.112	0.637	0.082	0.249	0.186	0.622	0.091
Portability	U	SVM	I 1.5	0.629	0.212	0.629	0.223	0.608	0.190	0.613	0.190	0.811	0.111
			U/B	0.843	0.065	0.757	0.248	0.762	0.188	0.623	0.173	0.800	0.100
Documentation	B	SVM	I 5.6	0.588	0.181	0.428	0.170	0.492	0.175	0.476	0.181	0.705	0.088
			U/B	0.729	0.066	0.826	0.104	0.772	0.072	0.527	0.149	0.759	0.072
Bug	U	SVM	I 4.2	0.573	0.163	0.499	0.163	0.507	0.118	0.456	0.159	0.696	0.077
			U/B	0.849	0.106	0.742	0.106	0.786	0.080	0.598	0.158	0.794	0.078
Legal	U	SVM	I 1.1	0.696	0.279	0.460	0.180	0.523	0.188	0.455	0.195	0.688	0.083
			U/B	0.877	0.133	0.820	0.140	0.835	0.101	0.640	0.165	0.810	0.083
OnlySentiment	B	SVM	I 7.7	0.613	0.142	0.429	0.142	0.501	0.139	0.478	0.145	0.704	0.073
			U/B	0.814	0.101	0.702	0.106	0.750	0.090	0.546	0.162	0.769	0.079
General Features	U	SVM	I 37.6	0.610	0.067	0.665	0.055	0.635	0.056	0.401	0.096	0.703	0.047
			U/B	0.694	0.026	0.619	0.072	0.653	0.048	0.351	0.070	0.674	0.035

As we observed, bigrams and trigrams are necessary to detect six of the aspects. (2) The topics between aspects have one or more overlapping words, which then labelled a sentence erroneously as corresponding to both aspects while it may be representing only one of them. For example, the word 'example' is both in Usability and Documentation and thus can label the following sentence as both aspects: "Look at the example in the documentation", whereas it should only have been about documentation.

Analysis of Misclassifications. While the precisions of nine out of the 10 detectors are at least 0.5, it is only 0.39 for the aspect 'Community' in the imbalanced dataset. While the detection of the aspect 'Compatibility' shows an average precision of 0.5, there is a high degree of diversity in the detection results. For example, in second column under 'Precision' of Table 17, we show the standard deviation of the precisions across the 10 folds and it is 0.5 for 'Compatibility'. This happened because the detection of this aspect showed more than 80% precision for half of the folds and close to zero for the others. We observed two primary reasons for the misclassifications, both related to the underlying contexts required to detect an aspect: (1) **Implicit**: When a sentence was labelled based on the nature of its surrounding sentences. Consider the following two sentences: (1) "JBoss is much more popular, ... it is easier to find someone ...", and (2) "Sometimes this is more important ...". The second sentence was labelled as community because it was a continuation of the opinion started in the first sentence which was about community support towards the API JBoss. (2) **Unseen**: When the features (i.e., vocabularies) corresponding to the particular sentence were not present in the training dataset. The gain in precision for those aspects using the balanced dataset shows the possibility of improved precision for the classifiers with more labelled data. In the future we will investigate this issue with different settings, e.g., using more labelled data.

Automatic Detection of API Aspects (RQ4)
Supervised Classification

The supervised classifiers outperform the rule-based classifiers, i.e., aspects cannot be detected by merely looking for keywords/topics in the dataset. The precisions of the supervised classifiers using imbalanced dataset range from 39.1% to 77.8%. The performance of the classifiers using the undersampled dataset was superior to the classifiers using the imbalanced dataset for all aspects (except 'Performance').

8 Automatic Mining of API Opinions

In Opiner, we mine opinions about APIs using the following steps:

1) **Loading and Preprocessing.** We load Stack Overflow posts and preprocess the contents of the posts as follows: (1) We identify the stopwords. (2) We categorize the post content into four types: a) *code terms*; b) *code snippets*;[10] c) *hyperlinks*[11]; and d) *natural language text* representing the rest of the content. (3) We tokenize the text and tag each token with

10. We detect code terms and snippets as the tokens wrapped with the <code> tag in the Stack Overflow post.

11. We detect hyperlinks using regular expressions.

its part of speech.[12] (4) We detect individual sentences in the *natural language text*.

2) **Opinionated Sentence Detection.** We detect sentiments (positive and negative) for each of the sentences of a given Stack Overflow thread. An opinionated sentence contains at least one positive or negative sentiment.
3) **API Mention to Opinion Association.** We detect API mentions (name and url) in the textual contents of the forum posts. We link each API mention to an API in our API database. We associate each opinionated sentence to an API mention.

We developed a sentiment detection engine in Opiner due to the low performance we observed in the off-the-shelf sentiment detectors, as well as due to the absence of any sentiment detection tool developed specifically for software engineering during our online deployment of Opiner in January 2017. None of the currently available sentiment detection tools for software engineering were available that time. However, the modular architecture in Opiner allows the incorporation of any other sentiment detection engine, by simply replacing the existing engine in the pipeline with another sentiment detector. This design was motivated by the research goal of Opiner, i.e., facilitate a good search engine for API reviews. In Section 8.1, we discuss the two sentiment detection algorithms we developed during our design and deployment of the Opiner search engine in 2017. We also compare the two algorithms against all the available sentiment detection tools that have been developed so far for the domain of software engineering. The purpose is to offer a data-driven guidance on the best possible sentiment detection tool to be included in Opiner as we move forward.

In Section 8.2 we present the algorithms to associate opinionated sentences to the API mentions in the forum posts. In Opiner online search engine, we also summarize the opinions about an API by the detected API aspects. The Opiner system architecture was the subject of our recent tool paper about Opiner [110]. In Section 8.3, we briefly discuss the system architecture of Opiner to support automatic mining and categorization of API reviews.

8.1 Automatic Detection of Opinions (RQ5)

In this section, we first describe the two sentiment detection algorithms we developed during our development of Opiner: OpinerDSO and OpinerDSOSenti. We then compare the performance of the two algorithms against four other sentiment detection tools frequently used to detect sentiments in software artifacts. The Opiner online website currently uses the results of OpinerDSO.

8.1.1 Algorithm

• **Algorithm 1. OpinerDSO.** The detailed steps describing DSO is described in the paper [44]. We are not aware of any publicly available implementation of the algorithm. We took cues from Blair-Goldensohn et al [11], who also implemented the algorithm to detect sentiments in the local service (e.g., hotels/restaurants) reviews. DSO assigns a polarity to a sentence based only on the presence of sentiment words. We implement the algorithm in our coding infrastructure as follows.

12. We used the Stanford POS tagger [104].

1) *Detect potential sentiment words.* We record the adjectives in the sentence that match the sentiment words in our sentiment database (discussed below). We give a score of +1 to a recorded adjective with positive polarity and a score of -1 to an adjective of negative polarity.
2) *Handle negations.* We alternate the sign for an adjective if a negation word is found close to the word. We used a window of length one (i.e., one token before and one token after) around an adjective to detect negations. Thus, 'not good' will assign a sentiment value of -1 to the adjective 'good'.
3) *Label sentence.* We take the sum of all of the sentiment orientations. If the sum is greater than 0, we label the sentence as 'positive'. If less than 0, we label it as 'negative'. If the sum is 0 but the sentence does indeed have sentiment words, we label the sentence with the same label as its previous sentence in the same post.

Sentiment Database in OpinerDSO. Our database of sentiment words contain 2746 sentiment words (all adjectives) collected from three publicly available datasets of sentiment words: 547 sentiment words shared as part of the publication of DSO, 1648 words from the MPQA lexicons [119], 123 words from the AFINN dataset [67]. The DSO dataset was developed to detect sentiments in product (e.g., computer, camera) reviews. The MPQA lexicons are used in various domains, such as, news articles. The AFINN dataset is based on the analysis of sentiments in Twitter. From these three datasets, we only collected the sentiment words that were strong indicators of sentiments (discussed below).

In MPQA lexicons, the strength of each word is provided as one of the four categories: strong, weak, neutral, and bipolar. For AFINN datasets, the strength is provided in a scale of -5 to +5. A value of -5 indicates strong negative and +5 as strong positive. For the DSO dataset, we used the sentiment strength of the words as found in the SentiWordnet. SentiWordnet [29] is a lexicon database, where each Wordnet [64] synset is assigned a triple of sentiment values (positive, negative, objective). The sentiment scores are in the range of $[0, 1]$ and sum up to 1 for each triple. For instance (positive, negative, objective) = (.875, 0.0, 0.125) for the term 'good' or (.25, .375, .375) for the term 'ill'.

