Automatic Summarization of API Artifacts from Informal Documentation

by
Gias Uddin

School of Computer Science
McGill University, Montréal, Quebec

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
DOCTOR OF PHILOSOPHY

Abstract

APIs (Application Programming Interfaces) offer interfaces to reusable software components. With the advent and proliferation of online developer forums as informal documentation, developers often share their opinions about the APIs they use. Thus, opinions of others can shape developer perception and decisions related to software development. For example, the choice of an API or how to reuse the functionality the API offers are, to a considerable degree, conditioned upon what other developers think about the API. While many developers refer to and rely on such opinion-rich information about APIs, we found little research that investigates the use and benefits of public opinions about APIs. To understand how and why developers share opinions about APIs, we conducted two surveys with a total of 178 software developers. Through an exploratory analysis of the survey responses, we found that developers seek for opinions about APIs to support diverse development needs, such as, to select an API amidst multiple choices, to fix a bug, to implement or enhance a feature, and so on. The developers wish for tool support to analyze the huge volume of such opinions and usage scenarios scattered across many forum posts.

The developers consider opinions about APIs as a form of API documentation, and analyze both the opinions and the usage scenarios posted about an API during the adoption of the API in their development tasks. They reported that they rely on opinions posted in forum posts to validate a usage examples in API formal documentation, because the official documentation is often incomplete and ambiguous. To understand the common problems that developers face while using API formal documentation, we conducted two surveys with a total of 323 software developers at IBM. We asked the developers to give us examples of three problems with API official documentation they recently encountered. Each example contained the description of a task and a link to the corresponding API documentation unit that was considered as problematic during the completion of the task. We observed 10 common documentation problems that manifest in the development practices of software developers. To compensate for the problems in the API formal documentation, the API reviews and usage scenarios posted in the forum posts can be leveraged.

To assist the developers in their exploration of opinions about APIs from developers, we developed a suite of techniques to mine and summarize opinions about APIs from the forum posts. We implemented the techniques in our tool, named Opiner. Opiner is an online search and summarization engine for API artifacts. In Opiner, developers can search for an API by its name and see the mined opinions about the API in our two proposed summarization algorithms (Statistical and Aspect-based) and six other summarization algorithms (e.g., contrastive viewpoint summarizer). We evaluated the API review summaries in Opiner by conducting two user studies, both involving professional and Industrial software developers. We observed promising results of leveraging the Opiner API review summaries to support diverse development needs, such as, API selection, documentation, being staying aware, and so on. In two API selection tasks involving four open source APIs, we found that the Industrial developers made more correct API choices while using Opiner and Stack Overflow together than while using only Stack Overflow.

To assist the developers in their usage of APIs, we further developed a framework to automatically mine and summarize usage scenarios about APIs from developer forums. In Opiner, we developed four different summarization algorithms for API usage scenarios mined from developer forums. In three user studies involving both professional software developers and students, we found that the participants were able to complete their coding tasks with more accuracy, and in less time and effort while using Opiner usage summaries compared to when they used Stack Oveflow only or

API official documentation only. The developers mentioned that the usage summaries in Opiner can offer improvements to both API formal and informal documents. More than 80% developers wished for Opiner usage summaries to be integrated into the API formal documentation. The Opiner online search and summarization engine website is hosted at: `http://opiner.polymtl.ca`

Résumé

Les API abstraites (interfaces de programmation d'applications) offrent des interfaces aux composants logiciels réutilisables. Avec l'avènement et la prolifération des forums de développeur en ligne, les développeurs partagent souvent leurs opinions sur les APIs qu'ils utilisent. Ainsi, les opinions des autres peuvent façonner la perception des développeurs et influencer leurs décisions relatives aux APIs . Par exemple, le choix d'une API ou la façon de réutiliser les fonctionnalités que l'API offre sont, dans une large mesure, conditionnées sur ce que les autres développeurs pensent de l'API. Bien que de nombreux développeurs se réfèrent et se fient aux opinions des pairs au sujet des APIs, nous avons trouvé peu de recherches qui étudient l'utilisation et les avantages des opinions publiques sur les APIs. Pour comprendre comment et pourquoi les développeurs partagent des opinions sur les APIs, nous avons mené deux sondages avec un total de 178 développeurs de logiciels. Grâce à une analyse exploratoire des réponses de l'enquête, nous avons constaté que les développeurs recherchent des avis sur les APIs pour soutenir divers besoins de développement, comme, pour sélectionner une API au milieu de plusieurs choix, corriger un bogue, implémenter ou améliorer une fonctionnalité, et ainsi de suite. Les développeurs souhaitent un support d'outil pour analyser le volume énorme de ces avis, ainsi que les multiples scénarios d'utilisation d'API dispersés dans de nombreux messages postés sur les forums.

Les développeurs considèrent les avis sur les APIs comme une forme de documentation de ces API, et analysent à la fois les opinions et les scénarios d'utilisation affichés sur une API au cours de l'adoption de l'API pour leurs tâches de développement. Ils ont rapporté qu'ils comptent sur les opinions affichées sur les forums pour valider des exemples d'utilisation dans la documentation officielle de l'API, car la documentation est souvent incomplète et ambiguë. Pour comprendre les problèmes courants rencontrés par les développeurs lors de l'utilisation de la documentation formelle des APIs, nous avons réalisé deux sondages avec un total de 323 développeurs de logiciels chez IBM. Nous avons demandé aux développeurs de nous donner trois exemples de problèmes qu'ils ont récemment rencontrés en travaillant avec la documentation officielle d'un API. Chaque exemple contenait la description d'une tâche et un lien vers l'unité de documentation correspondant à l'API qui a était considérée comme problématique durant l'exécution de la tâche. Nous avons observé 10 problèmes de documentation communs qui se sont manifestés dans les pratiques de développement de ces développeurs. Pour compenser les problèmes dans la documentation formelle des APIs, les opinions émises sur les APIs ainsi que les scénarios d'utilisation partagés sur les forums peuvent être exploités.

Pour aider les développeurs dans leur exploration des opinions partagées sur les forums, nous avons développé et implémenté dans notre outils Opiner, une suite de techniques permettant de résumer les avis émis sur les APIs dans les messages partagés sur un forum. Opiner est un moteur de recherche et de synthèse en ligne d'information sur des APIs. Dans opiner, les développeurs peuvent rechercher une API par son nom et voir les avis partagés sur l'API grâce à nos deux algorithmes de synthèse (statistiques et à base d'aspect) et six autres algorithmes de synthèse (par exemple, une synthèse de point de vue contrastés sur un API). Nous avons évalué les résumés de Opiner en effectuant deux études d'utilisateurs, impliquant des professionnels et les développeurs de logiciels industriels. Nous avons observé que ces résumés ont le potentiel d'aider les développeurs dans diverses tâches de développement, tels que, la sélection des APIs, l'élaboration de la documentation, et ainsi de suite. Lors de deux tâches de sélection d'APIs impliquant quatre APIs à code source ouvert, nous avons constaté que les développeurs industriels ont fait de meilleurs choix d'API lorsqu'ils utilisaient Opiner et Stack Overflow ensemble, que lorsqu'ils n'utilisaient que Stack Overflow. Dans

Opiner, nous avons aussi développé quatre algorithmes de synthèse de scénarios d'utilisation des API extraits des forums de développeurs. Dans trois études d'utilisateurs impliquant à la fois des développeurs de logiciels professionnels et des étudiants, nous avons constaté que les participants étaient en mesure de compléter leurs tâches de codage avec plus de précision, et en moins de temps et d'effort lorsqu'ils utilisaient Opiner, que lorsqu'ils utilisaient seulement Stack Overflow ou la documentation officielle de l'API. Les développeurs ont mentionnés que les résumés d'utilisation fournies par Opiner permettent de compenser les manques de la documentation officielle des API. Plus de 80% de développeurs souhaitaient que les résumés d'Opiner soient intégrés dans la documentation formelle des APIs. Le site Web du moteur de recherche et de synthèse Opiner est hébergé à: http://opiner.polymtl.ca

Acknowledgments

This thesis would not have been possible without the support and encouragement of my amazing supervisor, Foutse Khomh. Together, we did research of which we can be proud of. We discussed passionately to find the best possible way to evaluate a research project, and we delivered with great care and extreme hard work. I will remember the nights we were awake to meet the paper submission deadlines, and the numerous edits Foutse made tirelessly and with a smile in this face – even at 5AM. I will never forget the night of our submissions to the ASE 2017 – when Foutse was awake with me the whole night with a flu, but he told me about that after we submitted the paper at 7AM. Foutse, I am fortunate to have you as my supervisor. I hope you will always be my mentor.

I am grateful to Prof Jérôme Waldispühl to become my supervisor at McGill, to support me when I needed, and to respond to my questions whenever I asked him. My sincere thanks to Prof Prakash Panagaden for the support during the last years of my PhD. My sincere thanks to Prof Robin Beech. Prof Beech has been an amazing person, a great mentor, and a warm supporter for me since the first day I met him. My sincere thanks and gratitude to my thesis examiners, Prof David Lo, Prof Benjamin Fung, Prof Yann-Gaël Guéhéneuc, and Prof Claude Crépeau.

I am indebted to Prof Chanchal K Roy of University of Saskatchewan. Among many things we did together we also wrote papers, one of which is a crucial part of this dissertation. Prof Roy is an amazing researcher and a person to talk to and to get advice from. I am grateful to Prof Roy for helping me when I needed and for giving me advice whenever I asked for. I am thankful to Prof Ahmed E Hassan of Queen's University for taking the time to talk to me whenever I asked for his advice and suggestion.

It takes a village to complete a PhD. I am grateful to all my friends in the Academia and in the Industry who helped me during my PhD. I thank all the members of the SWAT lab and from Prof Roy's lab, for helping me with the data collection in a short time. I thank all the human coders who participated during the data analysis and the creation of the different benchmarks of this thesis. I thank all the numerous respondents to my surveys that I conducted as part of this thesis. I thank the open-source software developers for access to their repositories and data. Over the last two years of my stay as a Data Scientist at Apption Software, my colleagues have been exceptional. I have learned a lot from all of the co-authors of my papers (in no particular order): Prof Martin P. Robillard, Barthélémy Dagenais, Prof Olga Baysal, and Prof Latifa Guerrouj.

My family suffered with me during the darkest hours of my PhD, and celebrated with me during the bright moments. Without the support and encouragement of Dr. Kalam Shahed (my father-in-law) and Nasima Shahed (my mother-in-law), it would have been impossible to complete this thesis. I am thankful to my mother, brothers and sisters for believing in me, for giving me the courage to pursue a PhD, and for praying for me when I needed the most.

Finally, my hearty gratitude and love towards my amazing and beautiful wife, Anonna. Anonna accepted me when I had nothing, remained with me during the darkest hours of my PhD, stood beside me when I had no one else to talk to, took care of our little one when I was hopelessly busy with my full-time job at day time and my PhD research at night time. It took exceptional patience, courage, and love from Anonna to accompany me in this journey.

Dedication

To the two most beautiful ladies in my life: 1) My wife, Anonna Shahed, for becoming the love of my life, for accepting me when I had nothing to offer you, and for pushing me all the way during the darkest periods of my PhD. 2) My daughter, Nandini, for becoming my sunshine and my motivation. My baby, I hope you will never give up in the faces of adversity and you will always win with a smile in your face. I am thankful to the Almighty for giving the two of you in my life.

Related Publications & Submissions

The following is a list of published papers that are on topic of this thesis:

1) Gias Uddin and Foutse Khomh, Automatic Summarizing API Reviews, in Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp 159–170, 2017 (in Chapter 5).

2) Gias Uddin and Foutse Khomh, Opiner: A search and summarization engine for API Reviews, in Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp 978-983, 2017 (in Chapters 4, 5).

3) Gias Uddin and Martin P. Robillard, How API Documentation Fails, in IEEE Software, Vol 32, Issue 4, pp 68–75, 2015 (in Chapter 3)

The following is a list of papers that are currently under review and are on topic of this thesis:

1) Gias Uddin and Foutse Khomh, Automatic Opinion Mining from API Reviews, in IEEE Transactions of Software Engineering (TSE), pp 34 (in Chapter 4).

2) Gias Uddin, Foutse Khomh and Chanchal K Roy, Automatic Summarization of Crowd Sourced API Usage Scenarios, in IEEE Transactions of Software Engineering (TSE), pp 35 (in Chapter 6)

3) Gias Uddin, Olga Baysal, Latifa Guerrouj and Foutse Khomh, Understanding How and Why Developers Seek and Analyze API Reviews, in IEEE Transactions of Software Engineering (TSE), pp 27 (in Chapter 2).

The following is a list of papers that were published in parallel to the duration and topic of the PhD studies, but are not included in this PhD Thesis:

1) Gias Uddin, Barthélémy Dagenais, and Martin P. Robillard, Temporal Analysis of API Usage Concepts, In Proceedings of the 34th International Conference on Software Engineering (ICSE), pp 804–814, 2012.

2) Gias Uddin, Barthélémy Dagenais, and Martin P. Robillard, Analyzing Temporal API Usage Patterns, In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 456–459, 2011.

Attribution

The research work described in this thesis was conducted by me as the primary author and primary investigator. For the research work described in Chapter 3, Prof Robillard was the co-author of the corresponding paper that was published in IEEE Software [1]. Prof Robillard (1) advised during the design and the data analysis of the study described in the paper, and (2) provided useful and valuable comments and suggestions on the write-ups of the paper. I (1) initiated the idea of the surveys, designed and conducted the study, (2) wrote the initial and final drafts of the paper, (3) communicated with Prof Robillard on the investigation and the resulting paper. For the other research work described in Chapters 2, 4 – 6, I followed similar approaches with the co-authors of the corresponding papers/manuscripts.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

APIs (Application Programming Interfaces) offer interfaces to reusable software components. By leveraging APIs, developers can benefit from the *reusing* of already implemented API features over the *re-invention* or *re-implementation* of those features. Such benefits can thus increase *productivity* and improve the *quality* of the software [6, 7].

Modern day rapid software development is often facilitated by the plethora of open-source APIs that can be available for any given development task. The online development portal GitHub [8] 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 [9], Sourceforge [10], Freecode [11], Maven Central [12], and so on. For example, Ohloh currently indexes more than 650,000 projects, demonstrating an increase of 100,000 projects from 2015.[1]

There can be two major challenges associated with the abundance of such APIs available towards supporting the completion of a given task:

Selection. Identifying an API amidst choices to support the development task in hand. Such a development task can be diverse, such as, bug fixing, prototype development, business analysis, new feature development or enhancement, and so on.

Usage. If the development task is indeed about the reusing of the API features to complete a coding task, then a developer would also need to properly learn and use the API.

The advent and proliferation of online developer forums to communicate ideas and usage of APIs has opened up an interesting avenue for developers to look for solutions of their development tasks in the forum posts. While developer forums serve as communication channels for discussing the implementation of the API features, they also enable the exchange of opinions or sentiments expressed on numerous APIs, their features and aspects.

As an example, in Figure 1.1, we present the screenshot of seven Stack Overflow posts. In Figure 1.1, the four red-colored circles are answers ①, ③-⑤ and three green-colored circles are comments ②, ⑥, ⑦. The oldest post (at the top) is dated November 06, 2009 and the most recent one (at the bottom) is from February 10, 2016. These posts express developers' opinions about two Java APIs (Jackson [13] and Gson [14]) offering JSON parsing features for Java. None of the posts contain any code snippets. The first answer ① representing a positive opinion about the Gson API motivates the developer 'binaryrespawn' to use it ②. In the next answer ③, the user 'StaxMan' compares Gson with Jackson, favouring Jackson for offering better support. Based on this feedback, 'mickthomson' ④ decides to use Jackson instead of Gson. Three out of the four answers ③–⑤ imply a positive sentiment towards Jackson but a negative one about Gson. Later, the developer 'Daniel Winterstein' develops a new version of Gson by fixing the reported issues in ⑤ and shares his API ⑦ in Github. The APIs and the problems discussed by the developers are all prevalent, until now, even though the first post ① was last answered in November 2009. This is

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

Figure 1.1: The discussions with positive and negative reviews about two APIs in Stack Overflow. The red circled posts are answers. The green circled posts are comments.

shown in first and the last posts. While the first post ① was answered in November 2009, it was last edited in April 2017. The last post ⑦ was actually posted on February 10, 2017. This example illustrates how developers share their experiences and insights, as well as how they influence and are influenced by the opinions. A developer looking for only code examples for Gson would have missed the important insights about the API's limitations, which may have affected her development activities. Thus, opinions in the developer forums can drive developers' decision making during the usage and selection of APIs.

To understand how and why developers seek and analyze opinions about APIs in developer forums, we conducted two surveys with a total of 178 professional software developers (see Chapter 2). The developers reported that they seek for opinions about APIs in forum posts to support diverse development needs, such as, API selection, documentation, learning how to use an API, and so on. Developers also reported that they rely on opinions of other developers to determine the

Table 1.1: Number of threads in Stack Overflow for the three languages between 2014 and 2017

	Javascript		Java		C#	
	Total	**Increase**	**Total**	**Increase**	**Total**	**Increase**
January 2014	523,889	–	554,917	–	577,100	–
November 2017	1,511,268	188.5%	1,341,292	141.7%	1,156,775	100.4%

quality of a provided code example (e.g., if it is well-received) in the forum posts, and to decide about the *trustworthiness* and *validity* of a given API usage scenario in the official documentation of the API. Due to the plethora and diversity of opinions and usage scenarios about APIs in the forum posts, the developers asked for the support of automated tools to properly analyze API reviews and usage in the developer forums.

Stack Overflow as a developer forum has experienced rapid growth in the number of user participation. In Table 1.1, we show the total number of threads in Stack Overflow for the top three programming languages (JavaScript, Java and C#) between January 2014 and November 23, 2017.[2] The top three languages are taken from the Stack Overflow tags in November 2017 [16]. For each language, there is at least 100% increase in the number of threads within less than four years. For JavaScript, it was a little over half a million in 2014 and it became more than 1.5 millions in 2017 (188.5%↑). For Java, the increase was 141.7% (from half a million to 1.3 million). Each of the threads can have more than one post. A post can be a question/answer/comment. For example, for Java, the number of total posts under the threads tagged as 'Java' was 4.2 millions in 2014 and more than 9.7 millions in 2017 (129.7%↑).

Each post can have more than one opinion about diverse APIs. For example, the full post circled as ⑤ in Figure 1.1 has a total of seven sentences, each with positive and negative opinions. Besides two APIs (GSON and Jackson), the post also contained opinions about another API, XStream. Indeed, the 22.7 thousands posts tagged as 'Java+JSON' in Stack Overflow in 2014, contained more than 87 thousands sentences, with more than 1.08 millions words and participation from 7.5 thousands users. Out of the 87 thousands sentences in those 22.7 thousand posts, more than 15 thousand sentences are opinionated (positive or negative).[3] Out of those 22.7 thousand posts, more than five thousand posts contained at least one code example (see Table 1.2). [4]

These statistics are just from the Stackoverflow.com site. There is a total of 171 sites under the Stack Exchange network. Many of which are similar to the Stack Overflow site, with developers sharing opinions and usage about diverse APIs. For example, 'Arduino' is one of those other sites, where developers discuss their experience on the development of open source hardware and software based on the Arduino microcontrollers. [5] Besides, Stack exchange sites, there are many other developer forums online, where developers can share opinions and usage about APIs (e.g.,

[2]We used the Stack Overflow data dump of January 2014 to collect the usage statistics of 2014. We used the Stack Exchange online data explorer [15] to collect the usage statistics on November 2017.

[3]We identified the opinionated sentences by using our sentiment detection technique for API reviews. The technique is described in Chapter 4.

[4]A code snippet is detected if there were more than one code line inside the `<code>` tag in Stack Overflow or when there was at least one assignment operator (if just one line). We used the Stack Overflow data dump of January 2014.

[5]Arduino: `https://arduino.stackexchange.com/`

Table 1.2: Sentiment coverage in Stack Overflow posts tagged as "Java" and "Json"

Posts		Sentences			Words	Users
Total	**With Code**	**Total**	**Positive**	**Negative**	**Total**	**Distinct**
22,733	5,918	87,033	10,055	5,572	1.08M	7.5K

Gmane [17], reddit [18], and so on).

The developers in our survey consider opinions about APIs as a form of API documentation, and analyze both the opinions and the usage scenarios posted about an API during the adoption of the API in their development tasks. They reported that they rely on opinions posted in forum posts to validate usage examples from API formal documentation, because the official documentation can be incomplete or ambiguous [19]. To understand the common problems that developers face while using API formal documentation, we conducted two surveys with a total of 323 software developers at IBM. We asked the developers to give us examples of three problems with API official documentation they recently encountered. Each example contained the description of a task and a link to the corresponding API documentation unit that was considered as problematic during the completion of the task. We observed 10 common documentation problems that manifest in the development practices of software developers. To compensate for the problems in the API formal documentation, the API reviews and usage scenarios posted in the forum posts can be leveraged.

However, given the multitudes of forum posts where APIs can be discussed, it is challenging for a developer to analyze reviews and usage scenarios posted about different competing APIs for a given development task, let alone leveraging those to get a quick but informed decision on whether and how an API can be used. Indeed, when we asked the developers in our survey on how they navigate and search for opinions about APIs in developer forums, most of their answers were about the scanning over the first few search results from Google or Stack Overflow. One developer mentioned he *"usually uses google and select the top hits from relevant web sites (such as stackoverflow) "*. Another developer mentioned to have leveraged both search and sentiment analysis: *"Honestly, to find drawbacks, I search for strong negative phrases such as "X sucks" for API X. "*. Clearly, such an approach to decide on whether and how APIs can be used is *sub-optimal*. For example, in Figure 1.1, a developer only finding the post ① would have selected the Google GSON API. However, such a decision could have been different if the developer had the knowledge of the other posts (e.g, ④-⑥).

In this dissertation, we aim to assist developers to get quick and informed insights about APIs from developer forums. We formally define our problem statement as follows.

Problem – API Review and Usage Analysis from Developer Forums

The plethora and diversity of opinions and usage scenarios discussed about APIs in the online developer forums can pose challenges to the developers to gain quick and informed insights about APIs within the contexts of their development tasks.

Given the plethora of reviews and usage scenarios available about an API in developer forums, it can be extremely beneficial to produce an *informative* but *concise* summary of the reviews and

usage scenarios about an API. A concise but complete summary can be useful to get quick but useful insights about an API. Indeed, the developers in our survey envisioned diverse potential summarization approaches to help them to gather useful insights about APIs. They mentioned that in the absence of any automated summarization technique to help them, they leverage different features in forum posts to get summarized viewpoints of the APIs, e.g., skimming through high ranked posts, using tags to find similar APIs, etc. The summarized insights about API usages can facilitate the design and development of a more complete and better API documentation [20].

The thesis of this dissertation is that by aggregating and summarizing reviews and usage scenarios about APIs from informal documents, we can help developers to get quick but informed insights about APIs. We formally present our thesis statement as follows.

Thesis – Summarization of API Reviews and Usage Scenarios from Developer Forums

The aggregation and summarization of reviews and usage scenarios about APIs from online developer forums can support efficient API usage and learning, by offering quick but informed insights, thereby facilitating diverse development needs and improving API documentation.

1.1 Background & Definitions

We discuss the key concepts upon which this dissertation is founded.

1.1.1 Opinion

An opinion, as defined by Bing Liu [21] 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.

The aspects of an entity about which opinions are provided can be of two types [21]:

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

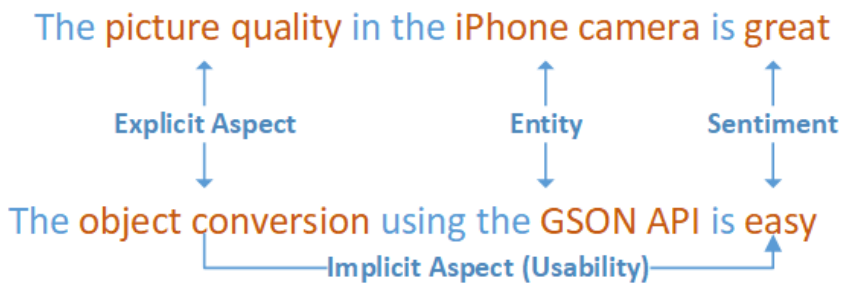

Figure 1.2: The presence of aspect and sentiment in reviews about entities (Camera and API)

in Figure 1.2. 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.

• **Sentiments vs Emotions in Opinion Analysis**. Sentiment detection focuses on the detection of subjectivity in a given input (e.g, a sentence). A subjectivity can be of three types: (1) Positive, (2) Negative, and (3) Neutral. 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 [22]: (1) Joy, (2) Love, (3) Surprise, (4) Anger, (5) Sadness, and (6) Fear. This dissertation 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 [21, 23]. The analysis of emotions in API reviews is our immediate future work.

1.1.2 Opinion Summarization Dimensions

Our research on API reviews takes cues from similar research in the other domains. Automated sentiment analysis and opinion mining about entities (e.g., cars, camera products, hotels, restaurants) have been a challenging but practical research area due to their benefits for consumers (e.g., to guide them in choosing a hotel or selecting a camera product).

In Figure 1.3, we show the screenshots of three separate initiatives on the automatic collection and aggregation of reviews about camera products. The first two tools are developed in the Academia and the the third tool was developed as part of Microsoft Bing Product Search. The first two circles ① and ②, show two preliminary outlines of such a tool presented by Liu et al [2]. The third circle ③ shows a similar tool in the Microsoft Bing Product Search. In all of the three tools, positive and negative opinions about a camera product are collected and aggregated under a number of aspects (e.g., picture quality, battery life, etc.). When no predefined aspect is found, the opinions are categorized under a generic aspect 'GENERAL' (aspect is named as 'feature' in ②). The opinions about the camera can be collected from diverse sources, e.g., online product reviews, sites selling the product, etc. For example, Google collect reviews about hotels and restaurants through a separate gadget in its online search engine.

Figure 1.3: Screenshots of opinion summarization engine for camera reviews. The aspects (e.g., picture quality) are used to present summarized viewpoints about the camera. The circle 3 shows a similar interface in the now defunct Bing product search. The screenshots are taken from [2].

• **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 1.1, there are three posts (one answer and two comments). Each of the posts can be considered as a separate document. In this dissertation, we analyze the reviews about APIs at the sentence level. For example, in Figure 1.1, the first post (①) contains six sentences. The three posts in Figure 1.1 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 1.1 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 1.1 are about two different API aspects (usability and performance). The analysis of sentiments at the sentence is applied to develop many research and Industrial tools and techniques in other domains [21].

• **Summarization Algorithms.** The opinions about an entity can be summarized using five different techniques [2, 24]:

1) **Statistical:** An approach to statistical summaries is the *"basic sentiment summarization*, by *simply counting and reporting the number of positive and negative opinions"* [2]. The statistical summaries of opinions are normally visualized, for example, by using star ratings in Amazon or eBay product search.

2) **Aspect-based:** Aspect-based summarization [25, 26] involves the generation of summaries of opinions around a set of aspects, each aspect corresponding to specific attributes/features of an

entity about which the opinion was provided. For example, for a camera, picture quality can be an aspect. Aspects can be different depending on the domains. Thus the detection of domain specific aspects is the first step towards aspect-based opinion summarization [21].

3) **Contrastive:** Given as input all the positive and negative opinions about an entity, a Contrastive Opinion Summarizer (COS) *"generates contrastive paired sentences"* [27]. Such contrastive pairs thus show how for a given aspect/feature, contrastive opinions are provided. For example, "The object conversion in Google GSON is easy to use" vs "The object conversion in Google GSON is buggy and slow".

4) **Extractive:** In an extractive summary of opinions about an entity, selective *representative* text segments are extractive as a summary from the input document of all opinions. Such representative text segments are usually individual positive or negative opinionated sentences, which are considered as *significant* to be *representative* of the entire document. Both supervised and unsupervised algorithms have been developed to produce extractive summaries [28–31].

5) **Abstractive:** Unlike the extractive summaries, abstractive summaries do not use the existing sentences from the input data. Instead, an algorithm to produce abstractive summaries generates sentences by analyzing the input data [32].

1.2 Detection of Aspects in API Reviews

Among the five different types of summaries that can be applied to opinions about an entity, aspect-based summaries are considered as the most domain-centric [21]. For example, for a restaurant, 'food' can be considered as an aspect. For a camera, 'food' cannot be considered as such, but 'picture quality' can be considered as an aspect. The summaries of the opinions of an entity by aspects, can offer added benefits of comparing similar entities by the aspects. For example, we can compare restaurants in a local area by the 'quality of food' they serve or the 'quality of their service', irrespective of the type and ethnicity of the restaurants [26].

For the domain of API reviews, we are aware of no aspects that can be leveraged to produce aspect-based summaries of the reviews. As a first step towards facilitating aspect-based summaries of API reviews, in our survey of 178 software developers, we asked them about their preference of API aspects in the opinions about APIs in a series of open and closed-ended questions. We analyzed the responses to the open-ended questions using open coding. Three coders participated in the open coding. 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 that emerged in the open codes were scalability, faster or slower processing, and so on.

Usability: The opinion is about how usable an API to support a development need (e.g., easy to use vs complicated designs, and so on). Related themes that emerged in the responses were the design principles of an API and their impacts on the usage of the API. As Stylos et al. [33] 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 that emerged from the open codes were authentication, authorization, encryption, and so on.

Documentation: The opinion is about how the API is documented in different 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 used in the different Linux distributions?". The themes that 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?" The 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?". The 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 in 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."

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 generalized 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). As such, 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). In this dissertation, we focus on the analysis and detection of the above aspects.

1.3 Combining API Code Examples with Reviews

While the API review summaries can be useful for the developers in their selection of an API, the developers would need to learn the API to actually use it in their development tasks. Ponzanelli et al. [34] observed that 85% of the participants of their survey usually used official API documentation in their development tasks. More than 90% of the participants also used informal API documents (e.g., Stack Overflow, forums, mailing lists) to learn and use an API. The developers in our survey considered the reviews and usage scenarios posted about APIs in the forum posts as a form of API documentation. They leverage both the code examples and the associated opinions to learn about the usage of an API. They also leverage the reviews to validate that the usage examples in the official documentation work as expected. One reason behind such exploration is that API formal documentation can be insufficient or not up to date to the current version of the API [19]. For example, in our surveys with 323 developers at IBM, we found that the *incompleteness* of the contents of the formal documentation is the most severe problem the developers faced while using APIs. One possible solution to address the incompleteness of API formal documentation is to leverage the the API usage examples shared and discussed in the developer forums [35].

However, as we discussed before, the plethora of usage discussions available for an API in the forum posts can pose challenges to the developers. Challenges in such exploration include, but not limited to, finding the actual solution amidst thousands of already available solutions that are scattered across many different forum posts, finding multiple relevant usage scenarios from different forum posts that can be used together to complete a complex development task, and so on.

As an example to further demonstrate the needs to summarize API usage scenarios, we show three pairs of question and answer in Figure 1.4. We show the questions chronologically, with the first question ① being the oldest (September 2013) and the last question being the most recent (January 2014) ⑤. All of the questions are related to the API Google GSON ①, ③, and ⑤. Each of the code snippets ②, ④, ⑥ are taken from the accepted/top-voted answers for the corresponding questions. The problems discussed in three questions are almost similar: Given as input a JSON string of nested key value pairs, how can the JSON be parsed into a list of Java dictionaries (e.g., HashMap) using the GSON API? As such, the code snippets also provide similar solutions using the same GSON types (`TypeToken`, `Gson`) and methods (`getType()`, `fromJson()`). The questions share the same tags (java, json, gson). However, none of the questions are tagged as duplicate or recommended as similar by the Stack Overflow community as of now. For each question, Stack Overflow also shows a list of related questions. None of the questions are suggested as similar to each other by Stack Overflow. Therefore, a developer while searching for the above problem may only find one of the questions. The developer looking for a solution in January 2014 (⑤ in Figure 1.4), could have easily used the solution posted in the previous months (check the ② and ④ in Figure 1.4) and thus would not have needed to post his question ⑤ at all. A search in Stack Overflow showed that many questions with similar topics (parse JSON with GSON) using the same GSON API were asked as recently as in November 2017 (e.g., see [36]). Indeed, the Stack Overflow online guidelines [37] encourage a user to only post a question when a solution is not found in the forum posts.

Indeed, the API usage examples available in the forum posts are beneficial for the developers to learn and use an API, as well as, to compensate for the absence of such information in the API official documentation [35]. However, the plethora of such information can also lead to developers' frustration to find the right information. Clearly, there is a need to summarize the usage scenarios

Figure 1.4: Similar usage scenarios in Stack Overflow

about API from developer forums, so that the right information can be found quickly and easily.

Recent efforts in code fragment summarization by Ying and Robillard [38] advocates the needs to only show *interesting* parts of a code example in the summary. The focus of such summaries is to provide an easy to understand representation of a code example in a search query result. According to developers in our surveys, the API usage scenarios and reviews in the forum posts are a form of API documentation, e.g., *"Because I want to know of real use, rather than just documentation . . . It's especially useful to learn about the problems others faced that might only be evident after considerable time is spent experimenting with or using the API in production"*. Therefore, their expectation from the API usage summaries from developer forums could be more about getting the *right and proper* contents to assist in their coding tasks. One developer in our surveys at IBM summarized his needs from API official documentation as *"I like to have a brief overview but lots of examples – ideally a code snippet using the API that I can cut and paste"*.

With a view to produce API usage summaries as a form of API documentation, we made the following design decisions while presenting a code example in the summary:

Length. We show the code example as it is shared in the forum post, instead of only showing its key elements (as it was done by Ying and Robillard [38]). This is to ensure that a developer can easily reuse the code examples in a summary.

Review. We detect the positive and negative opinions in the replies to a forum post where the code example was found, and associate those to the code example. This is to ensure that a developer can learn about the *quality* of the code example from the reactions of the other developers.

1.4 API Usage Scenario Summarization Dimensions

Our focus to generate API usage summaries from developer forums is to assist developers in their coding tasks. This is similar to how developers would like to use the API formal documentation to learn and use an API. Therefore, to understand how usage summaries about APIs can be produced, we can leverage our knowledge of API documentation, how it is structured, how contents are organized, and more importantly what problems developers normally face in API formal documentation [19]. We can produce API usage summaries by following the examples of good API documentation. In addition, by understanding the problems of API formal documentation, we can design API usage summarization algorithms that can be useful to address those problems.

We discussed previously about our surveys of 323 software developers at IBM. The developers shared their experience with regards to the quality of API formal documentation. Out of the 10 common documentation problems as reported by the developers, Six of the problems were related to the presence of the content, and four were related to the presentation of the contents. The insights about the documentation problems helped us in our design of the algorithms for API usage summaries. For example, the most severe content-related problem was the incompleteness in the documentation. One reason for this shortcoming in the API formal documentation is that API formal documentation lacks task-based documentation, such as, how a development task can be solved using one or a set of API elements together [20, 39]. As Carroll et al. [20] observed, the traditional hierarchy-based documentation (e.g., Javadoc) are ill-suited for the normal development needs of the developers, which are usually task-based. In the forum posts, developers discuss the usage of an API within the context of development task. For example, the title of a thread in Stack Overflow can be considered as a description of a development task. The combination of a question and the answers with code examples posted in response to the question can be considered as task-based informal documentation for the APIs.

To offer a task-based documentation experience in our API usage summaries, we made two design decisions in our usage summary algorithms:

Usage Scenario Title. We associate each code example with the title of the question, e.g., the title of a thread in Stack Overflow.

Usage Scenario Description. We associate with the code example a short text describing how the task is supported by the code example.

While the lack of contents in API formal documentation can be addressed by such task-based API usage scenarios, as we noted in our previous section, such scenarios can still be aplenty. Therefore, the scenarios need to summarized *concisely*, but should still offer a complete insight about the usage of an API [40]. A documentation can be incomplete when it does not have all the necessary steps to complete a development task. A usage summary of an API can be considered as incomplete for the same reasons. An example of such incompleteness can be attributed to the lack of *concepts* in the API documentation [41]. In our previous study [42], we observed that developers in multiple unrelated software often used a predefined set of of the same API types as an *API Usage Concept* to complete a development task. For example, a developer using the HttpClient [43] API, would first establish an HTTP connection to send a GET or POST message. We also observed that the formal API documentation of HttpClient do not document many such concepts. Therefore, the

Figure 1.5: Two questions in Stack Overflow with relevant usage scenarios involving the API Jackson

identification of concepts in the usage scenarios of APIs from developer forums could be one way to summarize usage scenarios about API.

Given as input all the code examples of an API as mined from developer forums, we identify the usage concepts of an API from developer forums based on a combination of heuristics, such as, the intersection of *common* API types (e.g., class), the *similarity* of the code examples (e.g., using clone detection), and so on. For example, we cluster the two code examples in Figure 1.5 in one concept. The answers in ② and ④ are the accepted answers for the two questions ① and ③, respectively. Both answers use the Java Jackson API. The first question is about a development task involving the deserialization a JSON string from a HTTP response. The second question is about a task involving the conversion of a JSON file to XML to be served to a web server. The two code examples use the same API, com.fasterxml.jackson. The reason we consider both corresponding to the same concept is that, both use the same API type `ObjectMapper` for mapping of the JSON fields. Besides `ObjectMapper`, the solution in ④ uses only one other type of the same API, `XmlMapper`. The two types are structurally connected, because the input to `XmlMapper` is first processed through the `ObjectMapper`. Therefore, a developer looking for features related to the deserialization of JSON contents can only look at this concept to learn about the two related features (i.e., JSON and XML processing using Jackson). Indeed, the deserialization of a JSON input and its conversion to an XML can be related tasks during the communication of SOAP (XML) and REST (JSON)-based web servers (e.g., the developer program APIs [44] of the Canada Post).

With regards to the presentation of API usage summaries, one approach could be to link code examples about an API type from the forum posts in the Javadoc of the API type [35]. Unfortunately, this approach does not address the major presentation problems with API documentation as reported

by the developers in our survey at IBM. The developers complained about the *fragmented* nature of API official documentation, e.g., when they need to click through multiple pages of API documentation to learn the functionality and usage of a single API element. In our API usage summaries, we adopted the following three design decisions with regards to the presentation of the summaries:

Single Page View. We present the usage summaries of an API in one single web page (see next section). This can mitigate 'fragmented' documentation problem. To avoid overwhelming the developers in the presence of too many code examples in one single page, we only show the title of the code example and hide the rest (i.e., description, the code example, and the reviews) under the title as a collapsible paragraph.

Recency. One major problem with API formal documentation is that the contents can be outdated [19, 45]. In fact, this is one of the major documentation problems we observed in our surveys at IBM. This problem can be largely mitigated in our usage summaries, because the developers tend to discuss about new API features in the forum posts [35]. In our usage summaries, we rank the usage examples based on their recency, i.e., the most recent example is be shown at the top of all examples included in the summary.

Traceability. We add a hyperlink to each code example in a summary. The link points to the corresponding post in the developer forum, where the code example was found. In multiple user studies that we conducted to investigate the effectiveness of our usage summaries, we found that the developers considered the combination of summaries and the traceability to the actual forum posts as useful to form clear but comprehensive insights about the API.

1.5 Automatic Summarization of API Reviews & Usage Scenarios

To assist the developers in their analysis of the huge volume of reviews about APIs posted in the developer forums, we have designed a framework to automatically mine and summarize opinions about APIs from Stack Overflow. We design statistical and aspect-based summaries for API reviews by introducing API specific contextual information into the design of the algorithms:

Statistical Summarization. We aggregate the sentiments expressed towards an API in multiple forum posts. We visualize the statistics in three different formats, e.g., by computing the star rating, by showing the progression of sentiment towards an API across time, and by showing the other APIs that were co-mentioned in the same post where an API is reviewed positively or negatively.

Aspect-Based Summarization. We investigate a suite of rule-based and supervised classifiers to automatically detect API aspects in the API reviews. We group reviews by aspects. We present aspect-specific API insights, such as, most reviewed APIs by an aspect (e.g., performance). To detect the API aspects, we leverage the catalog of API aspects we presented in Section 1.2.

We also investigate the feasibility of six other cross-domain opinion summarization algorithms for the domain of API reviews.