In addition, we automatically collected 750 software domain specific sentiment words by crawling Stack Overflow posts. We applied the following two approaches to collect those words:

Gradability. First adopted by Hatzivassiloglou and Wiebe [42], this technique utilizes the semantic property in linguistic comparative constructs that certain word can be used as an *intensifier* or a *diminisher* to a mentioned *property*. For example, the adverb 'very' can be a modifying expression, because it can intensify the sentiment towards an entity, e.g., 'GSON is very usable'. The lexicons of interest are the adjectives, which appear immediately after such an adverb, and thus can be good indicators of polarity. To apply this technique, we used the adverbs provided in the MPQA lexicons.

Coherency. We collect adjectives that appear together via conjunctions (we used 'and' and 'but') [56]. For example, for the sentence, 'GSON is slow but thread-safe', this technique will consider 'slow' and 'thread-safe' as co-appearing adjectives with opposing sentiment orientation. Thus, if we know that 'slow' is a negative sentiment, using this technique we can

TABLE 18: Confusion Matrix for the Three Class Sentiment Classification Performance Analysis

		Predicted		
		Positive (P)	**Negative (N)**	**Neutral (O)**
Actual	**Positive (P)**	TP_P	FP_N, FN_P	FP_O, FN_P
	Negative (N)	FP_P, FN_N	TP_N	FP_O, FN_N
	Neutral (O)	FP_P, FN_O	FP_N, FN_O	TP_O

mark 'thread-safe' as having a positive sentiment.

Both approaches are commonly used to develop and expand domain specific vocabularies [56]. We applied the two techniques on 3000 randomly sampled threads from Stack Overflow. Among the collected words, we discarded the words that are less frequent (e.g., with less than 1% support). None of the threads in our benchmark were part of the 3000 randomly sampled threads.

• **Algorithm 2. OpinerDSOSenti.** We detect the polarity of a sentence as follows:

1) We apply DSO and Sentistrength on the sentence. DSO produces a label for the sentence as either 'p' (positive), 'n' (negative), or 'o' (neutral). Sentistrength produces a two scores (one negative and one positive). We add the two scores and translate the added score into one of the three polarity levels as follows: A score of 0 means the sentence is neutral, greater than 0 means it is positive and negative otherwise.
2) We assign the final label to the sentence as follows:
   a) If both agree on the label, then we take it as the final label.
   b) If they disagree, we take the label of the tool that has the most coverage of sentiment vocabularies in the sentence.
   c) If both tools have similar coverage of sentiment words in the sentence and they still disagree, we assign the label of DSO to the sentence, if the sentence has one or more domain specific words. Otherwise, we take the label of the Sentistrength. The purpose was to put more emphasis on domain specific sentiment words, when those words were not present or had different sentiment labels in Sentistrength. For example, consider the following sentence: "The API is simple to use". The word 'simple' can be considered negative for movie reviews [57], but for the above sentence it is a positive review.

8.1.2 Evaluation

We analyze the performance of OpinerDSO and OpinerDSOSenti, along with the three sentiment detection techniques developed for the domain of software engineering, i.e., 1) SentistrengthSE [47] 2) SentiCR [1], and 3) Senti4SD [13]. We also compare each tool against a cross-domain sentiment detection tool (i.e., baseline), Sentistrength [103], following [71]. Sentistrength has been widely used to mine sentiments from software repositories [49].

• **Experimental Setting.** Senti4SD and SentiCR are supervised sentiment detectors, while the other four tools are rule-based. To enable a fair comparison among the tools, we adopted the experimental setting of Novieli et al. [71] as follows: 1) We split our benchmark dataset into training (70%) and test (30%) using the stratified sampling module (`StratifiedShuffleSplit`) from Python Scikit Learn [90] package. In total, training set has 3,165 sentences and the testing set has 1357 sentences. Both the training and testing datasets are highly imbalanced, i.e., skewed towards the neutral sentences (58.3% neutral sentences, 23.1% positive, 18.6% negative). This distribution is similar to previous sentiment polarity benchmarks created for software engineering [13], [70]. 2) We retrain the two supervised classifiers on the training set, by replicating the training steps provided in the corresponding paper and website (SentiCR and Senti4SD websites were accessed on Oct 5, 2018). 3) The performance of each tool including the unsupervised classifiers is then assessed on the test set.

To assess the performance of each tool, we use the confusion matrix in Table 18, which is consistent with the state of the art [71], [13], [92]. We report the performance for each polarity class as well as the overall performance using both micro and macro averages. Macro average is useful when we would like to emphasize the performance on classes with few instances. This is useful when the detection of the minority class is more important (e.g., positive/negative in our case). In contrast, micro-averaging is influenced mainly by the performance of the majority class (e.g., neutral), because it merely takes the average of each polarity class. We use F1-score to report the best detector following state of the art [71], [61].

• **Results.** In Table 19, we report the performance of each sentiment detection tool. For each tool, we first report the precision, recall and F1-score for the three polarity classes and then the Macro and Micro averages. For each class, we highlight the best value in bold. All the tools perform better than the baseline, i.e., Sentistrength. The best performing tool is Senti4SD (F1-score micro = 0.510, macro = 0.624). OpinerDSO achieves the best performance when detecting positive (F1-score = 0.429) and negative sentences (F1-score = 0.368). However, both the supervised classifiers (SentiCR and Senti4SD) show the best precision when detecting positive and negative sentences. The two supervised classifiers show lower recalls than OpinerDSO, i.e., they are more conservative in their detection of the polarized sentences, but are more accurate in their detection of the polarity of the sentences. OpinerDSO is the worst performer when detecting the neutral classes, while Senti4SD is the best performer for neutral classes (F1-score = 0.510). Indeed, the decision to optimize the performance of a tool by precision, recall, or F1-score is context and problem specific (see Subramanian et al. [101] and Calefato et al. [13]).

In Table 20, we report the agreement of the tools: first with the manual labels and then with each others. We report the agreement using the weighted Cohen κ and percent agreement measures defined in Section 5.4. Overall, the tools show more agreement than disagreement with the manual levels as well as with each other. Out of the two types of disagreements (i.e., severe and mild), the tools show a much greater amount of mild disagreement. Therefore, the tools show disagreement with the manual labeling more in the following cases: 1) When the manual label is neutral but the tool labels it as non-neutral, or 2) When the manual label is non-neutral but the tool labels it as neutral. The tools disagree much less with the manual labeling (maximum 8.7% by OpinerDSO) in the following two cases: 1) When the manual label is positive and tool labels it as negative, 2) When the manual label

TABLE 19: Accuracy analysis of the opinionated sentence detection techniques on the benchmark in Section 5

	Positive			Negative			Neutral			Macro			Micro		
	P	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
OpinerDSO	0.383	**0.489**	**0.429**	0.376	**0.361**	**0.368**	**0.716**	0.646	0.679	0.492	**0.499**	0.495	0.557	0.557	0.557
OpinerDSOSenti	0.397	0.371	0.383	0.435	0.266	0.330	0.675	0.777	0.722	0.502	0.471	0.486	0.588	0.588	0.588
Senti4SD	0.507	0.326	0.397	**0.496**	0.222	0.307	0.661	0.870	**0.751**	**0.555**	0.473	**0.510**	**0.624**	**0.624**	**0.624**
SentiCR	**0.530**	0.169	0.257	0.324	0.044	0.077	0.609	**0.941**	0.739	0.488	0.385	0.430	0.596	0.596	0.596
SentistrengthSE	0.451	0.147	0.222	0.477	0.163	0.243	0.616	0.909	0.734	0.515	0.406	0.454	0.595	0.595	0.595
Sentistrength	0.390	0.339	0.362	0.365	0.286	0.321	0.661	0.741	0.699	0.472	0.455	0.463	0.564	0.564	0.564

TABLE 20: Agreement between the sentiment classifiers and Manual labels on the benchmark in Section 5

		Manual	OpinerDSOSenti	Senti4SD	SentiCR	SentistrengthSE	Sentistrength
OpinerDSO	**Weighted κ**	0.229	0.724	0.290	0.149	0.245	0.744
	Perfect Agg	55.7%	83.3%	62.6%	57.1%	60.4%	85.3%
	Disagreement Severe	8.7%	0.9%	4.3%	1.3%	1.5%	1.9%
	Disagreement Mild	35.6%	15.8%	33.0%	41.6%	38.0%	12.7%
OpinerDSOSenti	**Weighted κ**	0.213	–	0.278	0.191	0.439	0.441
	Perfect Agg	58.8%		69.0%	69.3%	77.1%	71.3%
	Disagreement Severe	6.3%		3.5%	0.9%	0.7%	2.3%
	Disagreement Mild	34.9%		27.4%	29.8%	22.3%	26.5%
Senti4SD	**Weighted κ**	0.244			0.263	0.274	0.323
	Perfect Agg	62.4%			78.3%	77.2%	69.2%
	Disagreement Severe	3.9%			0.7%	1.7%	2.6%
	Disagreement Mild	33.7%			20.9%	21.1%	28.2%
SentiCR	**Weighted κ**	0.108			–	0.200	0.163
	Perfect Agg	59.6%				82.7%	66.8%
	Disagreement Severe	1.7%				0.5%	0.9%
	Disagreement Mild	38.7%				16.8%	32.4%
SentistrengthSE	**Weighted κ**	0.141				–	0.308
	Perfect Agg	59.5%					70.7%
	Disagreement Severe	2.1%					0.9%
	Disagreement Mild	38.4%					28.4%
Sentistrength	**Weighted κ**	0.188					–
	Perfect Agg	56.4%					
	Disagreement Severe	6.3%					
	Disagreement Mild	37.3%					

is negative and the tool labels it as positive.