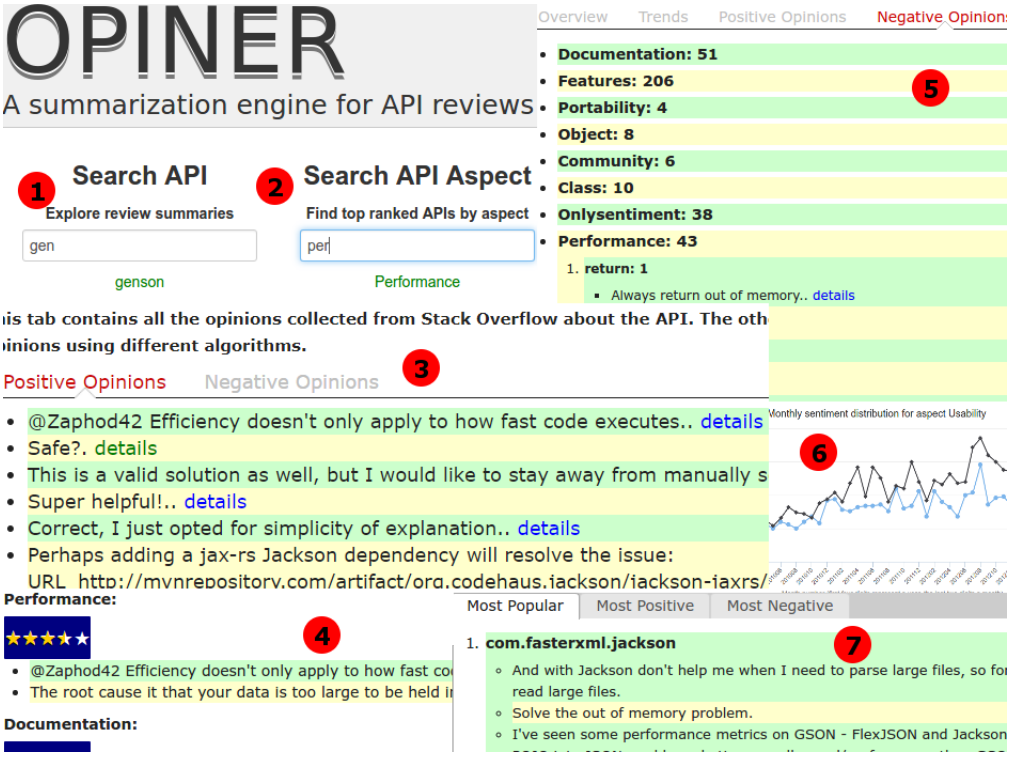

Figure 1.6: Screenshots of Opiner opinion search engine for APIs.

To assist the developers in their learning and usage of APIs from forum posts, we have developed a framework to automatically mine and summarize usage scenarios of an API from forum posts. We design four algorithms to summarize the usage scenarios about an API from developer forums:

Statistical Summarization. Presents the usage statistics using visualization.

Single Page Summarization. Presents all the usage scenarios in a single page.

Type-Based Summarization. Groups usage scenarios by the API types

Concept-Based Summarization. Groups usage scenarios by similar concepts, each concept denoting specific API feature.

We developed the techniques in our tool, named Opiner. Opiner is a mining and summarization framework for API reviews and usage scenarios. The development infrastructure of Opiner supports the automatically crawling Stack Overflow posts to mine and summarize opinions and usage scenarios about APIs.

In Figure 1.6, we show screenshots of the API review summaries in 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, 'genson' in ①, the resulting query produces a link to all the mined opinions of the API. When the

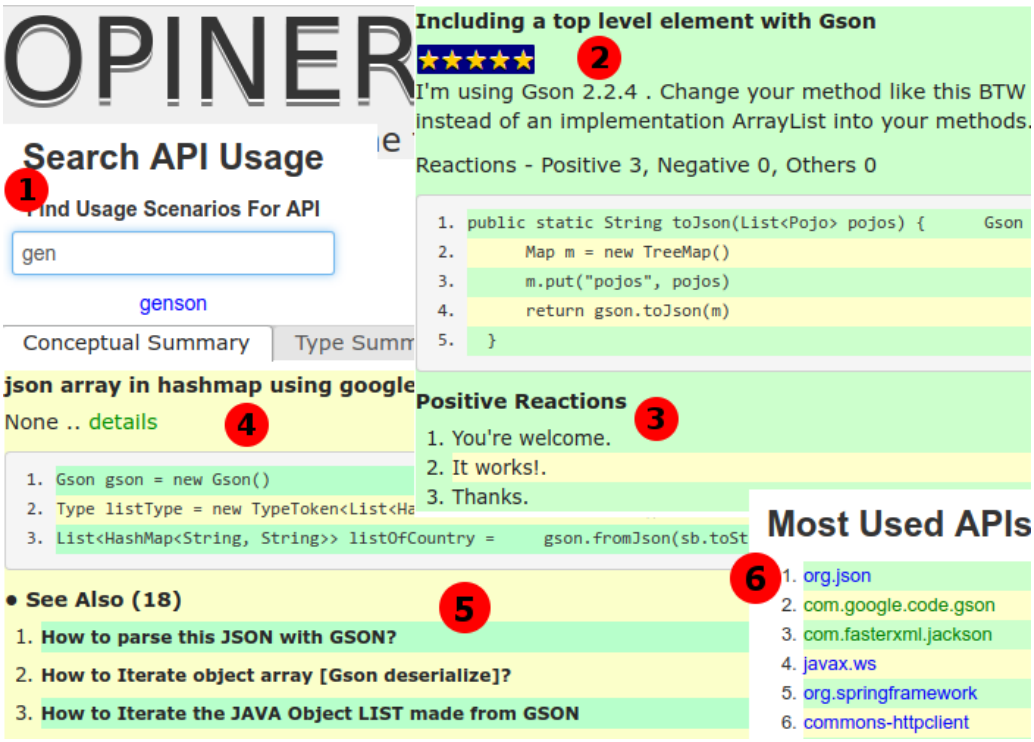

Figure 1.7: Screenshots of the Opiner API usage summarizer

user clicks the links (i.e., the link with the same name of the API as 'genson'), all the opinions about the API are shown. The opinions are grouped as positive and negative (see ③). The opinions are further categorized into the API aspects by applying the aspect detectors on the detected opinions ④. By clicking on each aspect, a user can see the opinions about the API (see ⑤). Each opinion is linked to the corresponding post from where the opinion was mined (using 'details' link in ③ or ⑤). A visualization on the the positivity and negativity towards each aspect of an API is also presented to show how developers reviewed the aspect about the API over time (e.g., 'Usability' in ⑥). Finally, a user can search for the most reviewed APIs for a given aspect using ②. For example, for the aspect 'performance', ⑦ in Opiner shows that the most reviewed API for JSON parsing in Java was 'com.fasterxml.jackson' API. It also shows the most recent five positive and negative opinions for the API in ⑦.

In Figure 1.7, we show screenshots of Opiner usage scenario summarizer. The user can search an API by its name to see the different usage summaries of the API ①. Upon clicking on the search result, the user is navigated to the usage summary page of the API ②. A usage scenario consists of a code example with a short description and the positive and negative reactions from developers ③. In ④, we show a summary of the scenarios for the GSON API, where all the relevant code examples in Figure 1.4 are grouped together, along with 15 other similar usage scenarios ⑤. Opiner's front page also shows the top 10 APIs with the most code examples ⑥.

1.6 Effectiveness of Opiner

We investigated the effectiveness of the API review and usage summaries in Opiner using five user studies. In a study with 10 professional software developers on the analysis of the usefulness of Opiner API review summaries, we found that developers favored our proposed summaries against other summaries (more than 80% vs less than 50% ratings). In another study of Industrial software developers involving two development tasks related to the selection of an API amidst choices, we found that the developers showed more precision in their selection while using Opiner and Stack Overflow together, compared to while using only Stack Overflow. While the developers found Opiner as usable, they considered it as confusing for a new user.

To assess the effectiveness of the Opiner usage summaries, we conducted a user study involving 34 participants that included both professional software developers and students. The participants completed four separate coding tasks, first using Stack Overflow, second using API official documentation, third using Opiner, and fourth using all the three resources (i.e., Opiner, Stack Overflow, and official documentation). We observed that developers wrote more correct code, spent less time and effort while using Opiner. More than 80% of the participants reported that Opiner summaries offered improvements over formal and informal documentation by presenting usage summaries *relevant* to the development tasks.

1.7 Opiner Website and Demos

- The Opiner online search and summarization engine is hosted at: `http://opiner.polymtl.ca`
- A video demo of Opiner explaining the opinion summarizers is available at: `https://youtu.be/XAXpfmg5Lqs`
- A video demo of Opiner explaining the usage scenario summarizers is available at: `https://youtu.be/-QxU5F_yhzI`

1.8 Thesis Contributions

This dissertation offers insight and algorithm support to automatically mine and summarize API reviews and usage scenarios from informal documents. The chapters are divided into two parts:

Part 1. Supporting API Review Analysis in API Informal Documentation
We report the findings of two surveys that we conducted to understand how developers seek and analyze API-related opinions from developer forum posts. A total of 178 developers responded to the surveys. Motivated by the findings, we developed a suite of techniques to automatically mine and summarize opinions about APIs from Stack Overflow.

Part 2. Supporting API Usage Analysis from API Informal Documentation
We report the findings of two surveys that we conducted to understand the problems developers face while using API formal documentation. A total of 323 IBM developers responded to

the surveys. We then present the usage scenario summarization framework of Opiner, where we designed the summarization algorithms based on the findings of the surveys.

This dissertation is organized as follows.

Chapter 2. Understanding How and Why Developers Seek and Analyze API Reviews

We present the design and results of two surveys involving a total of 178 professional software developers. We asked questions related to their seeking and analysis of opinions about APIs from developer forums. The design of the surveys and the insights gained from the results form the first contribution of this dissertation.

The research work of this chapter is currently under review as a journal [46] in the IEEE Transactions of Software Engineering.

Chapter 3. How API Formal Documentation Fails to Support API Usage

We present the design and results of two surveys involving 323 software developers from IBM. We conducted the study to understand the common problems that developers faced while using formal API documentation. The design and the analysis of the results form the second contribution of this thesis.

The research work of this chapter has been published as an IEEE Software article [1].

Chapter 4. Automatic Mining of Opinions from API Reviews

We 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 labeled as (1) the API aspect that is discussed in the sentence and (2) the overall polarity (positive, negative, neutral) expressed in 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 in the opinions. The dataset and the case study form the third contribution of this thesis. We leverage the dataset to investigate the automatic detection of API aspects.

We then present an analysis framework to automatically mine opinions about APIs. We proposed two algorithms to resolve API mentions in forum posts and to associate those mentions to the opinionated sentences in the posts. The framework with the proposed algorithms and its evaluation form the fourth contribution of this thesis.

The research work of this chapter is currently under review as a journal [47] in the IEEE Transactions of Software Engineering.

Chapter 5. Automatic Summarization of API Reviews

We designed two algorithms (aspect-based and statistical) to summarize opinions about APIs. We compared the algorithms against six other algorithms from literature. We present the results of two user studies that we conducted to analyze the effectiveness of the summaries to assist developers in some development tasks. The design of the proposed two summarization algorithms for API reviews, and their evaluation form the fifth contribution of the thesis.

The research work of this chapter was the subject of two publications (full research paper [48] and tool demo [49]) in the International Conference of Automated Software Engineering.

Chapter 6. Automatic Summarization of Crowd-Sourced API Usage Scenarios

We present a framework to automatically mine and summarize usage scenarios about APIs.

We present three user studies to evaluate the usage summaries in Opiner. The design of the framework and its evaluation form the sixth contribution of this thesis.

The research work of this chapter is currently under review as a journal [50] in the IEEE Transactions of Software Engineering.

Chapter 7. Related Research

We survey the related research and summarize the key differences between our research and the related research.

Chapter 8. Conclusions and Future Work

We discuss future work involving the various summaries produced in Opiner. We conclude by summarizing the contributions of this dissertation.

Chapter 2

Understanding How and Why Developers Seek and Analyze API Reviews

Opinions can be key determinants to many of the activities related to software development, such as developers' productivity analysis [51], determining developers burnout [52], improving software applications [53], and developing awareness tools for software development teams [54–56]. Research on APIs has produced significant contributions, such as automatic usage inference mining [57, 58], automatic traceability recovery between API elements (types, methods) and learning resources [59–61], and recommendation systems to facilitate code reuse [6, 62] (see Chapter 7).

However, we are aware of no research that focuses on the analysis of developers' perception about API reviews and how such opinions affect their decisions related to the APIs. As illustrated in Figure 1.1 of Chapter 1, while developer forums serve as communication channels for discussing the implementation of the API features, they also enable the exchange of opinions or sentiments expressed on numerous APIs, their features and aspects. Given the presence of sentiments in the forum posts and opinions about APIs, such insights can be leveraged to develop techniques to automatically analyze API reviews in forum posts. Such insights can also contribute to the development of an empirical body of knowledge on the topic and to the design of tools that analyze opinion-rich information.

To fill this gap in the literature, we conducted two surveys involving a total of 178 professional software developers. The *goals* of our study are to *understand* (1) how professional software developers seek and value opinions about APIs, and (2) what tools can better support their analysis and evaluation of the API reviews. The *subjects* are the survey participants and the *objects* are the API reviews that the developers encounter in their daily development activities from diverse resources. The *contexts* are the various development activities that can be influenced by the API reviews.

Through an exploratory analysis of the survey responses, we answer the following research questions:

RQ1: How do developers seek and value opinions about APIs in developer forums?

The developers reported that they seek for opinions about APIs in forum posts to support diverse development needs, such as API selection, documentation, learning how to use an API, etc.

The developers valued the API reviews. However, they were also cautious while making informed decisions based on those reviews due to a number of factors related to the quality of the provided opinions, such as, lack of insight into the prevalence of the issue reported in the opinion, the trustworthiness of the provided opinion (e.g., marketing initiatives vs subjective opinion), and so on. The developers wished for the support of different mechanisms to assess the quality of the provided opinions about APIs in the developer forums.

RQ2: What tool support is needed to help developers with analyzing and assessing API reviews in developer forums?

The developers consider that automated tools of diverse nature can be developed and consulted to properly analyze API reviews in forum posts, e.g., visualizations of the aggregated sentiment about APIs to determine their popularity, understanding the various aspects (e.g., performance) about APIs and how other developers rate those aspects about APIs based on their usage of the APIs, and so on.

The developers mentioned that the huge volume of available information and opinions about APIs in the forum posts can hinder their desire to get quick, digestible and actionable insights about APIs. The developers also mentioned that in the absence of any automated summarization technique to help them, they leverage different features in forum posts to get summarized viewpoints of the APIs, e.g., skimming through high ranked posts, using tags to find similar APIs, etc. Developers also envision diverse potential summarization approaches to help them to address such needs, e.g., a dedicated portal to show aggregated opinions about APIs.

Chapter organization. Section 2.1 explains the context of this research by outlining the sub-questions that we explored under the two research questions. In Section 2.2, we discuss the design and results of our first survey. In Section 2.3, we describe the design of the second survey, and in Section 2.4, we present the findings of the survey. In Section 2.5, we discuss the implications of our work. The scope and potential threats to the validity of the study are discussed in Section 2.6. Finally, Section 2.7 concludes the chapter.

2.1 Research Context

In this section, we discuss the motivation behind each research question, along with its sub-questions and detailed rationale underlying each question.

2.1.1 Reasons for Seeking Opinions About APIs (RQ1)

We aim to understand how and why developers seek and analyze such opinions about APIs and how such information can shape their perception and usage of APIs. The first goal of our study is to learn how developers seek and value the opinions of other developers.

2.1.1.1 Motivation

By analyzing how and why developers seek for opinions about APIs, we can gain insights into the role of API reviews in the daily development activities of software developers. The first step towards understanding the developers needs, is to learn about the resources they currently leverage to seek information and opinions about APIs. The developers may use diverse resources to seek for opinions about APIs. If a certain resource is used more than other resources, the analysis of the resource can be given more priority over others.

Our observation of the API reviews show that opinions about APIs are prevalent in the developer forums. Therefore, an understanding of how the different development activities can be influenced and supported through the API reviews can offer us insights into the factors that motivate developers

Table 2.1: The formulation of the research questions in the study

RQ1	**How do developers seek and value opinions about APIs?**
1.1	***Theme - Opinion Needs: How do developers seek opinions about APIs?***
1.1.a	*Where do developers seek opinions?*
1.1.b	*What motivates developers to seek opinions about APIs?*
1.1.c	*What challenges do developers face while seeking API reviews?*
1.2	***Theme - Opinion Quality: How do developers assess the quality of the provided opinions about APIs?***
RQ2	**How can the support for automated processing of opinions assist developers to analyze API reviews?**
2.1	***Theme - Tool Support: What tool support is needed to help developers to assess API reviews?***
2.2	***Theme - Summarization Needs: What are the developers' needs for summarization of API reviews?***
2.2.a	*What problems in API reviews motivate the need for summarization?*
2.2.b	*How summarization of API reviews can support developer decision making?*
2.2.c	*How do developers expect API reviews to be summarized?*

to seek for opinions and the challenges they may face during this process. By learning about the challenges that developers may face while seeking reviews about APIs, we can gain insights into the complexity of the problem that needs to be addressed to assist developers in their exploration of API reviews.

Finally, opinions are by themselves *subjective*, i.e., opinions about APIs stem from the personal belief or experience of the developers who use the APIs. Therefore, developers may face challenges while assessing the validity of someone's claim. By analyzing what factors can hinder and support the developers' assessment of the quality of the provided API reviews, we can gain insights into the challenges developers face while leveraging the API reviews.

2.1.1.2 Approach

We examine developer needs for API reviews via the following questions:

- RQ1.1: How do developers seek opinions about APIs?
- RQ1.2: How do developers assess the quality of the provided opinion?

To answer these questions exhaustively, we further divide RQ1.1 into three sub-questions:

- *RQ1.1.a: Where do developers seek opinions about APIs?*
- *RQ1.1.b: What motivates developers to seek for opinions?*
- *RQ1.1.c: What challenges do developers face while seeking for opinions?*

2.1.2 Tool Support to Analyze API Reviews (RQ2)

The second goal of our study is to understand the needs for tool support to facilitate automated analysis of API reviews.

2.1.2.1 Motivation

Before we start to develop any tool or do any feasibility analysis of such tools, we need to understand what tools developers may be using currently to analyze API reviews. We can learn from the developers about the tools they may have at their disposal to analyze API reviews. We can also learn about the tools the developers would like to have to process API reviews efficiently. Such analysis can offer insights into how research in this direction can offer benefits to the developers through future prototypes and tool supports.

A predominant direction in the automated processing of reviews in other domains (e.g., cars, cameras, products) is to summarize the reviews. For the domain of API reviews, it can also help to know how developers determine the needs for opinion summarization about APIs. The domain of API reviews can be different from other domains (e.g., car reviews, camera reviews). The first step is to determine the feasibility of the existing cross-domain opinion summarization techniques to the domain of API reviews. Such analysis can provide insights into how the summarization approaches adopted in other domains can be applicable to the domain of API reviews. Clearly, the summarization of API reviews needs to address the pain points of the developers, i.e., their specific development needs. By learning about the specific development needs that can be better supported though the summarization of API reviews, we can gain insights into the potential use cases API review summaries can support. By understanding how developers expect to see summaries of API reviews, we can gain insights into whether and how different summarization techniques can be designed and developed for the domain of API reviews. It is, thus, necessary to know what specific problems in the API reviews should be summarized and whether priority should be given to one API aspect over another.

2.1.2.2 Approach

We pose two research questions to understand the needs for tool support to analyze API reviews:

- RQ2.1: What tool support is needed to help developers with analyzing and assessing API-related opinions?
- RQ2.2: What are the developers' needs for summarization of opinion-rich information?

To answer these two questions exhaustively, we further divide RQ2.2 into three sub-questions:

- *RQ2.2.a: What problems in API reviews motivate the needs for summarization?*
- *RQ2.2.b: How can summarization of API reviews can support developer decision making processes?*
- *RQ2.2.c: How do developers expect API reviews to be summarized?*

2.1.3 The Surveys

We learn about the developer needs for API reviews and tool support for analyzing the reviews through two surveys. We conducted the first survey as a pilot survey and the second as the primary one. The purpose of the pilot survey is to identify and correct potential ambiguities in the design of the primary survey. Both the pilot and the primary surveys share the same goals. However, the questions of the primary surveys are refined and made more focused based on the findings of the pilot survey. For example, in the pilot survey, we mainly focused on the GitHub developers. We found that most of the respondents in our pilot survey considered the developer forums as the primary source of opinions about APIs. Therefore, in the primary survey, our focus was to understand how developers seek and analyze opinions in developer forums.

In Section 2.2, we discuss the design and the summary of results of the pilot survey. In Sections 2.3 and 2.4, we discuss the design and detail results of the primary survey.

2.2 The Pilot Survey

The pilot survey consisted of 24 questions: three demographic questions, eight multiple-choice, five Likert-scale questions, and eight open-ended questions. In Table 2.2, we show all the questions of the pilot survey (except the demographic questions) in the order they appeared in the survey questionnaires. The demographic questions concern the participants role (e.g., software developer or engineer, project manager or lead, QA or testing engineer, other), whether they are actively involved in software development or not, and their experience in software development (e.g., less than 1 year, more than 10 years, etc.). The survey was hosted in Google forms and can be viewed at `https://goo.gl/forms/8X5jKDKilkfWZT372`.

2.2.1 Pilot Survey Participants

We sent the pilot survey invitations to 2,500 GitHub users. The 2,500 users were randomly sampled from 4,500 users from GitHub. The 4,500 users were collected using the GitHub APIs, which returns GitHub users starting with the ID 1. From the number of emails sent to GitHub users, 70 emails were bounced back for various reasons, e.g., invalid (domain expired) or non-existent email addresses, making it 2,430 emails being actually delivered. A few users emailed us saying that they were not interested in participating due to the lack of any incentives. Finally, a total of 55 developers responded. In addition, we sent the invitation to 11 developers in a software development

Table 2.2: Pilot survey questions with key highlights from responses.

RQ1.1	**Opinion Needs**
1	Do you value opinion of other developer when deciding on what API to use? *(yes / no)* ***1) Yes 95%, 2) No 5%***
2	Where do you seek help/opinions about APIs? *(five sources and others)* ***1) Developer Forums 83.3%, 2) Co-worker 83.3%, 3) Mailing List 33.3%, 4) IRC 26.7%, 5) Others 35%***
3	How often do you refer to online forums (e.g, Stack Overflow) to get information about APIs? *(five options)* ***1) Every day 23.3%, 2) Two/three times a week 36.7%, 3) Once a month 25%, 4) Once a week 10%, 5) Never 5%***
4	When do you seek opinions about APIs? *(5-point likert scale for each option)* ***1) Select amidst choices 83.3%, 2) Select API version 18.3%, 3) Improve a feature 61.7%, 4) Fix bug 63.3%, 5) Determine a replacement 73.3%, 6) Validate a selection 45%, 7) Develop a competing API 58.3%, 8) Replace an API feature 51.7%***
5	What are other reasons of you referring to the opinions about APIs from other developers? *(text box)* ***1) Opinion trustworthiness analysis 18.8% 2) API usage 15.6%, 3) API suitability 15.6%, 4) API maturity 6.3%, 5) Usability analysis 6.3%***
6	What is your biggest challenge while seeking opinions about an API? *(text box)* ***1) Trustworthiness analysis 19.4% 2) Too much info 16.1%, 3) Biased opinion 12.9%, 4) Documentation 9.7%, 5) Staying aware 6.5%***
RQ2.1	**Tool Support**
7	What tools can better support your understanding of opinions about APIs in online forum discussions? *(5-point likert scale for each option)* ***1) Opinion mine & summarize 68.3%, 2) Sentiment mine 45%, 3) Comparator analyze 73.3%, 4) Trends 56.7%, 5) Co-mentioned APIs 76.7%***
8	What other tools can help your understanding of opinions about APIs in online forum discussions? *(text box)* ***1) Reasoning 18.2%, 2) Extractive summaries 18.2%, 3) Dependency info 12.1%, 4) Documentation 9.1%, 5) Expertise find 9.1%***
RQ2.2	**Summarization Needs**
9	How often do you feel overwhelmed due to the abundance of opinions about an API? *(four options)* ***1) Every time 5%, 2) Some time 48.3%, 3) Rarely 33.3%, 4) Never 13.3%***

10	Would summarization of opinions about APIs help you to make a better decision on which one to use? *(yes/no)* ***1) Yes 83.3%, 2) No 16.7%***
11	Opinions about APIs need to be summarized because? *(5-point likert scale for each option)* ***1) Too many posts with opinions 60%, 2) Opinions can evolve over time 65%, 3) Opinions can change over time 71.7%, 4) Interesting opinion in another post 66.7%, 5) Contrastive viewpoints missed 55%, 6) Not enough time to look for all opinions 56.7%***
12	Opinion summarization can improve the following decision making processes. *(5-point likert scale for each option)* ***1) Select amidst choices 85%, 2) Select API version 45%, 3) Improve a feature 48.3%, 4) Fix bug 40%, 5) Determine a replacement 78.3%, 6) Validate a selection 53.3%, 7) Develop a competing API 48.3%, 8) Replace an API feature 46.7%***
13	What other areas can be positively affected having support for opinion summarization? *(text box)* ***1) API usage 21.4%, 2) Trustworthiness analysis 21.4%, 3) Documentation 9.5%, 4) Maturity analysis 14.3%, 5) Expertise 7.1%***
14	What other areas can be negatively affected having support for opinion summarization? *(text box)* ***1) Trustworthiness analysis 33.3%, 2) Info overload 33.3%, 3) API usage 8.3%, 4) Reasoning 8.3%***
15	An opinion is important if it contains discussion about one or more of the following API aspects? *(5-point likert scale for each option)* ***1) Performance 86.7%, 2) Security 85%, 3) Usability 85%, 4) Documentation 88.3%, 5) Compatibility 75%, 6) Community 73.3%, 7) Bug 86.7%, 8) Legal 43.3%, 9) Portability 50%, 10) General Features 48.3%, 11) Only sentiment 15%***
16	What are the other factors you look for in API opinions? *(text box)* ***1) API usage 28.6%, 2) Usability 19%, 3) Expertise 14.3%, 4) Standards 9.5%, 5) Reasoning 9.5%***
17	What type of opinion summarization would you find most useful? *(five types)* ***1) In a paragraph 83.3%, 2) Divided into aspects 78.3%, 3) Top N by most recent 36.7%, 4) Divided into topics 35%, 5) Others 1.7%***
18	How many keywords do you find to be sufficient for topic description? *(five options)* ***1) Five words 61.7%, 2) 5-10 words 28.3%, 3) 10-15 words 5%, 4) 15-20 words 3.3%, 5) Not helpful at all 1.7%***
19	What are the other ways opinions about APIs should be summarized? *(text box)*

	1) API Usage 25%, 2) Expertise 8.3%, 3) Documentation 8.3%, 4) Popularity 8.3%, 5) Testing 8.3%
RQ1.2	**Opinion Quality**
20	How do you determine the quality of a provided opinion in a forum post? (e.g., Stack Overflow) *(seven options)* ***1) Date 55%, 2) Votes 66.7%, 3) Supporting links 70%, 4) User profile 45%, 5) Code presence 73.3%, 6) Post length 16.7%, 7) Others 1.7%***
21	What other factors in a forum post can help you determine the quality of a provided opinion? *(text box)* ***1) Documentation 63.3%, 2) Expertise 10.5%, 3) Trustworthiness 10.5%, 4) Situational relevance 10.5%, 5) Biased opinion 5.3%***

company in Ottawa, Canada. Out of the 11, nine responded. Among the GitHub participants: 1) 78% of respondents said that they are software developers (11% are project managers and 11% belong to "other" category), 2) 92% are actively involved in software development, and 3) 64% have more than 10 years of software development experience, 13% of them have between 7 and 10 years of experience, 9% between 3 to 6 years, 8% between 1 to 2 years and around 6% less than 1 year of experience. Among the nine Industrial participants, three were team leads and six were professional developers. All nine participants were actively involved in software development. The nine participants had professional development experience between five to more than 10 years.

2.2.2 Pilot Survey Data Analysis

We analyzed the survey data using statistical and qualitative approaches. For the open-ended questions, we applied an open coding approach [63]. As we analyzed the quotes, themes and categories emerged and evolved during the open coding process.

We created all of the "cards", splitting the responses for eight open-ended questions. This resulted into 173 individual quotes; each generally corresponded to individual cohesive statements. In further analysis, Gias Uddin and Prof Olga Baysal acted as coders to group cards into themes, merging those into categories. We analyzed the responses to each open-ended question in three steps:

1) The two coders independently performed card sorts on the 20% of the cards extracted from the survey responses to identify initial card groups. The coders then met to compare and discuss their identified groups.

2) The two coders performed another independent round, sorting another 20% of the quotes into the groups that were agreed-upon in the previous step. We then calculated and report the coder reliability to ensure the integrity of the card sort. We selected two popular reliability coefficients for nominal data: percent agreement and Cohen's Kappa [64].

   Coder reliability is a measure of agreement among multiple coders for how they apply codes to text data. To calculate agreement, we counted the number of cards for each emerged group

for both coders and used ReCal2 [65] for calculations. The coders achieved the *almost perfect* degree of agreement; on average two coders agreed on the coding of the content in 96% of the time (the average percent agreement varies across the questions and is within the range of 92–100%; while the average Cohen's Kappa score is 0.84 ranging between 0.63–1 across the questions).

3) The rest of the card sort (for each open-ended question), i.e., 60% of the quotes, was performed by both coders together.

2.2.3 Summary of Results from the Pilot Survey

In this section, we briefly discuss the major findings from the pilot survey, that influenced the design of the primary survey. A detail report of the findings of our pilot survey is in our online appendix [66].

In Table 2.2, the type of each question (e.g., open or closed-ended) is indicated beside the question (in *italic* format). The *key findings* for each question are provided under the question in Table 2.2 (in ***bold italic*** format). The key findings are determined as follows:

Responses to open-ended questions. The response to an open-ended question is provided in a text box (e.g., Q5). For each such question, we show the top five categories (i.e., themes) that emerged from the responses of the question. The frequency of categories under a question is used to find the top five categories for the question.

Likert-scale questions Each such question can have multiple options for a user to choose from (e.g., Q7). Each option can be rated across five scales: Strongly Agree, Agree, Neutral, Disagree, and Strongly Disagree. For each question, we calculate the percentage of respondents that agreed (= Total Agreed + Total Strongly Agreed) with the provided option.

Multiple choice questions. There were two types of multiple choice questions. 1) Only Select One: Each such question can have more than one choice and a participant can pick one of the the options (e.g., Q1). 2) Can Select Multiple. The participant can pick one, more than one, or all the choices, when the options are of type multiple select (e.g., Q17). For each option, we calculate the percentage of respondents that picked the option.

Needs for API Reviews (RQ1). 95% of the participants reported that they consider opinions of other developers while making a selection of an API. The primary sources of such opinions for them are both developer forums and co-workers (83.3%). 69.3% of the participants reported to have consulted developers forums to gain information about APIs. Most of the participants (83.3%) reported that they consider opinions about APIs while making a selection of an API amidst multiple choices. They also seek opinions to determine a replacement of an API (73.3%), as well as fixing a bug (63.3%), and so on. Among the other reasons for the participants to seek for opinion were the learning of API usage and the analysis of validity of a provided API usage (e.g, if it actually works). The participants reported that making an informed decision amidst too much opinion, and judging the trustworthiness of the provided opinions are the major challenges they face while seeking and analyzing opinions.

Most of the participants (73.3%) considered an opinion, accompanied by a code example, to be of high quality. 70% of the participants considered the presence of links to supporting documents as a good indicator of the quality of the provided opinion, while 55% believed that the posting date is an important factor. Stack Overflow uses upvotes and downvotes of a post as a gauge of the public opinion on the content. 66.7% of developers attributed a higher count of votes on the post to their assumption of the higher quality of the corresponding opinion in the post. The user profile, another Stack Overflow feature, was found as useful by 45% of participants to evaluate the quality of the provided opinion. The length of the post where the opinion is provided was not considered as a contributing factor during the assessment of the opinion quality (agreed by only 16.7% of the responders). One participant did not agree with any of the choices, while two participants mentioned two additional factors: (1) real example, i.e., the provided code example should correspond to a real world scenario; and (2) reasoning of the provided opinion.

Tools for API Review Analysis (RQ2). More than 83% of the respondents agreed that opinion summarization is much needed for several reasons. Since opinions change over time or across different posts, developers wanted to be able to track changes in API opinions and to follow how APIs evolve. The vast majority of responders also thought that an interesting opinion about an API might be expressed in a post that they have missed or have not looked at. The need for opinion summarization was also motivated by the myriad of opinions expressed via various developer forums. Developers believed that the lack of time to search for all possible opinions and the possibility of missing an important opinion are strong incentives for having opinions organized in a better way.

While developers were interested in a tool support for mining and summarizing opinions about APIs, they also wanted to link such opinions to other related dimensions, such as, usage, maturity, documentation, and user expertise. 85% of the respondents believed that opinion summarization could help them select an API amidst multiple choices. More than 78% believed that opinion summaries can also help them find a replacement to an API. Developers agreed that summaries can also help them develop a new API to address the needs that are currently not supported, improve a software feature, replace an API feature, validate the choice of an API, select the right version of an API, and fix a bug (48.3%, 48.3%, 46.7%, 53.3%, 45%, and 40%, respectively).

2.2.4 Needs for the Primary Survey

The responses in our pilot survey showed that opinions about APIs are important in diverse development needs. However, the survey had the following limitations:

Design.

A number of similar questions were asked in pairs. For example, in Q4, we asked the developers about the reasons to seek for opinions. The developers were provided eight options to choose from. In Q5, we asked the developers to write about the other reasons that could also motivate them to seek for opinions. Q5 is an open-ended questions. Both Q4 and Q5 explore similar themes (i.e., needs for opinion seeking). Therefore, the responses in Q5 could potentially be biased due to the respondents already being presented the eight possible options in Q4. A review of the manuscript based on the pilot survey (available in our online appendix [66]) both by the colleagues and the reviewers in Transaction of Software Engineering pointed out that a better approach would have been to ask Q5 before Q4, or only ask

Q5. Moreover, the respondent should not be given an option to modify his response to an open-ended question, if a similar themed closed-ended question is asked afterward.

Sampling.
We sampled 2500 Github developers out of the first 4500 Gibhub IDs as returned by the Github API. We did not investigate the relevant information of the developers, e.g., are they still active in software development? Do they show expertise in a particular programming languages?, and so on. This lack of background information on the survey population can also prevent us from making a formal conclusion out of the responses of the survey.

Response Rate.
While previous studies involving Github developers also reported low response rate (e.g., 7.8% by Treude et al. [67]), the response rate in our pilot survey was still considerably lower (only 2.62%). There can be many reasons for such low response rate. For example, unlike Treude et al. [67], we did not offer any award/incentives to participate in the survey. However, our lack of enough knowledge of the Github survey population prevents us from making a definitive connection between the low response rate and the lack of incentives.

We designed the primary survey to address the above limitations. Specifically, we took the following steps to avoid the above problems in our primary survey:

1) We asked all the open-ended questions before the corresponding closed-ended questions. The respondents only saw the closed-ended questions after they completed their responses to all the open-ended questions. The respondents were not allowed to change their response to any open-ended question, once they were asked the closed-ended questions.

2) We conducted the primary survey with a different group of software developers, all collected from Stack Overflow. To pick the survey participants, we applied a systematic and exhaustive sampling process (discussed in the next section).

3) We achieved a much higher response rate (15.8%) in our primary survey.

In the next section, we discuss in details about the design of the primary survey. In Section 2.5, we briefly compare the results of the pilot and primary survey on the similar-themed question pairs.

2.3 Primary Survey Design

We conducted the primary survey with a different group of professional software developers. Besides the three demographic questions, the primary survey contained 24 questions. In Table 2.3, we show the questions in the order they were asked. We show how the questions from the pilot survey were asked in the primary survey in the last column of Table 2.3. The survey was conducted using Google forms and is available for view at: `https://goo.gl/forms/2nPVUgBoqCcAabwj1`.

In Table 2.3, the horizontal lines between questions denote sections. For example, there is only one question in the first section (Q1). Once a developers responds to all the questions in a section, he is navigated to the next section. Depending on the type of the question, the next section is

Table 2.3: Questions asked in the primary survey. The horizontal lines between questions denote sections

No	Question	Theme	Map (Pilot)
1	Do you visit developer forums to seek info about APIs? *(yes/no)*	RQ1.1.a	
2	List top developer forums you visited in the last two years *(text box)*	RQ1.1.a	
3	Do you value the opinion of other developers in the developer forums when deciding on what API to use? *(yes/no)*	RQ1.2	1
4	What are your reasons for referring to opinions of other developers about APIs in online developer forums? *(text box)*	RQ1.1.b	5
5	How do you seek information about APIs in a developer forum? How do you navigate the multiple posts?	RQ1.1.c	
6	What are your biggest challenges when seeking for opinions about an API in an online developer forum? *(text box)*	RQ1.1.c	6
7	What factors in a forum post can help you determine the quality of a provided opinion about an API? *(text box)*	RQ1.2	21
8	Do you rely on tools to help you understand opinions about APIs in online forum discussions? *(yes/no)*	RQ2.1	
9	If you don't use a tool currently to explore the diverse opinions about APIs in developer forums, do you believe there is a need of such a tool to help you find the right viewpoints about an API quickly? *(yes/no)*	RQ2.1	
10	You said yes to the previous question on using a tool to navigate forum posts. Please provide the name of the tool. *(text box)*	RQ2.1	8
11	What are the important factors of an API that play a role in your decision to choose an API? *(text box)*	RQ2.2.c	16
12	Considering that opinions and diverse viewpoints about an API can be scattered in different posts and threads of a developer forum, what are the different ways opinions about APIs can be summarized from developer forums? *(text box)*	RQ2.2.c	19
13	What areas can be positively affected by the summarization of reviews about APIs from developer forums? *(text box)*	RQ2.2.b	13
14	What areas can be negatively affected by the summarization of reviews about APIs from developer forums? *(text box)*	RQ2.2.b	14
15	An opinion is important if it contains discussion about the following API aspects? *(5-point likert scale for each option)*	RQ2.2.c	15
16	Where do you seek help/opinions about APIs? *(5 opinions + None + Text box to write other sources)*	RQ1.1.a	2
17	How often do you refer to online forums (e.g., Stack Overflow) to get information about APIs? *(five options)*	RQ1.1.a	3

18	When do you seek opinions about APIs? *(5-point likert scale for each option)*	RQ1.1.b	4
19	What tools can better support your understanding of API reviews in developer forums? *(5-point likert scale for each option)*	RQ2.1	7
20	How often do you feel overwhelmed due to the abundance of opinions about an API? *(four options)*	RQ2.2.a	9
21	Would a summarization of opinions about APIs help you to make a better decision on which one to use? *(yes/no)*	RQ2.2.a	10
22	Opinions about an API need to be summarized because *(5-point likert scale for each option)*	RQ2.2.a	11
23	Opinion summarization can improve the following decision making processes *(5-point likert scale for each option)*	RQ2.2.b	12
24	Please explain why you don't value the opinion of other developers in the developer forums. *(text box)*	RQ1.2	

determined. For example, if the answer to first question (i.e., "Do you visit developer forums to seek info about APIs?") is a 'No', we did not ask him any further questions. If the answer is a 'Yes', the respondent is navigated to the second question. The navigation between the sections was designed to ensure two things: 1) that we do not ask a respondent irrelevant questions. For example, if a developer does not value the opinions of other developers, it is probably of no use asking him about his motivation for seeking opinions about APIs anyway, and 2) that the response to a question is not biased by another relevant question. For example, the first question in the third section of Table 2.3 is Q4 ("What are your reasons for referring to opinions of other developers about APIs in online developer forums?"). This was an open-ended question, where the developers were asked to write their responses in a text box. A relevant question was Q18 in the sixth section ("When do you seek opinions about APIs?"). The developers were given eight options in a Likert scale (e.g., selection of an API amidst choices). The developers were able to answer Q18 only after they answered Q4. The developers were not allowed to return to Q4 from Q18. We adopted similar strategy for all such question pairs in the primary survey. In this way, we avoided the problem of potential bias in the developers' responses in our primary survey.

The second last column in Table 2.3 shows how the questions are mapped to the two research questions (and the sub-questions) that we intend to answer. The pilot and the primary surveys contained similar questions. The last column of Table 2.3 shows how 18 of the 24 questions in the primary survey were similar to 18 questions in the Pilot survey. While the two sets of questions are similar, in the primary survey the questions focused specifically on developer forums. For example, Q4 in the primary survey (Table 2.3) was "What are your reasons for referring to opinions of other developers about APIs in online developer forums?" The similar question in the pilot survey was Q5 (Table 2.2): "What are other reasons for you to refer to the opinions about APIs from other developers?"

To ensure that we capture developers' experience about API reviews properly in the primary survey, we also asked six questions that were not part of the pilot survey. The first two questions (discussed below) in the primary survey were asked to ensure that we get responses from developers who indeed seek and value opinions about APIs. The first question was "Do you visit developer

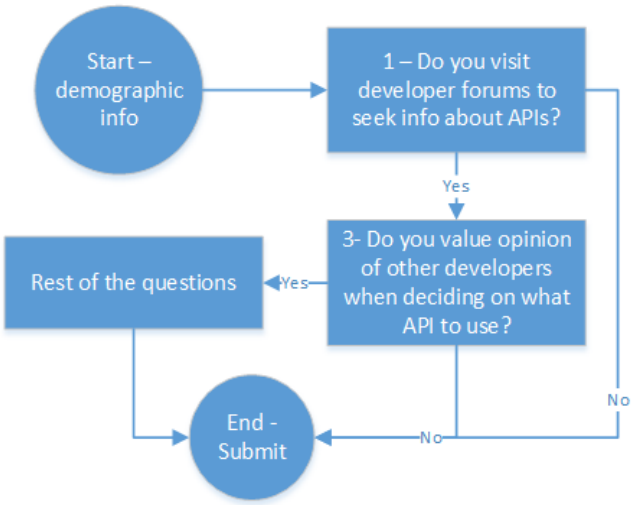

Figure 2.1: Navigation between sections in the primary survey.

forums to seek info about APIs?". If a developer responded with a 'No' to this question, we did not ask the him any further questions. We did this because 1) developer forums are considered as the primary resource in our pilot survey and 2) automated review analysis techniques can be developed to leverage the developer forums. The second new question was about asking the participants about the top developer forums they recently visited. Such information can be useful to know which developer forums can be leveraged for such analysis. The third new question was "Do you value the opinion of other developers in the forum posts while deciding on what API to use?". If the response was a 'no', we asked the participant only one question (24): "Please explain why you don't value the opinion of developers in the forum posts". We asked this question to understand the potential problems in the opinions that may be preventing them from leveraging those opinions. In Figure 2.1, we show how the above two questions are used to either navigate into the rest of the survey questions or to complete the survey without asking the respondents further questions.

2.3.1 Participants

We targeted developers that participated in the Stack Overflow forum posts (e.g., asked a question or provided an answer to a question in Stack Overflow). Out of a total of 720 invitations sent, we received 114 responses (response rate 15.8%). Among those, 72.8% responded that they visit developers forums and they value the opinion of other developers in the forums. The distribution of the profession of those participants is: 1) Software developers 79.5%, 2) Tech/Team Lead 13.3%, 3) Research Engineer 3.6%, 4) Student 3.6%, and 5) Other 1.2% . The distribution of experience of the participants is: 1) 10+ years 56.6%, 2) Seven to 10 years 28.9%, and 3) Three to six years 14.5%. To report the experience, the developers were given five options to choose from (following Treude et al. [67]): 1) less than 1 year, 2) 1 to 2 years, 3) 3 to 6 years, 4) 7 to 10 years, and 5) 10+ years. None of the respondents reported to have software development experience of less than three years. Therefore, we received responses from experienced developers in our primary survey. 97.6% of them were actively involved in software development.

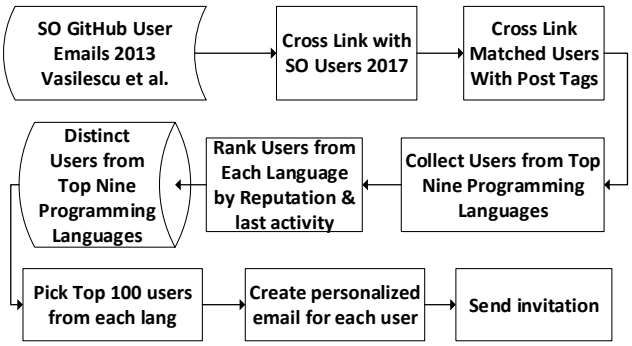

Figure 2.2: Sampling of participants for the primary survey.

2.3.1.1 Sampling Strategy

To recruit the participants for an empirical study in software engineering, Kitchenham et al. [68] offered two recommendations:

Population *"Identify the population from which the subjects and objects are drawn"*, and

Process *"Define the process by which the subjects and objects were selected"*.

We followed both of the two suggestions to recruit the participants for our primary survey (discussed below).

The contact information of users in Stack Overflow is kept hidden from the public to ensure that the users are not spammed. Vasilescu et al. [69] correlated the user email hash from Stack Overflow to those in Github. To ensure that the mining of such personal information is not criticized by the Stack Overflow community in general, Bogdan Vasilescu started a question in Stack Overflow with the theme, which attracted a considerable number of developers from the Stack Overflow community in 2012 [70]. The purpose of Vasilescu et al. [69] was to see how many of the Stack Overflow users are also active in Github. In August of 2012, they found 1,295,623 Stack Overflow users. They cross-matched 93,772 of those users in Github. For each of those matched users in Github, they confirmed their email addresses by mapping their email hash in Stack Overflow to their email addresses as they shared in Github. Each user record in their dataset contains three fields: (1) UnifiedId: a unique identifier for each record (an auto-incremental integer), (2) GithubEmail: the email address of the user as posted in Github, (3) SOUserId: the ID of the user in Stack Overflow. We used this list of 93,772 users as the *potential population source* of the primary survey. In our survey invitation, we were careful not to spam the developers. For example, we only sent emails to them twice (the second time as a reminder). In addition, we followed the Canadian Anti-Spam rules [71] while sending the emails, by offering each email recipient the option to 'opt-out' from our invitation. We did not send the second (i.e., reminder) email to the users who decided to opt-out.

We sampled 900 users out of the 93,772 users as follows (Figure 2.2):

1) **Match:** Out of the 93,772 users in the list of Vasilescu et al. [69], we found 92,276 of them in

Table 2.4: Summary statistics of the final survey population and sampled users from the population

	Total	Reputation				
	Users	Min	Max	Median	Avg	Standard Dev
Population	88,021	1	627,850	159	1,866.5	10,169.1
Sample	900	5471	627,850	18,433.5	43,311.22	66,738

	Threads	Created			Last Active		
	Contributed	2012	2011	<=2010	2017	2016	<=2015
Population	3,233,131	16,080	26,257	3,190,794	40,780	10,409	3,181,942
Sample	772,760	47	151	772,562	900	0	0

the Stack Overflow dataset of March 2017. We cross-linked the user 'ID' in the Stack Overflow dataset to the 'SOUserId' field in the dataset of Vasilescu et al. [69] to find the users. For each user, we collected the following information: (1) Name, (2) Reputation, (3) User creation date, and (4) Last accessed date in Stack Overflow.

2) **Discard:** Out of the 92,276 users, we discarded 4,255 users whose email addresses were also found in the list of 4500 Github users that we used to sample the 2500 Github users for the pilot survey. Thus the size of the target population in our primary survey was 88,021.

3) **Tag:** For each user out of the population (i.e., 88,021), we then searched for the posts in the Stack Overflow data dump of 2017 where s/he has contributed by either asking the question or answering the question. For each user, we created a user tag frequency table as follows: a) For each post contributed by the user, we collected the id of the thread of the post. b) For each thread, we collected the list of tags assigned to it. We put all those tags in the tag frequency table of the user. c) We computed the occurrence of each tag in the tag frequency table of the user d) We ranked the tags based on frequency, i.e., the tag with the highest occurrence in the table was put at the top.

4) **Programming Languages:** We assigned each user to one of the following nine programming languages:(1) Javascript, (2) Java, (3) C#, (4) Python, (5) C++, (6) Ruby, (7) Objective-C, (8) C, and (9) R. The nine programming languages are among the top 10 programming languages in Stack Overflow. We assigned a user to a language, if the language had the highest occurrence among the nine languages in the user frequency table of the user. If a user did not have any tag resembling any of the nine languages, we did not include him/her in the sample.

5) **Ranking of Users.** For each language, we ranked the users by reputation and activity date, i.e., first by reputation and then if two users had the same reputation, we put the one at the top between the two, who was active more recently.

6) **Create sample** For each language, we picked the top 100 users. For each user, we created a personalized email and sent him/her the survey invite.

In Table 2.4, we show the summary statistics of the primary survey population and the sampled users. The 88,021 users contributed to more than 3.2M threads. As of September 2017, Stack

Overflow hosts 14.6M threads and 7.8M users. Thus, the users in the population corresponds to 0.012% of all the users in Stack Overflow, but they contributed to 22.1% of all threads in Stack Overflow. With a 99% confidence interval, a statistically significant subset of this 88,021 users would contain a total of 661 users. Our sample contains a total of 900 users. The sampled users contributed to more than 23.9% of all threads contributed by the 88,021 users. All of the 900 users were active in Stack Overflow as early as 2017, even though most of them first created their account in Stack Overflow on or before 2010. Therefore, the sampled users are expected to be well aware of the benefits and problems of developer forums, such as, Stack Overflow. Moreover, each of the sampled users was highly regarded in the Stack Overflow community, if we take their reputation in the forum as a metric for that – the minimum reputation was 5471 and the maximum reputation was 627,850, with a median of 18,433.5. Therefore, we could expect that the answers from these users would not only ensure quality but also could be representative of the overall Stack Overflow developer community.

2.3.1.2 Participation Engagement

While targeting the right population is paramount for a survey, convincing the population to respond to the survey is a non-trivial task. As Kitchenham et al. [68] and Smith et al. [72] noted, it is necessary to consider the contextual nature of the daily activities of software developers while inviting them to the survey. While sending the survey invitations, we followed the suggestions of Smith et al. [72] who reported their experience on three surveys conducted at Microsoft. Specifically, we followed five suggestions, two related to *persuasion* (Liking, Authority and credibility) and three related to *social factors* (Social benefit, compensation value, and timing).

Liking. Nisbett and Wilson [73] explained the cognitive bias *halo effect* that people are *more likely to comply with a request from a person they have positive affect afterwards* [72]. Their advice to leverage the positive affection is to communicate with the people in the study by their name. For our survey, we created a personalized email for each user by (i) addressing the developer by his/her name in the email, and (ii) including the reputation of the developer in the email. In Figure 2.3, we show a screenshot of an email sent to one of the users ①.

Authority and credibility. Smith et al. [72] pointed out that the *"compliance rates rise with the authority and credibility of the persuader"*. To ensure that the developers considered our survey as authentic and credible, we highlighted the nature of the research in both the email and the survey front page. We also cited that the survey was governed by the formal Ethics Approval from McGill University and that the reporting to the survey would be anonymized.

Social Benefit. Edwards found that participants are more likely to respond to survey requests from universities [74]. The reason is that, participants may be more likely to respond to a survey if they know that their response would not be used for commercial benefits, rather it will be used to benefit the society (e.g., through research). We contacted the survey participants using the Academic emails of all the authors of the paper. We also mentioned in the email that the survey is conducted as a PhD research of Gias Uddin, and stressed on the fact that the survey was not used for any commercial purpose.

Compensation value. Smith et al. [72] observed in the surveys conducted at Microsoft that people will likely comply with a survey request if they owe the requestor a favor. Such reciprocity

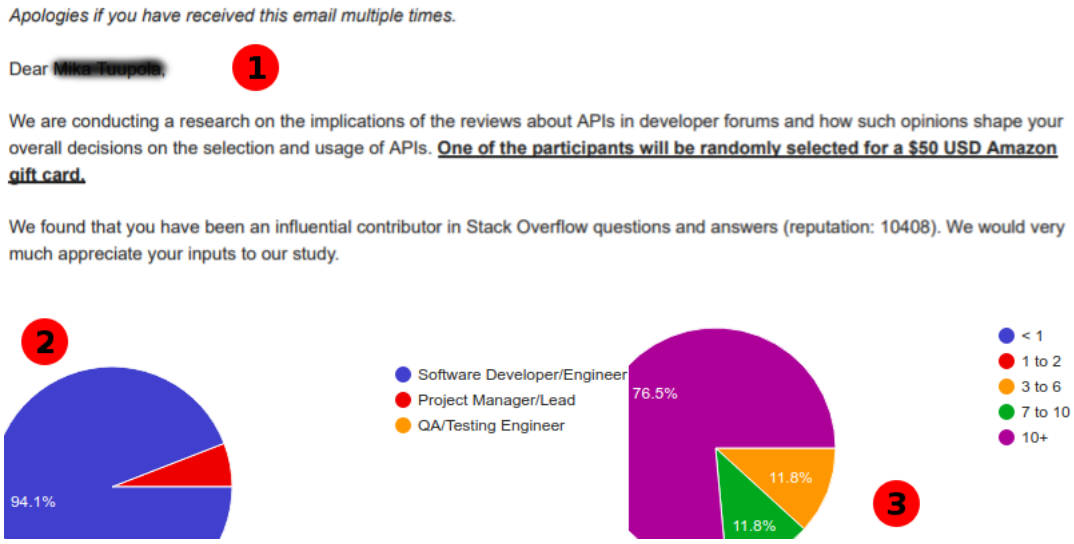

Figure 2.3: Snapshots of the primary survey email and responses.

Table 2.5: The agreement level between the coders in the open coding

	Q4	**Q5**	**Q6**	**Q7**	**Q11**	**Q12**	**Q13**	**Q14**
Percent	98.9	98.6	99.0	96.4	98.9	98.4	100	100
Cohen κ	0.90	0.76	0.80	0.66	0.80	0.75	1	1
Scott π	0.90	0.77	0.80	0.67	0.81	0.76	1	1
Krippen α	0.90	0.77	0.80	0.67	0.81	0.76	1	1
Quotes	112	104	112	121	150	122	101	108

can be induced by providing an incentive, such as a gift card. Previous research showed that this technique alone can double the participation rate [75]. In our primary survey, we offered a 50 USD Amazon Gift card to one of the randomly selected participants.

Timing. As Smith et al. [72] experienced, the time when a survey invitation is sent to the developers, can directly impact their likelihood of responding to the email. They advise to avoid sending the survey invitation emails during the following times: 1) Monday mornings (when developers just quickly want to skim through their emails) 2) Most likely out of office days (Mondays, Fridays, December month). We sent the survey invitations only on the three other weekdays (Tuesday-Thursday). We conducted the survey in August 2017.

Out of the 900 emails we sent to the sampled users, around 150 bounced back for various reasons (e.g., old or unreachable email). Around 30 users requested by email that they would not want to participate in the survey. Therefore, the final number of emails sent *successfully* was 720. We note that the email list compiled by Vasilescu et al. [69] is from 2013. Therefore, it may happen that not all of the 720 email recipients may have been using those email addresses any more. Previously, Treude et al. [67] reported a response rate of 7.8% for a survey conducted with the Github developers. Our response rate (15.8%) is more than the response rate of Treude et al. [67].

2.3.2 Survey Data Analysis

We analyzed the primary survey data using statistical and qualitative approaches, using the same approach we followed for the pilot survey. For the open-ended questions, we created a total of 947 quotes out of all the responses as follows: 1) For each response, we divided it into individual sentences 2) For each detected sentence, we further divided into individual clauses. We used semi-colon as a separator between two clauses. Each clause is considered as a quote.

For the open-ended questions, there were two coders. The author of this thesis (Gias Uddin) was the first coder. The second coder was selected as a senior software engineer working in the Industry in Ottawa, Canada. The second coder was not an author of this manuscript, nor was he involved in this project by any means.

The two coders together coded eight of the nine open-ended questions (Q4-Q7, Q11-Q14 in Table 2.3). The other open-ended question was Q24 ("explain why you don't value the opinion of other developers ..."). Only 10 participants responded to that question, resulting in 17 quotes. The first coder labeled all of those. In Table 2.5, we show the agreement level between the two coders for the eight open-ended questions. The last row in Table 2.5 shows the number of quotes for each of the questions. To compute the agreement between the coders, we used the online recal2 calculator [65]. The calculator reports the agreement using four measures: 1) Percent agreement, 2) Cohen κ [64], 3) Scott's Pi [76], and 4) Kripendorff's α [77] The Scott's pi is extended to more than two coders in Fleiss' κ. Unlike Cohen κ, in Scott's Pi the coders have the same distribution of responses. The Krippendorff's α is more sensitive to bias introduced by a coder, and is recommended over Cohen κ [78]. The agreement (Cohen κ) between the coders is above 0.75 for all questions except Q7. For Q7, the agreement is above the substantial level [4].

2.4 Primary Survey Results

During the open-coding process of the nine open-ended questions, 37 categories emerged (excluding 'irrelevant' and responses such as 'Not sure'). We labeled a quote as 'Not Sure' when the respondent in the quote mentioned that s/he was not sure of the specific answer or did not understand the question. For example, the following quotes were all labeled as 'Not sure': 1) *"Don't know"* 2) *"I am not sure"* 3) *"This is a hard problem."* 4) *"I'm not sure what this question is asking."* The 37 categories were observed a total of 1,019 times in the quotes. 46 quotes have more than one category. For example, the following quote has three categories (Documentation, API usage, and Trustworthiness): *"Many responses are only minimally informed, have made errors in their code samples, or directly contradict first-party documentation."* In Table 2.6, we show the distribution of the categories by the questions. We discuss the results of the open-ended and other questions along the two research questions.

2.4.1 Reasons for Seeking Opinions about APIs (RQ1.1)

We report the responses of the developers along the three sub-questions: 1) Sources for development and opinion needs, 2) Factors motivating developers to seek for opinion about APIs from

Table 2.6: The codes emerged during the open coding of the responses (#TC = Total codes)

	Q4		Q5		Q6		Q7		Q11		Q12		Q13		Q14		Q24		
	#C	#R	#C	#R	#C	#R	#C	#R	#C	#R	#C	#R	#C	#R	#C	#R	#C	#R	#TC
Documentation	11	11	1	1	4	4	54	49	47	43	11	9	6	5	1	1			135
Search			94	79	15	11					12	12	1	1					122
Usability	1	1			2	2			87	56	2	2	2	2					94
Community	11	9			9	7	12	10	31	22	6	3	2	2			2	1	71
Trustworthiness	13	11	1		21	20	8	5	2	1	2	2			21	17	4	3	68
Expertise	31	31	2	2	3	3	13	12	3	3	3	3	4	4			2	2	59
API Selection	13	12			6	6	1	1					30	25	6	6			56
API Usage	17	16			8	8	2	2	10	9	7	6	7	7	2	2			53
Situational Relevance					26	22	12	9	8	7	3	3	3	3			2	2	52
Reputation			1	1	1	1	35	33	5	4									42
Performance	8	8							30	23			3	3					41
Recency			1	1	13	13	6	6			1	1	3	3	3	3			27
Productivity	9	8			1	1							12	11					22
Compatibility					4	4			18	14									22
Aggregation Portal											18	15	1	1					19
Missing Nuances															18	12			18
Legal									15	14									15
Sentiment Statistics			4	4							8	5							12
General Insight					1	1			4	4	1	1	2	2	4	4			12
API Maturity									6	5	3	2							9
API Adoption							2	1	4	4	3	3							9
Portability									5	5			4	4					9
Opinion Categorization											8	8	1	1					9
Opinion Reasoning													1	1	7	6			8
Contrastive Summary											7	7							7
Info Overload					3	3					1	1			2	2			6
Similarity			3	3							2	2							5
Security									2	2	1	1							3
Learning Barrier															3	3			3
Bug									1	1			1	1					2
Extractive Summary											2	2							2
API Improvement													1	1	1	1			2
FAQ											1	1							1
Problem Summaries											1	1							1
Machine Learning											1	1							1
Wrong Info															1	1			1
Lack of Info					1	1													1
Not Sure											10	9	23	22	26	25	1	1	59
Irrelevant	7	4	5	5	10	8	7	4	11	7	25	18	8	7	14	13	6	3	87
Total	121	83	112	83	128	83	152	83	289	83	139	83	115	83	109	83	17	10	1019
# Categories	9		8		17		10		17		21		19		14		5		37

Notes: #C = the number of code, #R = the number of respondents, Q4 = Reasons for referring to opinions of other developers about APIs? Q5 = How do you seek information about APIs in a developer forum?, Q6 = Challenges when seeking for opinions about an API? Q7 = Factors in a forum post to determine the

developer forums, and 3) Challenges developers face while seeking for opinions about APIs from developer forums.

2.4.1.1 Sources for Opinions about APIs (RQ1.1.a)

We asked the developers three questions (Q2, 16, 17 in Table 2.3) about the sources from where they seek opinion and other information about APIs, and how they navigate such information in the developer forums.

Q2 *We asked them to give a list of the top developer forums they visited in the last two years.* The respondents reported 40 different developer forums. Stack Overflow and its companion sites in Stack Exchange were reported the most (94 times), at least once by each respondent. The other major forums listed were (in order of frequency): 1) Github issue tracker (9), 2) Google developer forum (8), 3) Apple developer forum (6), 4) Twitter/Quora/Reddit/Slack (3) The other mentions were blog lists, internal mailing lists, XDA, Android forums, and so on.

Q16 *We further asked them which developer forums they visit to seek for opinions about APIs.* We gave them six options to choose from: 1) Developer forums, e.g., Stack Overflow (Picked by all 83 developers who mentioned that they valued the opinions of other developers and that they visit developer forums). 2) Co-workers (63 developers), 3) IRC chats (22 developers), 4) Internal mailing lists (24 developers), 5) None (0 developers), 6) Others. For 'Others', we gave them a text box to write the name of the forums. Among the other sources, developers picked a variety of online resources, such as, Google search engine, Hacker news, blogs, slacks (for meetups), github, twitter, and so on.

Q17 *We asked the developers about their frequency of visiting the online developer forums (e.g., Stack Overflow) to get information about APIs (Q17).* There were six options to choose from: 1) Every day (picked by 36.1% of the developers), 2) Two/three times a week (32.5%), 3) Once a week (14.5%), 4) Once a month (16.9%), 5) Once a year (0%), 6) Never (0%). Therefore, most of the developers (83.1%) reported that they visit developer forums at least once a week, with the majority (36.1%) visiting the forums every day. Each developer reported to visit the developer forums to seek opinions about APIs at least once a month.

The Google and Apple developer forums were present in the list of forums that the developers visited in the last two years, but they were absent in the list of forums where developers visit to seek for opinions. Stack Overflow was picked in both questions as the most visited site by the developers both as a general purpose forum to find information about APIs and as a forum to seek for opinions about APIs. The presence of Twitter in both lists shows that developers view Twitter as an interesting place to visit with regards to their development needs as well as to seek for opinions about APIs, showing the necessity of recent efforts to harness Twitter for development needs [79]

Stack Overflow was considered as the major source to seek information and opinions about APIs by the developers. Besides Stack Overflow, developers also reported diverse informal documentation resources, such as, blogs, Twitter, and so on. The Github issue tracking system is also considered as a source for such information.

2.4.1.2 Factors motivating opinion seeking (RQ1.1.b)

We asked the developers two questions (Q4,18 in Table 2.3) about the factors that motivate them to seek for opinions about APIs from developer forums. For each category as we report below (e.g., EXPERTISE^{31,31} below), the superscript (n, m) is interpreted as follows: n for the number of quotes found in the responses of the questions, and m is the number of total distinct participants provided those responses. A similar format was previously used by Treude et al. [67] (except m, i.e., number of respondents).

Q4 *We asked the developers to write about the reasons for them to refer to opinions about APIs from other developers.* The developers seek for opinions for a variety of reasons: **(1)** To gain overall EXPERTISE^{31,31} about an API by learning from experience of others. The developers consider the opinions from other expert developers as indications of real-word experience and hope that such experience can lead them to the right direction. **(2)** To learn about the USAGE^{17,16} about an API by analyzing the opinion of other developers posted in response to the API usage scenarios in the posts. Developers consider that certain edge cases of an API usage can only learned by looking at the opinions of other developers. They expect to learn about the potential issues about an API feature from the opinions, before they start using an API. **(3)** To be able to SELECT^{13,12} an API amidst multiple choices for a given development. The developers think that the quality of APIs can vary *considerably*. Moreover, not all APIs may suit the needs at hand. Therefore, they expect to see the pros and cons of the APIs before making a selection. **(4)** To improve PRODUCTIVITY^{9,8} by saving time during a decision making process.

EXPERTISE^{31,31} and PRODUCTIVITY^{9,8} • *"Getting experience of others can save a lot of time if you end up using a better API or end up skipping a bad one."* • *"Because there is always a trick and even best practice which can only be provided by other developers."*

API USAGE^{17,16} • *"Because I want to know of real use, rather than just documentation."* • *"Knowing issues with APIs before using them in a project is very helpful."* • *"Practical experience beats a sales pitch"* • *"It's especially helpful to learn about problems others' faced that might only be evident after consider time is spent experimenting with or using the API in production."*

API SELECTION^{13,12} • *"Often they have evaluated alternatives and present comparisons."* • *"If I don't know an API and I need to pick one, getting the opinion of a few others helps make the decision."*

The developers cited different API aspects about which they look for opinions, such as, 1) the DOCUMENTATION^{11,11} support, e.g., when the official documentation is not sufficient enough, 2) the COMMUNITY^{11,9} engagement in the forum posts and mailing lists, 3) the PERFORMANCE^{8,8} of the

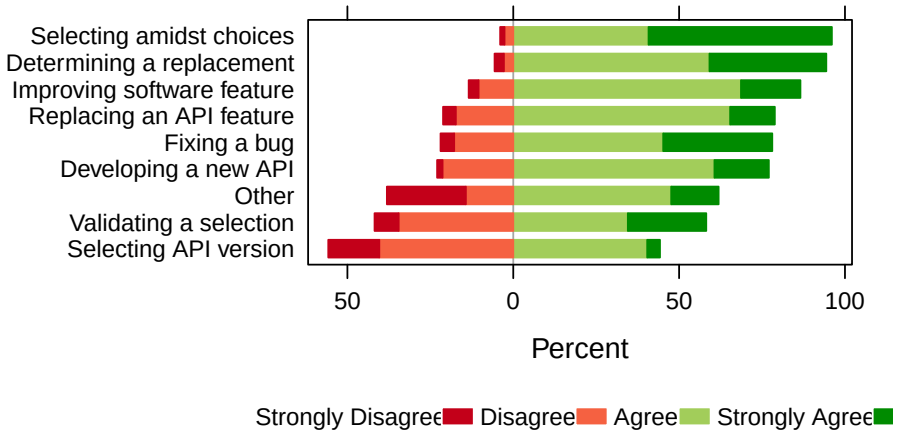

Figure 2.4: Developer motivation for seeking opinions about APIs.

API in terms of speed and scalability, etc., and 4) the USABILITY[1,1] and design principles of the API. To make a selection or to meet the specific development needs at hand, the developers leverage the knowledge about the API aspects expressed in the opinions about the competing APIs.

DOCUMENTATION[11,11] • *"APIs may include quirks which are not fully documented"* • *"possibility to get an answer to a specific question (which may not be explained in other sources like the API's documentation)"*

COMMUNITY[11,9] • *"Stack Overflow is reliable resource, as it has a really wide audience."* • *"for instance, Firebase team members are active on Stack Overflow."*

PERFORMANCE[8,8] • *"API stability, API safety"* • *"It allows me to see common gotchas and workarounds, so I can estimate the quality of the API based on those opinions, not just my own."*

The developers also leverage the opinions to TRUST[13,11] and to *validate* the claims about a specific API usage or feature (e.g., if a posted code example is good/safe to use).

TRUSTWORTHINESS[13,11] • *"They represent hands-on information from fellow devs, free of marketing and business."* • *"Other people are saying that this should work so maybe there is something in it."*

Q18 *We asked the developers about the specific development needs that may motivate them to seek for opinions about APIs.* We solicited opinions using a five-point Likert-scale question. The options were picked from similar questions used in our pilot survey. The analysis of the Likert-scale question (Figure 2.4) shows that developers find selection-related factors (i.e., determining replacement of an API and selection of an API among choices) to be the most influential for seeking for opinions (88% and 80.7% of all respondents agreed/strongly agreed, respectively). Developers

also seek opinions when they need to improve a software feature, for which they hope to find a well-reviewed API (69.9% agreement). 68.7% of the developers agree that they turn to online help when they need to fix a bug. 56.6% of the respondents seek help during the development of a new API, such as, addressing the shortcomings of the existing API. Developers are less enthusiastic to rely on opinions for validating their choice of an API to others (38.6%), or replacing one version of an API with another version (27.7%). Only 15.7% of the respondents mentioned that they seek for opinions for reasons not covered by the above options. Note that while the 'Neutral' responses are not shown in Figure 2.4, we include the neutral counts in our calculation of percent agreements above. Our charts to show the results of Likert-scale questions (e.g., Figure 2.4) follow similar formats as adopted in previous research [80, 81].

> Developers seek for opinions about APIs in developer forums to support diverse development needs, such as building *expertise* about API aspects, learning about nuances about specific API usage, making a selection of an API amidst choices, and so on.

2.4.1.3 Challenges in opinion seeking (RQ1.1.c)

We asked developers two questions (Q5, 6 in Table 2.3) to determine the way developers seek information and opinions about APIs and the challenges they face while seeking and analyzing the opinions.

Q5 *We asked the developers to write about how they seek information about APIs in a developer forums and how they navigate through multiple forums posts to form an informed insight about an API.* More than 83% of the respondents include some form of SEARCHING[94,79] using a general purpose search engine (e.g., Google) or using the provided search capabilities in the forums.

SEARCH[94,79] • *"Google and patience"* • *"I use Google and trust Stack Overflow's mechanisms for getting the best posts into the top search results"*

Because such search can still return many results, the developers employ different mechanisms to navigate through the results and posts and to find the information that they can consider as the *right* result. Such mechanisms include, the ranking (e.g., top hits) of the results, the manual SIMILARITY[3,3] assessment of the search results by focusing on the *recency* of the opinion, the analysis of the presence of SENTIMENTS[4,4] in the post about the API, and so on.

SIMILARITY[3,3] • *"Relevance, timestamp and titles"* • *"Search for a specific problem, targeting specific tags"*

SENTIMENT PRESENCE[4,4] • *"I usually google keyword and then look for the positive and negative response"*

Q6 *We asked developers to write about the challenges they face while seeking for opinions about APIs from developer forums.* The major challenge developers reported was to get the SITUATIONAL RELEVANCY[26,22] of the opinions within their usage needs. Such difficulties can stem from diverse

sources, such as, finding the right question for their problem, or realizing that the post may have only the fraction of the answers that the developer was looking for, and so on. Another relevant category was RECENCY[13,13] of the provided opinion, as developers tend to struggle whether the provided opinion is still valid within the features offered by an API. The assessment of the TRUSTWORTHINESS[21,20] of the provided opinion as well as the opinion provider were considered as challenging due to various reasons, such as, lack of insight into the expertise level of the opinion provider, bias of opinion provider, and so on.

SITUATIONAL RELEVANCY[26,22] • *"Evaluating the relevancy of the opinion to the problem I'm facing."* • *"Asking the right question, or using the right search terms."* • *"It's hard to know what issues people have with APIs so it's difficult to come up with something to search for."*

TRUSTWORTHINESS[21,20] • *"bias or vendor plugs"* • *"Sometimes it's hard to determine a developer's experience with certain technologies, and thus it may be hard to judge the API based on that dev's opinion alone"* • *"Sometimes the opinions aren't sufficiently detailed to understand why the user has a positive or negative opinion."*

RECENCY[13,13] • *"Information may be outdated, applying to years-old API versions"*

The INFORMATION OVERLOAD[3,3] in the forum posts while navigating through the forum posts was mentioned as a challenge. The absence of answers to questions, as well as the presence of low quality questions and answers, and the LACK OF INFORMATION[1,1] in the answers were frustrating to the developers during the analysis of the opinions. The developers mention about their extensive reliance on the SEARCH[15,11] features offered by both Stack Overflow and general purpose search engines to find the right information. Finding the right keywords was a challenge during the searching. Developers wished for better search support to easily find duplicate and unanswered questions.

INFORMATION OVERLOAD[3,3]**.** • *"They provide many different opinions, and I have to aggregate all information and make my own decision "* • *"Too many opinions sometimes"*

SEARCH[15,11]**.** • *"guessing appropriate search terms"* • *"I need to filter out noob opinions."* • *"You have to frame the question to game the forum, to make it easy to answer"* • *"If I'm inexperienced or lacking information in a particular area, I may not be using the right terminology to hit the answers I'm looking for."* • *"It would be awesome if Google didn't show me unanswered questions from Stack Overflow!"*

Developers expressed their difficulties during their SELECTION[6,6] of an API by leveraging the opinions. The lack of opinions and information for newer APIs is a challenge during the selection of an API. Developers can have specific requirements (e.g., performance) for their USAGE[8,8] of an API, and finding the opinion corresponding to the requirement can be non-trivial (e.g., is the feature offered by this API scalable?).

SELECTION[6,6] • *"Find the limitation of an API, and which ones are simpler."* • *"My biggest challenge is comparing different API's and technologies, because the 'truth' tends to get subjective or very dependent on your use case."* • *"With newer APIs, there's often very little information."*

USAGE[8,8] • *"As an example, I often has stricter performance requirements, like I need to draw 50000 points so opinions on performance with 50 points are irrelevant to me. "* • *"That's why I usually try something like "API XYZ issues" or "API XYZ problems" into google and see what comes up. "*

The necessity to analyze opinions based on specific API aspects was highlighted by the developers, e.g., link to the DOCUMENTATION[4,4] support in the opinions, COMPATIBILITY[4,4] of an API feature in different versions, learning about the USABILITY[2,2] of the API, and the activity and engagement of the supporting COMMUNITY[9,7]. The lack of a proper mechanism to support such analysis within the current search engine or developer forums make such analysis difficult for the developers.

DOCUMENTATION[4,4] • *"Ensuring that the information is up-to-date and accurate requires going back-and-forth between forums and software project's official docs to ensure accuracy (in this case official docs would be lacking, hence using forums is the first place)"*

COMPATIBILITY[4,4] • *"filtering for a particular (current) API version is difficult."* • *"Different versions of an API - what might have worked once may no longer work now"*

COMMUNITY[9,7] • *"That there are too few users of a new and not yet recognized API, but it could also be a niche API that no one has ever heard of."* • *"But looking at several questions in a particular tag will help give a feeling for the library and sometimes its community too."*

Finally, getting an instant insight into the EXPERTISE[3,3] of the opinion provider is considered as important by the developers during their analysis of the opinions. The developers consider that getting such insight can be challenging.

EXPERTISE[3,3] • *"Need to figure out if the person knows what they are talking about and is in a situation comparable to ours."*

Majority of the developers leverage search engines to look for opinions. They manually analyze the presence of sentiments in the posts to pick the important information. The developers face challenges in such exploration for a variety of reasons, such as the absence of a specialized portal to search for opinions, the difficulty in associating such opinions within the contexts of their development needs, and the lack of enough evidence to validate the trustworthiness of a given claim in an opinion, and so on.

2.4.2 Needs for API Review Quality Assessment (RQ1.2)

We asked the developers three questions (two open-ended) to understand their needs to assess the quality of the provided opinions about APIs (Q3,7,24 in Table 2.3).

Q3 *We asked the developers whether they value the opinion about APIs of the other developers in the developer forums.* 89.2% of the respondents mentioned that they value the opinion of other

developers in the online developer forums, such as, Stack Overflow. 10.8% reported that they do not value such opinions.

Q24 *We asked the 10.8% participants who do not value the opinion of other developers to provide us reasons of why do not value such opinions.* The developers cited their concern about TRUSTWORTHINESS[4,3] as the main reason, followed by the lack of SITUATIONAL RELEVANCY[2,2] of the opinion to suit specific development needs. The concerns related to the trustworthiness of the opinion can stem from diverse factors, such as, the *credibility* and the *experience* of the poster, as well the inherent bias someone may possess towards a specific computing platform (e.g., Windows vs Linux developers).

TRUSTWORTHINESS[4,3] • *"They are usually biased and sometimes blatant advertisements."*

SITUATIONAL RELEVANCY[2,2] • *"I do to a degree but other factors will ultimately decide in selection"*

We note that both of these two categories were mentioned also by the developers who value opinions of other developers. However, they mentioned that by looking for diverse opinions about an entity, they can form a better insight into the trustworthiness of the provided claim and code example.

In the rest of paper, we report the responses from the 89.2% developers who reported that they value the opinion of the other developers.

Q7 *The open-ended question asked developers to write about the factors that determine the quality of the provided opinion.* The major category is associated with the DOCUMENTATION[54,49] quality. Forum posts are considered as informal documentation. The developers expected the provided opinions to be considered as an alternative source of API documentation.

DOCUMENTATION[54,49]

Clarity of the provided opinion with proper writing style and enough details with links to support each claim are considered as important when opinions can be considered as of good quality.

- **Clarity.** • *"I mean when example is clean + has some documentations links"*
- **Completeness.**• *"Exhaustiveness of the answer and comparison to others"*
- **Writing Quality.**• *"Well written (grammar, formatting etc.)"* • *"You can usually tell an insightful post very quickly from an ill-informed one by the quality of the language and the logic."* • *"Spelling, punctuation and grammar are helpful heuristics."*
- **Facts.** Presentation of the information using supporting facts, e.g., *accurate technical terms*, using *screenshots or similar suggestions*, or *examples from real world applications*. • *"Cross-referencing other answers, comments on the post, alternate answers to the same question, testing, and checking against the documentation. "*
- **Brevity.** • *"brief and to the point, copy paste solution, knowledge of the poster, reputation points of the poster "*
- **Context.** • *"Amount of additional context provided in answers, identified usage patterns and recognized API intent"*

- **Code with Reaction.** Code examples about an API accompanied with positive reactions are considered as good sources of documentation for an API. • *"Code samples, references to ISOs and other standards, clear writing that demonstrates a grasp of the subjects at hand "* • *"Community rating, thoroughness, link to outside support, helpfulness. "*

The other major factors developers reported to analyze the quality of the provided opinions are (1) REPUTATION[35,33] of the opinion provider and upvote to the forum post where the opinion is found, (2) The perceived EXPERTISE[13,12] of the opinion provider within the context of the provided opinion, (3) The TRUSTWORTHINESS[8,5] of the opinion provider, (4) The SITUATIONAL RELEVANCE[12,9] of the provided opinion within a given development needs, and (5) The RECENCY[6,6] of the provided opinion.

REPUTATION[35,33] • *"If the poster has a high Stack Overflow score and a couple of users comment positively about it, that weighs pretty heavily on my decision."*

EXPERTISE[13,12] • *"It's easy to read a few lines of technical commentary and identify immediately whether the writer is a precise thinker who has an opinion worth reading."*

SITUATIONAL RELEVANCE[12,9] • *"I typically do not simply look at accepted answers."* • *"Often questions seem to be about an API but eventually it turns out that the problem was something else and not the API at all."*

TRUSTWORTHINESS[8,5] The opinion must be free from bias, demonstrating *apparent fairness and weighing pros/cons*. • *"Corroboration across multiple sources of information: rather than taking a single post in isolation, check across several to get a broader impression. "*

RECENCY[6,6] • *"Recency: posts more than a few years old are likely to be out of date and possibly counterproductive."*

Developer analyze the quality of the opinions about APIs in the forum posts by considering the opinions as a source of *API documentation*. They employ a number of *metrics* to judge the quality of the provided opinions, e.g., the clarity and completeness of the provided opinion, the presence code examples, the presence of detailed links supporting the claims, and so on.

2.4.3 Tool Support for API Review Analysis (RQ2.1)

We asked four questions (one open-ended) to understand whether and how developers prefer tool support to analyze the opinions about APIs in the developer forums (Q8-10, 19 in Table 2.3).

Q8 *We asked developers whether they currently rely on tools to analyze opinions about APIs from developer forums.* 13.3% developers responded with a 'Yes' and the rest (86.7%) responded with a 'No'.

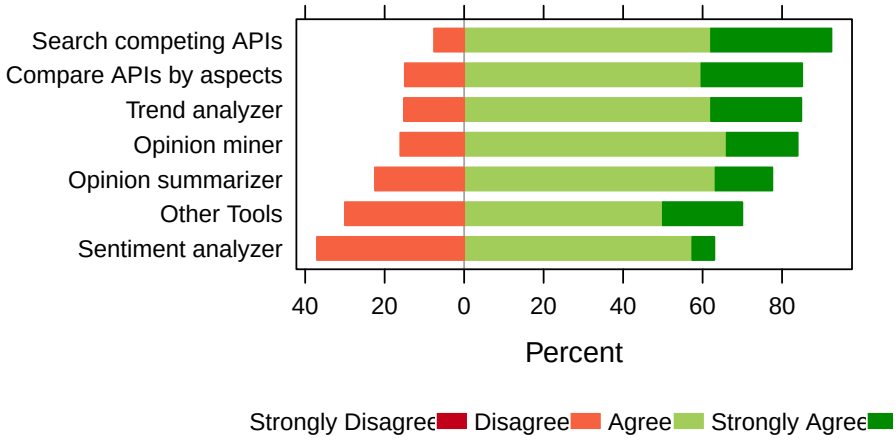

Figure 2.5: Tool support for analyzing API opinions.

Q9 *We further probed the developers who responded with a 'No'. We asked them whether they feel the necessity of a tool to help them to analyze those opinions.* The majority (62.5%) of the developers were unsure ('I don't know'). 9.7% responded with a 'Yes' and 27.8% with a 'No'.

Q10 *We further probed the developers who responded with a 'Yes'. We asked them to write the name of the tool they currently use.* The developers cited the following tools: 1) Google search, 2) Stack Overflow votes, 3) Stack Overflow mobile app, and 4) Github pulse.