Misclassification Analysis. We investigated the misclassifications of the sentiment detection tools following the methodology originally proposed by Novieli et al. [71]: 1) We picked the sentences for which each tool disagrees with the manual labels, i.e., all tools were wrong. 2) We manually label the reason of misclassification for each such sentence. For the manual labeling, we use the same seven error categories as originally observed by Novieli et al. [71] in their analysis of misclassifications by sentiment tools in software engineering: 1) Polar facts by neutral, 2) General error, 3) Politeness, 4) Implicit sentiment polarity, 5) Subjectivity in sentiment annotation, 6) Inability of classifiers to deal with context information, and 7) Figurative language. Figure 12 shows the distribution of the misclassification categories in our evaluation, i.e., where all tools were wrong. Around half (47%) of the misclassifications occurred due to the inability of the tools to deal with contextual information. The presence of implicit sentiment polarity is the reason of 24% of the misclassifications, followed by the presence of general error (20%). While our distribution of general error is similar to Novieli et al. [71], they observed much less distribution of errors due to presence of contexts (only 10%). We did not observe the error category "Polar facts by neutral" in our assessment. We discuss the misclassifications below.

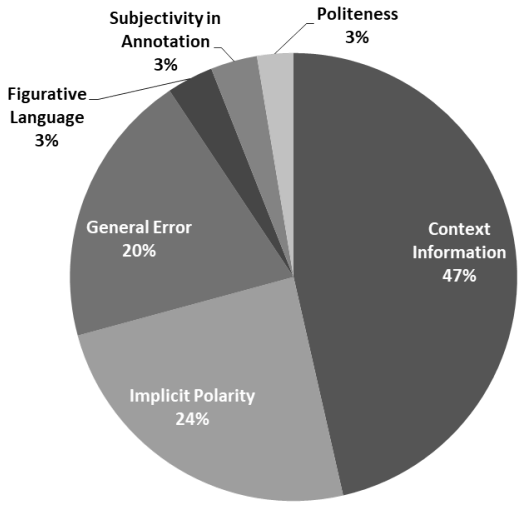

Fig. 12: Distribution of categories in sentiment misclassifications

1) **Inability of classifiers to deal with pragmatics or context information.** The tools were unable to detect the polarity (positive or negative) of sentence and instead labeled it as 'neutral' when the polarity was determined by the contextual information *before*

the sentence. The following sentence is labeled as positive by the annotators: *"It does not depend on the Swing Application Framework at all."* All the tools label it as neutral. This sentence is labeled as positive by the annotator, because the previous sentence is polarized *"Yeah that is not true."* and the post is on the needs to avoid the dependency on the Swing framework for user interface development in Java. Novieli et al. [71] observed 20% of the misclassifications attributed to the presence of contextual and pragmatic information in their Stack Overflow dataset used to train the Senti4SD [13] (10% overall across all four datasets analyzed in [71]). In our test dataset, we observed a greater concentration of such sentences (47%). This is due to the fact that our benchmark dataset consists of all the sentences from 71 Stack Overflow threads. Each annotator was presented all the sentences of a thread in the same order as they appeared in the thread itself. Therefore, the annotators were able to use the contextual information during their polarity labeling of sentences. For the Senti4SD Stack Overflow benchmark dataset, the sentences were sampled opportunistically from Stack Overflow, based on the presence of polarity lexicon (using the Sentistrength tool). None of the tools was developed to handle such contextual information. To detect polarity in such cases, the overall *intention* in the post as well as the polarity of the preceding sentences need to be analyzed.

2) **Implicit sentiment polarity.** In 24% of the cases the tools fail to detect the polarity because the sentiment is implicit, such as when the sentiment vocabulary is not present or there was not explicit polarized cues. The following sentence was labeled as negative by the human annotators, because of the *incompatibility* of the API (simmetrics) with another software, *"simmetrics looks to be GPL v2, however, so not compatible with commercially developed software."*. The cue in the above sentence is "compatible", but it is not included as a sentiment lexicons of any of the tools. The 'compatibility' of an API can be an important aspect, as we already noted in our API aspect analysis.

3) **General Error.** In 20% of the cases, the tools fail to detect the polarity due to two major reasons: a) *Convoluted Sentences*, such as when the sentence has multiple clauses and the polarity is determined based on one of the clauses. For example, the human annotators labeled the following sentence as negative due to the negativity expressed after the clause that started with the word *but*, *"I looked at the grammar of some PL/SQL Parsers as well as Lexers and Parsers but was unable to fathom how to use one."* In such cases, a clause-level sentiment detector could have been a better choice. b) *Emoticons*, such as the presence of informal languages, icons, or symbols used to express sentiment.

4) **Figurative Language.** Similar to Novieli et al. [71], we also observed the presence of sarcasms in the sentences which carried sentiment. For example, the human raters considered the following sentence as positive, but three tools (Sentistrength, OpinerDSO, and Senti4SD) labeled it as negative and the other tools labeled it as neutral: *"2005 is not Jurassic period and I think you shouldn't dismiss a library (in Java, at least) because it hasn't been updated for 5 years."*. Sarcasm detection is a challenging sub-field of sentiment detection. None of the tools in our study are designed to detect such sarcasms.

5) **Subjectivity in Annotation.** Similar to Novieli et al. [71], we also observed that in 3% of the cases our human raters were too conservative to label a sentence as positive or negative, but the tools detected it as such. For example, the following sentence was erroneously labeled as neutral by the human raters, but the tools labeled it as positive, *"It is a great alternative that might work better for you."*

6) **Politeness.** The human raters considered politeness expressed in some sentences as indicative of positivity, which the tools were unable to detect as such. For example, the following sentence was labeled as neutral by all the tools, *"@Pascal - You are right."*

OpinerDSO vs OpinerDSOSenti. We expected the OpinerDSOSenti to offer better performance than OpinerDSO. However, OpinerDSO shows slightly better performance than OpinerDSOSenti (Macro F1-score 0.495 for OpinerDSO vs 0.486 for OpinerDSOSenti). While OpinerDSO offers better performance to detect the positive and negative classes, OpinerDSOSenti offers better performance to detect the neutral classes. The superiority of OpinerDSOSenti for the detection of neutral classes as well as its lower performance when detecting negative classes are due to the bias of Sentistrength tool towards the recognition of negative and neutral sentences. Indeed, similar observations about the Sentistrength tool were previously reported by both Novieli et al. [71] and Jongeling et al. [49].

Performance Cross-Check. We sought to understand whether the low performance of the tools could be due to any particular aspect of our benchmark. To investigate this, we repeated our comparison of the performance of all the six tools using another benchmark created from Stack Overflow by Lin et al. [54]. Besides our benchmark and the benchmark of Lin et al. [54], the benchmark used to train Senti4SD tool also used Stack Overflow. We refer to [71] for a comparison of the four tools (Senti4SD, SentiCR, SentistrengthSE, and Sentistrength) on the Senti4SD benchmark.

In Table 21, we show the performance of the tools using the benchmark of Lin et al. [54]. In Table 22, we show the agreement of the tools with the manual labels as well as with each other. Similar to our benchmark dataset, Senti4SD performed the best. However, the difference is only 0.1 (F1-score Macro). This slightly higher performance could be attributed to the difference in how the two datasets are sampled. For our benchmark dataset, we picked all the sentences of a given Stack Overflow thread. In the benchmark of Lin et al. [54], the sentences were picked randomly. As we noted above, the major reasons for low performance of the tools in our benchmark is due to the inability of the tools to detect contextual information. The human raters for the dataset of Lin et al. [54] did not have access to such contextual information, and as such they used a given sentence as is in their annotation.