Q19 The last question was a multiple choice question, with each choice referring to one specific tool. The choice of the tools was inspired by research on sentiment analysis in other domains [24]. To ensure that the participants understood what we meant by each tool, we provided a one-line description to each choice. The choices were as follows (in the order we placed those in the survey):

1) **Opinion miner**: for an API name, it provides only the positive and negative opinions collected from the forums.

2) **Sentiment analyzer**: it automatically highlights the positive and negative opinions about an API in the forum posts.

3) **Opinion summarizer**: for an API name, it provides only a summarized version of the opinions from the forums.

4) **API comparator**: it compares two APIs based on opinions.

5) **Trend analyzer**: it shows sentiment trends towards an API.

6) **Competing APIs**: it finds APIs co-mentioned positively or negatively along an API of interest and compares those.

The respondents' first choice was the 'Competing APIs' tool (6), followed by a 'Trend Analyzer' to visualize trends of opinions about an API (5), an 'Opinion Miner' (1) and a 'Summarizer' (3)

(see Figure 2.5). The sentiment analyzer (2) is the least desired tool, i.e., developers wanted tools that do not simply show sentiments, but also show context of the provided opinions. Note that an opinion search and summarization engine in other domain (e.g, camera reviews) not only show the mined opinions, but also offer insights by summarizing and showing trends. The engines facilitate the comparison among the competing entities through different aspects of the entities. Such aspects are also used to show the summarized viewpoints about the entities (ref Figure 1.3). Therefore, it was encouraging to see that developers largely agreed with the diverse summarization needs of API reviews and usage information, which corroborate to the similar summarization needs in other domains.

> To analyze opinions about APIs in the forums, developers leverage traditional search features in the absence of a specialized tool. The developers agree that a variety of opinion analysis tools can be useful in their exploration of opinions about APIs, such as, opinion miner and summarizer, a trend analyzer, and an API comparator engine that can leverage those opinions to facilitate the comparison between APIs.

2.4.4 Needs for API Review Summarization (RQ2.2)

As we observed from the responses of the developers in our survey in the previous section, in a potential opinion summarization engine about APIs, we need to show summarized viewpoints about an API, and support the comparison of APIs based on those viewpoints. To properly understand how and whether such summarization can indeed help developers, we probed the participants in our survey with a number of question. Specifically, we present the analysis of their responses along three themes: 1) Factors motivating the needs for opinion summarization about APIs (RQ2.2a), 2) Developers preference of summarized viewpoints to support their development needs (RQ2.2.b), and 3) The perceived impact of different summarization types used in other domains to the domain of API reviews (RQ2.2.c).

2.4.4.1 Factors motivating summarization needs (RQ2.2.a)

We asked the developers three questions (Q20-22 in Table 2.3). We explored the relative benefits and disadvantages of opinion summarization by asking developers whether opinions do indeed need to be summarized, and how the developers envision such summaries may impact their decision making processes related to software development involving APIs.

Q20 *We asked developers how often they feel overwhelmed due to the abundance of opinions about APIs in the developer forums.* 3.6% of the developers mentioned that they are 'Always' overwhelmed, 48.2% are 'Sometimes' overwhelmed and 37.3% are 'Rarely' overwhelmed. Only 10.8% developers mentioned that they are 'Never' overwhelmed by the abundance of opinions about APIs in the developer forums.

Q21 *We then asked them whether a summarization of those opinions would help them make better decisions about APIs?* 38.6% responded with a 'Yes' and 13.3% responded with a 'No'. 48.2%

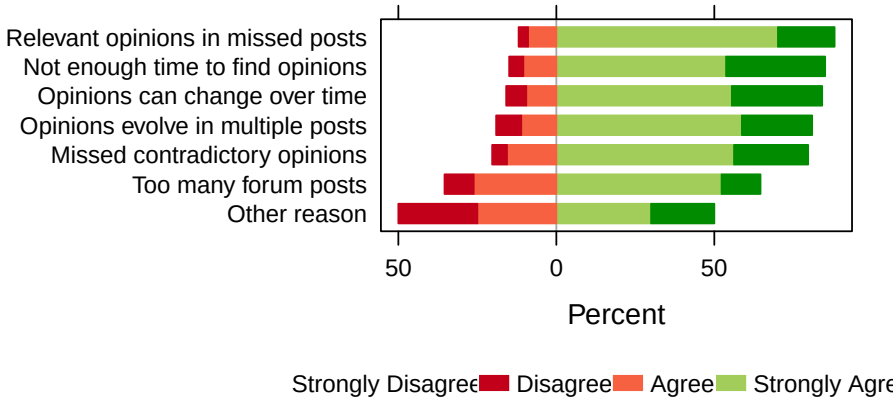

Figure 2.6: Developer needs for opinion summarization about APIs.

were unsure ('I don't know'). The reason for the majority to be unsure is probably due to the fact that developers were not aware of any existing summarization tools for API reviews.

Q22 *Finally, We examined the needs to opinion summarization using a five-point Likert-scale question.* There were seven options: 1) Too many forum posts with opinions, 2) Opinions can evolve over time in different posts, 3) Opinions can change over time, 4) The interesting opinion may not be in the posts that you have looked in, 5) Contradictory opinions about an API may be missed, 6) Not enough time to look at all opinions, and 7) Other reason. The options were selected based on our empirical analysis of API reviews in forum posts. The same options were also used in the pilot survey and we observed agreement from the developers across the options. The analysis of the Likert-scale question (Figure 2.6) shows that nearly all developers agree that opinion summarization is much needed for several reasons. Since opinions change over time or across different posts, developers want to be able to track changes in API opinions and follow how APIs evolve. The vast majority of the respondents also think that an interesting opinion about an API might be expressed in a post that they have missed or have not looked at. The need for opinion summarization is also motivated by the myriad of opinions expressed in the various developer forums. Developers believed that the lack of time to search for all possible opinions and the possibility of missing an important opinion are strong incentives for having opinions organized in a better way.

> The majority of the developers feel overwhelmed due to the abundance of opinions about APIs in the developer forums. The developers were in agreement that such difficulty arise due to a variety of reasons, such as, opinions about APIs being scattered across multiple unrelated posts, the huge amount of time it normally takes to form an informed insights from such disparate resources, the risk of potentially missing the relevant insight due to the absence of a specialized opinion processor, and so on.

2.4.4.2 Summarization preferences (RQ2.2.b)

We asked three questions (two open-ended) to understand the relative benefits and problems developers may encounter during their usage of API review summaries from developer forums (Q13,14, and 23 in Table 2.3).

Q13 *We asked developers to write about the areas that can be positively impacted by the summarization of opinions about APIs from developer forums.* The majority of the developers considered that API SELECTION[30,25] is an area that can be benefited from the summarization of API reviews. They also considered that API review summarization can improve the PRODUCTIVITY[12,11] of the developers by offering quick but informed insights about APIs.

API SELECTION[30,25] • *"It would make the initial review and whittling down of candidates quicker and easier."* • *"Will make picking/choosing between multiple APIs which solve the same problem much easier."* • *"There's a real problem that Android developers face among open source libraries: which library should I choose?"* • *"Easier to compare different APIs serving the same purpose"*

PRODUCTIVITY[12,11] • *"speed up time taken to decide on an API"* • *"Improve the quality of software by providing valuable information, and the speed of development"* • *"It would save a lot of time."*

The developers expected the summaries to assist in their API USAGE[7,7], such as, by showing reactions from other developers for a given code example that can offer SITUATIONALLY RELEVANT[3,3] insights about the code example (e.g., if the code example does indeed work and solve the need as claimed). The developers consider that they can use a summary as a form of documentation to explore the relative strengths and weaknesses of an API for a given development need. The developers do not expect the official documentation of an API to have such insights, because official documentation can be often incomplete [19]. Carroll et al. [20] advocated the needs for *minimal API documentation*, where API usage scenarios should be shown in a task-based view. In a way, our developers in the survey expect the API usage scenarios combined with the opinions posted in the forum posts as effective ingredients to their completion of the task in hand. Hence, they expect that the summaries of opinions about API can help them with the efficient usage of an API in their development tasks.

API USAGE[7,7] • *"You can see how the code really works, as opposed to how the documentation thinks it works"* • *"If summarization is beneficial to understanding API, then any problem even remotely related to that use stands to achieve a network benefit."*

As we noted before in this section, developers leverage the opinions about diverse API aspects to compare APIs during their selection of an API amidst choices. In the absence of any such automatic categorization available to automatically show how an API operates based on a given aspect (e.g., performance), the developers had expectations that an opinion summarizer would help them with such information. For example, the developers considered that opinion summaries can also improve the analysis about different API aspects, such as, USABILITY[2,2], COMMUNITY[2,2], PERFORMANCE[3,3], DOCUMENTATION[6,5], and so on.

DOCUMENTATION[6,5] • *"Any area with hard and long documentation. "* • *"Disambiguation, clarification of API documentation"*

PERFORMANCE[3,3] • *"API developers might decide to improve the attributes reviewers care about . . . "*

Finally, developers envisioned that opinion summaries can improve the SEARCH[6,6] for APIs and help them find better EXPERTISE[4,4] to learn and use APIs.

Q14 *We next asked developers to write about the areas that can be negatively impacted by the summarization of opinions about APIs from developer forums.* The purpose was to be aware of any potential drawback that can arise due to the use of API opinion summaries. Developers considered that opinion summaries may MISS NUANCES[18,12] in the API behavior that are subtle, may not be as frequent, but that can still be important to know. The REASONING[7,6] about opinions can also suffer because of that.

MISSING NUANCES[18,12] • *"The summary may miss key design choices or the meta-information . . . "* • *"I'd say probably 10-20% of the time, I glean something from a relatively obscure comment on a post that proves to be the key to understanding something."*

REASONING[7,6] • *"Probably will be hard to follow the chain of conclusion by the poster"*

While developers considered the selection of an API as an area that can be positively affected by opinion summaries in general, they also raised the concern that summaries may negatively impact API SELECTION[6,6] when more subtle API aspects are not as widely discussed as other aspects, and when new APIs with less number of opinions are not ranked higher in the summaries.

API BARRIER[3,3] • *"Simply a lack of traffic about it."* • *"using API less/known or used but more appropriate in the context"*

API SELECTION[6,6] • *"Popularity and fashion trends may mask more objective quality criteria "* • *"Just relying on the summarization for deciding for or against an API will probably not be enough and may lead to decisions that are less than optimal. "*

Q23 *We next asked developers about how opinion summaries can support their tasks and decision making process.* There were nine options, the same options we used to solicit the needs for opinions about APIs in Figure 2.4. The purpose of an opinion summarizer should be to facilitate the easier and efficient consumption of opinions. Therefore, the end goal remains the same in both cases, and so the options. In Figure 2.7, we present the responses of the developers against the options. More than 70% of the developers believe (agree + strongly agree) that opinion summarization can help them with two decisions: 1) determining a replacement of an API and 2) selecting the right API among choices. Developers agree that summaries can also help them improve a software feature, replace an API feature, and validate the choice of API (45.8%, 42.2%, 48.2%, and 48.2%, respectively). Fixing a bug, while receiving 38.6% of the agreement, might not be well supported by summaries since this task requires certain detailed information about an API that may not be present in the

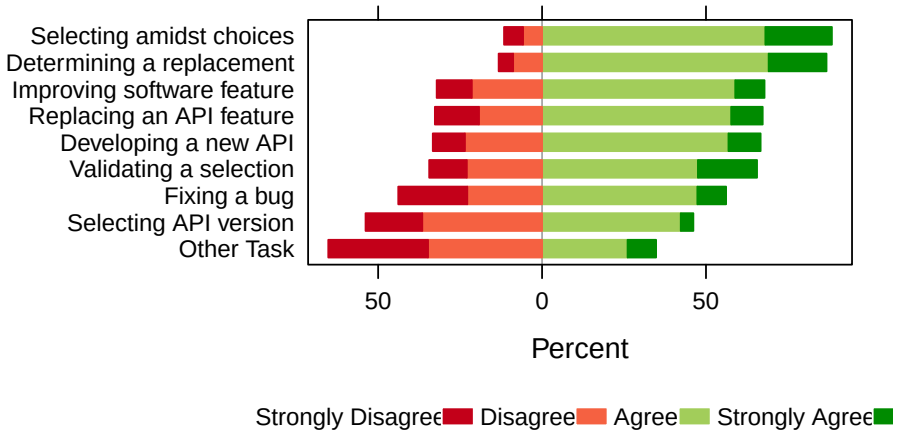

Figure 2.7: Impact of opinion summaries on decision making about APIs.

opinions (e.g., its version, the code itself, the feature). The developers mostly disagreed with the use of opinions to select an API version (28.9% agreement only).

> The summarization of opinions about APIs can be effective to support diverse development needs, such as, the selection of an API amidst choices, the learning of specific API usage cases, and so on. Developers expect a gain in productivity in supporting such needs by the opinion summarizer. However, developers expressed their cautions with a potential opinion summarizer for APIs, when for example, subtle nuances or use cases of an API can be missed in the summary. Another cause of concern for them with a summarizer that it can make it difficult for a new API to enter into an ecosystem that may already be populated by many APIs. For example, the developers will keep using the existing APIs, because they are possibly reviewed more than the new APIs.

2.4.4.3 Summarization types (RQ2.2.c)

We asked developers three questions (Q11,12,15 in Table 2.3) to learn about their preferences on how they would like to have opinion summaries about APIs to be produced.

Q11 *We asked developers to write about the different factors in an API that play a role during their decision to choose the API.* The purpose was to learn what specific API aspects developers consider as important. Intuitively, developers would like to see opinions about those aspects in the forum posts to make a decision on the API. The developers mentioned about the following API aspects as factors contributing to their decisions to choose an API:

USABILITY[87,56] • *"Is it easy to use?"* • *"Simplicity, compliance with standards"* • *"Productivity: how much of my time will it take, and will I enjoy it?"*

DOCUMENTATION[47,43] • *"easy to use, common format, clean, good documentation"* • *"good doc-*

umentation, has to be intuitive (consistent naming)" • *"First of all good documentation, secondly good SDKs."*

COMMUNITY[31,22] • *"Active community."* • *"Strong Stack Overflow community (to show that the API is relatively mature). "* • *"Good documentation and wide adoption"* • *"documentation) and track record of its maintenance."*

PERFORMANCE[30,23] • *"stability, safety (e.g., do how well the modeled internals are protected on faulty use/misuse"* • *"In general, I look for a balance of power and complexity, whether the technology has good documentation, how mature it is, how widely adopted, etc."*

COMPATIBILITY[18,14] • *"compatibility (will it work for project I'm working on)"* • *"Implemented in the language I need it for"*

LEGAL[15,14] • *"openness"* • *"I also think it's important that the API is open source and maintained."*

PORTABILITY[5,5] • *"Thread safety, sync/async, cross-language and cross-os support and so on. "* • *"used by multiple API consumers, fit to my problem, ease of use, stability, multiple implementations preferred"*

SECURITY[2,2] • *"Are security issues addressed quickly?"*

BUG[1,1] • *"Features, ease of use, documentation, size and culture of community, bugs, quality of implementation, performance, how well I understand its purpose, whether it's a do-all framework or a targeted library, . . . "*

The USAGE SCENARIOS[10,9] of an API, and the FEATURES[4,4] it supports, and the MATURITY[6,5] of the API to support the different features are also considered to be important factors to be part of the opinion summaries.

USAGE SCENARIOS[10,9] • *"Code quality, maintainer activity, ratio of answered/unanswered questions about the API."* • *"Has it been used in production grade software before?"*

MATURITY[6,5] • *"The maturity of API."* • *"Is the API mature, so it doesn't change every 10 days?"*

Q12 *We asked developers to provide their analysis on how opinions about APIs can be summarized when such opinions are scattered across multiple forum posts.* The purpose was to know what techniques developers consider as valuable to aggregate such opinions from diverse posts. The developers wished for an OPINION PORTAL[18,15], where all opinions about an API are aggregated and presented in one place.

OPINION PORTAL[18,15] • *"aggregate similar information, collect 'A vs B' discussions "* • *"like an aggregated portal about an API with all organized or non-organized pages"* • *"It would be very interesting for this to be automated via machine learning."* • *"Then I can read through them all and synthesize a solution."*

The developers mention about using the SEARCH[12,12] features in Stack overflow, but some acknowledged that they never thought of this problem deeply.

SEARCH[12,12] • *"Never thought about that deeply - I simply scan Google results until I'm satisfied"* • *"Often there is a sidebar with similar questions/discussions, but no easy way to "summarize" that I can think of."*

Once the opinions are aggregated, developers also wished to see those CATEGORIZED[8,8] into different API aspects, as well as view the distribution of SENTIMENTS[8,5] in formats, such as, star rating (similar to other domains). The CONTRASTIVE[7,7] viewpoints about an were also desired to learn the strengths and weakness of an API feature from multiple forum posts.

CATEGORIZATION[8,8] • *"Summarization by different criteria, categorization"* • *"download and do cluster analysis"*

SENTIMENT STATISTICS[8,5] • *"do an x-out-of-5 rating for portability, stability, versatility and other attributes"* • *"In general, I think the only useful summarization is just raw statistics, e.g.* `http://stateofjs.com/`*"*

CONTRASTIVE VIEWPOINTS[7,7] • *"I make a research if the there are contradicting opinions on stack overflow / etc. "* • *"You could divide it in like 'highlights' and 'pain points', maybe also give a score for maturity . . . "*

Developers asked for better support to find EXPERT[3,3], get concise PROBLEM SUMMARIES[1,1] of the question, learning about API POPULARITY[3,3] via the aggregation of the sentiments expressed towards the API, and the MATURITY[3,2] of an API.

Finally, the developers wished for the aggregation of API USAGE SCENARIOS[7,6] with the opinions, as well as a unification of the API FORMAL DOCUMENTATION[11,9] with those aggregated insights.

API USAGE SCENARIOS[7,6] • *"most interesting would be typical use cases"* • *"Usually opinions are evaluated via blog posts and other use cases."*

UNIFIED DOCUMENTATION[11,9] • *"Unified documentation on Stack Overflow (beta) looks like a positive step in the right direction. "*

Unfortunately, the Stack Overflow documentation site was shut down on August 8, 2017 [82], due mainly to a lower than the expected number of visits to the documentation site. Therefore, we can draw insights both from the problems faced by the Stack Overflow documentation site and from the responses of the developers in our survey to develop an API portal that can offer better user experiences to the developers.

Q15 *We asked developers about 11 API aspects, whether they would like to explore opinions about APIs around those aspects.* The 11 aspects are: 1) Performance, 2) Usability, 3) Portability, 4) Compatibility, 5) Security, 6) Bug, 7) Community, 8) Documentation, 9) Legal, 10) General Features, 11) Only sentiment We note that each of these options are found in the open-ended questions already. Thus the response to this question can be used to further ensure that we understood the responses of the developers. The results of the relevant Likert-scale survey question are summarized in Figure 2.8. The vast majority of the developers agrees that the presence of documentation,

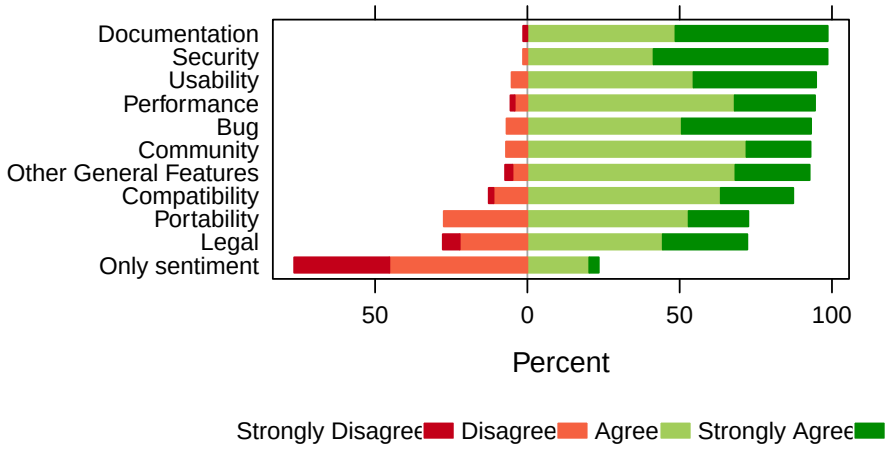

Figure 2.8: Developers' preference to see opinions about API aspects.

discussion of API's security features, mention of any known bugs in an API, its compatibility, performance, usability, and the supporting user community are the main attributes that contribute to the opinion quality. Developers could not agree whether a useful opinion should include a mention of any legal terms; while they agree that posts containing only one's sentiment (e.g., "I love API X" or "API Y sucks") without any rationale are not very useful. We note that there was a similar themed open-ended question in Q11. However, the developers were not shown Q15 before their response of Q11. Moreover, the two questions were placed in two separate sections, Q11 in section 4 and Q15 in section 6. There was only one question (i.e., Q11) in section 4. Therefore, the participants did not see Q15 or the options in Q15 during their response of Q11.

> Developers expect to see opinions about diverse API aspects, such as, usability, performance, security, compatibility, portability, and so on. Developers wished for a dedicated opinion portal engine for APIs where they can search for an API and see all the opinions collected about the APIs from the developer forums.

2.5 Discussions

In this section, we summarize the key points from our primary survey with regards to the two research questions that we set forth to address (Section 2.5.1). We then explore the potential implications out of the findings (Section 2.5.2).

2.5.1 Summary of Results

In Table 2.7 we show the key insights from all the questions that we asked the developers to learn about their needs to seek and analyze opinions about APIs from developer forums. In Table 2.8 we show the key insights from all the questions that we asked developers to learn about their needs

Table 2.7: Highlights of the Primary survey about developers' needs to seek opinions about APIs (RQ1)

RQ1.1	**Needs for seeking opinions about APIs from developer forums**
1	Do you visit developer forums to seek info about APIs? ***1) Yes 79.8%, 2) No 21.2%***
2	List top developer forums you visited in the last two years ***1) Stack Overflow & Exchange 56%, 2) Blogs & mailing lists 18.1%, 3) Github 5.4% 4) Google dev 4.8%, 5) Apple dev 3.6%***
4	What are your reasons for referring to opinions of other developers about APIs in online developer forums? ***1) Expertise 25.6%, 2) API usage 14%, 3) API selection 10.7%, 4) Opinion trustworthiness 10.7%, 5) Documentation/Community 9.1%***
5	How do you seek information about APIs in a developer forum? ***1) Search 83.9%, 2) Sentiment statistics 3.6% 3) Similarity analysis 2.7%, 4) Expertise 1.8%, 5) Recency/Documentation 0.9%***
6	What are your biggest challenges when seeking for opinions about an API in an online developer forum? ***1) Situational relevance 20.3%, 2) Trustworthiness 16.4% 3) Search 11.7%, 4) Recency 10.2%, 5) Community engagement 7%***
16	Where do you seek help/opinions about APIs? ***1) Stack Overflow 34.7%, 2) Co-workers 26.4%, 3) Internal mailing lists 10%, 4) IRC Chats 9.2%, 5) Search/Diverse sources 19.7%***
17	How often do you refer to developer forums to get information about APIs? ***1) Every day 36.1%, 2) Two/three times a week 32.5%, 3) Once a month 16.9%, 4) Once a week 14.5%***
18	When do you seek opinions about APIs? ***1) Selection amidst choices 88%, 2) Determining replacement 80.7%, 3) Feature enhancement 69.9%, 4) Fixing a bug 68.7%, 5) Develop competing API 56.6%, 6) Replace a feature 49.4%, 7) Validate a selection 38.6%, 8) Select API version 27.7%, 9) Other task 15.7%***
RQ1.2	**Needs for opinion quality assessment**
3	Do you value the opinion of other developers in the developer forums when deciding on what API to use? ***1) Yes 89.2%, 2) No 10.8%***
7	What factors in a forum post can help you determine the quality of a provided opinion about an API? ***1) Documentation 35.5%, 2) Reputation 23%, 3) Expertise 8.6%, 4) Situational relevance/Community 7.9%, 5) Trustworthiness 5.3%***

24	Please explain why you don't value the opinion of other developers in the developer forums. ***1) Trustworthiness 23.5% 2) Situational relevance 11.8%, 3) Community 11.8%, 4) Expertise 11.8%,***

Table 2.8: Highlights of the Primary survey about developers' needs for tool support to analyze opinions about APIs (RQ2)

RQ2.1	**Tool support**
8	Do you rely on tools to help you understand opinions about APIs in online forum discussions? ***1) Yes 13.3%, 2) No 86.7%***
9	If you do not use a tool to currently explore the diverse opinions about APIs in developer forums, do you believe there is a need of such a tool to help you find the right viewpoints about an API quickly? ***1) Yes 9.7%, 2) No 27.8%, 3) I don't know 62.5%***
10	You said yes to the previous question on using a tool to navigate forum posts. Please provide the name of the tool ***1) Google/Search 4, 2) Stack Overflow vots/app/related post 3, 3) Github issu/pulse 1, 4) Safari 1, 5) Reference documentation 1***
19	What tools can better support your understanding of API reviews in developer forums? ***1) Opinion mining 56.6%, 2) Sentiment analysis 41%, 3) Opinion summarization 45.8%, 4) API comparator 68.7%, 5) Trend analyzer 67.5%, 6) Co-mentioned competing APIs 73.5%, 7) Other Tools 8.4%***
RQ2.2	**Needs for Opinion Summarization**
11	What are the important factors of an API that play a role in your decision to choose an API? ***1) Usability 30.1%, 2) Documentation 16.3%, 3) Community 10.7%, 4) Performance 10.4%, 5) Compatibility 6.2%, 6) Legal 5.2%,***
12	Considering that opinions and diverse viewpoints about an API can be scattered in different posts and threads of a developer forum, what are the different ways opinions about APIs can be summarized from developer forums? ***1) Portal for opinion aggregation 12.7%, 2) Search 8.5% 3) Documentation 7.7%, 4) Sentiment statistics/Opinion categorization 5.6%, 5) Contrastive summaries/Usage scenario summaries 4.9%***
13	What areas can be positively affected by the summarization of reviews about APIs from developer forums?

	1) API selection 26.1%, 2) Productivity 10.4% 3) API usage 6.1% 4) Documentation 5.2% 5) API popularity analysis 3.5%
14	What areas can be negatively affected by the summarization of reviews about APIs from developer forums? ***1) Opinion trustworthiness 19.3% 2) Missing nuances 16.5% 3) Opinion reasoning 6.4% 4) API selection 5.5% 5) New API Entry 2.8%***
15	An opinion is important if it contains discussion about the following API aspects? ***1) Usability 88% 2) Documentation 85.3% 3) Security 83.1% 4) Performance 81.9% 5) Bug 81.9% 6) Compatibility 66.3%, 7) Community 63.9%, 8) Legal 47%, 9) Portability 44.6%, 10) General API Features 45.8%, 11) Only Sentiment 18.1%***
20	How often do you feel overwhelmed due to the abundance of opinions about an API? ***1) Sometimes 48.2% 2) Rarely 37.3% 3) Never 10.8% 4) Always 3.6%***
21	Would a summarization of opinions about APIs help you to make a better decision on which one to use? ***1) I don't know 48.2% 2) Yes 38.6% 3) No 13.3%***
22	Opinions about an API need to be summarized because ***1) The interesting opinion may not be in the posts that you have looked in 71.1% 2) Not enough time to look at all opinions 68.7% 3) Opinions can change over time 63.9% 4) Contradictory opinions about an API may be missed 61.4% 5) Opinions can evolve over time in different posts 61.4% 6) Too many forum posts with opinions 50.6% 7) Other reason 12%***
23	Opinion summarization can improve the following decision making processes ***1) Selection amidst choices 73.5%, 2) Determining replacement of an API 71.1%, 3) Validating an API selection 48.2%, 4) Feature enhancement 45.8% , 5) Replacing an API feature 42.2%, 6) Fixing a bug 38.6%, 7) Selecting API version 28.9%, 8) Other 9.6%***

for tool support to analyze opinions about APIs from developer forums. The insights are calculated using the same rules that we used in Section 2.2.

As we observe in Table 2.7 , the developers leverage opinions about APIs to support diverse development needs, such as, API selection, usage, learning about edge cases, etc. As we analyze such needs in Table 2.8 within the context of opinion summarization, we find that developers agree that opinion summaries can be useful to facilitate those needs by offering an increase in productivity (e.g., saving time, quick but informed insights). In the absence of a specific opinion summarization portal available for APIs, though, we find that developers were unsure whether such summaries can be feasible. However, they recommend to leverage machine learning techniques to facilitate an opinion portal for APIs.

There are a number of questions that are paired as open-ended and closed questions, with the open-ended questions being asked before the corresponding closed-end question. For example, we asked developers the reasons for them to seek for opinions about APIs using two questions (Q4

and Q18). The responses to the open-ended question (Q4) show a slight variation from the options provided in the closed question (Q18). For example, in Q4, the main reason developers mentioned was to build and look for expertise about APIs while seeking for opinions. Q18 does not have that option. Nevertheless, the majority of agreement for the options in Q18 show that those needs are also prevalent.

In another pair of open and closed questions, we asked developers about what specific API aspects/factors they expect to see in the opinions about APIs. In this case, the responses to both questions produced an almost similar set of API aspects, such as, performance, usability, documentation, and so on. This is perhaps due to the fact that developers expected to use opinions to compare APIs during their development needs. Aspects, such as, performance or usability can be used across APIs of any domains to facilitate such comparison.

2.5.2 Implications

• *Evaluating benefits of API opinion summaries.* While the majority of the developers agreed that opinion summaries can help them evaluating APIs, three types of summarization were found to be very useful: 1) categorization of opinions by API aspects; 2) opinions summarized as topics; and 3) most important opinions summarized into a paragraph. It would be interesting to further investigate the relative benefit of each summarization type and compare such findings with other domains. For example, Lerman et al. [83] found no clear winner for consumer product summarization, while aspect-based summarization is predominantly present in camera and phone reviews [24], and topic-based summarization has been studied in many domains (e.g., identification of popular topics in bug report detection [84], software traceability [85], etc.).

• *Augmenting API usage examples with opinions to facilitate crowd-sourced API documentation.* Developers mentioned that opinions about different API elements can help them better understand the benefits and shortcomings of the API. Summaries extracted from Stack Overflow can be effective about the purpose and use of API elements (classes, methods, etc.) [86]. While the official API documentation still lacks completeness, clarity, and context [1, 7], our results suggest to augment it by building opinion summarizer tools that leverage informal documentation. This motivates future extension of the recent research on augmenting insights about APIs from the forums to the formal API documentation [87]. Such tools can be integrated within IDEs to assist developers during their APIs-based tasks.

• *API expertise recommendation.* Our survey participants seek opinions from the experts and consider such opinions as "trusted". The identification of domain experts is an active research area [88–90]. Research on recommending API experts from the developer forum posts would help developers seeking and evaluating opinions. As we found in the survey, the specific facets (e.g., developer profile and reputation) of the developer forums such as Stack Overflow may be useful.

2.6 Threats to Validity

The accuracy of the open coding of the survey responses is subject to our ability to correctly detect and label the categories. The exploratory nature of such coding may have introduced the *researcher bias*. To mitigate this, we coded 20% of the cards for eight questions independently and measured the coder reliability on the next 20% of the cards. We report the measures of agreement in Table 2.5. While we achieved a high level of agreement, we, nevertheless, share the complete survey responses, along with the results of the open-coding in our online appendix [66].

Due to the diversity of the domains where APIs can be used and developed, the provided opinions can be contextual. Thus, the generalizability of the findings requires careful assessment. Such diversity may introduce *sampling bias* if developers from a given domain are under- or overrepresented. To mitigate this, we did not place any restriction on the developers' knowledge or experience to be eligible to participate in the survey.

2.7 Summary

Opinions can shape the perception and decisions of developers related to the selection and usage of APIs. The plethora of open-source APIs and the advent of developer forums have influenced developers to publicly discuss their experiences and share opinions about APIs. To better understand the role of opinions and how developers use them, we conducted a study based on two surveys of a total of 178 professional developers. The survey responses offer insights into how and why developers seek opinions, the challenges they face, their assessment of the quality of the available opinions, their needs for opinion summaries, and the desired tool support to navigate through opinion-rich information. We found that developers feel frustrated with the amount of available API opinions and face several challenges associated with noise, trust, bias, and API complexity when seeking opinions. High-quality opinions are typically viewed as clear, short and to the point, bias free, and supported by facts. We also found that while API opinion summaries can help developers evaluate available opinions, establishing traceability links between opinions and other related documentation can further support their API-related tasks and decisions.

Chapter 3

How API Formal Documentation Fails to Support Developers' API Usage

In our previous chapter, we described the results of two surveys to understand how and why developers seek and analyze API reviews in online developer forums. One of the observations was that developers leverage opinions posted about APIs to validate the provided code examples about API. According to one developers of the study: *"You can see how code really works, as opposed to how the documentation thinks it works"*. The developers consider that a summarization of reviews about APIs that involves the code examples posted in the forum posts, can assist them in the learning and usage of the APIs.

Leveraging usage scenarios about APIs posted in the developer forums can be necessary, when API official documentation does not include those scenarios and can often be outdated [91]. Indeed, documentation that does not meet the expectations of its readers can lead to frustration and a major loss of time, or even in an API being abandoned [19]. To address the shortcomings in API official documentation, research efforts focused on the linking of API types in a formal documentation (e.g., Javadoc) to code examples in forum posts where the types are discussed [35], presenting interesting textual contents from Stack Overflow about an API type in the formal documentation [87], and so on. However, our study in the previous chapter shows that the plethora of usage discussions available for an API in the forum posts can pose challenges to the developers to get quick and actionable insights into how an API can be used for a given development task. One possible way to assist developers is to generate on-demand developer documentation from forum posts [92]. However, to be able to do that, we first need to understand what specific problems persist in the API official documentation, that should be addressed through such documentation efforts. If we know of the common documentation problems, we can then prioritize those problems and investigate techniques to leverage API usage scenarios posted in the developer documents to address those problems.

To provide an evidence-based foundation for planning API documentation efforts, we conducted two surveys of API documentation quality with a total of 323 software developers at IBM. A first survey allowed us to catalog and understand how ten common documentation problems manifest themselves in practice. Six of the problems we learned about have to do with content, and four with presentation. With a second survey, we studied the frequency and severity of the API documentation problems we had observed, leading to the conclusion that incompleteness, incorrectness, and ambiguity are the most pressing problems with API documentation. With regards to the presentation of the documentation, the fragmentation in the documentation (e.g., one page for each class in a Javadoc) is considered as the most severe. In the rest of the chapter, we denote the first survey as the 'Exploratory Survey' and the second survey as the 'Validation Survey'.

Chapter Organizations. We discuss the design and data analysis process of the exploratory survey in Sections 3.1, 3.2. In Section 3.3, we present the 10 documentation problems. We discuss the results of the validation survey in Section 3.4. In Section 3.5, we discuss potential approaches to address the documentation problems by taking cues from the findings of our surveys in Chapter 2. We conclude in Section 3.6.

Q1 What is the last development task you completed that required you to consult API documentation?

— Task description | Online link to task URL

Q2 Give up to 3 examples of API documentation that you found useful. Explain why the documentation was useful.

— URL of the API documentation unit

— Textual explanation of why the documentation unit was useful to complete the task

Q3 Give up to 3 examples of API documentation that you did not find useful. Explain why the documentation was not useful.

— URL of the API documentation unit

— Textual explanation of why the unit was not useful

Figure 3.1: Exploratory survey questions

3.1 Exploratory Survey

We sent out a survey to explore the experience that developers have with documentation. Our goal was to collect both good and bad examples of reference API documentation. The survey contained five questions, two for demographics and three related to API documentation (Figure 3.1).

We selected participants from the population of employees at the IBM Canada Ottawa and Toronto lab locations. We sent the survey invitation to 698 employees whose job role contained the keyword 'software' and one or more of the following keywords: engineering, development, architect, testing, and consultant. 69 employees responded (9.8%): 47 were developers, 15 architects, three consultants, three managers, and one tester. The experience of the respondents ranged between one year and 40 years (average 13.1, standard deviation 10.8). The consultants were engaged in proof-of-concept prototype development. The tester consulted API documentation to write test cases. Two out of the three managers were product managers. In this chapter, we refer to survey respondents using a tuple $(i : j)$ denoting the survey (i) and the respondent (j). The 'Exploratory Sruvey' is denoted by the ID 1 and the 'Validation Survey' is denoted by the ID 2 in the tuple.

Each of the 69 responses contained a reference/description to an actual development task, 20 of which also contained a URL for the task in an issue tracking repository. This enabled us to study the responses within the context of the task, the related code changes (when we had an access to their code repository), and the API documentation units that developers referred to to complete the task.

The respondents provided insights on documentation from a wide variety of technologies: 72 APIs from six types of programming languages (see Figure 3.2). In our subsequent analysis, we considered an API unit to roughly correspond to a coherent set of elements that could be distributed together (such as jar files in the Maven repository). Fourteen respondents provided examples from publicly available corporate APIs, three from private corporate APIs, the rest from open source

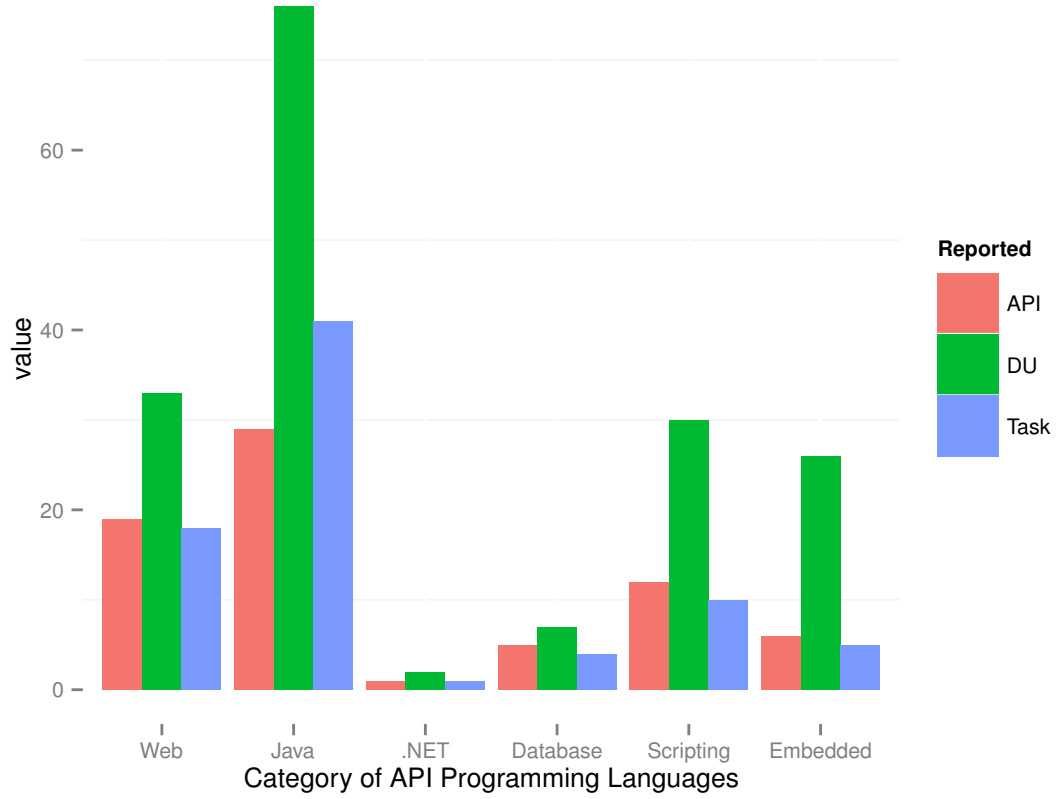

Figure 3.2: APIs reported in the exploratory survey

APIs. The author of this thesis (Gias Uddin) had access to all referenced corporate APIs in the context of a research internship at IBM.

Figure 3.2 shows the number of APIs mentioned by the respondents, organized by the type of programming language. The first bar for each language shows the total number of APIs mentioned, the second bar shows the total number of documentation units, and third the number of development tasks. A documentation unit is an explicitly designated block in the documentation related to an API element (such as the documentation for a method) or to the API itself (such as an overview page). Forty-four percent of the APIs (58/131) were Java-based. Web-based APIs constituted 29% of the total and were mostly for JavaScript. Embedded APIs were mostly for C/C++. Scripting APIs were mostly for Python.

3.2 Data Analysis

From the answers to questions 2 and 3 (Figure 3.1), we collected a total 179 documentation examples: 90 examples of good documentation and 89 examples of bad documentation. After inspection, we realized that we could learn very little from the examples of good documentation: Most of these were justified as generally fulfilling the developer's information needs. We thus

focused on learning about documentation quality by studying examples where documentation fails.

Following a card-sorting approach, we grouped the examples of bad documentation according to the nature of the problem reported. Both authors (Gias Uddin and Martin P. Robillard) were involved in the card sort analysis. We performed this task by taking into account the respondent's comment and the API documentation unit provided in the URL (when applicable). We discarded six examples as invalid because they contained comments too generic to qualify, such as "All API docs help more or less...".

Among the remaining 83 examples, we discarded another four because we could not confidently interpret the respondent's answer. In the end, this phase resulted in the mapping out of 10 common types of documentation problems illustrated by 79 comments describing concrete examples of inadequate documentation. Five examples were linked to two or more problems, and the remaining 74 were linked to a single problem.

3.3 Documentation Problems

The respondents described six problems related to the content and four related to presentation of the documents. We explain the problems below (summarized in Table 3.1).

3.3.1 Content Problems

C1. Incomplete

In the trivial case, documentation can be incomplete due to a lack of investment:[1]

> I have had to deal with lots of auto-generated documentation with no documentation at all for many classes/methods. ($R_{2:3}$)

However, even when earnest effort is invested in documenting an API, it can be difficult to anticipate all the different ways in which the API can be used. Understanding how to use an API element may require more than the description of its functionality when the API element can be involved in complex interactions with other API elements.

For example, the Java Ehcache API offers functionality to set up distributed caching across nodes. The interaction among distributed nodes needs to be established based on the Java RMI (Remote Method Invocation) API. $R_{1:29}$ was tasked with the set up of a Java-based distributed caching system and for this purpose he experimented with the Ehcache API. While he found the summary overview of the API useful, he was frustrated when the documentation did not discuss adequately on how it communicates with the RMI API:

[1] In our Validation survey, we asked developers to write their comments about the documentation problem (Q7 in Figure 3.3). A number of quotes in this section are taken from those responses.

Table 3.1: API documentation problems reported in the exploratory survey

Category	Category Description/Related Problems With Description		#R	#D
Content	arise when the content of the documentation does not meet needs		61	57
Problems	C1. Incomplete	The description of an API element or topic is missing from where it is expected	20	20
	C2. Ambiguous	The description of an API element is provided and mostly complete, but the description itself is unclear	16	15
	C3. Unexplained examples	A code example is provided, but it is insufficiently explained	10	8
	C4. Outdated	The documentation on a topic exists but refers to a previous version of the API	6	6
	C5. Inconsistent	The documentation of elements meant to be combined does not agree	5	4
	C6. Incorrect	Some information in the documentation is not correct	4	4
Presentation	arise when the presentation can hinder the understanding of contents		25	22
Problems	D1. Bloated	The description of an API element or topic was overly verbose or excessively extensive	12	11
	D2. Fragmented	The information related to an element or topic was fragmented/scattered in too many different pages/sections	5	5
	D3. Excessive Structural Info	The description of an element contained redundant information about the syntax or structure of the element, which could be easily obtained through modern IDEs	4	3
	D4. Tangled	The description of an API element or topic was tangled with other information developers did not need	4	3

#R = # examples mentioning a problem. #D = # distinct developers reporting a problem

> The experiment involved extending the existing RMI replication mechanism and the javadoc didn't go into sufficient detail as to when/how the various methods would be called. ($R_{1:29}$)

C2. Ambiguous

Ambiguous documentation refers to the case where a topic of interest appears to be covered by the documentation, but the text leaves out important details so that multiple interpretations are possible. In a way ambiguous documentation is a form of incomplete documentation, but where the missing information is of a *specific, clarifying* nature. Nevertheless, there is a relatively clear distinction between the two categories in terms of reaction from the respondents: whereas they speak of missing information in the case of incompleteness, the comments have to do with confusion or incomprehension in the case of ambiguity.

The ASM 3.2 API is used to manipulate Java bytecode. It contains a method `int getItem(int item)` in the class `ClassReader`. The javadoc of `getItem` states: "Returns the start index of the constant pool item in b, plus one.". A constant pool in Java is a table of structures that stores the identifiers of types, variables and string constants; hence the start index can point to a type (e.g., 'String'), or its constants/identifiers.

> Documentation for ClassReader did not adequately describe the getItem() method used to index into a constant pool. Specifically, it wasn't clear whether the return value was the index of the constant pool entry type tag or the start of the data for the constant pool entry. ($R_{1:1}$)

C3. Unexplained Examples

Although code examples are generally appreciated by developers, some respondents indicated being frustrated when an example did not have an adequate explanation.

The DOJO JavaScript APIs are used to develop AJAX-based applications. A respondent liked that many methods in the APIs are documented with code examples, but got frustrated when it was hard to figure out how an input in the example should be configured. DOJO, being a JavaScript API, needs to explain the code examples in terms of how the code can be deployed in a browser environment. This requires the configuration of input objects for each of the documented code examples in a method. In the DOJO documentation, the configuration objects are not always discussed before a code example:

> DOJO's APIs [...] should specify the configuration object that each method takes in detail (e.g., the methods/strings that the object can contain), because simply saying that it takes an object is not usable. ($R_{1:22}$)

Indeed, as Mehdi et al. [93] reported in their study on the code examples posted in StackOverflow, "explanations accompanying an example are as important as the examples themselves".

C4. Outdated

When APIs go through rapid development, their documentation can become quickly outdated [45], and the new changes often are not reflected in the current documentation.

The Apache HBase API is used to host distributed Hadoop database for big data. The API goes through frequent version changes, often changing elements or removing them altogether. For example, the `ClusterStatus` class has a method `Collection<ServerName> getServers()` in version 0.90x which was changed to `int getServers()` in 0.92. `Collection<ServerName> getServerInfo()` from 0.90 was changed to `Collection<HServerInfo> getServerInfo()`.

> Even though the method names are similar, they have different functionality in 0.92x. ($R_{1:5}$)

While it is not reasonable to expect an API to have documentation of every change, significant functionality changes related to an element requires feedback in the form of documentation or something to overcome the information mismatch between versions, especially when the version change is expected to be backward-compatible [94].

C5. Inconsistent

Software development can be incremental, with multiple products often combining together into a new end product. These intermediate products can be developed by more than one development team, leading to a risk of inconsistent documents with insufficient explanation on how the interoperability between the intermediate products can be supported.

> I'm looking at ways to improve integration between different products ... One approach would be to use the different product API's to construct new "glue" type applications as part of our portfolio ... The problem is that there is no consistency across the different products, ..., not in documentation, ... Obviously that does not promote interoperability between products. ($R_{1:45}$)

C6. Incorrect

The description of an API element can be incorrect. For example, the actual input parameters and output type of an API method as implemented in the source code can be different from the documentation. To do routine update, a reported API offered scheduling functionality using a set of UNIX-like triggers:

> there is a Cron trigger with wrong information: "it says the "schedule" parameter must be a string with the same format as a UNIX crontab file; however it does not implement everything that the UNIX contrab format provides ... inaccurate documentation (it says `prev_trigered_time` is available, but it was not .... ($R_{1:36}$)

3.3.2 Presentation Problems

D1. Bloated

The risk with associating large chunks of text to a specific element or topic is that readers will not readily be able to determine whether the text provides the information they seek, especially in cases where the title or header are general. An example was provided by a respondent who was trying to understand some of the advantages of using a cluster installation:

> This introductory section is so bloated that it was hard to understand why it has so much text at all, when the introduction gives neither an overview nor the purpose of clustering. ($R_{1:31}$)

D2. Fragmented

When developers had to click through multiple pages of an API documentation to learn the functionality and usage of a single API element, they found the separation of description at such micro-level to be unnecessary (as Robillard and DeLine have similarly observed [19]). Unlike other APIs, DOJO APIs adhere to a two-layer description format of each functions. The first page provides an overview of the functionality with a code example (referred to as the 'API' document), the second page describing the related parameters, e.g., the underlying environment or server (referred to as the 'usage' document):

> ...This information (functionality description and usage configuration) should be part of the API, instead of separated into two documents: API and usage. ($R_{1:22}$)

Developers found it difficult to navigate through the myriad of pages in an API document to find information:

> fragmented documentation I find really difficult to use where you have to have 10's of clicks through links to find the information you need and page after page to read. ($R_{2:69}$)

D3. Excessive Structural Information

The description of a type (class, interface) in object-oriented APIs normally includes structural relations with other types of the API. For example, all the classes in the Java SE APIs are subclasses of the class `java.lang.Object`. Respondents indicated that excessive structural information, which often can be readily obtained on-demand from the IDE, was an obstacle to focusing on the important content.

> again too much information on how the class is related to other classes, I don't need docs for this. ($R_{1:11}$)

Q1 Do you consult API documentation as part of your development tasks?

—— Yes/No

Q2 Approximately how many hours on average per week do you spend consulting API documentation?

—— numbers [decimals ok]

Q3 Based on your experience of the last 3 months, how frequently did you observe the documentation problems?

► List of documentation problems as identified from survey 1

——Never | Once or Twice | Occasionally | Frequently | No opinion

Q4 How severe was the documentation problem to complete your development task, when you last observed it?

► List of documentation problems as identified from survey 1

—— Not a problem | Kind of irritating (moderate) | I wasted a lot of time on this, but figured it out (severe) | I could not get past it and picked another API (blocker) | No opinion

Q5 Based on your experience of the last 3 months, list the three APIs whose documentation caused you the most severe problems, as reported in the survey.

—— Name of API and the documentation problem

Q6 Given a strict budget to solve the 10 documentation problems, which 3 should be prioritized?

___ List of documentation problems
numeric value [3=Top, 2=Fair, 1=Low, 0=No priority]

Q7 Any comment about your experience with the problems?

—— comments

Figure 3.3: The validation survey questions

D4. Tangled

The explanation of APIs with specific usage scenarios were considered very helpful, but not when multiple usage scenarios were tangled with each other in one single description. $R_{1:9}$ was unhappy with the documentation of the JSONObject class of the json.org API:

> the class says that for get and put, you can cast to a type which is fine, but the next line dives to a different topic of type coercion that I did not need. ($R_{1:9}$)

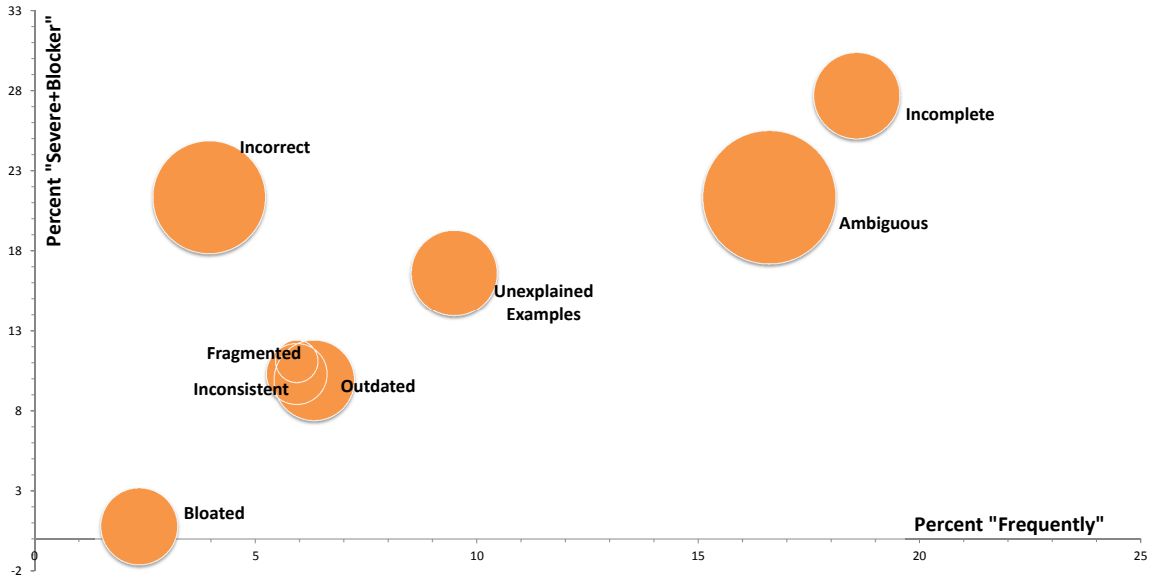

Figure 3.4: Percent "Frequently" is the ratio of the response "Frequently" over all responses (Q4 in Figure 3.3). Percent "Severe + Blocker" is the ratio of response "Severe + Blocker" over all responses (Q5 in Figure 3.3). The size of each circle is the based on the *percent top priority* of the participants to solve the documentation problems if they have been given a budget to solve any three of the 10 problems (Q6 in Figure 3.3).

3.4 Validation Survey

We conducted a second survey with a different group of participants at IBM, to measure the frequency and severity of the problems we documented based on the first survey. The survey questionnaire had seven questions, with 10 sub questions for Q3 and Q4 (Figure 3.3).

We sent the survey invitation to the software developers and architects at IBM Canada and Great Britain. From our first survey, we found that 65 out of the 69 respondents were either software developers or architects. Therefore, we only sent the survey invitation if the job role of the employee contained one of the three phrases: "software developer", "software engineer", "software architect". The target population thus included 1064 participants (610 from Canada and 454 from the UK). We received 254 responses (24.9%). Among the respondents, 203 (80%) were developers (114 from Canada and 89 from the UK). Among the 51 architects (20%), 28 were from Canada and 23 from the UK. The years of experience of the respondents ranged between 1 and 45 (average 13.17, standard deviation 9.9).

We analyzed three aspects of the responses: frequency and severity of the problems and the necessity to solve the problems (Q3, Q4, and Q6 in Figure 3.3, respectively). The severity scale was constructed by discussing with three respondents from the first survey.

For the frequency scale, for each documentation problem, we computed the ratio of the response "Frequently" over all responses. For example, for the problem "Ambiguous", we had 42 respondents mention that they observed the problem frequently in the last three months, 68 occasionally, 38

once or twice, and 10 never. Ninety-six of the respondents did not provide any opinion. We follow the strategy of Bird and Zimmermann [81], and do not include the number of no opinions in the calculation of the frequency ratio, leading to a 26.5% ratio in this case. For the severity scale, we computed the ratio of the response "Severe + Blocker" over all responses, again excluding no opinion responses.

In Figure 3.4, the x-axis represents the ratio of "Frequently" responses and the y-axis represents the ratio of (Severe + Blocker) responses. The size of each circle is the based on the *percent top priority* of the participants to solve the documentation problems if they have been given a budget to solve any three of the 10 problems. For each problem, the respondents were given an ordinal scale of three values: top (3), fair (2), and low (1). Each respondent chose at least one problem as worth solving. The size of the circle for each problem is the percentage of the respondents who selected "top" priority for the problem. For example, ambiguity (C2) was chosen 54 times as the top priority, 18 times fair, 28 times low, and 4 times as no priority. Thus, ambiguity was chosen 51.9% times as the top priority. The problem, "excessive structural information" is absent because it was never selected as the top problem to solve.

Six problems were mentioned as blocker at least once: Incomplete, Ambiguous, Outdated, Incorrect, Inconsistent, and Unexplained examples. In terms of priority for addressing documentation problems, ambiguity was selected as the most important. In fact, the problems of ambiguity (C2) and incompleteness (C1) identified as the most critical in Figure 3.4 are in the top three priorities for improving documentation. The difference in ranking may be a function of how we presented the data. For example, ambiguity was the problem with the highest proportion of "blocker" severity ratings. An interesting new observation from the budget priority question, however, is the perceived importance to fix incorrect documentation. Incorrect documentation was observed infrequently, but instances of this problem were judged very severe, showing that for some API projects it may make more sense to address problems in order of severity than combined frequency and severity. In any case, the analysis clearly confirms incompleteness, incorrectness, and ambiguity as three documentation problems in need of attention.

In interpreting the results of the study, one should keep in mind that the respondents were from a single software company. Even though IBM is a large international company with software products from diverse domains, it is possible that the responses may be influenced by corporate culture. However, the responses came from software engineers from two countries situated in two different continents. We also point out that the survey focused on common API documentation problems, and may not include all possible API documentation problems.

3.5 Discussions

The findings from the surveys show that developers considered API usage scenarios in the official documentation as often incomplete, ambiguous and difficult to use. This observation leads to the question of how we can improve API documentation if the only people who can accomplish this task are unavailable to do it. One potential way forward is to develop recommendation systems that can reduce as much of the administrative overhead of documentation writing as possible, allowing experts to focus exclusively on the value-producing part of the task.

Another potential way is to produce API documentation by leveraging the crowd, such as the live API documentation [35], which links Stack Overflow posts with related reference documentation. Although these kinds of solutions do not have the benefit of authoritativeness, our surveys in Chapter 2 show that developers leverage the reviews about APIs to determine how and whether an API can be selected and used, as well as whether a provided code example is good enough for the task for which it was given. Thus, the combination of API reviews and code examples posted in the forum posts may constitute an acceptable expedient in cases of rapid evolution or depleted development resources, offering ingredients to on-demand developer documentation [92]. In this section, we take cues from our findings in Chapter 2 and discuss potential approaches to address the documentation problems in Section 3.3. Specifically, we discuss how API reviews and code examples posted in developer forums may help us find solutions to the documentation problems.

The major content-related problem developers faced while using the API formal documentation is the lack of *useful* contents in the 'Incomplete' documentation pages. By automatically mining and aggregating usage examples posted about API element in the developer forums, we can make the content of the formal documentation page of an API element (e.g., an API class in the Javadoc) more complete. The 'Incompleteness' and the 'Ambiguity' in the documentation contents were considered as the two most sever problems, that could force developers to abandon an API. The third major content problem is the 'Unexplained Examples'. For the code examples about an API that can be mined from forum posts, we can further mine the description and the reviews (e.g., strengths and weaknesses) as posted in forum posts about the example, into the documentation pages. This can thus compensate for both the ambiguity problem that arise when the usage of an API element is not adequately discussed, and the unexplained problems that can arise when an example is not adequately discussed. The problem of 'Outdated' documentation can also be addressed by continuously mining and aggregating such usage examples about APIs from developer forums. The other content problems can also be addressed by leveraging usage scenarios about APIs posted in the forum posts. For example, how an API is portable across different computing platform is an often missing part in the documentation contents (C5). As we observed in the survey in Chapter 2, developers look for opinions provided by other developers in the forum posts to determine the *portability* of an API across the computing platforms. In fact, this is one of the API aspects about which the developers in our survey also wished to explore opinions about (see Section 2.4 of Chapter 2).

The opportunity of leveraging API reviews, discussions and code examples posted in the forum posts to improve API documentation begs the question of how to do it *properly*. Indeed, one approach is to link code examples about an API type from the forum posts in the Javadoc of the API type, as proposed by Subramanian et al. [35]. However, this approach does not address the problems developers face with the 'fragmented' as well as 'bloated' presentation formats of the API documentation. For example, developers complained about their needs to click through multiple pages of API documentation to learn the functionality and usage of a single API element (D2). Therefore, one desirable approach would be to mine and summarize usage scenarios about APIs from forum posts, in a way that can not only assist developers find the relevant API usage quickly and with less effort, but also will help them find those information in a less fragmented and bloated format.

3.6 Summary

The major outcome of our surveys is that quality content is what our respondents most cared about. The most frequent and common problems had to do with content, and the requests for addressing problems prioritized five content-related problems above any presentation problem. Perhaps not surprisingly, the biggest problems with API documentation are also the ones that require the greatest amount of technical expertise: completing (C1, C3), clarifying (C2), and correcting (C6) documentation requires a deep and authoritative knowledge of the API's implementation, making it difficult to accomplish by non-developers or recent contributors to a project. In Chapter 6, we describe a framework to automatically mine and summarize usage scenarios about APIs from forum posts. We show that developers can leverage such summaries to complete their coding tasks with more accuracy, in less time and effort than when they use API official documentation.

Chapter 4

Automatic Opinion Mining from API Reviews

In the surveys about the developers' needs to seek and analyze opinions about APIs (Chapter 2), we observed that the respondents preferred to explore opinions about certain API aspects in the forum posts, e.g., performance, usability, portability, and so on.

The diverse nature of 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 [95, 96], detecting similar technological terms (e.g., techniques similar to 'JSON') [97], or automatically detecting problematic API features [98], and so on. For example, the automatic categorization dataset used by Hou and Mo [95] 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.

With a view to understand the role of sentiments and 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 subjectivity of provided opinions in all the 4,522 sentences from 1334 Stack Overflow posts. An aspect denotes what specific API attribute was discussed in the sentence. An aspect can be an attribute (e.g., usability) or a contributing factors towards the usage or development of an API (e.g., licensing or community support). Subjectivity denotes whether the sentence was positive, negative or neutral. A sentence is considered as an opinion, if it is either positive or negative. Total eight different 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 in the API discussions, and how API aspects are discussed across the positive and negative opinions and via the communication among the API stakeholders (e.g., API authors and users). We found that developers offer insights about diverse API aspects in their discussions about APIs. We found that such insights are often associated 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 mine API reviews and to categorize those reviews across aspects. We investigated the feasibility of several rule-based and supervised aspect detection techniques. The 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 77.8 precision and 99.1 accuracy.

Step 4. Automatic Mining of Opinions from API Reviews

Motivated by the prevalence of API reviews in forum contents, we designed a framework to automatically mine API reviews from forum posts. First, we detect API mentions in the textual contents of the forum posts with upto 98.8 precision. Second, we detect opinionated sentences with upto 72.6 precision. Third, we associate the opinionated sentences to the APIs about which the opinions were provided with upto 95.9 precision.

Step 5. Opiner - A Search Engine for API Reviews

We developed a tool named Opiner that supports 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 online developer forums, such as, Stack Overflow. 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) discussed in those opinions.

Chapter Organization. We describe the methodology used to conduct and report the research in Section 4.1. We discuss the detail of the data collection process used to construct our API review benchmark in Section 4.2. In Section 4.3, we describe our exploratory analysis on the role of sentiments and API aspects in API reviews. We present a suite of techniques to automatically categorize the API aspects in API reviews in Section 4.4. We present a suite of techniques to automatically mine opinionated sentences about APIs from forum posts in Section 4.5. We discuss the threats to the validity in Section 4.6 and conclude the chapter in Section 4.7.

4.1 Research Method

In this section, we describe the study procedures, and present the research questions that we answered through the study.

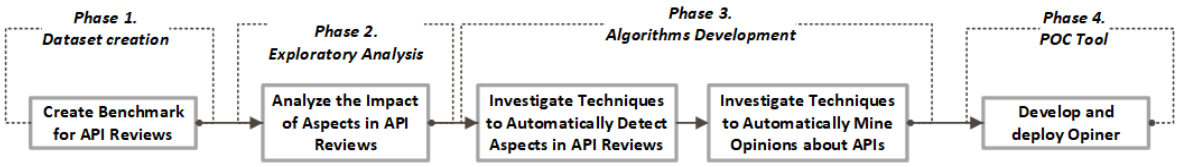

Figure 4.1: The five steps we followed in this study. The five steps were grouped under four different phases. First phase was data collection. Last phase was the development of Opiner, our POC (Proof of Concept) tool.

4.1.1 Study Procedures

The five steps can be grouped under four distinct phases (see Figure 4.1):

Phase 1. Dataset Creation

We create the benchmark dataset of 4522 sentences. Each sentence is manually coded with two labels:

Aspect:

What API aspect is discussed in the sentence. We noted in Section 1.2 of Chapter 1 that in our survey of software developers to learn about their needs to seek and analyze opinions about APIs (Chapter 2), the developers were mostly interested to see opinions about the following 11 API aspects.

1) **Performance**: The opinion is about the performance of an API.
2) **Usability:** The opinion is about how usable an API to support a development need (e.g., easy to use vs complicated designs, and so on).
3) **Security:** The opinion is about the security principles, the impact of security vulnerabilities and the understanding of the security best practices about an API.
4) **Documentation:** The opinion is about how the API is documented in different development resources.
5) **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.
6) **Portability:** The opinion is about how an API can be used across different operating systems
7) **Community:** The opinion is about the presence and responsiveness of the people using and supporting an API.
8) **Legal:** The opinion is about the licensing and pricing issues of an API.
9) **Bug:** The opinion is about the presence, proliferation, fixing, or absence of specific bugs/issues in an API.
10) **Other General API Features**: Opinions about any feature or usage of an API.
11) **Only Sentiment:** Opinions about an API without specifying any aspect or feature.

We label each sentence as the above 11 categories. As we discussed in Chapter 1, the first nine aspects are considered as *implicit aspects*. An opinion about an API may not be categorized to any of the nine implicit aspects, but it can contain information about certain API features. In such cases, we label the aspect as 'Other API General Features'. If only sentiment towards an API is expressed in a sentence, we merely label it as 'Only Sentiment'.

Sentence Polarity:
The presence of positive or negative sentiments in the opinion. A sentence may not contain any sentiment. In such case, we label the sentence as 'neutral'.

Phase 2. Exploratory Analysis
We analyze the dataset to investigate the presence of aspects in the API reviews, and how such aspects influence developers' sentiments towards the APIs.

Phase 3. Algorithms Development
We investigated the feasibility and design of the following techniques:

1) Automatic aspect detection. We investigated both rule-based and supervised classifiers.
2) Automatic API mention detection. We designed a rule-based technique.
3) Automatic detection of opinionated sentences. An opinion about an API is a sentence with positive or negative sentiments towards the API expressed in the sentence. We investigated the fusion of two popular sentiment detection algorithms from literature.
4) Automatic association of opinions towards the detected API mentions. We designed a rule-based technique.

Phase 4. POC (Proof of Concept) Tool
To facilitate the automatic mining and summarization of opinions around API aspects, we developed Opiner, our proof of concept (POC) tool. We deployed Opiner as a search engine. Using Opiner, developers search for opinions about APIs and analyze those based on the different aspects.

We analyze and report the performance of the each of the techniques developed and experimented as part of the algorithms development phase. We report the performance of the techniques using four standard measures of information retrieval: precision (P), recall (R), F-measure ($F1$), and Accuracy (A) (Equations 4.1 - 4.4).

$$P = \frac{TP}{TP + FP} \tag{4.1}$$

$$R = \frac{TP}{TP + FN} \tag{4.2}$$

$$F1 = 2 * \frac{P * R}{P + R} \tag{4.3}$$

$$A = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.4}$$

Here, $TP =$ Nb. of true positives, and $FN =$ Nb. false negatives.

4.1.2 Research Questions

We approach and report the exploratory analysis and the algorithms development (Phases 2 and 3) by answering six research questions (three for the exploratory analysis and three for the algorithms development). We present the research questions below.

4.1.2.1 Exploratory Analysis (Phase 2)

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

• **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. [26] of Google research observed that the opinions in a domain mostly followed the Zips law in that certain aspects are much more discussed than others. If we observe similar phenomenon for API reviews, that can eventually lead to the prioritization of the analysis of API aspects across domains (e.g., deeper analysis on the more prevalent aspect, etc.).

• **Approach.** For each aspect, we first count the total number of sentences labeled 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 to each aspect in the opinions (e.g., design, usage and other themes in the usability aspect). The opinions labeled 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 labeled each sentence for a given aspect based on the themes related to the aspect that was discussed in the sentence. (2) Once all the sentences of a given aspect are labeled, 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 [91].

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 into 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) **Intent/Action**: What were the intention and/or action expressed by the stakeholder in the opinions?

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

Stakeholder: We manually examined each sentence in the benchmark and labeled it based on the type of person who originally wrote the sentence. For example, we labeled 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.

Intent/Action: We identified the following intents in the opinions: (1) **Liked**: the opinion showed support towards a provided suggestion or promotion but the opinion provider 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. (2) **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. (3) **Used**: the opinion was from a user who explicitly mentioned that he had used the API as suggested.

We report the findings using the following metrics:

$$\text{Suggestion Rate} = \frac{\#\text{Suggestions (S)}}{\#\text{Posts (T)}}, \tag{4.5}$$

$$\text{Liked Rate} = \frac{\#\text{Liked}}{\#\text{S} + \#\text{P}} \tag{4.6}$$

$$\text{Promotion Rate} = \frac{\#\text{Promotion (P)}}{\#\text{Posts (T)}}, \tag{4.7}$$

$$\text{Unliked Rate} = \frac{\#\text{Unliked}}{\#\text{S} + \#\text{P}} \tag{4.8}$$

4.1.2.2 Algorithms Development (Phase 3)

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

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

RQ5. How can opinions be detected in API reviews?

• **Motivation.** The second step to mine opinions about APIs is to detect opinionated sentences in the forum posts. An opinionated sentence can have positive or negative sentiments about one or more APIs. Recent efforts on sentiment detection in software engineering show that sentiment detection techniques from other domains do not perform well for the domain of software engineering. To this end, two sentiment detection techniques have been developed for the domain of software engineering. Both of the two techniques were trained and tested on Jira posts. A Jira post can be considered as a document in sentiment analysis. Such a document can contain more than one sentence. None of the techniques were tested on the API reviews posted in Stack Overflow. 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.** We developed a technique in Opiner to automatically detect opinionated sentences from API reviews. 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). We describe the algorithm and compare its performance against the 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

Table 4.1: The Benchmark (Post = Answer + Comment + Question)

Domain	Tags	Posts	Answers	Comments	Sentences	Sentences/Threads
serialize	xml, json	189	68	113	531	66.38
security	auth, crypto	105	34	63	399	49.88
utility	io, file	297	85	204	1000	125
protocol	http, rest	126	49	69	426	53.25
debug	log, debug	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

our benchmark dataset.

4.2 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 labeled 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 labeled as sentence based on their subjectivity (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.

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

In Table 4.1, 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 labeled 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 for the benchmark from Stack Overflow using a two-step sampling strategy (see Section 4.2.1)

2) **Data Preprocessing**: We preprocessed the sampled posts and developed a benchmarking application to label each sentence in the post (see Section 4.2.2)

3) **Manual Labeling of the API aspects**: Total four coders were involved in the labeling of the aspects in the sentences (see Section 4.2.3)

4) **Manual Labeling of the Subjectivity**: Total seven coders were involved in the labeling of the subjectivity of the sentences (see Section 4.2.4)

4.2.1 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 [99]. In total, the 554,917 threads contained 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. In our future work, 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.

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 labeled 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 labeled 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 4.1, 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 [100]. 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 4.1, 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 4.2, 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'.

4.2.2 Data Preprocessing

To assist in the labeling process in the most seamless way possible, we created two web-based survey applications[1]: one to label the aspects in each sentence and another to label the subjectivity 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 subjectivity is hosted at: `http://sentimin.soccerlab.polymtl.ca:48080/`.

Aspect Coding App. In Figure 4.2, 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.

Subjectivity Coding App. In Figure 4.3, 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 subjectivity categories ①, and 2) each sentence is labeled as the subjectivity as shown in ④.

[1]The app is developed using Python on top of the Django web framework.

Table 4.2: 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

Table 4.3: 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 HTTP-Client 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	GSN 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.
General Features	The opinion about the aspect cannot be labeled using any of the above categories	N/A

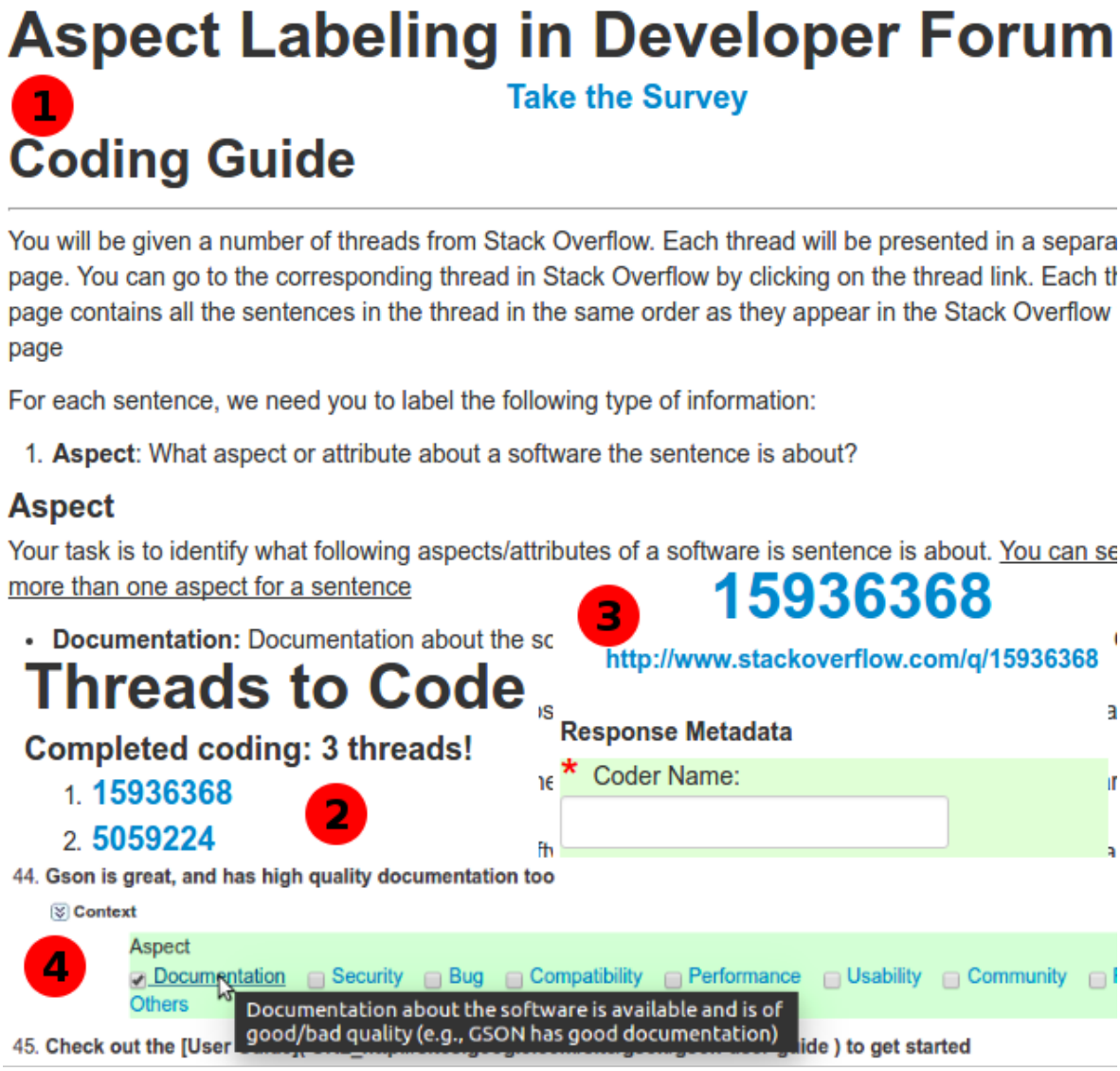

Figure 4.2: Screenshots of the benchmarking app to label aspects in API reviews.

4.2.3 Labeling of API Aspects in Sentences

The labeling of each sentence was followed by a coding guide as shown in Figure 4.2, where, each aspect was formally defined with an example (see circle ① in Figure 4.2). In Table 4.3, we present the definition and example used for each aspect in the coding guide. Besides, during the labeling, each check box corresponding to the aspect showed the definition as a tooltip text (see circle ④ in Figure 4.2).

4.2.3.1 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 chapter, nor were they associated with the work or the

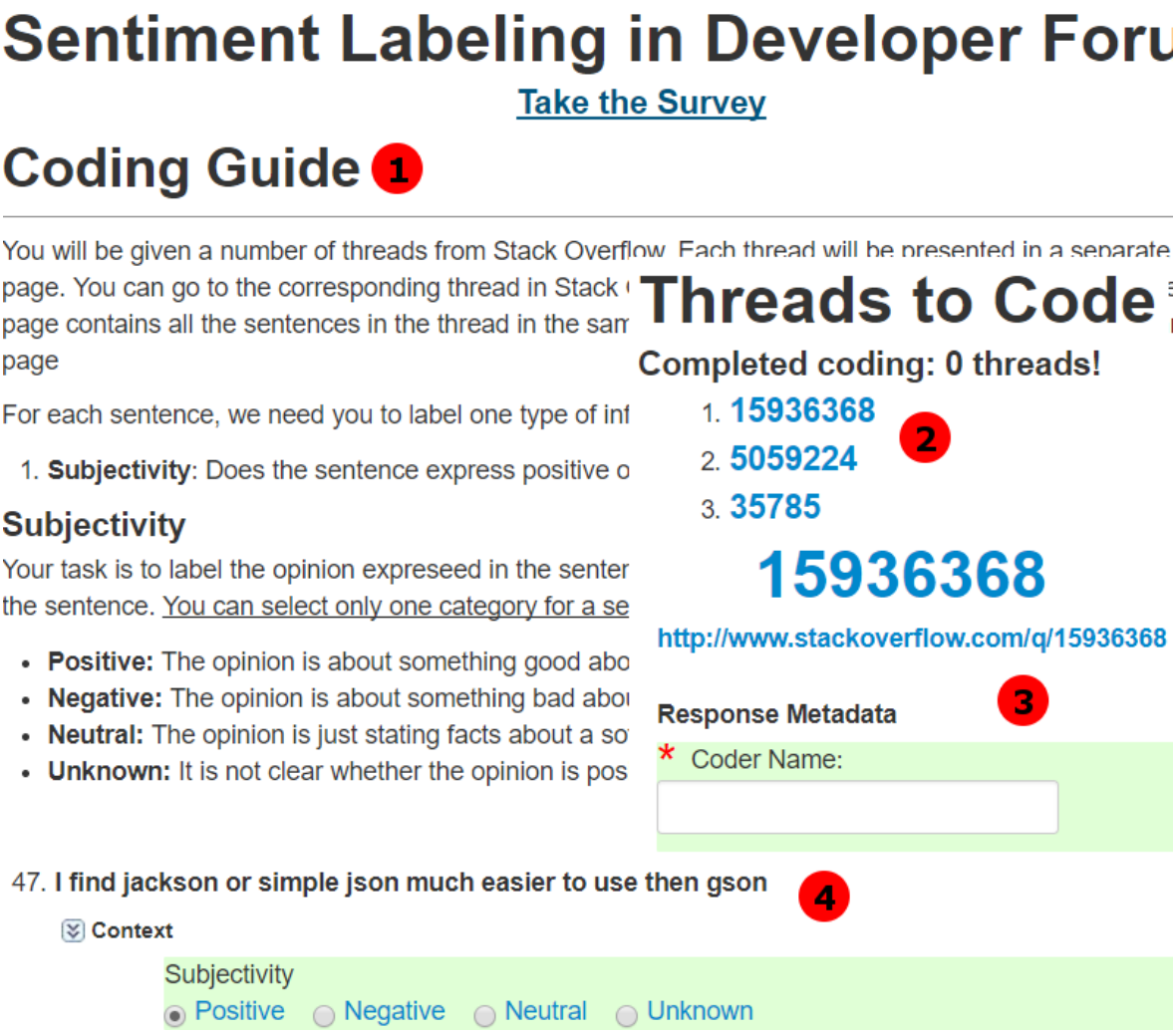

Figure 4.3: Screenshots of the benchmarking app to label sentiment subjectivity in API reviews.

project.

C1. The first coder was Gias Uddin (the author of this thesis). The first coder 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 Postdoc 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.

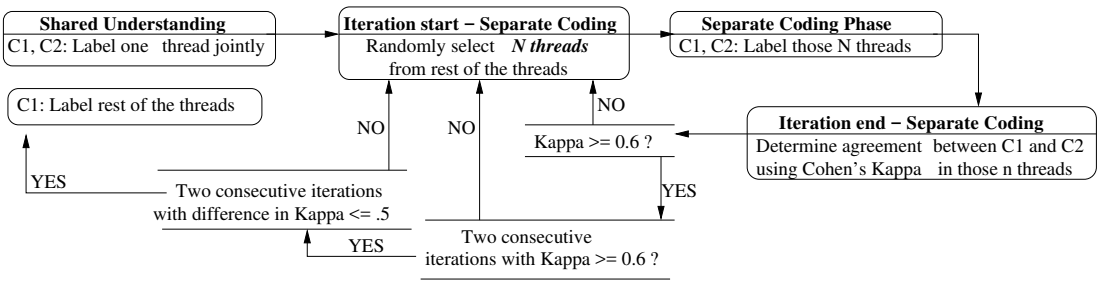

Figure 4.4: The steps used in the labeling process of the benchmark (C1 = First Coder, C2 = Second Coder)

4.2.3.2 Labeling Process

The coding approach is closely adapted from Kononenko et al. [80]. 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 labeled.
- **P2. Code Completion.** The first coder then labeled the rest of the 57 threads.
- **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 coding was 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 labeled 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 4.2, we show screen shots of the user interface of the application.
2) **Separate Coding.** The two coders separately labeled 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 4.4) and the change in agreements between *subsequent* iterations were below 5%, i.e., further improvements might not be possible. In Table 4.5 we show the number of sentences labeled in each iteration by the two coders and the level of agreement (Cohen Kappa value) for the iteration. 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 labeled as an aspect if it contains clear indicators (e.g., specific vocabularies) of the aspects and the focus of the message contain 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 aim to classify the sentences automatically.

P2. Code Completion.

The first coder then completed the coding of the rest of 57 threads. The first coder labeled 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 $\kappa = 0.72$ (percent agreement 96.84%) between C1 and C3, and Cohen $\kappa = 0.804$ (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).