The differences in performance of the tools (Senti4SD, SentistrengthSE, SentiCR) in our benchmark and the dataset of Lin et al. [54] are lower than their performances in the Senti4SD dataset [71]. This can be attributed to the way the benchmarks are created. The Senti4SD authors [13] followed Shaver's emotion framework to train the human annotators during the creation of their benchmark. We and Lin et al. [54] did not follow the Shaver framework in the benchmark creation process. However, as we noted in Section 5, we used a coding guide as well as group discussions to train the human annotators. A detailed investigation of an appropriate benchmarking approach for the calibration and

TABLE 21: Accuracy analysis of the opinionated sentence detection techniques on the benchmark from Lin et al. [54]

	Positive			Negative			Neutral			Macro			Micro		
	P	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
OpinerDSO	0.400	0.462	0.429	**0.500**	0.278	0.357	0.859	0.902	0.880	0.586	0.547	0.566	0.789	0.789	0.789
OpinerDSOSenti	0.289	**0.615**	0.393	0.381	**0.444**	**0.410**	**0.895**	0.762	0.823	0.522	**0.607**	0.561	0.711	0.711	0.711
Senti4SD	**0.867**	0.333	**0.481**	0.486	0.315	0.382	0.850	**0.952**	**0.898**	**0.734**	0.534	**0.618**	**0.822**	**0.822**	**0.822**
SentiCR	0.800	0.308	0.444	0.457	0.296	0.360	0.845	0.947	0.893	0.701	0.517	0.595	0.813	0.813	0.813
SentistrengthSE	0.276	0.205	0.235	0.200	0.056	0.087	0.805	0.916	0.857	0.427	0.392	0.409	0.751	0.751	0.751
Sentistrength	0.193	0.436	0.268	0.339	0.352	0.345	0.856	0.734	0.790	0.463	0.507	0.484	0.662	0.662	0.662

TABLE 22: Agreement between the sentiment classifiers and Manual labels on the benchmark of Lin et al. [54]

		Manual	OpinerDSOSenti	Senti4SD	SentiCR	SentistrengthSE	Sentistrength
OpinerDSO	**Weighted κ**	0.333	0.607	0.327	0.190	0.344	0.242
	Perfect Agg	78.9%	83.1%	82.9%	80.0%	84.2%	68.9%
	Disagreement Severe	1.6%	0.2%	0.4%	1.1%	0.4%	1.6%
	Disagreement Mild	19.6%	16.7%	16.7%	18.9%	15.3%	29.6%
OpinerDSOSenti	**Weighted κ**	0.317		0.325	0.225	0.280	0.711
	Perfect Agg	71.1%		73.3%	70.0%	72.7%	85.8%
	Disagreement Severe	2.9%		0.4%	1.1%	0.7%	1.3%
	Disagreement Mild	26.0%		26.2%	28.9%	26.7%	12.9%
Senti4SD	**Weighted κ**	0.371			0.487	0.287	0.242
	Perfect Agg	82.2%			90.2%	86.2%	70.4%
	Disagreement Severe	0.7%			0.9%	0.4%	0.7%
	Disagreement Mild	17.1%			8.9%	13.3%	28.9%
SentiCR	**Weighted κ**	0.341				0.175	0.209
	Perfect Agg	81.3%				84.4%	70.0%
	Disagreement Severe	0.7%				0.9%	1.6%
	Disagreement Mild	18.0%				14.7%	28.4%
SentistrengthSE	**Weighted κ**	0.104					0.364
	Perfect Agg	75.1%					75.8%
	Disagreement Severe	0.7%					0.2%
	Disagreement Mild	24.2%					24.0%
Sentistrength	**Weighted κ**	0.208					
	Perfect Agg	66.2%					
	Disagreement Severe	2.9%					
	Disagreement Mild	30.9%					

the evaluation of tools to detect opinionated sentences in Opiner is our future work.

Automatic Detection of Opinions (RQ5)

Given the superior performance of Senti4SD on our benchmark as well as on the benchmark of Lin et al. [54], one could consider replacing the current sentiment detection engine in Opiner (i.e., OpinerDSO) with Senti4SD. Given that OpinerDSO performs better than Senti4SD in the detection of positive and negative sentences, a fusion of the two engines may produce even better results. Given the considerable presence of contextual information in the misclassified sentences, a future improvement should also take into account *surrounding* context when detecting the polarity of a sentence.

8.2 Association of Opinions to API Mentions (RQ6)

In Opiner, we mine opinionated sentences about APIs by associating the detected opinions to the APIs mentioned in the forum posts. First, we detect potential API mentions (name and url) in the *textual contents* of the forum posts. Second, we link those mentions to the APIs listed in the online software portal, such as, Maven central. Third, we associate the opinionated sentences to the detected API mentions. We evaluate and report the performance of linking on our benchmark dataset.

TABLE 23: Statistics of the Java APIs in the database.

API	Module	Version	Link
62,444	144,264	603,534	72,589

8.2.1 Algorithm

The algorithm is composed of five major components:

1) Portal operation,
2) Name and url preprocessing,
3) API name detection, and
4) API hyperlink detection.
5) API mention to opinion association

Even though we developed and evaluated our API mention resolution technique using Java APIs only, the technique is generic enough to be applicable to detect APIs from any other programming languages. We describe the steps below.

Step 1. Portal Operation.
Our API database consists of all the Java official APIs and the open source Java listed in two software portals Ohloh [72] and Maven central [96].[13] Each entry in the database contains a reference to a Java API. An API is identified by a name. An API consists of one or more modules. For each API, we store the following fields: 1) *API name*; 2) *module names*; 3) resource *links*, e.g., download page, documentation page, etc.; 4) *dependency on another API*; 5) *homepage url*: Each module schema in Maven normally contains a homepage link named "url" . We discarded API projects that are of type 'demo', or 'hello-world' (e.g., `com.liferay.hello.world.web`). We crawled the javadocs of five official Java APIs (SE 6-8, and EE 6,7) and collected information about 875 packages and 15,663 types. We consider an official Java package as an API, following similar format adopted in the Java official documentation (e.g., the `java.time` package is denoted as the Java date APIs in the new Java official tutorial [73]). In Table 23, we show the summary statistics of the API database.

Step 2. Name and Hyperlink Preprocessing.
We preprocess each API name to collect representative tokens as follows: (a) *Domain names*: For an API name, we take notes of all the domain names, e.g., for `org.apache.lucene`, we identify that `org` is the internet domain and thus developers just mention it as `apache.lucene` in the post. (b) *Provider names*: we identify the provider names, e.g., for `apache.lucene` above, we identify that `apache` is the provider and that developers may simply refer to it as `lucene` in the posts. (c) Fuzzy combinations: We create fuzzy combinations for a name with multiple tokens. For example for `org.apache.lucene`, we create the following combinations: apache lucene, apache.lucene, lucene apache, and apache-lucene. (d) Stopwords: we consider the following tokens as stopwords and remove those from the name, if any: test, sample, example, demo, and code. (e) Country codes: we remove all two and three digit country codes from the name, e.g., cn, ca, etc. For each such API name, we create two representative tokens for the name, one for the full name, and another the name without the code. We preprocess each url as follows: 1) Generic links: We remove hyperlinks that are most likely not pointing to the repository of the API. For example, we removed this hyperlinks http://code.google.com/p, because it just points to the code hosting repository of Google code instead of specifying which project it refers to. 2) Protocol: For an API with hyperlink using the 'HTTP' protocol, we also created another hyperlink for the API using the 'HTTPS' protocol. This is because the API can be mentioned using any of the hyperlink in the post. 3) Base url: For a given hyperlink, we automatically created a base hyperlink by removing the *irrelevant* parts, e.g., we created a base as http://code.google.com/p/gson/ from this hyperlink http://code.google.com/p/gson/downloads/list.

Step 3. API Name Detection.
We match each token in the textual contents of a forum post against each API name in our database. We do two types of matching: exact and fuzzy. If we find a match using exact matching, we do not proceed with the fuzzing matching. We only consider a token in a forum post eligible for matching if its length is more than three characters long. For an API name, we start with its longest token (e.g., between code.gson.com and gson, code.gson.com is the longest token, between 'google gson' and 'gson', 'google gson' is the longest), and see if we can match that. If we can, we do not proceed with the shorter token entities of the API. If for a given mention in the post we have more than one exact matches from our database, we randomly pick one of them. Using contextual information to improve resolution in such cases is our future work. If we don't have any exact match, we do fuzzy match for the token in forum post against all the API names in our database. We do this as follows: (1) we remove all the non-alpha characters from the forum token; (2) we then make it lowercase and do a levenshtein distance; (3) we pick the matches that are above 90% threshold (i.e., more than 90% similar) between the token and an API and whose first character match (i.e., both token and API name starts with the same character) (4) If there are more than one match, we pick the one with the highest confidence. If there is a tie, we sort the matches alphabetically and pick the top one.