4.2.3.3 Analysis of the Disagreements

In this section, we discuss the major themes that emerged during the disagreements between the coders. In Table 4.6, 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 second coder labeled the following sentence as 'Security' initially: "The J2ME version is not thread safe". However, he agreed with the first coder that it should be labeled as

Table 4.4: Interpretation of Cohen κ values [4]

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 4.5: 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

'Performance', because 'thread safety' is normally associated with the performance-based features of an API. The first coder initially labeled 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 labeled 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 labeled 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 labeled 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 labeled 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 labeled 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.

4.2.4 Labeling of Subjectivity in Sentences

In this section, we describe the labeling process that we followed to code the subjectivity 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:

Table 4.6: 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

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

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. It is not clear whether the opinion is positive, negative, or neutral.

The labeling of each sentence was followed by a coding guide as shown in Figure 4.3, where, each subjectivity category was formally defined with an example (see circle ① in Figure 4.2). We used the same description for each subjectivity category as provided above. Besides, during the labeling, each checkbox corresponding to the subjectivity category showed the definition as a tooltip text.

4.2.4.1 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 chapter, nor were they associated with the project.

C1. The first coder was Gias Uddin (the author of this thesis). The first coder 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 Postdoc 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. C3 coded 53 of the threads. Each of C4-C6 coded 24 threads.

Table 4.7: The percent agreement between the coders in their labeling of the subjectivity of the sentences

Coder	C2	C3	C4	C5	C6	C7
C1	76%	40%	45.4%	71.5%	66%	34.6%
C2			46.3%	73.5%	64.8%	36.8%
C3			32.9%	42.2%	39%	16%
C4						25.1%
C5					73%	31.7%
C6						36.1%

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#.

4.2.4.2 Labeling Process

We followed the established labeling process in sentiment detection [101]. For each sentence, we collected the labels until we reached a majority vote. For a given sentence, there can be two scenarios:

1) We find a majority agreement from the first two/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 two/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 C7 to label the sentence. C7 was not given access to any of the labels of the other coders, to ensure that he was not biased by any particular coder. C7 was given access to all the contents of threads where the sentences were found, to ensure that his assessment of the subjectivity 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.

In Table 4.7, we show the percent agreements between the coders of the study. Percent agreements are commonly used to report the inter-coder agreements in sentiment labels. The agreement was the highest (76%) between the coders C1 and C4, followed by 73.6% between C4 and C5. The coder C7 was assigned to label the sentences where a majority was not reached previously. That means, the subjectivity of those sentences is not easy to understand. This difficulty is also reflected in his agreement level with the other coders (maximum 36.8% between C4 and C7).

4.2.4.3 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:

(1) **Sarcasm.** A sentence containing both subjectivity 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 labeled it as: C1(n), C3(o), C5(p). The coder C7 labeled it as positive, which we believe is the right label given that the opinion towards the 'library' is indeed positive. (2) **Convoluted Subjectivity.** 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 (especially not leveraging Java generics), there is a fork called CoffeeDOM which exactly addresses these aspects and modernizes the JDOM API. . . "*. The first three coders labeled it as: C1(p), C3(n), C5(o). C7 labeled it as positive, which we believe is correct since the sentiment towards the target which is the 'forked' API CoffeeDOM is positive.

4.3 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.

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

Figure 4.5 shows the overall distribution of the aspects in the benchmark. The top pie chart shows the overall distribution of all the implicit aspects against the two other aspects (Other General Features and Only Sentiment). Majority (56.8%) of the sentences are labeled 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 in the benchmark are labeled as 'Other General Features'. Among those labeled as 'Other General Features' only 1% also have one or more of the implicit aspects labeled.

In total, 95.2% of the sentences are labeled as only one aspect. Among the rest of the sentences, 4.6% are labeled as two aspects and around 0.2% are labeled 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 labeled as three aspects (performance, usability, bug): "HTML parsers . . . Fast .. Safe .. bug-free . . . Relatively simple . . . . . ".

Out of the sentences labeled as at least one of the nine implicit aspects, 53.3% of the sentences are labeled as 'Usability', followed by the aspect 'Performance' (12.9%). The two aspects (Performance and Usability) captured 58% of all of the sentences. While detecting the aspects in hotel and restaurant reviews for Google Research, Blair-Goldenshon et al. [26] observed that certain aspects

Table 4.8: 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	auth, crypto	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**
Compatible	6.5	16.1	2.2	14.0	6.5	8.6	**18.3**	15.1	12.9
Document	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

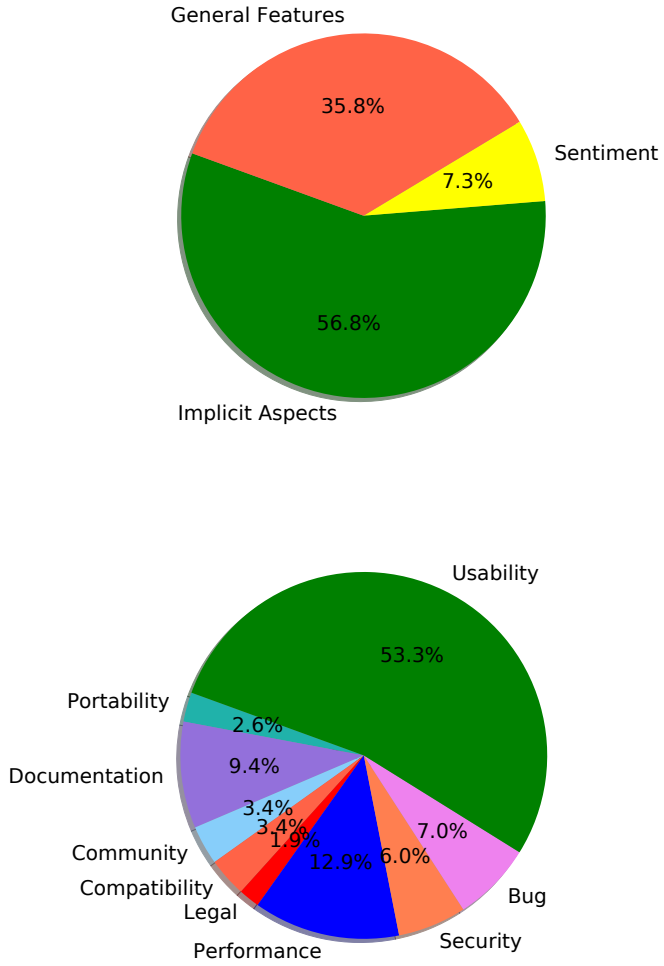

Figure 4.5: The distribution of aspects in the benchmark. The top pie chart shows the distribution of the implicit vs other aspects (OnlySentiment and Other General Features) in the benchmark. The bottom pie chart shows the distribution of each individual aspect among the implicit aspects.

(e.g., food, service) exhibited the Zipfian distribution, in that they were present in those reviews in a much greater proportion than the other aspects (e.g., decor). We observe that the discussion of API aspects in forum posts also follow a Zipfian distribution, i.e., some aspects (e.g., Usability, Performance) are discussed much more often than others. This distribution did not change when we applied our aspect classifiers on all the threads tagged as Java+Json (see Chapter 5).

The sentences labeled as the 'Usability' aspect exhibit 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. We observed that the two aspects 'Community' and 'Compatibility' were not reported as frequently as 'Usability' because the two aspects contained themes that were not directly related to the basic usage of an API. For example, discussions about 'Compatibility' arose when developers already familiar with a framework wanted to find an API compatible with the framework. For

example, one opinion labeled as 'Compatibility': "I would advise against Spring MVC for REST because it is not JAX-RS compliant." The next sentence was also labeled 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. "

In Table 4.8, we show the distribution of aspects across the 18 tags from which the benchmark was constructed. Four of the implicit aspects (Performance, Usability, Bug, and Documentation), along with the other two aspects (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 and write). The discussions related to 'Security' aspect are mostly found in the two tags related to security (authentication and cryptography). The discussions about community support have the largest concentration in the posts tagged as the framework-related tags (eclipse and spring). Similarly, the compatibility of the APIs in the frameworks are also discussed the most compared to the other tags in the benchmark. The aspect 'Legal' had 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 discussion around the two popular Java UI APIs (Swing and AWT) in those posts. The Java AWT API relies on native operating system for look and feel and swing avoids that.

In Table 4.9, we show the distribution of the themes in the sentences labeled 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.

A number of major themes are prevalent in the discussions of the aspects in our benchmark dataset:

Usability. The designing of an API is the major theme in the sentences labeled as the aspect 'Usability'. The another major theme for the 'Usability' aspect was the usage of the API, such as, whether it is easy or difficult to use, and so on.

Performance. The two major themes dominating the sentences labeled as the 'Performance' aspect are the 'speed' and the 'concurrency' of the provided API features. The scalability of the provided feature, and the memory footprint of the technique are the next major themes in the sentences labeled as the 'Performance' aspect.

Security. Similar to the 'Usability' aspect, the discussions about the 'Security' aspect in the sentences revolve around the 'design' and 'usage' of the API. However, for the 'Security' aspect, the design issues focus on the development of a secure API, such as, specification of access control policies, role based access control, authentication, authorization, and so on.

Compatibility. The sentences labeled as the 'Compatibility' aspect exhibit major theme as the 'compliance' of an API in a framework (e.g., which API is compatible with the Spring frame-

Table 4.9: The distribution of themes in the aspects

↓Theme|Aspect→	Performance	Usable	Security	Compatibility	Portability	Document	Bug	Legal	Community	Other General Features	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

work to offer JSON parsing) or the support/availability of a feature in different versions of an API.

Portability. The 'interoperability' of an API across different computing platform is the major theme in the sentences labeled as the 'Portability' aspect. 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.

Documentation. The discussion of both formal (e.g., Javadoc) and informal (e.g., blogs, other forum posts) resources comprise the major themes in the sentences labeled as the 'Documentation' aspect.

Bug. The usage and design of an API are major themes in discussion of 'Bugs'. For example, how Null Pointer should be designed and how they operate are discussed extensively in multiple posts in the benchmark.

Legal. The licensing and costing issues of an API are major sources of discussions in the sentences labeled as the 'Legal' aspect.

Community. The discussions about 'Community' support are mainly about how supportive/active the developers using an API, or about the developers *owning* (e.g., authors, maintainers) an API.

Other General Features. The sentences labeled as 'Other 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, and so on.

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

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

In Figure 4.6, we show the distribution of the positive and negative opinions for each aspect. The percentage number beside each aspect shows how many of the sentences that are labeled as the aspect are opinionated (i.e., either positive or negative). For the four aspects (Usability, Performance, Legal, and Bug), more than 40% of the sentences are labeled as opinionated. For other aspects, at least 32% of the sentences are labeled as opinionated. While considering the proportion of positive and negative opinions for a given aspect, developers are more positive in the discussions around 'Documentation', followed by 'Usability', 'Compatibility' and 'Security'. While 41.81% of sentences that are labeled as the 'Performance' aspect, the proportion of positivity and negativity in those opinions is almost equal, i.e., performance may be a pain point during the usage an API. With the same distribution (41.38%) of opinionated sentences for 'Usability', developers expressed

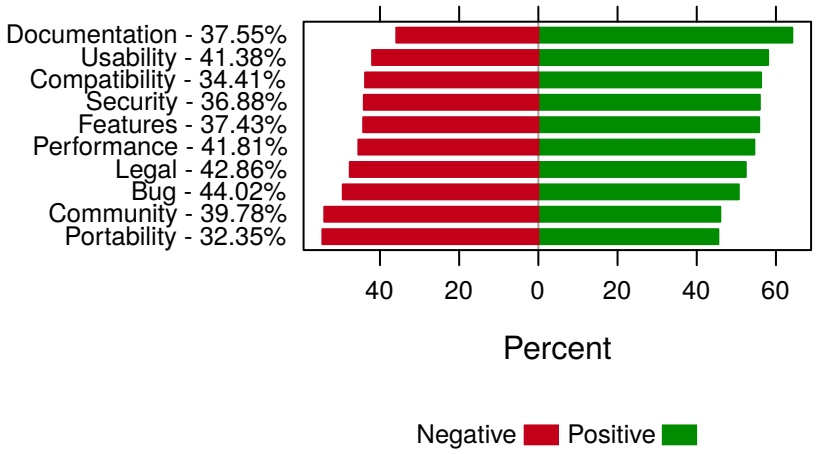

Figure 4.6: Distribution of Opinions across aspects in benchmark

more positivity in our benchmark. For the two implicit aspects (Community and Portability), we observe more negative opinions than the positive opinions. 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.

In Table 4.10, we show 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 (green bar for positive opinions and red bar negative opinions). The aspect 'Performance' was discussed favorably across the themes related to usage, speed, reliability and scalability issues regarding the APIs. However, developers were not happy with respect 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. However, the developers were mostly positive about the design of an API with regards to the 'Usability' of the API. The themes related to the 'Security' were discussed mostly favorably, with relatively minor complaints with regards to the usage and design of the API while developing security algorithms. The themes related to the 'Legal' were mainly about the costing, licensing support of the APIs. Because the discussed APIs in our benchmark were predominantly open-source, both of the themes offered worry-free adoption (e.g., cost-free, open-source licensing). This was evident in the sentiment distribution of those themes across the aspects. 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 community support was not available. However, they were mostly happy with regards to the compatibility of those APIs in diverse computing resources. The abundance of computing resources and operating systems offer significant challenges to the

Table 4.10: The distribution of polarity for themes across aspects (Red and Green 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										
Debugging										
Formal Docs										
Activity										
Concurrency										
Environment										
Compliance										
Ownership										
Resource										
Interoperability										
Reliability										
Footprint										
License										
Scalability										
Benchmarking										
Cost										
Vulnerability										
Extensibility										
Communication										
Building										
Authority										

developers while deciding on how to make their software compatible across all such resources. The availability of such resources (themes related to compliance and resource) were discussed mostly unfavorably.

In Table 4.11, we show the distribution of opinions around an aspect across the nine domains in our benchmark. For each aspect and each domain, we show the overall presence of positive and negative opinions. The percentage beside each aspect name in Table 4.11 shows how much of the percentage of sentences labeled as that aspect contained opinions with positive or negative sentiment. The bar for each aspect shows the relative proportion of positive and negative opinionated sentences for each domain. The concentration of opinions is more for some aspects than other aspects. Performance and Usability both have much more negative opinions than positive opinions for domains 'Protocol' and 'Debug'. Developers in the dataset complained about the performance of a program while debugging. 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 'Bug' mostly contained negative opinions because developers discussed about bugs while using an API and thus such discussions often contained frustrations. For example, one developer was not happy with the Null pointer exception in Java while tracing the root cause of a bug: "NPE is the most useless Exception in Java, period. ". The aspect 'Legal' mostly contained positive opinions because the APIs discussed in our dataset were open source and unlike special licenses (e.g., GPL) such licenses (e.g., MIT) promote worry-free usage of the API. 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. Developers did not have any complaints about the Legal aspects of the APIs from the Text domain, although they were not happy with their buggy nature.

In Figure 4.7, we show the overall presence of opinions per domain. The percentages beside each domain show the overall distribution of opinions (positive and negative) sentences in the domain. The bar for each domain further shows the relative proportion of positivity versus negativity in those opinions. Except two domains (Debug and Framework), developers expressed more positivity than negativity in all domains except two (Framework and Debug). In general, we observe greater concentration of opinions in the benchmark dataset than the presence of code examples or just code terms (e.g., mention of an API class name). 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. For example, in Table 4.12, 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. 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.

Table 4.11: Distribution of opinions across aspects and domains (Red and Green bars represent negative and positive opinions, resp.)

↓Aspect|Domain→	Security	Database	Serialize	Widget	Text	Utility	Protocol	Debug	Framework
Performance - 41.81%									
Usability - 41.38%									
Security - 36.88%									
Bug - 44.02%									
Community - 39.78%									
Compatibility - 34.41%									
Documentation - 37.55%									
Legal - 42.86%									
Portability - 32.35%									
General Features - 37.43%									

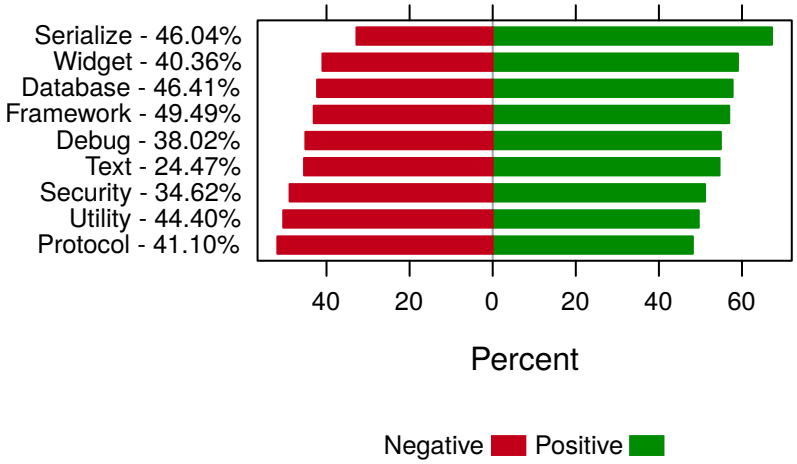

Figure 4.7: Opinion distribution across domains (red and green bars denote negative and positive opinions, respectively.

Table 4.12: 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

Developers expressed more opinions towards the discussions of specific API aspects (e.g., Usability, Performance) in our dataset. Diverse themes of those aspects emerged during the analysis of those discussions (e.g., concurrency, ownership, and so on.). The concentration of opinions in the dataset is much more than the presence of API code terms/snippets. Depending on the domain of the APIs (e.g., serialization APIs for XML or JSON), the distribution of opinions for a given aspect vary (e.g., Security aspect is almost fully positive reviewed for APIs offering HTTP and REST protocol-based features, but are less favored in the framework-related discussions).

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

We observed three types of stakeholders in our dataset:

User: who planned to use the features offered by an already available API or who responded to queries asked by another user in the post.

Author: who developed or authored an API.

Table 4.13: Percent distribution of aspects discussed by the 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	

User Turned Author: who developed a wrapper around one or more already available APIs and offered the wrapper to others in the post.

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

(1) Aspect: In Table 4.13, we show the percent distribution of the aspects in the discussions of the three types of stakeholders in our benchmark. We computed the distribution as follows:a) For each stakeholder, we identify the posts which the stakeholder posted. b) For each post, we collect the list of labeled aspects and the frequency of each aspect. c) For each stakeholder, we count the total number of times an aspect is found in the post which the stakeholder has posted. While the API users discussed about each of the aspects, we only observed a few number of those aspects were discussed by the authors. The majority of the responses of the authors were about the usability of the API, followed by specific feature support. The users who decided to develop an API during their communication in the posts, were interested in the improvement of the features, usability, and fixing of the bugs.

(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 (9.1%), e.g., "Give boon a try. It is wicked fast". Besides promotion, we observed the following types of interactions between users and authors that did not provide any specific promotions and 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) Intents/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. Therefore, users were more positive towards the provided suggestions. Two of the users explicitly mentioned that they used the API in the provided suggestions. We believe the usage rate per suggestion could be much more higher, if we considered more subtle notions of usage (e.g., "I will have a try on it"). 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. Consider the following comment:"Jackson sounds promising. The main reason I mention it is that the author, Tatu Saloranta, has done some really great stuff (including Woodstox, the StAX implementation that I use)."

> API authors and users provide opinions to promote and suggest 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.

4.4 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 4.4.1).
We investigate a rule-based technique based on topic modeling to automatically detect the 11 API aspects.

Supervised Aspect Detection (Section 4.4.2).
We investigate a suite of supervised classification techniques to automatically detect the 11 API aspects.

We describe the algorithms below.

4.4.1 Rule-Based Aspect Detection

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

Table 4.14: Performance of topic-based aspect detection (C = Coherence, N = Iterations, JT, JF = Jaccard, T for positives, F negatives. P = Precision. R = Recall. F1 = F1 score. A = Accuracy)

Aspect	Topic	C	N	JT	JF	P	R	F1	A
Performance	faster, memory, thread, concurrent, performance, safe, robust	0.52	100K	0.045	0.007	0.270	0.540	0.360	0.852
Usability	easy,works,work,uses,solution,support,example,need,simple	0.33	300K	0.026	0.011	0.501	0.436	0.467	0.683
Security	encrypted,decryption,bytes,crypto,security,standard,salt,cipher	0.54	200K	0.054	0.007	0.152	0.669	0.248	0.854
Bug	exception,nullpointerexception,throw,thrown,getters,error,bugs	0.55	300K	0.038	0.005	0.151	0.429	0.223	0.875
Community	years,explain,downvote,community,reputation,support	0.61	200K	0.045	0.005	0.093	0.523	0.158	0.884
Compatibility	android, reflection,equivalent,compatible,engine,backend	0.53	300K	0.040	0.008	0.053	0.452	0.094	0.821
Documentation	answer,example,question,article,link,tutorial,post	0.53	300K	0.044	0.012	0.137	0.509	0.215	0.795
Legal	free,commercial,license,licensing,open-source,illegal,evaluate	0.50	200K	0.080	0.004	0.093	0.780	0.166	0.913
Portability	windows,linux,platform,unix,redis,android,compact	0.53	200K	0.090	0.007	0.058	0.557	0.106	0.854
OnlySentiment	thanks,good,right,little,correct,sorry,rant,rhetorical,best	0.61	100K	0.028	0.005	0.219	0.371	0.276	0.850
General Features	array,write,need,implementation,service,server,size,lines	0.49	200K	0.014	0.012	0.370	0.203	0.262	0.571

4.4.1.1 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.) [102]. We used LDA (Latent Dirichlet Allocation) [102] 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 [103], and so on.

To determine representativeness of topics for a given aspect, we use the topic *coherence* measure as proposed by Röder et al [104].[2] 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 4.14, the third column (C) 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 labeled as the aspect and remove the stopwords[3] (2) We apply LDA on the sentences labeled as the aspect repeatedly until the coherence value of the LDA model no longer increases. In Table 4.14, 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 4.14, we show 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 [107] between the topic-words (T) of a given aspect and the words in the sentence (S) using the following equation:

$$J = \frac{Common(T, S)}{All(T, S)} \tag{4.9}$$

(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.

4.4.1.2 Evaluation

In Table 4.14, the last four columns show the performance in aspect detection for each aspect using the rule-based technique (Precision P, Recall R, F1 score $F1$, and Accuracy A). The precision

[2]We used gensim [105] to produce the topics and its coherencemodel to compute c_v coherence.

[3]We used the NLTK tokenization algorithm and the stopwords provided by Scikit-learn library [106].

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:

Specificity. While the keywords in a topic describing an aspect can be specific to the description of the aspect, the detection of an aspect may not be always possible using those keywords. For example, the keyword 'fast' may denote that the sentence can be labeled as 'Performance'. However, the performance of an API can be discussed using diverse themes, e.g., its adoption in a multi-threaded environment, and so on. One probable solution to this would have been to relax the constraints on the number of topics and on the number of words describing a topic. For example, instead of the top 10 words, we could have used more than 10 words from each topic description. However, such constraint relaxation can introduce the following problems , i.e., the increase of generic keywords in a topic description.

Generality. Some of the keywords in the topics were common across the aspects. The generic keyword 'standard' is found in the topics of both 'Usability' and 'Security', which does not necessarily denote the presence of any of the aspects. Even with 10 words, we experienced noise in the topics. For the aspect Bug, one of the words was 'getters' which does not indicate anything related to the buggy nature of an API. 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. The list do not include 'getters' as a stopword, because it is more specific to the domain of software engineering. In our future work, we will analyze the techniques to automatically identify domain-specific stopwords [108] 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 4.14, the two columns JT and JF show the average Jaccard index for all the sentences labeled 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.

> 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.

4.4.2 Supervised Aspect Detection

In this section, we describe the design and evaluation of 11 supervised aspect detectors. To train and test the performance of the classifiers, we used 10-fold cross-validation.

4.4.2.1 Algorithm

Because the detection of the aspects requires the analysis of textual contents, 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.[4]

We applied the SVM-based classification steps as recommended by Hsu et al. [109] 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.[5]

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

Imbalanced. As recommended by Chawla [110], 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. To produce a balanced subset of the labels for a given API aspect from the benchmark dataset, we used the following steps [110]. For each aspect labeled in our benchmark dataset (1) We transform the labels as 1 for the sentences where the aspect is labeled. We transform the labels of all other sentences as 0, i.e., the sentence is not labeled as the aspect. (2) We determine the label with the less 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.

From this subset, we picked our test and train set for 10-fold cross validation. In our future work, we will investigate different balancing strategies, such as, oversampling of the minority class using the SMOTE algorithm [111], and so on.

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 labeled 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 [112].

[4] We used the SGDClassifier of Scikit [106]

[5] We computed hyper parameters using the GridSearchCV algorithm of Scikit-Learn

(3) As recommended by Hsu et al. [109], 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[6], which keeps the ratio of target values similar across the folds.

(5) We took the average of the precision, recall, F1-score, accuracy 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). We decided to use F1-score instead of accuracy to account for the fact that the dataset was highly imbalanced for most of the aspects (i.e., accuracy would be above 95% with a baseline that would have labeled each input to the majority class).

4.4.2.2 Evaluation

In Table 4.15, we report the performance of the final classifiers for each aspect. Except for one aspect (Security), SVM-based classifiers were found to be the best. In Table 4.15, we show the performance of the final classifiers for each aspect for both of the dataset settings, i.e., balanced and imbalanced. Except for one aspect (Performance), the precisions of the classifiers developed using the balanced dataset are better than the same classifiers developed using the imbalanced dataset. 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 among all of the other classifiers. 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 +250). For aspects (Community, Legal, Compatibility), the number of sentences labeled 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 (five aspects and others), followed by bigrams (four aspects) and trigrams (Compatibility). 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 for each aspect outperformed the topic-based algorithm for two primary reasons: (a) The topics were based on words (unigrams) and as we observed, bigrams and trigrams

[6]We used the Stratified KFold implementation of sklearn

Table 4.15: Performance of the supervised API aspect detectors

				Precision		Recall		F1 Score		Accuracy	
Aspect	**Ngram**	**Classifier**	**Dataset**	**Avg**	**Stdev**	**Avg**	**Stdev**	**Avg**	**Stdev**	**Avg**	**Stdev**
Performance	Bigram	SVM	Imbalanced	0.778	0.096	0.455	0.162	0.562	0.125	0.948	0.012
			Balanced	0.680	0.060	0.796	0.103	0.732	0.072	0.711	0.070
Usability	Bigram	SVM	Imbalanced	0.532	0.045	0.749	0.101	0.620	0.055	0.710	0.040
			Balanced	0.649	0.033	0.780	0.091	0.707	0.052	0.680	0.048
Security	Unigram	Logistic	Imbalanced	0.775	0.272	0.578	0.174	0.602	0.160	0.969	0.025
			Balanced	0.876	0.081	0.786	0.111	0.823	0.079	0.835	0.070
Community	Unigram	SVM	Imbalanced	0.391	0.323	0.239	0.222	0.256	0.204	0.974	0.011
			Balanced	0.641	0.211	0.534	0.226	0.572	0.198	0.615	0.174
Compatibility	Trigram	SVM	Imbalanced	0.500	0.500	0.078	0.087	0.133	0.144	0.981	0.003
			Balanced	0.619	0.086	0.664	0.112	0.637	0.082	0.622	0.091
Portability	Unigram	SVM	Imbalanced	0.629	0.212	0.629	0.223	0.608	0.190	0.988	0.006
			Balanced	0.843	0.065	0.757	0.248	0.762	0.188	0.800	0.100
Documentation	Bigram	SVM	Imbalanced	0.588	0.181	0.428	0.170	0.492	0.175	0.952	0.016
			Balanced	0.729	0.066	0.826	0.104	0.772	0.072	0.759	0.072
Bug	Unigram	SVM	Imbalanced	0.573	0.163	0.499	0.163	0.507	0.118	0.960	0.013
			Balanced	0.849	0.106	0.742	0.106	0.786	0.080	0.799	0.074
Legal	Unigram	SVM	Imbalanced	0.696	0.279	0.460	0.180	0.523	0.188	0.991	0.004
			Balanced	0.877	0.133	0.820	0.140	0.835	0.101	0.840	0.102
OnlySentiment	Bigram	SVM	Imbalanced	0.613	0.142	0.429	0.142	0.501	0.139	0.935	0.017
			Balanced	0.814	0.101	0.702	0.106	0.750	0.090	0.769	0.079
General Features	Unigram	SVM	Imbalanced	0.610	0.067	0.665	0.055	0.635	0.056	0.712	0.048
			Balanced	0.694	0.026	0.619	0.072	0.653	0.048	0.674	0.035

are necessary to detect six of the aspects. (b) The topics between aspects have one or more overlapping words, which then labeled 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 4.15, 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% accuracy 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 labeled 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 labeled 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 data labeled data. In the future we will investigate this issue with different settings, e.g., using more labeled data.

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 classifiers did not perform well to detect 'Community' and 'Compatibility' due mainly to the implicit nature of the aspects in the opinions and the need to analyze surrounding sentences as contexts. The performance of the classifiers using the balanced dataset was superior to the classifiers using the imbalanced dataset for all aspects (except 'Performance').

4.5 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*; c) *hyperlinks*[7]; and d) *natural language text* representing the rest of the content. (3) We tokenize the text and tag each token with its part of speech.[8] (4) We detect individual sentences in the *natural language text*.

2) **Opinionated Sentence Detection.** We detect sentiments (positive and negative) for each of

[7]We detect hyperlinks using regular expressions.
[8]We used the Stanford POS tagger [113].

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 detect opinionated sentences by fusing two algorithms from literature. We present two algorithms to resolve API mentions and to associate the opinionated sentences to the API mentions. We discuss the technique to detect opinionated sentences in Section 4.5.1, and the algorithms to associate those opinionated sentences to the API mentions in the forum posts in Section 4.5.2. 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 [49]. In Section 4.5.3, we briefly discuss the system architecture of Opiner to support automatic mining and categorization of API reviews.

4.5.1 Automatic Detection of Opinions (RQ5)

In Opiner, we detect opinionated sentences by heuristically combining the results of two widely used sentiment detection techniques, Sentistrength [114], and Dominant Sentiment Orientation (DSO) [115]. Originally proposed by Hu and Liu [115] to mine and summarize reviews of computer products, this algorithm was also applied to mine customer reviews about local services, e.g., food, restaurants, etc. [26]. Sentistrength [114] has been used to detect sentiments in various software engineering projects [51].

4.5.1.1 Algorithm

Given as input a sentence, we detect its subjectivity 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 subjectivity 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 algorithm that has the most coverage of sentiment vocabularies in the sentence. If they have the same coverage, we apply the following rule.

   c) If both algorithms have similar coverage of sentiment words in the sentence and they still disagree, we then 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.

Table 4.16: Accuracy analysis of the opinionated sentence detection techniques

Sentiment Detector	TP	FP	TN	FN	Precision	Recall	F1-Score	Accuracy
Opiner	2311	874	195	771	0.726	0.750	0.738	0.604
Sentistrength-SE	2415	754	143	839	0.762	0.742	0.752	0.616
Sentistrength	2084	944	210	913	0.688	0.695	0.692	0.553
Stanford	1589	666	375	1521	0.705	0.511	0.592	0.473

The detailed steps describing DSO is described in the paper [115]. We are not aware of any publicly available implementation of the algorithm. We took cues from Blair-Goldensohn et al [26], 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.

1) *Detect potential sentiment words.* We record the adjectives in 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. 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 [116], 123 words from the AFINN dataset [117]. 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 [118] is a lexicon database, where each Wordnet [119] 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'.

Table 4.17: Percent agreement between the sentiment classifiers

Tool	SentistrengthSE	Sentistrength	Stanford
Opiner	68.76%	65.82%	48.20%
SentistrengthSE	–	67.38%	50.05%
Sentistrength	–	–	47.19%

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 [120], 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 subjectivity. To apply this technique, we used the adverbs provided in the MPQA lexicons.

Coherency This technique collects adjectives that appear together via conjunctions (we used 'and' and 'but') [121]. 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 mark 'thread-safe' as having a positive sentiment.

Both of the two approaches are commonly used to develop and expand domain specific vocabularies [121]. 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.

4.5.1.2 Evaluation

We developed the sentiment detection engine in Opiner in January 2017, motivated by the findings that cross domain sentiment detection techniques are not efficient enough to detect sentiments in software engineering. Similar findings were also independently observed by other researchers [122, 123]. Since then two sentiment detection tools have been developed and published online to address the needs of software engineering. We compare the performance of Opiner sentiment detection technique against one of those tools, SentistrengthSE [124]. The other tool, Senti4SD [125] could not be run on our benchmark dataset on numerous attempts, due to issues ranging from memory exception to file not found. We also compare the Opiner sentiment detection engine against two cross-domain sentiment detection techniques, Sentistrength [114], and Stanford Sentiment [126]. The Stanford Sentiment detection tool showed more than 80% precision on the movie reviews.

In Table 4.16, we show the performance of the four tools in our benchmark dataset. We compared the performance on the sentences where the majority of the agreement is one of the three labels,

positive, negative, and neutral. The SentistrengthSE tool performed the best among the four tools. The SentistrengthSE tool outperforms Opiner by precision (76.2% vs 72.6%), but Opiner shows better recall (75% vs 74.2%). The Stanford sentiment detection tool offers comparable precision (70.5%), but the accuracy (47.3%) is much lower than other tools. The major reason for this low accuracy for the Stanford tool is its aggressive labeling of the sentences as false negative (1521).

In Table 4.17, we show the percent agreement between the tools to detect the sentiments. The highest agreement is observed between Opiner and SentistrengthSE (68.76%) and the lowest is between Sentistrength and Stanford (47.19%). As Jongeling et al. [123] observed, cross domain sentiment detection tools show low agreement among themselves with regards to the detection of sentiments in software engineering. The higher agreement between Opiner and SentistrengthSE is encouraging and shows the needs of sentiment detection tools more specific to software engineering. Our immediate future work is focused on improving the sentiment detection in Opiner by combining it with other tools developed for software engineering, such as SentistrengthSE.

The sentiment detection engine in Opiner is a fusion of two existing algorithms from literature (Sentistrength [114] and Dominant Sentiment Orientation (DSO) [115]). The fused classifier leverages domain specific sentiment lexicons as well as sentiment lexicons used in other domains. The engine offers performance better than two cross domain sentiment detectors (Stanford [126] and Sentistrength). However, the precision of Opiner sentiment detection engine is slightly lower than the other software domain specific sentiment detector, SentistrengthSE [124]. Our future work will investigate techniques to improve the sentiment detection in Opiner by analyzing results from the other tools.

4.5.2 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.

4.5.2.1 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.

Table 4.18: Statistics of the Java APIs in the database.

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

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 [9] and Maven central [12].[9] 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 Javase official tutorial [127]). In Table 4.18, we show the summary statistics of the API database.

Step 2. Name and Hyperlink Preprocessing.

We preprocess each API name to collect representative token 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 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

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

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 [107]. 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 *closest* opinion to an API as follows. Consider the example, 'I like the object conversation in Gson, but I like the performance of Jackson'. In this sentence, we associate Gson to the opinion 'I like the object conversion in Gson' and Jackson to 'but I like the performance of Jackson'. We do this by splitting the sentence into parts based on the presence of sentiment words (e.g., like) and then associating it to its nearest API that was mentioned after the sentiment word.

• ***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.

4.5.2.2 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 by our algorithm 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.

Table 4.19: Accuracy analysis of the API mention resolver

Resolver	TP	FP	FN	Precision	Recall	F1	Acc
Name	1413	17	118	0.988	0.923	0.954	0.913
Link	301	3	18	0.99	0.944	0.966	0.935
Overall	1714	20	136	0.988	0.926	0.956	0.917

TP = True Positive, FN = False Negative, Acc = Accuracy

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.

We include the manually annotated list of APIs with our benchmark dataset and share those in our online appendix [128].

In Table 4.19, 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.

Table 4.20: Accuracy analysis of the API to opinion association

Rule	TP	FP	TN	FN	Precision	Recall	F1 Score	Accuarcy
Same Sentence	47	2	0	4	0.959	0.922	0.940	0.887
Same Post	23	3	1	3	0.885	0.885	0.885	0.800
Same Conversation	4	2	33	5	0.667	0.444	0.533	0.841
Overall	74	7	34	12	0.914	0.860	0.886	0.850

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 [129].

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 opinion to API mention 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 found, 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 another developer or another 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 that is 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 in Stack Overflow [130]). Thus, we consider true negatives only in the Stack Overflow threads where APIs are mentioned. We established the 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 4.20, we show the performance of our association algorithm. The overall precision is 91.4% with a recall of 86% and an accuracy of 85%. Among the three rules, the 'Same Sentence' rule showed the best precision (95.9%). The 'Same Conversation' rule performed the worst with a precision of 66.7% and a recall of 44.4% and an accuracy of 84.1%. The accuracy of the 'Same Conversation' rule is similar to the other rules, i.e., the rule was effective to fine 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 found to be 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.

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.

4.5.3 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 auto-completion 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

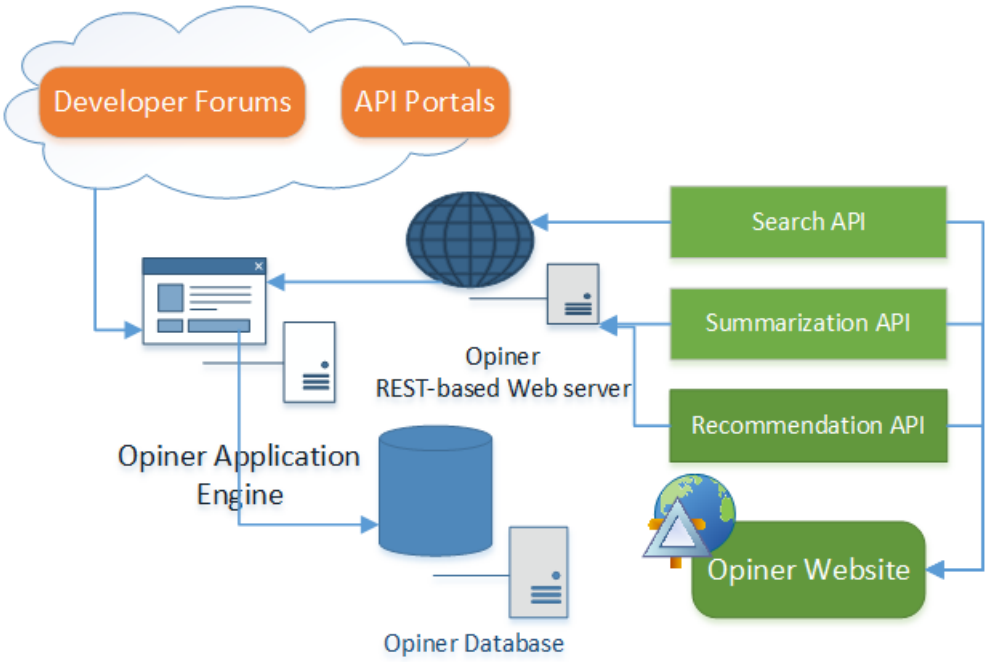

Figure 4.8: The client-server architecture of Opiner

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.

In Figure 4.8, we show the opiner architecture. 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.

4.6 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 [128]. We also share the benchmark we used to resolve API mentions in our online appendix [128].

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. While we observed a high level of agreement in both cases, we, nevertheless, share the complete evaluation corpus and the annotations, as well as the results in our online appendix [128].

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.

4.7 Summary

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 and categorize opinions by aspects. We developed and deployed a tool named Opiner, that supports the development infrastructure to automatically mine and categorize opinions about APIs from developer forums.

Chapter 5

Automatic Summarization of Opinions about APIs from Informal Documentation

Due to the diversity of opinions in online forums about different products (e.g., camera, cars), mining and summarization of opinions for products have become an important but challenging research area [24]. We are aware of no summarization techniques applied to API reviews. Given the plethora of reviews available about an API in developer forums, it can be extremely beneficial to produce an *informative* but *concise* representation of the summaries, so that quick but useful insights can be gathered.

We investigated opinion summarization algorithms for APIs from the following five major categories [2, 24]:

1) **Aspect-based**: positive and negative opinions are grouped around aspects related to the entity (e.g., picture quality). Aspects can be pre-defined or dynamically inferred using algorithms such as topic modeling.

2) **Contrastive**: Contrastive viewpoints are grouped.

3) **Extractive**: A subset of the opinions are extracted.

4) **Abstractive**: An abstraction of the opinions is produced.

5) **Statistical**: Overall polarity is transformed into numerical rating (e.g., star ratings).

We designed the aspect-based and statistical summaries for API reviews. Both aspect-based and statistical summaries are researched in other domains (e.g., camera reviews). In this thesis, we contribute to the two algorithms by introducing the aspects and summary presentation specific to the domain of software engineering and API usage. For example, for statistical summaries, one of our contributions is the calculation of the star rating for an API. We leverage off-the-shelf tools to produce extractive, abstractive, and topic-based summaries and implemented the algorithm proposed by Kim and Zhai [27] to produce contrastive summaries.

We implemented each summarization techniques in Opiner, our online API review summarization engine. In Figure 5.1, we present a screenshot of Opiner. Opiner is developed as a search engine, where developers can search opinions for an API ①. Upon clicking on API, developers can see all the reviews collected about the API both in summarized and original formats ②. A developer can also search for an API aspect (e.g., performance) ③ in Opiner, to find the most popular APIs based on the aspect ④.

We investigated the usefulness of the API review summaries in Opiner based on two research questions:

RQ1: How informative are the various API opinion summarization techniques to the developers?

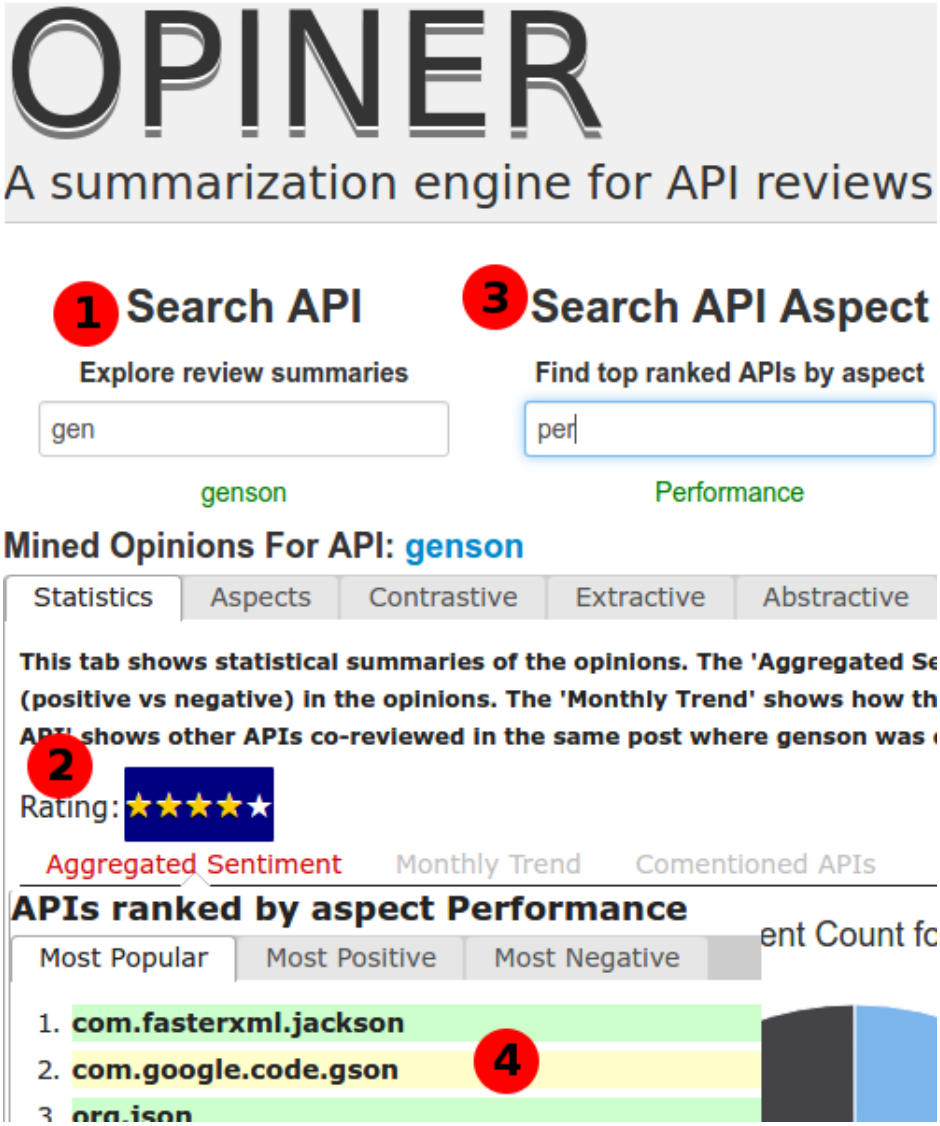

Figure 5.1: Screenshots of Opiner API review search engine

We investigated the informativeness of our proposed summaries against the summaries produced by six off-the-shelf summarization techniques by conducting a study involving 10 professional software engineers. The 10 developers rated the summaries of four different APIs using five development scenarios (e.g., selection of an API, etc.). We found that developers strongly favored our proposed summaries against other summaries (more than 80% vs less than 50% ratings).

RQ2: How useful is an API opinion summarization engine to support development decisions?

We conducted another study where we provided access to our tool to professional software engineers to help them in their selection of an API for two development tasks. We observed that while developers correctly picked the right API with 20-66% accuracy while just using Stack Overflow, they had 100% accuracy while they used Opiner and Stack Overflow together.

Table 5.1: Descriptive statistics of the dataset

Threads	Posts	Answer	Comment	Sentences	Words	Users
3048	22.7K	5.9K	13.8K	87K	1.08M	7.5K
Average	7.46	1.93	4.53	28.55	353.36	3.92

Chapter Organization. We introduce the dataset we used to develop and evaluate the different API review summaries in Section 5.1. We present our proposed statistical summaries of API reviews in Section 5.2, and the aspect-based summaries in Section 5.3. We discuss four summarization techniques we adopted for the domain of API reviews in Section 5.4. We describe a user study to compare the informativeness of the different API reviews summaries (RQ1) in Section 5.5. We investigate the effectiveness of Opiner API review search and summarization engine to assist developers in their API selection tasks in Section 5.6. We discuss the threats to the validity in Section 5.7 and conclude the chapter in Section 5.8.

5.1 Dataset

Our API review dataset was produced by collecting all the opinionated sentences for each Java API mentioned in the Stack Overflow threads tagged as "Java + JSON", i.e., the threads contained discussions and opinions related to the json-based software development tasks using Java APIs. We selected Java APIs because we observed that Java is the most popular Object-oriented language in Stack Overflow. As of April 2017, there were more than 12.3 million threads in Stack Overflow, behind only Javascript (13.5 million). We used JSON-based threads for the following reasons:

Competing APIs. Due to the increasing popularity in JSON-based techniques (e.g., REST-based architectures, microservices, etc.), we observed a large number of competing APIs in the threads offering JSON-based features in Java.

Diverse Opinions. We observed diverse opinions associated to the competing opinions from the different stakeholders (both API users and authors).

Diverse Development Scenarios. JSON-based techniques can be used to support diverse development scenarios, such as, serialization, lightweight communication between server and clients and among interconnected software modules, and growing support of JSON-based messaging over HTTP, using encryption techniques, and on-the-fly conversion of Language-based objects to JSON formats, and vice versa.

In Table 6.1 we show descriptive statistics of the dataset. There were 22,733 posts from 3048 threads with scores greater than zero. We did not considered any post with a negative score because such posts are considered as not helpful by the developers in Stack Overflow. The last column "Users" show the total number of distinct users that posted at least one answer/comment/question in those threads. To identify uniqueness of a user, we used the user_id as found in the Stack Overflow database. On average, around four users participated in one thread, and more than one user participated in 2940 threads (96.4%), and a maximum of 56 distinct users participated in one thread [131].

Table 5.2: Distribution of opinionated sentences across APIs

	Overall			Top Five		
API	**Total**	**+Pos**	**–Neg**	**Total**	**+Pos**	**–Neg**
415	15,627	10,055	5,572	10,330	6,687	3,643
Average	37.66	24.23	13.43	2,066	1,337.40	728.60

From this corpus, we identified all of the Java APIs that were mentioned in the posts. We collected the opinionated sentences about Java APIs using the API opinion technique we developed in our previous chapter (Chapter 4). The technique consisted of the following steps:

1) **Loading and preprocessing** of Stack Overflow posts.

2) **Detection of opinionated sentences**. We used a rule-based algorithm based on a combination of Sentistrength [114]; the Sentiment Orientation (SO) algorithm [115].

3) **Detection of API names** in the forum texts and hyperlinks and **association of APIs to opinionated sentences** based on a set of heuristics.

In Table 5.2, we present summary statistics of the opinionated sentences detected in the dataset. Overall 415 distinct APIs were found. While the average opinionated sentence per API was 37.66 overall, it was 2066 for the top five most reviewed API. In fact, the top five APIs contained 66.1% of all the opinionated sentences in the posts. The APIs are jackson, Google Gson, spring framework, jersey, and org.json. Intuitively, the summarization of opinions will be more helpful for the top reviewed APIs.

5.2 Statistical Summaries of API Reviews

An approach to statistical summaries is the *basic sentiment summarization*, by *simply counting and reporting the number of positive and negative opinions* [2]. In Figure 5.2, ② shows such a summary for API jackson. We propose the following statistical summaries for API reviews:

1) **Star rating.** can provide an overall representation using a rating on a five-star scale. We present a technique to compute a five star rating for an API based on sentiments.

2) **Sentiment Trends.** Because APIs can undergo different versions and bug fixes, the sentiment towards the API can also change over time.

3) **Co-Reviewed APIs.** We visualized other APIs that were mentioned in the same post where an API was reviewed. Such insights can help find competing APIs.

5.2.1 Star Rating

The determination of a statistical rating can be based on the input star ratings of the users. For example, in Amazon product search, the rating is computed using a weighted average of all star ratings. Thus, if there are 20 inputs with 10 five stars, two four stars, four three stars, one two stars, and three one stars, the overall rating can be computed as 3.75. Given as input the positive and negative opinionated sentences of an API, we computed an overall rating of the API in a five star scale by computing the proportion of all opinionated sentences that are positive as follows: $R = \frac{\#Positives \times 5}{\#Positives + \#Negatives}$. For example, for the API Jackson, with 2807 positive sentences and 1465 negative sentences, the rating would be 3.3. In Figure 5.2, ① shows the calcuated five-star rating for Jackson.

Our approach is similar to Blair-Goldensohn et al. [26] at Google research, except that we do not penalize any score that is below a manually tuned threshold. In our future work, we will investigate the impact of sentiment scores and popularity metrics available in developer forums.

5.2.2 Sentiment Trend

We produced monthly aggregated sentiments for each API by grouping total positive and negative opinions towards the API. In ③ of Figure 5.2, the line charts show an overview of the summary by months. We produced the summary as follows: (1) We assign a timestamp to each opinionated sentence, the same as the creation timestamp of the corresponding post where the sentence was found. (2) We group the timestamps into yearly buckets and then into monthly buckets. (3) We place all the opinionated sentences in the buckets based on their timestamp.

5.2.3 Co-Reviewed APIs

In Figure 5.2, ④ shows the other APIs that were reviewed in the same posts where the API Jackson was discussed. We computed this as follows: (1) For each opinionated sentence related to the given API, We identified the corresponding posts where the sentence was found. (2) We identified all the other API mentions in the same post. (3) For each of the other APIs in the post, we detect the opinionated sentences related to the APIs. (4) For each API, we grouped the positive and negative opinions for each post. (5) We then calculate a ratio of negativity vs positivity by taking the count of total positive opinions for the given API vs total negative opinions for each of the other APIs. If the ratio is greater than one, we say that the given API is more negatively reviewed around the other API. (6) For each of the other APIs, We count the number of times the other API is more positively reviewed around the given API.

5.3 Aspect-Based Summaries of API Reviews

Aspect-based summarization [25, 26] involves generating summaries of opinions around a set of aspects, each aspect corresponding to specific attributes/features of an entity about which the opinion was provided. For example, for a camera, picture quality can be an aspect. Aspects can

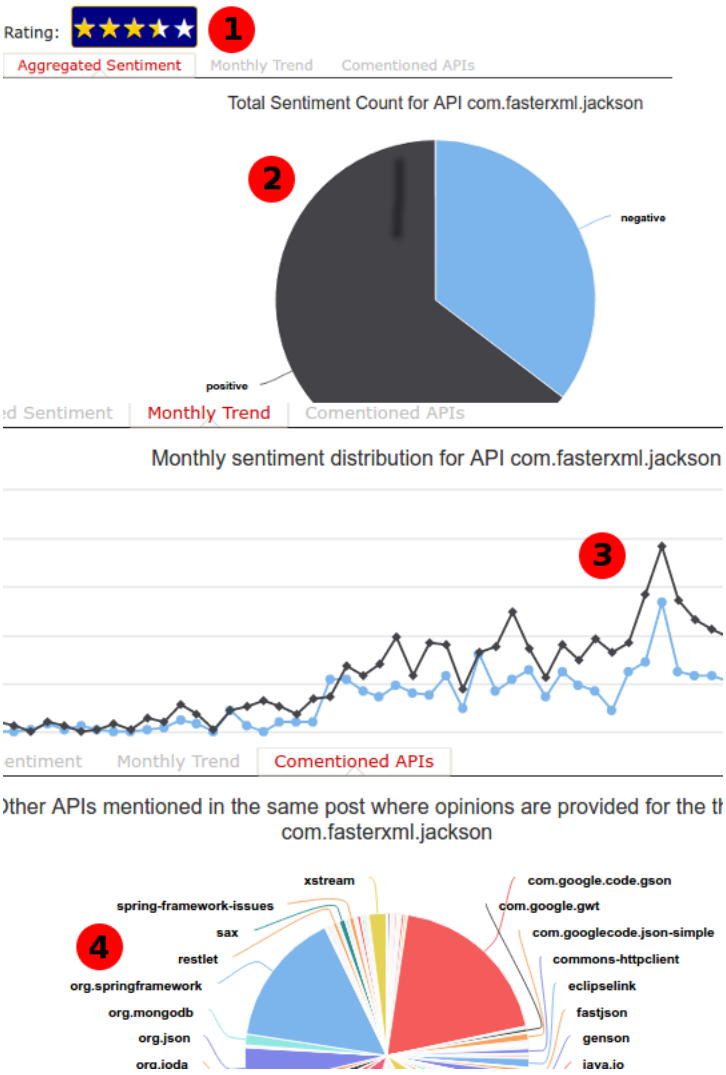

Figure 5.2: Statistical summarization of opinions for API Jackson.

be different depending on the domains and thus the detection of domain specific aspects is the first step towards aspect-based opinion summarization. In this section, we describe the components of our system that identifies the aspects of an API about which developers provided opinions. This includes finding the corresponding sentences that mention these aspects. Our approach contains the following steps, each supported by our development infrastructure:

1) **Static aspect detection:** We leveraged the fact that similar to other domains [26], we observed a Zipfian distribution for API reviews, i.e., some aspects are more likely to be discussed across the APIs (e.g., performance, usability). These are called static aspects [26]. We detect the 11 API aspects we described in Chapter 4.

2) **Dynamic aspect detection:** We observed that certain aspects can be more common in an API or a group of APIs rather than in all the APIs ('object conversion' for JSON parsing vs 'applet size' for user interfaces). The detection of these *dynamic* aspects required techniques different from the detection of static aspects. We detect dynamic aspects in the sentences labeled as 'Other

General Features'.

3) **Summarizing opinions for each aspect:** For each API, we produced a consolidated view by grouping the reviews under the aspects and by presenting different summarized views of the reviews under each aspect.

5.3.1 Static Aspect Detection

In Section 4.4, we described a suite of rule-based and supervised techniques to automatically detect the following aspects in the API reviews. For our aspect-based summaries, we use the best performing supervised classifiers for each aspect we developed using the imbalanced dataset.

Performance: How well does the API perform?

Usability: How usable is the API?

Security: How secure is the API?

Documentation: How good is the documentation?

Compatibility: Does the usage depends on other API/framework/version?

Portability: Can the API be used in different platforms?

Community: How is the support around the community?

Legal: What are licensing and pricing requirements?

Bug: Is the API buggy?

Other General Features: Opinions about other API features and general API usage.

Only Sentiment: Opinions without an API specifying any aspect.

In Figure 5.3, we show the distribution of the static aspects in our dataset of all Java JSON threads. Some of the aspects were much less represented due to the specific nature of the domain. For example, 'security' is less of a concern in JSON parsing than it is in network-based tasks. The aspect 'Compatibility' was the least represented, due to the nature of the domain. For example, APIs offering JSON parsing in Java can be applied irrespective of the underlying operating system. The aspect 'Usability' accounted for almost 60% of the opinions, followed by 'Other General Features'. The sentences belonging to the 'Other General Features' category contained opinions about API aspects not covered by the nine implicit API aspects. We applied our dynamic aspect detection on the opinions labeled as 'Other General Features' (discussed below).

5.3.2 Dynamic Aspect Detection

We detected dynamic aspects on the 15% of the sentences in our dataset that were labeled exclusively as 'Other General Features'. Our dynamic aspect detection algorithm is adopted from similar techniques used in other domains, e.g., electronic products [115], and local service reviews (e.g, hotels/restaurants [26]). Our approach consists of the following three steps:

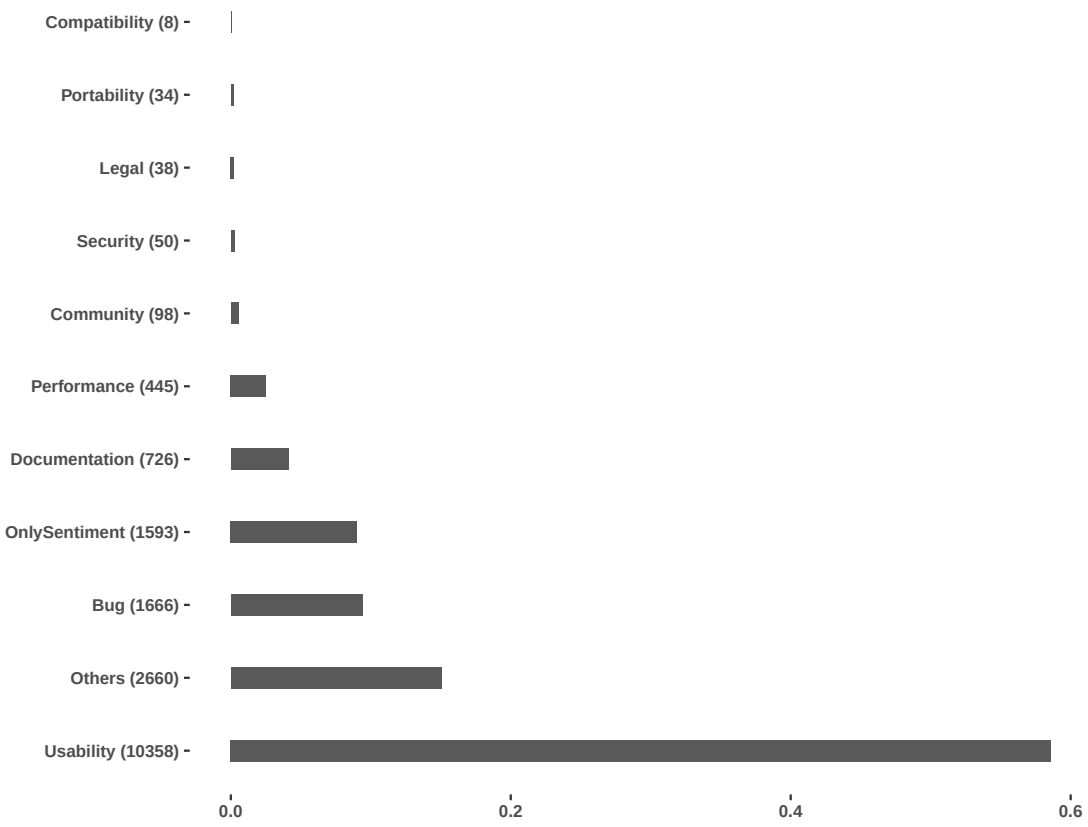

Figure 5.3: The distribution of static aspects in the dataset

.

1) **Identification**: We find representative keywords and phrases in the sentences labeled as 'Other General Features'.

2) **Pruning**: We discard keywords/phrases based on two filters. The remaining items are considered as dynamic aspects.

3) **Assignment**: We assigned each sentence to an aspect.

The remaining keywords are considered as dynamic aspects. We discuss the steps below.

Step 1. Keyword Identification

The first step in dynamic aspect detection is to find keywords/phrases that are most likely to be a representation of the underlying domain. Unlike Hu et al. [115] and Blair-Goldensohn et al. [26] who used frequent itemset mining, we used the keyphrase detection process used in the Textrank algorithm. Textrank uses the Google Page rank algorithm to find representative keyphrases in a given dataset. The page rank algorithm has been successfully applied in software engineering research [132]. We detected the keyphrases as followed: (1) We tokenized the sentences (2) We removed the stopwords (3) We detected Parts of Speech (4) We discarded any words that are not

Nouns (5) We created a graph of the words by putting the words as nodes and creating an edge between the words whenever they appeared together in the dataset. (6) On this network of words, we then applied the Page rank algorithm to find the ranking of each words.[1] (7) From this ranking, we kept the top 5% words as keywords. In our future work, we will investigate the algorithm with different percentages of keywords. We then merged two or more into one if they appeared together in the sentences.

Step 2. Pruning

Hu et al. [115] and Blair-Goldensohn et al. [26] observed that, the frequent keywords by themselves can still be too many to produce a concise list of aspects. We thus applied two filters on the keywords to remove the keywords that were not widely found across different APIs. Both of the filters were originally proposed by Hu et al. [115]

F1. Compactness pruning was applied to remove key phrases that were not present in at least three sentences. The words in a key phrase were considered as present in sentence if all the words in the key phrase were found in the sentence in the same sequence, with possibly at most one another word in between two keywords.

F2. Support-based pruning was applied to remove keywords that were not present in at least three sentences.

We then computed the TF-IDF (Term Frequency - Inverse Document Frequency) [107] of the remaining keywords and phrases in the sentences labeled as 'Other General Features' and ranked those based on score (i.e., the higher the value is the more representative the keyword is).

Step 3. Assignment

We assigned each sentence in the 'Other General Features' dataset to a keyword as identified in step two. If a sentence was not labeled as any of the keywords/phrases, we considered it as a 'general comment'. We labeled each such sentence as discussing about an API 'feature'. For a sentence labeled as more than one dynamic aspect, we assigned it to the one with the highest TF-IDF score.

In Figure 5.4, we show the distribution of the dynamic aspects (except the general comments) in the dataset. The four most prominent aspects were 'class', 'value', 'json', and 'object'. We note that such aspects are more specific to the domain of JSON parsing. For example, it's often a common requirement to produce an automatic conversion between a Java object and JSON-based data and vice versa. Using static aspect detection, we would not expect to find such aspects.

[1] We used the Page rank implementation from the Python NetworkX library [133]

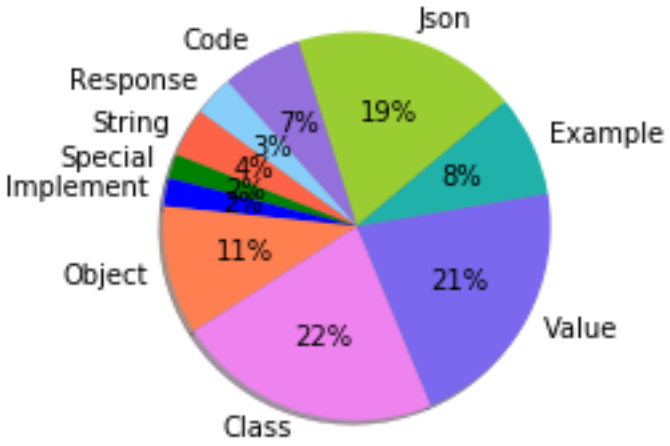

Figure 5.4: The distribution of dynamic aspects in the dataset

.

5.3.3 Aspect-Based Summarizing of Opinions

We produced the following summaries based on the aspects: Overview, Nested aspect, and Comparative. We explain the techniques behind each summaries below:

Aspect Overview.

For a given API, we produce a consolidated overview of the aspects detected in its reviews. The overview contains the rating of each aspect for the given API and the most recent positive and negative opinion. In Figure 5.5, ① shows the overview page. The aspects are ranked based on their recency, i.e., the aspect with the most recent positive or negative opinion is placed at the top. We further take cues from Statistical summaries and visualize sentiment trends per each aspect for the API. Such trends can be useful to compare how its different aspects have been reviewed over time.

Nested Aspect Views.

Given as input all the positive or negative opinions of an API, we group the opinions by the detected aspects. We rank the aspects based on their recency, i.e., the aspect with the most opinion is placed at the top. We observed that even after this grouping, some of the aspects contained hundreds of opinions (e.g., Usability). To address this, we further categorized the opinions under an aspect into sub-categories. We denote those as nested-aspect. We detected the nested aspects as follows. (a) We collected all the Stack Overflow tags associated to the threads in the dataset. (b) For each sentence under a given aspect of an API, we labeled each opinion as a tag, if the opinion contained at least one word matching the tag. (c) For an opinion labeled as more than one tag, we assign the opinion to the tag that covered the most number of opinions within the aspect for the API. (d) For an opinion that could not be labeled as any of the tags, we labeled it as containing 'general'. In Figure 5.5, the second circle(②) shows a screenshot of the nested views of the negative opinions of the API Jackson.

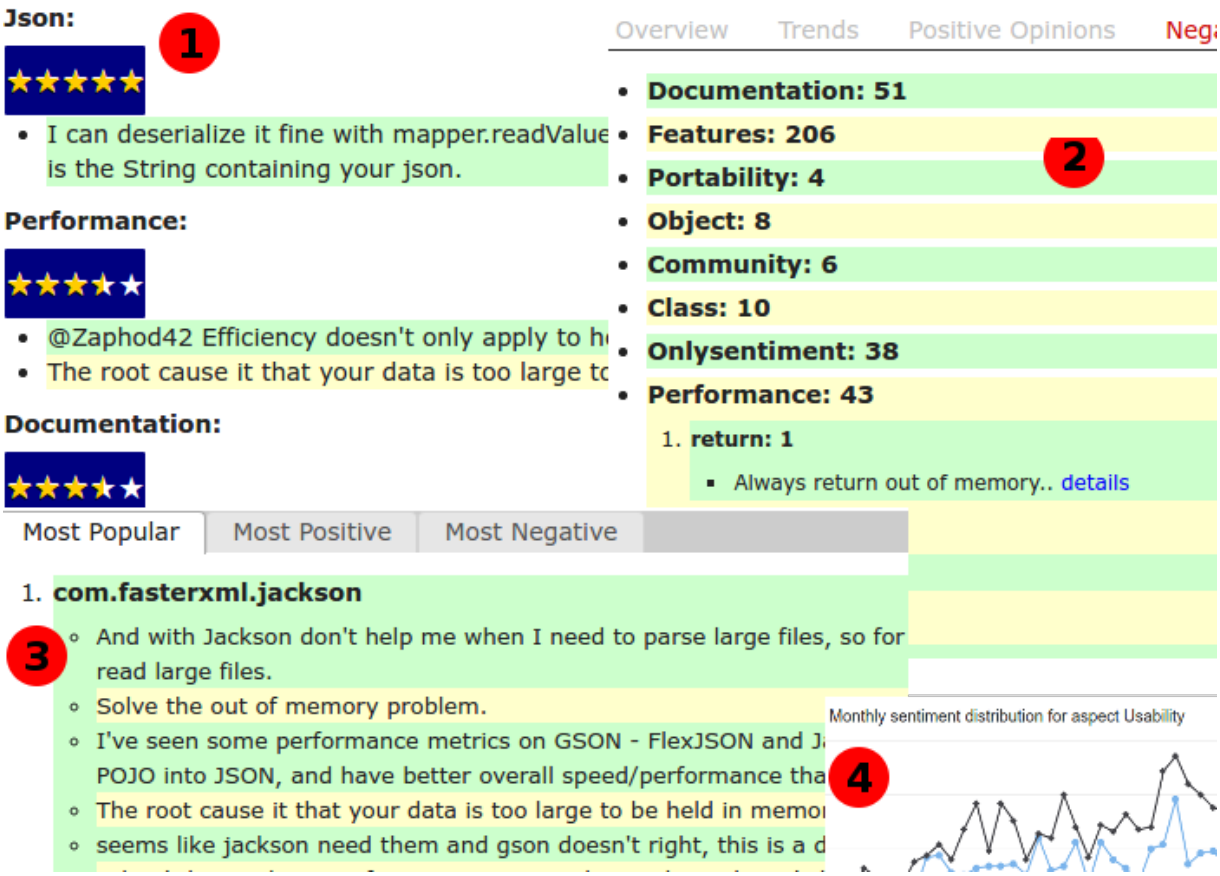

Figure 5.5: Screenshot of the aspect-based summarizer

.

By clicking the 'details' link the user is taken to the corresponding post in Stack Overflow where the opinion was provided.

Comparative By Aspect

While all the above three summarizations took as input the opinions and aspects of a given API, they do not offer a global viewpoint of how a given aspect is reviewed across the APIs. Such an insight can be useful to compare competing APIs given an aspect (e.g., performance). We show three types metrics: popularity $\frac{P_A+N_A}{P_C+N_C}$, positivity $\frac{P_A}{P_T}$, and negativity $\frac{N_A}{N_C}$ (P_A is the total number of positive opinions about an API, where P_C is the total number of positive opinions about all APIs in the dataset). In Figure 5.5, the third circle(③) shows a screenshot of the 'comparative by aspect' views for the aspect 'performance' and shows that the most popular API for json-based features in Java is jackson. A user can click the API name and then it will show the most recent three positive and three negative sentences, one after another. If the user is interested to explore further, a link is provided after those six opinions that will take the user to all the opinions and summaries of the reviews of the API.

5.4 Summarization Algorithms Adopted For API Reviews

In this section, we explain how we adopted currently available summarization algorithms to produce extractive, abstractive, contrastive, and topic-based summaries of API reviews.

5.4.1 Topic-based Summarization

In a topic-based opinion summarization [134–138], topic modeling is applied on the opinions to discover underlying themes as topics. There are two primary components in a topic-based opinion summarizer: (1) A title/name of the produced topics, and (2) A summary description of the topic. Three types of summaries are investigated: word-level [134, 135], phrase level [136] and sentence level [137, 138]. For each API in our dataset, we summarized the reviews by applying LDA (Latent Dirichlet Allocation) [102] once for the positive opinions and once for the negative opinions. LDA has been used to investigate topics in various software engineering artifacts, e.g., emerging topics in Stack Overflow [103].

We produced the summary using topic modeling as follows:

Polar Subset Generation: we divided the datasets into two subsets: positive and negative. The positive subset contained all the sentences for the API labeled as positive, and the negative subset contained all the negative sentences.

Optimal Number of Topics Detection: We determined the optimal number of topics for each subset using the technique proposed by Arun et al. [139].[2] The technique computes symmetric KL-Divergence metrics of the produced topics. An optimal number of topic is determined for the topics with the lowest KL-divergence values. For example, for the 2807 positive sentences for the API Jackson, the technique determined that the optimal number of topics would be 24 while for the 1465 negative sentences for the API, the algorithm returned the optimal number of topics to be only two.

Topic Detection: We applied LDA (Latent Dirichlet Allocation) [102] on each of the subsets repeatedly until the coherence value[3] of the LDA model no longer increased. The coherence measure of a given model quantifies how well the topics represent the underlying theme in the data. To determine the representativeness of topics, we used the topic *coherence* measure as proposed by Röder et al [104].

Topic Description Generation: For each topic, we produced a description by taking the top ten words of the topic.

Summary Generation: For each topic, we produced a summary by associating it with each *representative* sentence as follows. We assigned each sentence in the subset to a topic that showed the highest correlation value towards the sentence. A sentence could only be assigned to one topic. Following Ku et al. [138] and Mei et al. [137], we ranked each sentence in a topic based on the time, i.e., the most recent sentence in a list of all the sentences assigned to a topic, was placed at the top of all the sentences for the topic.

[2] We used the implementation from Tom Lib library [140]

[3] We used gensim [105] with c_v coherence to produce topics.

- **0. simple binding gson mapping stream hibernate deal mention model objectmapper: 33**
- **1. gson spring structure jaxb serialize format filter implementation find provide: 44**
- **2. find gson solution lot objectmapper case codehaus bind thing binding: 37**
- **3. case csv gson jaxb thing output jax write writevalue readvalue: 26**
- **4. gson deserialize standard jersey mapper add output step deserialization instance: 44**
- **5. parsing add case read structure input hibernate exist generate jira: 43**
- **6. jaxb spring require avoid databind jersey simple gson write thing: 27**

Figure 5.6: Topic-based summarization of the positive sentences for the API Jackson. Only the first seven topics are shown here.

processer?.

2. ◦ Well sometimes performance and usability can be at odds, but I've found GSON to b
   Jackson , org json and others .
   ◦ GSON was 5x slower than org json, and 10x slower than Jackson .
3. ◦ In particular, we are interested in the l jackson databind ObjectWriter html#withRoo
   withRootName method: CODESNIPPET JAVA2 .
   ◦ Finally, if you decide that enough is enough, and decide to use a JSON library (you r

Figure 5.7: contrastive opinion summarization for Jackson.

In Figure 5.6, we show a screenshot of the topic-based summaries for the API Jackson in Opiner. The numbers beside each topic show the total number of positive sentences that are assigned to it to produce the summaries for each topic.

5.4.2 Contrastive Viewpoint Summary

In contrastive summaries, contrastive viewpoints are grouped together (e.g., 'The object conversion in gson is easy to use' vs 'The object conversion in GSON is slow'). In the absence of any off-the-shelf API available to produce such summaries, we implemented the technique proposed by Kim and Zhai [27]. The technique works as follows:

1) We compute the similarity between each pair of sentences in our positive and negative subsets of each API. We used the Jaccard index-based similarity measure as follows: $jaccard = \frac{Common}{All}$. This is one of the two measures proposed by Kim and Zhai [27]. While computing the similarity, we filtered out the stopwords and sentiment words.

2) We then apply the Contrasitve-First approximation algorithm [27] to select k (top 1%) sentence pairs with the highest similarity score.

In Figure 5.7, we show a screenshot of the contrastive summaries for the API Jackson in Opiner.

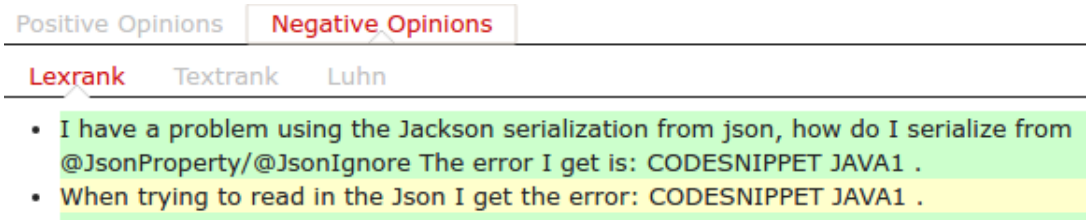

Figure 5.8: Extractive opinion summarization for Jackson.

5.4.3 Extractive Summarization

With the extractive summarization techniques, a summary of the input documents is made by selecting *representative* text segments, usually sentences from the original documents. Extractive summaries are the most prevalent for text summarization across domains (e.g., news, blogs). The techniques use diverse heuristics to determine the representative sentences, such as, using the frequency of important words (e.g., non stopwords) and their pairing in the documents (such as using TF-IDF). For each API, we applied three extractive summarization algorithms:

- **Luhn [28]**: The first extractive summarization algorithm. It uses heuristics to determine representative sentences from input documents, e.g., frequency of important words, etc.
- **Lexrank [141]** and **Textrank [29]**: Two of the most recent algorithms. Both are based on the concept of PageRank which formed the basis of Google's search engine. In both algorithms, a graph of the input sentences is created by creating a vertex for each sentence. The edges between sentences are generated based on semantic similarity and content overlap between the sentences. To compute the similarity, Lexrank uses cosine similarity of TF-IDF vectors of input sentences and Textrank uses a measure based on the number of words common between two sentences.

In our future work, we will investigate review summaries using more existing extractive algorithms [142].

Using each algorithm, we produced two summaries for each API, one for positives and one for negatives.[4]

For each API, we produce summaries containing 1% of the inputs or 10 opinionated sentences (whichever yields the most number of sentences in the summary). Each sentence in our dataset has 12 words on average. The most reviewed API in our dataset is Jackson with 28 sentences with a 1% threshold (i.e., 330-340 words). For 10 sentences, the summaries would have 120 words. Previous research considered lengths of extractive summaries to be between 100 and 400 words [31]. In Figure 5.8, we show a screenshot of the summaries for the API Jackson in Opiner (one tab for each algorithm).

[4]We used the Python sumy library implementation of Luhn, Lexrank and Textrank

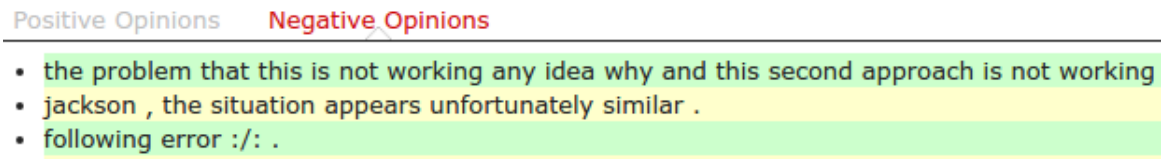

Figure 5.9: Abstractive opinion summarization for Jackson.

5.4.4 Abstractive Summarization

Unlike extractive summaries, abstractive summaries produce an abstraction of the documents by generating concise sentences. A limitation of extractive summarization for opinions is that a limit in the summary size may force the algorithm to remove important sentences. Moreover, when the sentences are long, even a small summary can be quite verbose and may not be suitable for small screens. We produced abstractive summary for an API once for positive sentences and once for negative sentences using Opinosis, an abstractive domain-independent opinion summarization engine. To address such limitations, Ganesan et al. [32] proposed Opinosis that produces a concise overview of the input sentences by producing new sentences with important keywords found in the sentences. To achieve this, the algorithm first creates a textual word graph (unlike sentence graph in extractive summarization) and then selects candidate words in different sub parts of the graph to create a list of sentences. The Opinosis tool is available and it is domain independent. We produced abstractive summary for an API once for positive sentences and once for negative sentences as follows. (1) We detected the Parts of Speech of each word in a sentence. (2) We ran the Opinosis jar file with the configuration similar to the extract summarization, i.e., produce summaries of length of only 1% of the total lengths. In Figure 5.9, we show a screenshot of the abstractive summaries for the API Jackson in Opiner.

5.5 Informativeness of the Summaries (RQ1)

Because the goal of the automatic summary of the reviews of an API is to provide developers with a quick overview of the major properties of the API that can be easily read and digested, we performed a study involving potential users, in a manner similar to previous evaluation efforts on software artifact summarization [40, 143]. In this section, we describe the design and results of the study.

5.5.1 Study Design

Our *goal* was to judge the *usefulness* of a given summary. The *objects* were the different summaries produced for a given API and the *subjects* were the participants who rated each summary. The *contexts* were five development tasks. The five tasks were designed based on our analysis of the diverse development needs that can be supported by the analysis of API reviews (Chapter 2).

Each task was described using a hypothetical development scenario where the participant was asked to judge the summaries through the lens of the software engineering professionals. Persona based usability studies have been proved effective both in the Academia and Industry [144]. We

briefly describe the tasks below.

T1. Selection - Can the summaries help you to select this API?
The persona was a 'Software Architect' who was tasked with making a decision on a given API based on the provided summaries of the API. The decision criteria were: *(C1)* ***Rightness***: contains the right and all the useful information. *(C2)* ***Relevance***: is relevant to the selection. *(C3)* ***Usable***: different viewpoints can be found.

T2. Documentation - Can the summaries help to create documentation for the API?
The persona was a 'Technical Writer' who was tasked with writing a software documentation of the API to highlight the strengths and weaknesses of a given API based on the reviews in Stack Overflow. The decision criteria were: *(C1)* ***Completeness***: complete yet presentable *(C2)* ***Readable***: easy to read and navigate.

T3. Presentation - Can the summaries help you to justify your selection of the API?
The persona was a development team lead who was tasked with creating a short presentation of a given API to other teams with a view to promote or discourage the usage of the API across the company. The decision criteria were: *(C1)* ***Conciseness***: is concise yet complete representation *(C2)* ***Recency***: shows the progression of opinions about the different viewpoints.

T4. Awareness - Can the summaries help you to be aware of the changes in the API?
The persona was a software developer who needed to stay aware of the changes to this API because she used the API in her daily development tasks. The decision criteria were: *(C1)* ***Diversity***: provides a comprehensive but quick overview of the diverse nature of opinions. *(C2)* ***Recency***: shows the most recent opinions first.

T5. Authoring - Can the summaries help you to author an API to improve its features?
The persona was an API author, who wanted to assess the feasibility of creating a new API to improve the features offered by the given API. The decision criteria were: *(C1)* ***Strength and Weakness highlighter***: shows the overall strengths and weakness while presenting the most recent opinions first. *(C2)* ***Theme identification***: presents the different viewpoints about the API.