Step 4. API Hyperlink Detection.
We do not consider a hyperlink in a forum post if the url contains the following keywords: stackoverflow, wikipedia, blog, localhost, pastebin, and blogspot. Intuitively, such urls should not match to any API hyperlink in our database. For other urls, we only consider urls that start with http or https. This also means that if a hyperlink in the post is not properly provided (i.e., broken link), we are unable to process it. For all other hyperlinks in the forum post, we match each of them against the list of hyperlinks in our database. Similar to name matching, we first do an exact match. If no exact match is found, we do a fuzzy match as follows. 1) We match how many of the hyperlinks in our database start with the exact same substring as the hyperlink in the post. We collect all of those. For example, for a hyperlink in forum post http://abc.d.com/p/d, if we have two hyperlinks in our database http://abc.com and http.abc.d.com, we pick both as the fuzzy match. From these matches, we pick the one with the longest length, i.e., http.abc.d.com.

Step 5. Associate an Opinionated Sentence to an API Mention.
Our technique leverages concepts both from the hierarchical structure of the forum posts and the co-reference resolution techniques in text mining [61]. We associate an opinionated sentence in a post to an API about which the opinion is provided using three filters:

F1) API mentioned in the **same sentence.**
F2) API in **same post.**
F3) API in **same conversation.**

We apply the filters in the order in which they are discussed.

• ***F1. Same sentence association.*** If the API was mentioned in the same sentence, we associate the opinion to the API. If more than one API is mentioned in the same sentence, we associate the opinion to all of the APIs. We are currently investigating an improvement of this approach by associating an API to its closest opinion in a sentence. The technique is still under development, and it is not part of the currently deployed version of Opiner.

13. We crawled Maven in March 2014 and Ohloh in Dec 2013. Ohloh was renamed to Black Duck Open Hub in 2014.

• ***F2. Same post association.*** If an API was not mentioned in the same opinionated sentence, we attempted to associate the sentence to an API mentioned in the same post. First, we look for an API mention in the following sentence of a given opinionated sentence. If one mention is found there, we associate the opinion to the API. If not, we associate it to an API mentioned in the previous post (if any). If not, we further attempt to associate it to an API mentioned in the following two sentences. If none of the above attempts are successful, we apply the next filter.

• ***F3. Same conversation association.*** We define a conversation as an answer/question post and the collection of all comments in reply to the answer/question post. First, we arrange all the posts in a conversation based on the time they were provided. Thus, the answer/question is the oldest in a conversation. If an opinionated sentence is found in a comment post, we associate it to its nearest API mention in the same conversation as follows: an API mentioned in the immediately preceding comment or the nearest API mentioned in the answer/question. A window is enforced to ensure minimum distance between an API mention and opinionated sentence. For example, for an answer, the window should only consider the question of the post and the comments posted as replies to the answer. For a comment, only the comments preceding it should be included, as well as the answer about which the comment is provided.

8.2.2 Evaluation

API Mention Resolution. We analyzed the performance of our API mention resolution algorithm by applying it on all the 71 threads of our benchmark dataset. The first author further annotated the benchmark dataset by manually identifying the list of APIs mentioned in each of the 4522 sentences of the benchmark. We used the following four sources of information to identify the APIs: (a) Our API database (b) The Stack Overflow thread where the mention was detected, (c) The Google search engine, and (d) The homepages of the candidate APIs for a given mention. If an API was not present in our database, but was mentioned in the forum post, we should considered that to be a missed mention and added that in the benchmark. All such missed mentions were considered as false negatives in the accuracy analysis.

For each sentence in our benchmark, we then compared the results of our mention resolution algorithm against the manually compiled list of APIs. For each sentence, we produced a confusion matrix as follows:

True Positive (TP). The resolved mention matches exactly with the API manually identified as mentioned.

False Positive (FP). The algorithm identified an API mention, but the benchmark does not have any API listed as mentioned.

False Negative (FN). The algorithm did not identify any API mention, but the benchmark lists one or more APIs as mentioned in the sentence.

True Negative (TN). None of the algorithm or the benchmark identify any API mentioned in the sentence. Intuitively, for a sentence where no mention is detected, the TN = 1. We observed that this approach can artificially inflate the accuracy of the algorithm, because the majority of the sentences do not have any API mentioned. We filter some API name specific noises during the preprocessing of the forum contents (ref. Step 2 - Name and Hyperlink Preprocessing). It was useful to ignore sentences with those *stopwords* that were not potentially pointing to any API. However, the focus of our mention resolution technique is to *correctly link an API mention to an API*. We thus discard all such TNs from our performance measurement and consider TN = 0.

TABLE 24: Accuracy analysis of the API mention resolver

Resolver	Precision	Recall	F1-Score
Name-Based Resolution	0.988	0.923	0.954
Link-Based Resolution	0.99	0.944	0.966
Overall	0.988	0.926	0.956

In Table 24, we present the performance of the mention resolution algorithm. The overall precision is 98.8% with a recall of 92.6% (F1 score = 95.6%) and an accuracy of 91.7%. The name resolution technique has a precision of 98.8%, with a recall of 92.3% and an accuracy of 91.3%. The link resolution technique has a precision of 99%, but with a recall of 94.4% and with an accuracy of 93.5%. We considered the following resolution of API names as wrong: 1) Database. The mention is about a database (e.g., cassandra) and not the reuse of the database using programming. 2) Specification. The mention is about an API standard, and not the actual implementation of the standard (e.g., JAAS). 3) Server. The mention is about a server (e.g., tomcat).

Both the name and link mention resolver techniques are precise at linking an API mention to an actual API. The algorithms show high precision, due to the fact that the matching heuristics were based mainly on the exact matching of the names/urls. Thus, when a match was found, it was mostly correct. However, both of the techniques show lower recall than the precision, due to around 100 API mentions completely missed by the mention detector. There are two reasons for the missing: 1) Missing in API Database. The mentioned API is not present in our API database. For example, one of the missed mentions was NekoHtml, which is not present in our API database. 2) Error in Parsing. The textual content is not properly parsed and thus the exact/fuzzy matcher in the mention detector missed the mention. For example, the parser could not clean the name 'groovy' from this input "[groovy's-safe-navigation-operator]". 3) Typo. The API mention was misspelled. 4) Abbreviation. The API name is mentioned using an abbreviation. For example, the Apache HttpClient was mentioned as 'HC' in this input "If HC 4.0 was perfect then that would be a good point, but it is not." While the performance of the resolver is promising, we note that the tool currently does not take into account the context surrounding a mention and thus may not work well for false mentions, such as 'Jackson' as an API name versus as a person name, etc. We did not encounter such ambiguous names in our benchmark dataset. We have been investigating the feasibility of incorporating contextual information into the resolution process as our immediate future work [112].

TABLE 25: Accuracy analysis of the API to opinion association

Rule	Precision	Recall	F1 Score
Same Sentence	0.959	0.922	0.940
Same Post	0.885	0.885	0.885
Same Conversation	0.667	0.444	0.533
Overall	0.914	0.860	0.886

Association of Opinions to API Mentions (RQ6)
API Mention Resolution

Our API name detection technique can detect and resolve API names with 98.8% precision and 92.6% recall. Further improvements can be possible with the analysis of contextual information (e.g., how forum contents are linked to API features) during the resolution process.

API Mention to Opinion Association. We applied our association technique on all the 71 threads of our benchmark dataset. For each opinionated sentence, the technique either associated the sentence to an API mention or left it blank when no potential API mention could be associated. For each association, the technique also logged which of the three rules (Same Sentence, Same Post, and Same Conversation) was used to determine the association. With a 99% confidence interval, a statistically significant subset of the 4522 benchmark sentences should have at least 580 sentences. We manually analyzed a random subset of 710 sentences from the benchmark. For each opinionated sentence in the 710 sentences, we created a confusion matrix as follows:

True Positive (TP). The opinionated sentence is correctly associated with the right API mentioned.

False Positive (FP). The opinionated sentence is incorrectly associated to an API. The error can be of the following types:

- **T1 - Generic Opinion.** The opinion is not about any API, rather towards a developer/entity in general.
- **T2 - Wrong Association.** The opinion should be associated to another API.

True Negative (TN). The opinionated sentence is correctly not associated to any API. Intuitively, any opinionated sentence not associated to an API can be considered as a true negative. However, this can *artificially* inflate the performance of the rules, when the post is not about an API at all. For example, we observed forum posts where developers discussed about the relative benefits of technology standards. Such discussions are opinionated, but do not have any mention or reference to API (e.g., XML vs JSON [75]). Thus, we consider true negatives only in threads where APIs are mentioned based on following guidelines:

- **TN - Yes.** The thread has a mention of one or more APIs. The opinionated sentences thus can be associated to the APIs. If an opinionated sentence is not associated to an API in such situation, and the algorithm did not associate the opinion to any API, then we consider that as true negative.
- **TN - No.** The thread does not have a mention of any API, but it has opinionated sentences. If the algorithm did not associate any such opinion to any API, we do not consider that as a true negative. We discard such opinionated sentences from our evaluation.

False Negative (FN). The algorithm did not associate the opinionated sentence to an API mention, when the opinion was indeed about an API.

In Table 25, we show the performance of our association algorithm. Among the three rules, the 'Same Sentence' rule showed the best precision. The 'Same Conversation' rule performed the worst. The rule was more effective to find the true negatives. In fact, out of the 34 detected true negatives, 33 were detected by the 'Same Conversation' rule. The reason for this is that developers provide opinions not only about APIs, but also about other developers or entities in the forum posts. The window enforced to determine a conversation boundary is effective to discard API mentions that are far away from a conversation window. Such filtering is found to be effective in our evaluation. However, compared to the other two rules (Same Sentence and Same Post), the *number of opinionated sentences* considered in the Same Conversation is more, i.e., the rule is more greedy while associating an opinion to an API mention. This is the reason why we obtained more false associations using this rule. The Same Sentence rule has the most number of true positives, i.e., developers in our benchmark offered most of their opinions about an API when they mentioned the API. The Same Post rule incorrectly associated an opinion to an API, when more than one API was mentioned in the same post and the opinion was not about the API that was the closest.

Association of Opinions to API Mentions (RQ6)
API Mention to Opinion Association

Our opinion mining technique can successfully associate an opinion to an API mentioned in the same thread where opinion was found with a precision of 66.7%-95.9%. The precision value drops when the distance between an API mention and the related opinion is more than a sentence, and when more than one API is mentioned within the same context of the opinion.

8.3 Opiner System Architecture

The Opiner website offers two online services to the users:

Search. An API can be searched by its name. Real-time suggestions of API names are provided using AJAX autocompletion principle to assist developers during their typing in the search box. Multiple APIs can be compared by their aspect. For example, a separate search box is provided, where users can type an API aspect to see the opinions related to the aspect in all the APIs.

Analytics. The mined opinions about each API mentioned in the Stack Overflow posts are shown in individual pages of Opiner. The opinions are grouped by API aspects (e.g., performance, security). The current version of Opiner uses the best performing supervised aspect detectors that we developed using the benchmark dataset (i.e., imbalanced setting).

The Opiner offline engine is separate from the online engine. The offline engine automatically crawls Stack Overflow forum posts to automatically mine opinions about the APIs using the techniques we described in this section. API aspects are detected in those mined opinions.

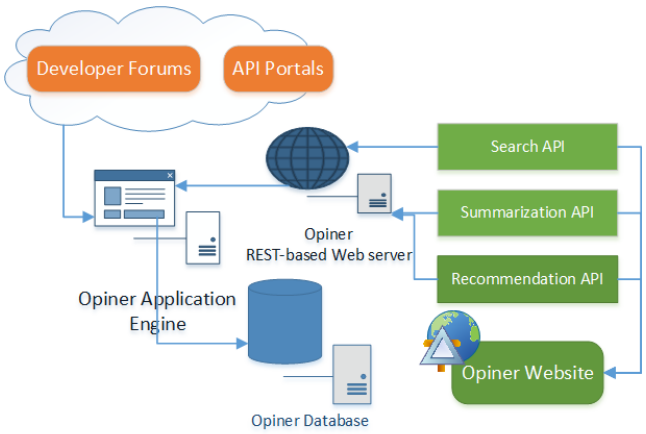

Fig. 13: The client-server architecture of Opiner

In Figure 13, we show the architecture of Opiner. The Opiner architecture supports the efficient implementation of the algorithms, and a web-based search engine with visual representation of the insights obtained from the collection and summarization of API reviews. The online crawlers and the algorithms are developed and deployed in the 'Opiner Application Engine'. The API database is hosted in a PostgreSQL relational database. The website is developed on top of the Python Django Web Framework. The communication between the Django webserver and the 'Opiner Application Engine' is handled via the Opiner REST-based middleware, which transforms data between the website and the middleware in JSON format over HTTP protocol. There are four major components in Opiner (see Figure 13):

- **Application Engine:** Handles the loading and processing of the data from forum posts and API portals and hosts all the algorithms and techniques used to collect and summarize opinions about APIs.
- **Database:** A hybrid data storage component with support for data manipulation in diverse formats, such as, relational, graph, tree, etc.
- **REST Web Server:** The middleware between the Application Engine and the Website.
- **Website:** The web-based user interface of Opiner that hosts the search, summarization and recommendation results based on API reviews

9 Threats to Validity

To ensure reproducibility of the analyses presented in this paper, we share our benchmark along with all the sentiment and aspect labels in our online appendix [111]. We also share the benchmark we used to resolve API mentions in our online appendix [111].

The accuracy of the evaluation of API aspects and mentions is subject to our ability to correctly detect and label each such entities in the forum posts we investigated. The inherent ambiguity in such entities introduces a threat to investigator bias. To mitigate the bias, we conducted reliability analysis on both the evaluation corpus and during the creation of the benchmark.

Due to the diversity of the domains where APIs can be used and developed, the generalizability of the approach requires careful assessment. While the current implementation of the approach was only evaluated for Java APIs and Stack Overflow posts discussing Java APIs, the benchmark was purposely sampled to include different Java APIs from domains as diverse as possible (18 different tags from 9 domains). The domains themselves can be generally understood for APIs from any given programming languages. In particular, the detection of API aspects is an ambitious goal. While our assessment of the techniques show promising signs in this new research direction, the results will not carry the automatic implication that the same results can be expected in general. Transposing the results to other domains requires an in-depth analysis of the diverse nature of ambiguities each domain can present, namely reasoning about the similarities and contrasts between the ambiguities.

10 Conclusion and Future Work

With the proliferation of online developer forums as informal documentation, developers share their opinions about the APIs they use. With the myriad of forum contents containing opinions about thousands of APIs, it can be challenging for a developer to make informed decision about which API to choose for his development task. In this paper, we examined opinions shared about APIs in online developer forums, and found that opinions about APIs are diverse and contain topics from many different aspects. As a first step towards providing actionable insights from opinions about APIs, we presented a suite of techniques to automatically mine opinions by aspects. We developed and deployed a tool named Opiner, that supports the development infrastructure to automatically mine opinions about APIs from developer forums. Our future work involves an in-depth exploration of API aspects in the forum posts by leveraging the aspect detection and opinion mining infrastructure we have developed in Opiner. In addition, we will extend the benchmark to explore reviews from other programming languages (e.g, javascript, python, etc.) as well as other domains, such as, platform (e.g., 'linux'), data structures (e.g., 'hashmap'), build systems (e.g., 'ant'), software project management system (e.g., 'maven'), etc.

Acknowledgments

We sincerely thank the anonymous reviewers for providing us valuable feedback to improve the paper. We are grateful to Prof Novielli for providing cues on how to train the SentiCR classifier, to run the Senti4SD classifier, and for kindly reviewing the contents of Section 8.1. We thank all the coders who participated in the development of the benchmark data and the evaluation of the algorithms. This work is partly supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] 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*, pages 106–111, 2017.

[2] A. Anderson, D. Huttenlocher, J. Kleinberg, and J. Leskovec. Steering user behavior with badges. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 95–106, 2013.

[3] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[4] A. Bacchelli, M. Lanza, and V. Humpa. Rtfm (read the factual mails) - augmenting program comprehension with remail. In *15th IEEE European Conference on Software Maintenance and Reengineering*, pages 15–24, 2011.

[5] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *32nd International Conference on Software Engineering*, pages 375–384, 2010.

[6] A. Bacchelli, T. D. Sasso, M. D'Ambros, and M. Lanza. Content classification of development emails. In *Proc. 34th IEEE/ACM International Conference on Software Engineering*, pages 375–385, 2012.

[7] J. Backon. *The decline of GPL?* https://opensource.com/article/17/2/decline-gpl, 2017.

[8] K. Bajaj, K. Pattabiraman, and A. Mesbah. Mining questions asked by web developers. In *In Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 112–121, 2014.

[9] 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*, pages 1–31, 2012.

[10] B. Bazelli, A. Hindle, and E. Stroulia. On the personality traits of stackoverflow users. In *In Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*, pages 460–463, 2013.

[11] S. Blair-Goldensohn, K. Hannan, R. McDonald, T. Neylon, G. A. Reis, and J. Reyner. Building a sentiment summarizer for local search reviews. In *WWW Workshop on NLP in the Information Explosion Era*, page 10, 2008.

[12] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3(4-5):993–1022, 2003.

[13] F. Calefato, F. Lanubile, F. Maiorano, and N. Novielli. Sentiment polarity detection for software development. *Journal Empirical Software Engineering*, pages 2543–2584, 2017.

[14] F. Calefato, F. Lanubile, M. C. Marasciulo, and N. Novielli. Mining successful answers in stack overflow. In *In Proceedings of the 12th Working Conference on Mining Software Repositories*, page 4, 2014.

[15] F. Calefato, F. Lanubile, and N. Novielli. Emotxt: A toolkit for emotion recognition from text. In *Proc. 7th Affective Computing and Intelligent Interaction*, page 2, 2017.

[16] S. Chang and A. Pal. Routing questions for collaborative answering in community question answering. In *In Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining ACM*, pages 494–501, 2013.

[17] N. V. Chawla. *Data Mining for Imbalanced Datasets: An Overview*, pages 875–886. Springer US, Boston, MA, 2010.

[18] C. Chen, S. Gao, and Z. Xing. Mining analogical libraries in q&a discussions incorporating relational and categorical knowledge into word embedding. In *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 338–348, 2016.

[19] C. Chen and Z. Xing. Mining technology landscape from stack overflow. In *The 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 14:1–14:10, 2016.

[20] C. Chen and Z. Xing. Mining technology landscape from stack overflow. In *10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 10, 2016.

[21] C. Chen, Z. Xing, and X. Wang. Unsupervised software-specific morphological forms inference from informal discussions. In *39th International Conference on Software Engineering*, pages 450–461, 2017.

[22] G. Chen, C. Chen, Z. Xing, and B. Xu. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. In *31st IEEE/ACM International Conference on Automated Software Engineering*, pages 744–755, 2016.

[23] N. Chen, J. Lin, S. C. H. Hoi, X. Xiao, and B. Zhang. Ar-miner: Mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, pages 767–778, 2014.

[24] Y. Chen, H. Xu, Y. Zhou, and S. Zhu. Is this app safe for children? a comparison study of maturity ratings on android and ios applications. In *Proceedings of the 22nd international conference on World Wide Web*, pages 201–212, 2013.

[25] J. Cohen. Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit. *Psychological Bulletin*, 70(4):213–220, 1968.

[26] D. Correa and A. Sureka. Chaff from the wheat: Characterization and modeling of deleted questions on stack overflow. In *In Proceedings of the 23rd international conference on World wide web*, pages 631–642, 2014.

[27] B. Dagenais and M. P. Robillard. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In *Proc. 18th Intl. Symp. Foundations of Soft. Eng.*, pages 127–136.

[28] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *Proc. 34th IEEE/ACM Intl. Conf. on Software Engineering*, pages 45–57, 2012.

[29] A. Esuli and F. Sebastiani. Sentiwordnet: A publicly available lexical resource for opinion mining. In *In Proceedings of the 5th Conference on Language Resources and Evaluation*, pages 417–422, 2006.

[30] FasterXML. *Jackson*. https://github.com/FasterXML/jackson, 2016.

[31] freecode.com. http://freecode.com/, 2013.

[32] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh. Why people hate your app: making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1276–1284, 2013.

[33] A. L. Ginsca and A. Popescu. User profiling for answer quality assessment in q&a communities. In *In Proceedings of the 2013 workshop on Data-driven user behavioral modelling and mining from social media*, pages 25–28, 2013.

[34] github.com. https://github.com/, 2013.

[35] M. Gómez, B. Adams, W. Maalej, M. Monperrus, and R. Rouvoy. App store 2.0: From crowdsourced information to actionable feedback in mobile ecosystems. *IEEE Software*, 34(2):81–89, 2017.

[36] Google. *Gson*. https://github.com/google/gson, 2016.

[37] L. Guerrouj, S. Azad, and P. C. Rigby. The influence of app churn on app success and stackoverflow discussions. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 321–330.

[38] 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*, pages 352–355, 2014.

[39] E. Guzman and B. Bruegge. Towards emotional awareness in software development teams. In *Proceedings of the 7th Joint Meeting on Foundations of Software Engineering*, pages 671–674, 2013.

[40] E. Guzman and W. Maalej. How do users like this feature? a fine grained sentiment analysis of app reviews. In *Proceedings of the 22nd International Requirements Engineering Conference*, pages 153–162, 2014.

[41] B. Hamner. *Metrics*. https://github.com/benhamner/Metrics, 2018. Last accessed on 24 October 2018.

[42] V. Hatzivassiloglou and J. M. Wiebe. Effects of adjective orientation and gradability on sentence subjectivity. In *In the 18th Conference of the Association for Computational Linguistics*, pages 299–305.

[43] D. Hou and L. Mo. Content categorization of API discussions. In *IEEE International Conference on Software Maintenance*, pages 60–69, 2013.

[44] M. Hu and B. Liu. Mining and summarizing customer reviews. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 168–177, 2004.

[45] IBM. *Alchemy sentiment detection*. http://www.alchemyapi.com/products/alchemylanguage/sentiment-analysis, 2016.

[46] L. Inozemtseva, S. Subramanian, and R. Holmes. Integrating software project resources using source code identifiers. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 400–403, 2014.

[47] M. R. Islam and M. F. Zibran. Leveraging automated sentiment analysis in software engineering. In *Proc. 14th International Conference on Mining Software Repositories*, pages 203–214, 2017.

[48] R. Jongeling, S. Datta, and A. Serebrenik. Choosing your weapons: On sentiment analysis tools for software engineering research. In *Proceedings of the 31st International Confernece on Software Maintenance and Evolution*, 2015.

[49] R. Jongeling, P. Sarkar, S. Datta, and A. Serebrenik. On negative results when using sentiment analysis toolsfor software engineering research. *Journal Empirical Software Engineering*, 22(5):2543–2584, 2017.

[50] D. Kavaler, D. Posnett, C. Gibler, H. Chen, P. Devanbu, and V. Filkov. Using and asking: Apis used in the android market and asked about in stackoverflow. In *In Proceedings of the International Conference on Social Informatics*, pages 405–418, 2013.

[51] O. Kononenko, O. Baysal, and M. W. Godfrey. Code review quality: How developers see it. In *Proc. 38th International Conference on Software Engineering*, pages 1028–1038, 2016.

[52] S. Lal, D. Correa, and A. Sureka. Miqs: Characterization and prediction of migrated questions on stackexchange. In *In Proceedings of the 21st Asia-Pacific Software Engineering Conference*, page 9, 2014.

[53] B. Lin, F. Zampetti, G. Bavota, M. D. Penta, M. Lanza, and R. Oliveto. Sentiment analysis for software engineering: How far can we go? In *40th IEEE/ACM International Conference on Software Engineering*, page 11, 2018.

[54] B. Lin, F. Zampetti, G. Bavota, M. D. Penta, M. Lanza, and R. Oliveto. Sentiment analysis for software engineering: How far can we go? In *Proceedings of the 40th International Conference on Software Engineering*, page 11, 2018.

[55] 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 2014, pages 83–94, New York, NY, USA, 2014. ACM.

[56] B. Liu. *Sentiment Analysis and Opinion Mining*. Morgan & Claypool Publishers, 1st edition, May 2012.

[57] B. Liu. *Sentiment Analysis and Subjectivity*. CRC Press, Taylor and Francis Group, Boca Raton, FL, 2nd edition, 2016.

[58] W. Maalej, Z. Kurtanović, H. Nabil, and C. Stanik. On the automatic classification of app reviews. *Journal of Requirements Engineering*, 21(3), 2016.

[59] W. Maalej and M. P. Robillard. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering*, xx(xx):22, 2013.

[60] M. Makrehchi and M. S. Kamel. Automatic extraction of domain-specific stopwords from labeled documents. In *Proc. European Conference on Information Retrieval*, pages 222–233, 2008.

[61] C. D. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. Cambridge Uni Press, 2009.

[62] M. Måntylå, B. Adams, G. Destefanis, D. Graziotin, and M. Ortu. Mining valence, arousal, and dominance – possibilities for detecting burnout and productivity? In *Proceedings of the 13th Working Conference on Mining Software Repositories*, pages 247–258, 2016.

[63] A. Marcus and J. I. Maletic. Recovering document-to-source-code traceability links using latent semantic indexing. In *Proc. 25th Intl. Conf. on Software Engineering*, pages 125–135, 2003.

[64] G. A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[65] L. Moonen. Generating robust parsers using island grammars. In *Proc. Eighth Working Conference on Reverse Engineering*, pages 13–22, 2001.

[66] A. Murgia, P. Tourani, B. Adams, and M. Ortu. Do developers feel emotions? an exploratory analysis of emotions in software artifacts. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014.

[67] A. Nielsen. A new ANEW: Evaluation of a word list for sentiment analysis in microblogs. In *In the 8th Extended Semantic Web Conference*, pages 93–98.

[68] NLTK. *Sentiment Analysis*. http://www.nltk.org/howto/sentiment.html, 2016.

[69] 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*, pages 33–40, 2015.

[70] N. Novielli, F. Calefato, and F. Lanubile. A gold standard for emotion annotation in stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories (Data Showcase)*, page 4, 2018.

[71] 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*, page 12, 2018.

[72] Ohloh.net. www.ohloh.net, 2013.

[73] Oracle. *The Java Date API*. http://docs.oracle.com/javase/tutorial/datetime/index.html, 2017.

[74] M. Ortu, B. Adams, G. Destefanis, P. Tourani, M. Marchesi, and R. Tonelli. Are bullies more productive? empirical study of affectiveness vs. issue fixing time. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015.

[75] S. Overflow. *JSON and XML comparison*. https://stackoverflow.com/questions/4862310, 2017.

[76] S. Overflow. *Tag Synonyms*. https://stackoverflow.com/tags/synonyms, 2017.

[77] D. Pagano and W. Maalej. User feedback in the appstore: An empirical study. In *Proceedings of the IEEE 21st IEEE International Requirements Engineering Conference*, pages 125–134, 2013.

[78] B. Pang, L. Lee, and S. Vaithyanathan. Thumbs up? sentiment classification using machine learning techniques. In *Conference on Empirical Methods in Natural Language Processing*, pages 79–86, 2002.

[79] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey. Crowd documentation: Exploring the coverage and dynamics of api discussions on stack overflow. Technical report, Technical Report GIT-CS-12-05, Georgia Tech, 2012.

[80] W. G. Parrott. *Emotions in Social Psychology*. Psychology Press, 2001.

[81] 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*, pages 348–351, 2014.

[82] L. Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *In Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111, 2014.

[83] D. M. W. Powers. Applications and explanations of zipf's law. In *Proc. Joint Conferences on New Methods in Language Processing and Computational Natural Language Learning*, pages 151–160, 1998.

[84] R. Řehůřek and P. Sojka. Software framework for topic modelling with large corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, 2010.

[85] P. C. Rigby and M. P. Robillard. Dicovering essential code elements in informal documentation. In *Proc. 35th IEEE/ACM International Conference on Software Engineering*, pages 832–841, 2013.

[86] M. P. Robillard and Y. B. Chhetri. Recommending reference API documentation. *Empirical Software Engineering*, 20(6):1558–1586, 2015.

[87] M. Röder, A. Both, and A. Hinneburg. Exploring the space of topic coherence measures. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, pages 399–408, 2015.

[88] C. Rosen and E. Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, page 33, 2015.

[89] C. R. Rupakheti and D. Hou. Satisfying programmers' information needs in API-based programming. In *Proceedings of the IEEE 19th International Conference on Program Comprehension*, pages 53–62, 2011.

[90] Scikit-learn. *Machine Learning in Python*. http://scikit-learn.org/stable/.

[91] scikit learn. *Machine Learning in Python*. http://scikit-learn.org/stable/index.html#, 2017.

[92] F. Sebastiani. Machine learning in automated text categorization. *Journal of ACM Computing Surveys*, 34(1):1–47, 2002.

[93] N. Silveira, T. Dozat, M.-C. D. Marneffe, S. Bowman, M. Connor, J. Bauer, and C. Manning. A gold standard dependency corpus for english. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation*, pages 2897–2904, 2014.

[94] J. Singer. Using the american psychological association (APA) style guidelines to report experimental results. In *Workshop on Empirical Studies in Software Maintenance*, pages 71–75, 1999.

[95] R. Socher, A. Perelygin, J. Wu, C. Manning, A. Ng, and J. Chuang. Recursive models for semantic compositionality over a sentiment treebank. In *Proc. Conference on Empirical Methods in Natural Language Processing (EMNLP)*, page 12, 2013.

[96] Sonatype. *The Maven Central Repository*. http://central.sonatype.org/, 22 Sep 2014 (last accessed).

[97] A. D. Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, and C. A. Visaggio. What would users change in my app? summarizing app reviews for recommending software changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 499–510, 2016.

[98] Sourceforge.net. http://sourceforge.net/, 2013.

[99] C. Stanley and M. D. Byrne. Predicting tags for stackoverflow posts. In *In Proceedings of the 12th International Conference on Cognitive Modelling*, pages 414–419, 2013.

[100] J. Stylos. *Making APIs More Usable with Improved API Designs, Documentations and Tools*. PhD in Computer Sscience, Carnegie Melon University, 2009.

[101] S. Subramanian, L. Inozemtseva, and R. Holmes. Live api documentation. In *Proc. 36th International Conference on Software Engineering*, page 10, 2014.

[102] Y. Tausczik, A. Kittur, and R. Kraut. Collaborative problem solving: A study of math overflow. In *In Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work and Social Computing*, pages 355–367, 2014.

[103] M. Thelwall, K. Buckley, G. Paltoglou, D. Cai, and A. Kappas. Sentiment in short strength detection informal text. *Journal of the American Society for Information Science and Technology*, 61(12):2544–2558, 2010.

[104] 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 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, pages 173–180, 2013.

[105] G. Uddin. *Pilot Survey - Needs for API reviews to support development tasks*. https://goo.gl/forms/88dfE15S5tPCtOQi2, 2016. Last accessed on 30 November 2017.

[106] G. Uddin. *Primary Survey - Needs for API reviews to support development tasks*. https://goo.gl/forms/0no2gVYuJJgxisDk1, 2017. Last accessed on 30 November 2017.

[107] G. Uddin, O. Baysal, L. Guerrouj, and F. Khomh. Understanding how and why developers seek and analyze API-related opinions. *IEEE Transactions on Software Engineering*, page 27, 2017. Under review.

[108] G. Uddin and F. Khomh. Automatic summarization of API reviews. In *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering*, page 12, 2017.

[109] G. Uddin and F. Khomh. Automatic summarization of api reviews. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 159–170, 2017.

[110] G. Uddin and F. Khomh. Opiner: A search and summarization engine for API reviews. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 978–983, 2017.

[111] G. Uddin and F. Khomh. *Opinion value analysis in API reviews*. https://github.com/giasuddin/OpinionValueTSE, 24 Nov 2018.

[112] G. Uddin and M. P. Robillard. Resolving API mentions in informal documents. Technical report, Sept 2017.

[113] B. Vasilescu, A. Serebrenik, P. Devanbu, and V. Filkov. How social q&a sites are changing knowledge sharing in open source software communities. In *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*, pages 342–354, 2014.

[114] A. J. Viera and J. M. Garrett. Understanding interobserver agreement: The kappa statistic. *Family medicine*, 37(4):360–363, 2005.

[115] S. Wang and C. D. Manning. Baselines and bigrams: simple, good sentiment and topic classification. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*, pages 90–94, 2012.

[116] A. B. Warriner, V. Kuperman, and M. Brysbaert. Norms of valence, arousal, and dominance for 13,915 english lemmas. *Behavior Research Methods*, 45(4):1191–1207, 2013.

[117] C. wei Hsu, C. chung Chang, and C. jen Lin. A practical guide to support vector classification.

[118] Wikipedia. *Application programming interface*. http://en.wikipedia.org/wiki/Application_programming_interface, 2014.

[119] T. Wilson, J. Wiebe, and P. Hoffmann. Recognizing contextual polarity in phrase-level sentiment analysis. In *In Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 347–354, 2005.

[120] E. Wong, J. Yang, and L. Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 562–567, 2013.

[121] R. K. Yin. *Case study Research: Design and Methods*. Sage, 4th edition, 2009.

[122] Y. Zhang and D. Hou. Extracting problematic API features from forum discussions. In *21st International Conference on Program Comprehension*, pages 142–151, 2013.

[123] B. Zhou, X. Xia, D. Lo, C. Tian, and X. Wang. Towards more accurate content categorization of API discussions. In *22nd International Conference on Program Comprehension*, pages 95–105, 2014.

[124] C. Zou and D. Hou. LDAAnalyzer: A tool for exploring topic models. In *International Conference on Software Maintenance and Evolution*, pages 593–596, 2014.