We assessed the ratings of the three tasks (Selection, Documentation, Presentation) using a 3-point Likert scale (the summary does not miss any info, Misses some info, misses all the info). For the task (Authoring), we used a 4-point scale (Full helpful, partially helpful, Partially Unhelpful, Fully unhelpful). For the task (Awareness), we asked participants how frequently they would like to use the summary (never, once a year, every month, every week, every day). Each of the decision criteria under a task was ranked using a five-point likert scale (Completely Agree – Completely Disagree).

Each participant rated the summaries of two APIs. We collected ratings for four different APIs. The four APIs (Jackson, GSON, org.json and jersey) were the four most reviewed APIs in our dataset offering JSON-based features in Java. The four APIs differed from each other on a number of aspects. For example, Jackson differentiates itself by providing annotation-based mixin to facilitate faster processing of JSON files. GSON focuses more on the usability of the overall usage, org.json is the oldest yet the natively supported JSON API in Android, and jersey is a framework (i.e., it offers

other features besides providing JSON features). In addition to collecting rates for each summary, we also collected demographic information about the participants, i.e., by collecting their experience and current roles in their development teams. We analyzed the responses using descriptive statistics.

5.5.2 Participants

We hired 10 participants from the online professional social network site (Freelancer.com). Each freelancer had at least a 4.5 star rating (the maximum possible star rating being 5). Sites like Amazon Mechanical turks and Freelancer.com have been gaining popularity to conduct studies in empirical software engineering research due to the availability of efficient, knowledgeable and experienced software engineers. We only hired a software engineer if he used at least one of the four APIs in the past. Each participant was remunerated between $7-$12, which is a modest sum given the volume of the work. Each participant was allowed to complete a study only once. To ensure future traceability, we collected the name and email address of each participant. The study was conducted using Google Doc Form. Each participant was given an access to the Opiner online tool and provided a short demo of the tool to ensure that they understood the tool properly. In addition, a coding guide was linked to the study questionnaire to explain all the summaries in details. Each reference to a summary of a given API was also linked to the corresponding web page in Opiner. The participants were allowed to only complete the study when they completed reading the coding guide. All of the participants were actively involved in software development and had software development experience ranging from three to more than 10 years. The occupation of the participants varied among software developers, team leads, and architects.

5.5.3 Results

In Table 6.10, we show the percentage of the ratings for each summary for the four tasks (Selection, Documentation, Presentation, Authoring). In Figure 6.14, we show the percentage of participants showing their preference towards the usage of the summaries to stay aware (Awareness task). The aspect-based summary was considered as the most useful, followed by statistical summaries. Among the other summaries, the topic-based summaries were the most favored. In Table 5.4, we show the percentages of the ratings for each criteria under each development task. We discuss the rationales below.

T1. Selection - Can the summaries help you to select this API?

While most participants completely agreed the most (60-65%) for aspect-based summaries, when we combined the ratings of completely agreed and agreed, statistical summaries were the most favored. When the participants were asked to write a short summary of their ratings based on the provided criteria, participant R6 summed it up well *"First I used the statistics summarization to get an overview on the API, whether I should go on and check it's details. Then I headed to Aspect-based summarization to check each aspect of the API. These gave me a good picture of the API and whether we should use it or not."* A close third was the contrastive summaries, showing that pairing similar themes based on the contrastive viewpoints can provide useful quick insights. Among the other summaries, extractive summaries were also well-regarded. According to one participant *"use extractive and contrast summary to*

Table 5.3: Impact of the summaries based on the scenarios

Task	Option	Statistical	Aspect	Topic	Contrastive	Abstractive	Luhn	Textrank	Lexrank
Selection	**NM**	**75**	**80**	25	40	10	20	35	35
	MS	25	20	65	50	40	65	60	65
	MA	0	0	10	10	50	15	5	0
Documentation	**NM**	**60**	**75**	40	25	20	25	35	30
	MS	25	20	40	60	50	50	50	55
	MA	15	5	20	15	30	25	15	15
Presentation	**NM**	**65**	**65**	40	20	20	20	30	10
	MS	30	30	35	50	50	60	65	80
	MA	5	5	25	30	30	20	5	10
Authoring	**FH**	**50**	**75**	15	10	5	10	20	20
	PH	40	10	55	65	50	55	40	45
	PU	5	10	10	0	25	5	10	10
	FU	0	0	0	5	10	10	5	0
	NT	5	5	20	20	10	20	25	25

NM = Not misses any info, MS = Misses some info, NT = Neutral

MA = Misses all info, S = Statistical Summary, A = Aspect

FH/PH = Full/Partially Helpful, FU/PU = Full/Partially Unhelpful

T = Topic-based, B = Abstractive, L = Lexrank, P = Texrank, U = Luhn

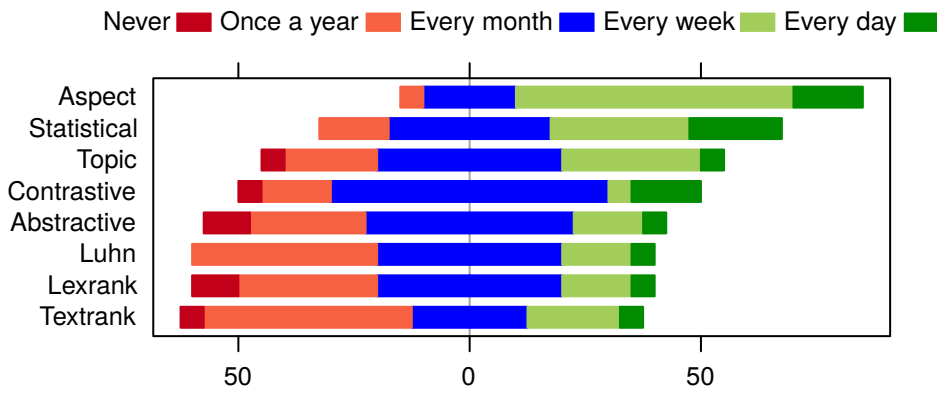

Figure 5.10: Developers' preference of usage of summaries

Table 5.4: Rationale Provided by the Participants to Support the Scenarios (O = Option, T = Scenario)

T	Criteria	O	Statistic	Aspect	Topic	Contrast	Abstract	Luhn	Text	Lex
S	Rightness	👍👍	**55**	**65**	25	15	5	10	20	25
E		👍	30	20	15	**55**	35	40	25	25
L		😐	15	15	45	15	25	20	35	30
E		👎	0	0	10	15	15	15	20	20
C		👎👎	0	0	5	0	20	15	0	0
T	Relevant	👍👍	**65**	**60**	25	10	5	15	25	10
I		👍	25	20	20	**55**	40	30	25	40
O		😐	10	10	35	25	20	15	20	35
N		👎	0	10	15	10	10	30	25	10
		👎👎	0	0	5	0	25	10	5	5
	Usable	👍👍	40	**60**	25	30	10	15	25	10
		👍	45	20	30	30	30	35	35	45
		😐	15	15	30	35	25	20	10	25
		👎	0	5	10	5	15	20	30	20
		👎👎	0	0	5	0	20	10	0	0
D	Complete	👍👍	**55**	**60**	35	15	15	20	30	30
O		👍	15	20	20	45	35	30	20	20
C		😐	25	15	20	30	15	25	35	35
U		👎	0	5	15	5	15	15	10	10
M		👎👎	5	0	10	5	20	10	5	5

E	Readable	👍👍	**60**	**65**	35	30	20	20	20	20
N		👍	20	30	35	35	45	35	40	35
T		😐	15	5	10	25	10	25	20	30
		👎	0	0	15	0	15	10	10	5
		👎👎	5	0	5	10	10	10	10	10
P	Concise	👍👍	**60**	**65**	30	15	10	10	25	15
R		👍	25	20	25	**50**	40	35	20	35
E		😐	5	5	25	20	25	35	35	40
S		👎	5	10	15	5	10	10	15	0
E		👎👎	5	0	5	10	15	10	5	10
N	Recency	👍👍	**65**	**70**	35	15	25	20	25	25
T		👍	15	10	20	30	20	30	25	20
		😐	20	20	35	35	30	30	30	30
		👎	0	0	10	10	15	10	15	15
		👎👎	0	0	0	10	10	10	5	10
A	Diversity	👍👍	**60**	**75**	25	15	15	10	25	15
W		👍	30	10	40	**60**	40	55	50	60
A		😐	10	10	25	20	25	30	15	20
R		👎	0	5	10	0	10	0	10	5
E		👎👎	0	0	0	5	10	5	0	0
N	Recency	👍👍	**55**	**70**	30	10	15	10	15	10
E		👍	15	10	25	40	45	40	30	30
S		😐	30	20	35	40	25	40	40	45
S		👎	0	0	10	5	5	0	15	15
		👎👎	0	0	0	5	10	10	0	0
A	Highlight	👍👍	**50**	**70**	25	15	10	15	25	15
U		👍	25	15	35	**50**	**50**	**50**	35	45
T		😐	20	15	40	30	30	25	30	35
H		👎	5	0	0	0	0	5	5	0
O		👎👎	0	0	0	5	10	5	5	5
R	Theme	👍👍	25	**65**	20	20	10	15	20	15
I		👍	**50**	20	25	35	40	30	25	30
N		😐	20	15	**55**	45	40	45	40	40

G	👎	0	0	0	0	0	5	10	10
	👎👎	5	0	0	0	10	5	5	5

👍👍= Completely Agree, 👍= Agree, 👎👎= Completely disagree, 👎= Disagree 😐= Neutral

determine usefulness and to see if the library fits my teams needs.". The abstractive summary was not considered as useful *"...however the abstractive summarization didn't give enough detail even at an abstract level."*. Among the extractive summaries, the summaries produced by Luhn algorithm were considered as the most useful *" I could extract further interesting information from Extractive Summarization "Luhn". The other ones gave me not meaningful results, so I didn't take them into consideration when I made my decision"*.

T2. Documentation - Can the summaries help to create documentation for the API?
The aspect-based summary was favored followed by statistical summaries. Among the other summaries, topic-based summaries were the most favored, showing the needs of grouping opinions by themes as well as offering visualizations. According to a participant *"Aspect-based summarization's documentation part was a huge help making the decision. All the other summarization were quite useless regarding deciding about documentation."* However, some other participants were in favor of all summaries *"the information presented in all the summaries was clean and clear enough so as to make a selection for a particular API."* Visualization is favored *"Statistical shows all, including charts which saves time shows comparisons with other APIs too"*.

T3. Presentation - Can the summaries help you to justify your selection of the API?
While aspect and statistical summaries were again strongly favored, contrastive summaries were the most favored among the rest, showing a similarity with the 'Selection' task. This inclination could mainly be due to the fact that both tasks asked them to provide justification of their selection/usage of the API. According to one participant *"Statistical is ideal for presentation Aspect covers every thing trends, positive and negative response, response summary"*.

T4. Awareness - Can the summaries help you to be aware of the changes in the API?
While the aspect and statistical summaries were the most favored based on each criteria, topic-based was considered as the most helpful among others. Intuitively, the same trends were observed for the task 'Documentation', and we note both were focused towards helping developers while 'Selection' and 'Presentation' were mainly for software develop ers who are not involved in daily programming activities. Participants wanted a combination of both statistical and aspect-based summaries *"from both points the aspect-based summary helped a lot to make the decision. Aspect-based mainly because it's opinions are ordered with the most recent first, which helps keep track of changes/updates about the API. From the point of diversity the statistical summarization helps a lot, because it provides a good overview."*

T5. Authoring - Can the summaries help you to author an API to improve its features?
Aspect-based summaries were still favored the most, because of the ratings provided to each aspect in the overview page *'Aspect-based summary helped a lot with it's rating system, where I could see exactly what are the weak spots of the API, so what should I concentrate on if I decide to make a new, better API."*

5.6 Effectiveness Analysis of Opiner (RQ2)

Because the purpose of developing Opiner was to add benefits over the amazing usefulness Stack Overflow provides to the developers, we sought to seek the usefulness of Opiner by conducting another study of professional software developers who completed two tasks using Stack Overflow and Opiner.

5.6.1 Study Design

The *goal* of this study is to analyze the *usefulness* of Opiner to assist in a development task. The *objects* of this study are the APIs and their summaries in Opiner and Stack overflow. The *subjects* are the participants completing the tasks. Our study consisted of two parts:

(P1) Assessment of Opiner's usefulness in a development setting

(P2) Opportunities of Industrial adoption of Opiner.

To conduct the study, we developed an online data collection tool with the following features: 1) Login features for each user 2) Passive logging of user activities 3) Storage of user inputs in a relational database.

P1. Usefulness Analysis.

We designed two tasks, both corresponding to the selection of an API from a pool of two APIs. The two tasks corresponded to two different sets of APIs.

Task 1. The participants were asked to make a selection out of two APIs (GSON and org.json) based on two criteria: 1) Usability, and 2) Licensing usage. The right answer was GSON. The licensing agreement of the API org.json is generally considered not safe for Industrial usage [145].

Task 2. The participants were asked to make a selection out of two APIs (Jackson and json-lib) based on two criteria:1) Performance and 2) Pre-installed base in leading frameworks (e.g., Spring, Restlet, etc.) The correct answer was Jackson which is the pre-packaged JSON API in the major frameworks.

We asked each developer to complete a task in two settings:

1) **SO only**: Complete only using Stack Overflow
2) **SO + Opiner**: Complete using Stack Overflow + Opiner

For a task in a each setting, each developer was asked to provide following answers:

Selection: Their choice of API

Confidence: How confident they were while making the selection. We used a five-point scale: Fully confident (value 5) - Fully unsure (1).

Rationale: The reason of their selection in one or more sentences.

Table 5.5: Progression of learning from Stack Overflow to Opiner

Task ID	Tool	Correctness	Conversion	Confidence	Time
T1	SO	20.0%	–	4.3	12.3
	SO + Opiner	100%	100%	4.5	7.5
T2	SO	66.7%	–	2.7	18.6
	SO + Opiner	100%	100%	4.6	6.8

In addition, we logged the time it took for each developer to complete a task under each setting.

P2. Industrial Adoption.

After a developer completed the tasks, we asked her to share her experience of using Opiner:

- **Usage**: Would you use Opiner in your development tasks?
- **Usability**: How usable is Opiner?
- **Improvement**: How would you like Opiner to be improved?

The developers were asked to write as much as they can.

5.6.2 Participants

We invited nine professional developers from a software company and two developers from Freelancer.com to participate in the study. The experience of the developers ranged between 1 year and 34 years. The developers carried on different roles ranging from software developer to architect to team lead. The team leads were also actively involved in software development. Except one developer from the company all others participated in the study. We first provided an online demo of Opiner to them within the company during the lunch time. The Freelancers were provided the demo by sharing the screen over Skype. After the demo, each developer was given access to our data collection tool.

5.6.3 Study Data Analysis

We analyzed the responses along the following dimensions:1) Correctness: How precise the participants were while making a selection in the two settings. 2) Confidence: How confident they were making the selection? 3) Time: How much time did the developers spend per task? In addition, we computed a 'conversion rate' as the ratio of developers who made a wrong selection using Stack Overflow but made the right selection while using both Stack Overflow and Opiner.

5.6.4 Results

In this section, we present the results of the study.

5.6.4.1 Usefulness Analysis

Nine developers completed task 1 using the two different settings (10 using Stack Overflow, one of them could not continue further due to a sudden deadline at work). Five developers completed task 2 using both of the settings (nine using Stack Overflow, four of them could not complete the task. In Table 5.5, we present the impact of Opiner while completing the tasks. To compute the Conversion rate, we only considered the developers that completed a task in both setting. For task 1, only 20% of the developers using Stack Overflow only selected the right API, but all of those who later used Opiner picked the right API (i.e., 100% conversion rate). For task 2, only 66.7% of the developers using Stack Overflow only selected the right API, but all of them picked the right API when they used both Opiner and Stack Overflow (conversion rate = 100%). The developers using the setting 'SO+Opiner' on average took only 7.5 minutes total to complete T1 and 6.8 minutes to complete T2. When the same developers used SO alone, they took on average 12.3 minutes to complete T1 and 18.6 minutes to complete T2. The developers also reported higher confidence levels when making the selection using Opiner with Stack Overflow. For task 1, the confidence increased from 4.3 to 4.5 and for Task 2, it increased from 2.6 (Partially unsure) to 4.6 (almost fully confident). For Task 1 one developer did not select any API at all while using Stack Overflow and wrote *"not a lot of actual opinions on the stackoverflow search for net.sf.json-lib and same for com.fasterxml.jackson. Most Stack Overflow questions are related to technical questions"*. One of the developers (with 7 years of experience) spent more than 23 minutes for Task 1 while using Stack overflow only and made the wrong selection, but then he picked the right API while using Opiner by citing that he found all the information in the summary *"com.google.code.gson has an open source license, while I was only able to find a snippet citing 'bogus license' for org.json . . . "* The developers found the information about license while using Aspect-based summaries.

5.6.4.2 Industrial Adoption

All developers answered to the three questions. We summarize major themes below.

Would you use Opiner in your daily development tasks?

Nine out of the 10 developers mentioned that they would like to use Opiner. The participants wanted to use Opiner as a starting point, *"Yes I would use Opiner. The summaries were a great way to form a first opinion which could then be improved with specific fact checking. If I was not constrained by today's exercise to only use Stack Overflow then I would then challenge the my first opinion with general searches."* For one developer the usage may not be daily *"Yes, perhaps not daily . . . the value is the increased confidence. The co-mentioned APIs are extremely valuable."* Another developer pointed out the potential significance of Opiner to a newcomer software developer "I don't think I would use it daily, but it would be a great starting point for determining which libraries to use if I am unfamiliar with the options. . . As well, I think it has good potential in helping newer programmers decide on a starting point."

How usable is Opiner?

The participants were divided about the usability of Opiner. Half of the participants considered it usable enough *"It is very user-friendly to use. Got used to it quickly."*. For another participant *"The UI is simple. But is a bit confusing to go through the tabs to find out exactly*

what the user needs. The search bar is a big help." The participants recommended two major usability improvements for Opiner:

Speed: "It's slow, but it provide a great information."

Documentation: "It's pretty straightforward, I'm unfamiliar with certain terms (Contrastive, Extractive, etc). I can sort of guess as to what they mean, but for a layman I'm not as sure as I'd like to be."

How would you improve Opiner?

Moving forward, the participants offered a number of features that would like to see in Opiner:

Enhanced contrastive: *"Focus on contrastive view, . . . instead of its technical qualities (e.g. why use one vs another)."*

Documentation: *"A brief description regarding what 'Aspects', 'Contrastive', 'Extractive', 'Abstractive' mean would be helpful"*

Sorting: *For aspect-based summaries "I think the categories need to be in alphabetical order. It makes it hard when I was comparing to API to another because I had to ctrl-f to find category."*

Opinion from multiple sources: *"Another thing that could improve Opiner is if it mined from other websites too. I think it would give an even better opinion and I would be tempted to use it more than google then."*

5.7 Threats to Validity

Construct validity threats concern the relation between theory and observations. In this study, they could be due to measurement errors. In fact, 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. We relied on the manual labelling of aspects. To assess the performance of participants, we use time and the percentages of correct answer which are two measures that could be affected by external factors, such as fatigue.

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our study. The anonymized survey responses are provided in our online appendix [146]. Nevertheless, generalizing the results of this study 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 in the detection of API aspects mentioned in forum posts.

5.8 Summary

Opinions can shape the perception and decisions of developers related to the selection and usage of APIs. The plethora of open-source APIs and the advent of developer forums have influenced developers to publicly discuss their experiences and share opinions about APIs. In this chapter, we have presented a suite of summarization techniques for API reviews. We implemented the

algorithms in our tool Opiner. Using Opiner, developers can gather quick, concise, yet complete insights about an API. In two studies involving Opiner we observed that our proposed summaries resonated with the needs of the professional developers for various tasks.

Chapter 6

Automatic Summarization of API Usage Scenarios from Informal Documentation

The huge volume of questions and answers posted on developer forums and the richness and recency of such information provide unique opportunities to compensate for the shortcomings in formal documentation. Recent such research efforts focused on the generation of on-demand developer documentation from forum posts [92], linking API types in a formal documentation (e.g., Javadoc) to code examples in forum posts where the types are discussed [35], presenting interesting textual contents from Stack Overflow about an API type in the formal documentation [87], and so on. The source code and version history of application are analyzed using static and dynamic analyses to automatically mine API usage specifications from the execution traces of the application software [147–150]. However, given the plethora of usage discussions available for an API in the forum posts, it is challenging to get quick and actionable insights into how an API can be used for a given development task. Challenges in such exploration include, but not limited to, finding the actual solution amidst thousands of already available solutions that are scattered across many different forum posts, finding multiple relevant usage scenarios from different forum posts that can be used together to complete a complex development task, and so on.

In this chapter, we present a framework to automatically mine and summarize usage scenarios of an API from forum posts. An API usage scenario consists of five items: (1) **Code snippet.** A code example from a forum post, (2) **API.** The API associated to the code example, (3) **Summary description.** A short description of what the code snippet does, (4) **Reactions.** The positive and negative reactions of other developers *potentially* towards the solution provided in the code example, and (5) **Title.** A descriptive title of the usage scenario. We design four algorithms to summarize the usage scenarios of a given API:

Statistical: We visualize the usage statistics.

Single Page Summary. We present all the usage scenarios in one single page.

Type-Based Summary. We group usage scenarios by API types (class, interface).

Concept-Based Summary. We group usage scenarios by similar *concepts* [42]. Each concept denotes a potential API feature.

The design of the algorithms is motivated by our findings on the improvement suggestions of API documentation (in Chapter 3), and how the summarization of API reviews can offer insights into the API documentation (Chapter 5).

We have implemented the algorithms in our tool Opiner. In Figure 6.1, we show screenshots of Opiner usage scenario summarizer. The user can search an API by name to see the different usage summaries of the API ①. The front page also shows the top 10 APIs for which most number of code examples were found ②. Upon clicking on the search result, the user is navigated to the usage summary page of the API ③, where the summaries are automatically mined and summarized from Stack Overflow. A user can also click on each of the top 10 APIs listed in the front page. An example usage scenario in Opiner is shown in ④.

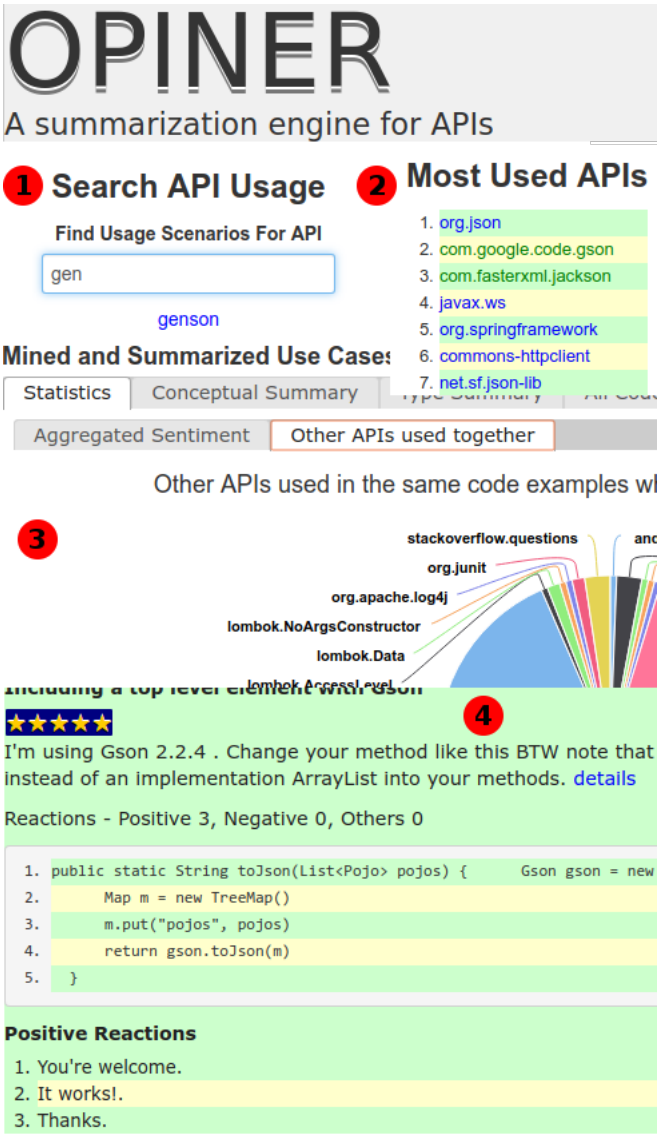

Figure 6.1: Screenshots of the Opiner API usage summarizer

We assessed the usefulness of the API usage summaries in Opiner by answering four research questions in order:

RQ1: How accurate is the association of a code snippet to an API in Opiner?

- ***Motivation.*** The first step for mining and summarizing usage scenarios about API is to associate each code example in forum post to an API about which the code example is provided. It is necessary to be able to link a code example to an API precisely (i.e., precision) [35]. Moreover, for summarization, it is necessary to be able to link as many code examples as possible (i.e., recall).

- ***Approach.*** Our developed algorithm can heuristically associate a code example to an API mention in a Stack Overflow forum post with a precision of 88.2% and a recall of 98.2%.

RQ2: How useful is the Opiner API usage scenario summarization engine to support development tasks?

• ***Motivation.*** We developed the usage summaries in Opiner to assist developers in their programming tasks. Based on our previous findings on the shortcomings of API formal documentation, it was necessary that the usage summaries in Opiner would be able to alleviate some of the key pain points of developers while using an API, such as, the amount of time and effort required to learn to use an API.

• ***Approach.*** We conducted a user study involving 34 participants. The participants completed four separate coding tasks, first using Stack Overflow, second using API official documentation, third using Opiner, and fourth using all the three resources (i.e., Opiner, Stack Overflow, and official documentation). We observed that developers wrote more correct code, spent less time and effort while using Opiner.

RQ3: How informative are the four API usage scenarios summarization techniques in Opiner to the developers?

• ***Motivation.*** The development needs of developers can vary depending on the needs. As we previous research [19] observed, the specific needs from API documents can also vary depending on the programming tasks. We designed and developed the four types of API usage summarization techniques in Opiner to assist developers in their diverse development needs. As such, it was necessary to determine whether and how the different summaries are useful and under what specific development scenarios.

• ***Approach.*** We conducted a survey with 31 of the 34 participants who participated in the coding tasks that we used to answer RQ2 above. The participants completed the survey after the completion of their coding tasks. We asked them to compare the four summarization techniques in Opiner under five different development scenarios (selection, documentation, presentation, awareness, and authoring). The participants were asked to provide their rating in a Likert scale. In a series of open-ended questions, we also asked the participants to justify their ratings.

We found that different summarization techniques can be useful for the different development needs. For example, statistical summaries are mainly useful to produce presentation slides to overview an API, while the concept-based summaries are useful for both novice and experienced developers in their completion of a programming task. Our findings confirm the observations of previous research [19] that the specific needs from API documents can vary depending on the programming tasks.

RQ4: How useful are the Opiner usage summaries over API documentation?

• ***Motivation.*** The API documents (formal or developer forums) are not designed to summarize API usage scenarios. As such the navigation along API documents can be troublesome for a developer for various reasons, e.g., when the information is not readily available, etc. To the best of our knowledge, the generation, Opiner is the first framework developer to automatically generate summaries of usage scenarios from forum posts. Therefore, it is necessary that the usage summaries in Opiner can offer improvements over API documents and show that Opiner can add complementary

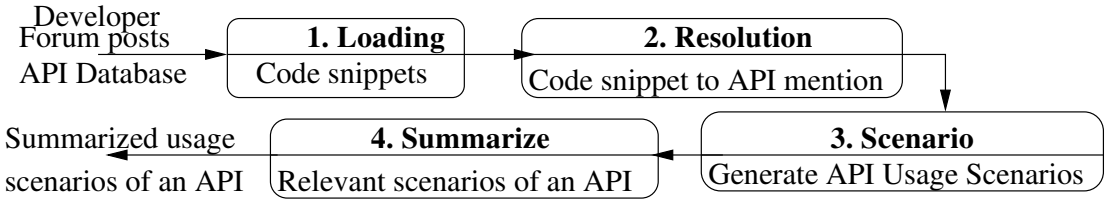

Figure 6.2: The major steps of the analysis framework to produce summarized usage scenarios of an API

values to the existing API documents, thereby, improving the usage of an API.

• ***Approach.*** We conducted another survey with 31 of the 34 participants who participated in the coding tasks that we used to answer RQ2 above. We asked them questions with regards to the observed value Opiner might have added in their completion of the tasks over the API documents in a series of open and closed-ended questions. More than 80% of the participants reported that Opiner summaries offered improvements over formal and informal documentation by presenting usage summaries *relevant* to the development tasks.

Chapter Organization. We present the framework to mine and summarize usage scenarios in Opiner in Section 6.1. We evaluate the resolution accuracy of our code examples to API linking algorithm in Section 6.2. In Section 6.3, we describe a user study that we conducted to evaluate the effectiveness of opiner usage summaries to assist developers in their coding tasks. In Section 6.4, we describe another user study that we conducted to compare the usefulness of the four usage summaries in Opiner. In Section 6.5, we describe our final user study that we conducted to compare the usefulness over both API formal and informal documentation. We discuss the threats to the validity in Section 6.6 and conclude the chapter in Section 6.7.

6.1 Analysis Framework of Opiner Usage Summarizer

The input to the framework supporting Opiner is a list of forum posts. The output is a list of APIs each associated with the summarized usage scenarios discussed about the API in the forum posts. Our analysis framework operates in four major steps (see Figure 6.2), each supported by Opiner's development infrastructure.

1) **Dataset Creation:** We load and preprocess posts from Stack Overflow. A post can be a question, answer, or a comment.

2) **API Resolution:** We heuristically associate a code snippet to an API mentioned in the same thread.

3) **Scenario Generation:** For each code snippet, we create a scenario by producing a short textual description of the snippet and by linking it with the sentiments of the developers.

4) **Scenario Summarization:** We mine all the usage scenarios of an API and group those using our four proposed API usage scenario summarization algorithms. We present the summaries in the Opiner web-based user interface.

Table 6.1: Descriptive statistics of the dataset

Threads	Posts	Sentences	Words	Valid Snippet	Lines	Discarded	Users
3048	22.7K	87K	1.08M	8596	68.2K	4826	7.5K
Average	7.5	28.6	353.3	2.8	7.9	1.6	3.9

Table 6.2: Descriptive statistics of the Java APIs in the database.

API	Module	Version	Link	AL	AM	AV
12,451	144,264	603,534	72,589	5.8	11.6	4.2
AM = # Avg Modules Per API, AL = #Avg Links per API, AV = #Avg Versions Per Module						

6.1.1 Usage Scenario Dataset Creation

Our API usage scenario dataset was produced by collecting all the code examples and textual contents found in the Stack Overflow threads tagged 'Java+JSON', i.e., the threads contained discussions and opinions related to the json-based software development tasks using Java APIs. As we discussed in the previous chapter (Section 5.1), this dataset contains discussion and usage examples about different competing Java APIs for JSON parsing, to support diverse development needs.

We 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*; (c) *hyperlinks*[1]; and (d) *natural language text* representing the rest of the content. (3) We tokenize the text and tag each token with its part of speech.[2] (4) We detect individual sentences in the *natural language text*. (5) We discard the following *invalid* code examples during our parsing: (a) Non-code snippets (e.g., XML, JSON extract), (b) Non-Java snippets (e.g., JavaScript). We identified the invalid code examples using techniques originally developed by Dagenais and Robillard [59].

In Table 6.1 we show descriptive statistics of the dataset. There were 22,733 posts from 3,048 threads with scores greater than zero. We did not consider any post with a negative score because such posts are considered as not helpful by the developers in Stack Overflow. There were 8,596 *valid* code snippets and 4,826 invalid code snippets. We only included the valid code snippets in our usage summaries. On average each valid snippet contained at least 7.9 lines. The last column "Users" show the total number of distinct users that posted at least one answer/comment/question in those threads. To identify uniqueness of a user, we used the user_id as found in the Stack Overflow database. On average, around four users participated in one thread, and more than one user participated in 2,940 threads (96.4%), and a maximum of 56 distinct users participated in one thread [131].

To locate APIs in the code examples, we use 1) all the Java APIs collected from the online Maven Central repository [12][3], and 2) all the Java official APIs from JavaSE (6-8) and EE (6 and 7). We

[1] We detect hyperlinks using regular expressions.
[2] We used the Stanford POS tagger [113].
[3] We crawled the entire Maven Central database in January 2014

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 JavaSE official tutorial [127]). In Table 6.2, we show the summary statistics of the API database used in this study. All of the APIs (11,576) hosted in Maven central are for Java. We picked the Java APIs from the Maven Central repository for the following reasons:

1) It is considered as the primary resource for hosting and searching for Java APIs with over 70 million downloads every week [151].

2) It offers a comprehensive set of APIs to search the code bases of the hosted APIs, by the API types (class, interface) and version [152]. This feature is useful for the following reasons: (i) We can leverage the APIs to search for API elements and use that to link code examples in the forum posts to the APIs. (ii) We do not have to create an offline database to develop our algorithms to associate code examples to APIs. As of now, offline databases needed to be constructed to trace API elements in the forum posts [35]. (iii) The database is continuously updated, thus we can get the most up to date information about an API. With an offline database, such as the one used by Baker [35], we cannot have the most recent information about an API. Indeed, when we contacted the authors of Baker about the availability of their database, we were informed that the database was not updated and it was missing API fields (e.g., method information) that were used in their development of Baker.

6.1.2 Resolving a Code snippet to an API

Given as input a code snippet from a forum post, we associate the snippet to one or more API about which the code snippet was provided in the corresponding post. Thus, we associate a code example to an API mentioned in the same post or thread. Consider the different ways such mentions are found in forum posts in Figure 6.3. The five code examples are taken from three different posts in Stack Overflow, each from three different threads, ① from one thread [3], ②-④ from another thread [153], and ⑤ from another thread [154]. For ①, our algorithm associates the code example to the Google GSON API, because there was a mention 'GSON' in the textual contents of the post containing the code example and the Google GSON API has a class type `com.google.gson.Gson`. We associate the code example ② to Google GSON API again, because the immediately preceding text of the code example has the mention 'Google Gson' and the two types `JsonObject` and `JsonParser` were found in the API. We assign code example ③ to the org.json API because the API is mentioned in the text preceding the code example. We do not assign ③ to the Gson API even though it is also mentioned in the preceding text, because even though the type in the code example `JsonObject` can be found both in the Gson and org.json API, in all the Java JSON posts, the type `JsonObject` is more frequently associated to org.json than to Gson. Similarity, we associate ④ to the com.fasterxml.jackson API because it has a type `ObjectMapper` and there is a mention of 'Jackson' in the preceding text. We assign ⑤ to the API com.fasterxml.jackson because even though the answer does not have any mention of 'Jackson', the title of the thread containing the answer has a mention of 'Jackson' and the annotation type `JsonInclude` can be found in the com.fasterxml.jackson.

Therefore, the association of a code example to an API in our algorithm leverages both the API mentions in the forum posts and the API types found in the code examples using a series of heuristics. Our algorithm operates in two steps:

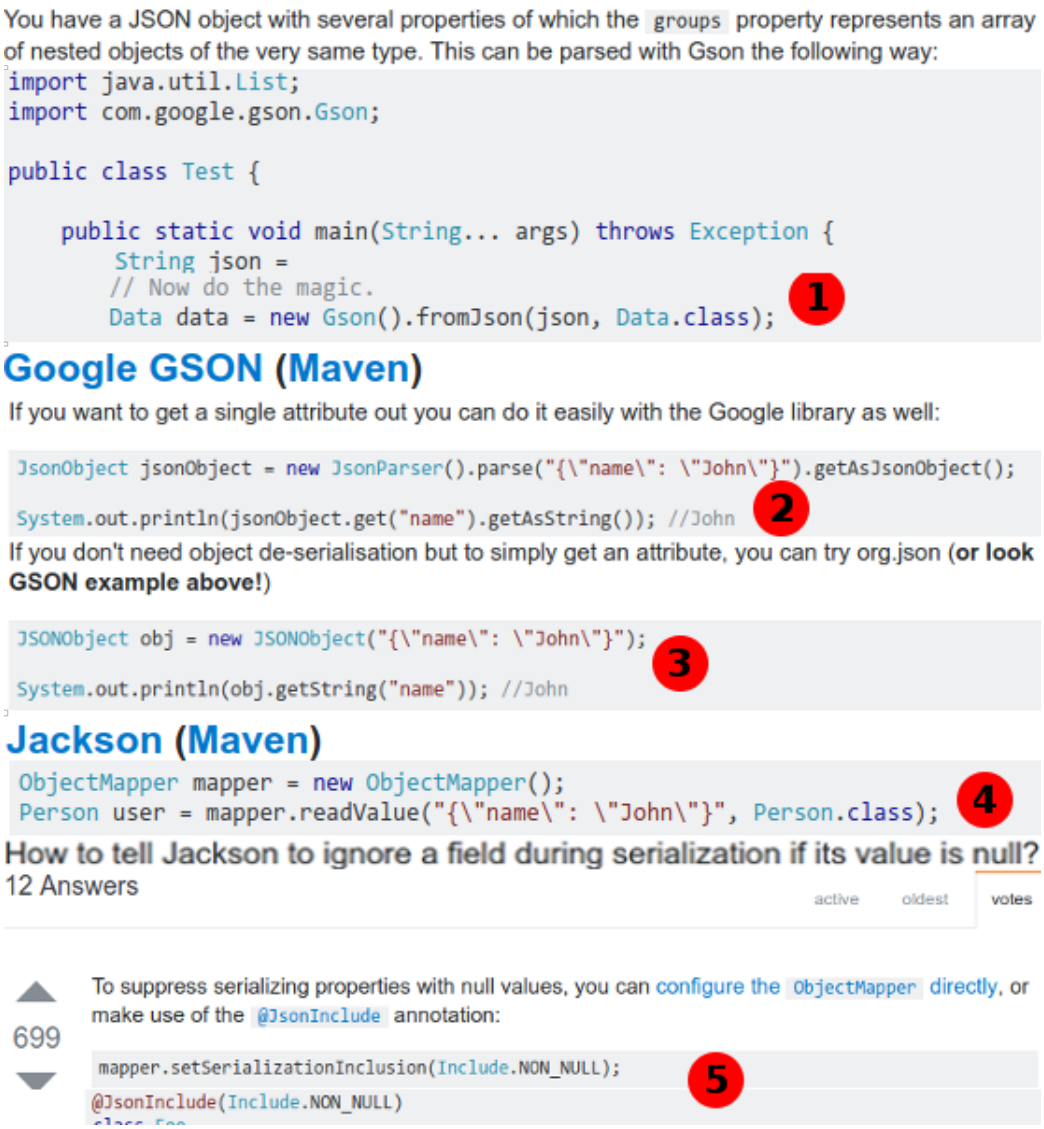

Figure 6.3: How APIs & snippets are discussed in forum posts.

1) **Generation of Association Context:** For each code example found in a thread, we identify the *valid* code types in the example and the probable API mentions that may be associated to the code example.

2) **Association of a code example to an API:** We apply a set of heuristics to associate a code example to an API.

We elaborate each step with examples below.

6.1.2.1 Generation of Association Context

Given as input a code snippet s_i and forum thread where the snippet was found, we construct an *Association Context* a_i for the snippet as: $a_i = \{C, M\}$, where C corresponds to all the *valid* code

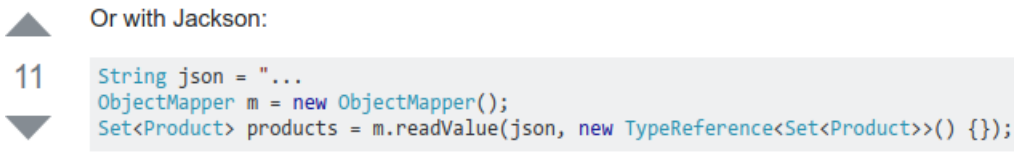

Figure 6.4: A popular code example with a syntax error [3]

types (discussed below) in the code example and M corresponds to the API mentions that are most likely associated to the code snippet s_i.

(1) Code Types: We identify types (class, interface, and annotation) in each post using Java naming conventions, similar to previous approaches [59, 60] (e.g., camel case). We collect types that are most likely not declared by the user. Consider the following example [155]:

```
import com.fasterxml.jackson.databind.*
private void tryConvert(String jsonStr) {
    ObjectMapper mapper = new ObjectMapper();
    Wrapper wrapper = mapper.readValue(...);}
```

We add `ObjectMapper` into our code context, but not the type `Wrapper`, which was declared in the same post before as: `public Class Wrapper.`

Code Parser. We parse code snippets using a hybrid parser combining ANTLR [156] and Island Parser [157], based on the observation that code examples in the forum posts can contain syntax errors which an ANTLR parser is not designed to parse. However, such errors can be minor, and thus the code example can still be considered as useful by other developers. Consider the code example in Figure 6.4. An ANTLR parser with Java grammar fails to parse it at line 1 and stops there. However, the post was still considered as helpful by others (upvoted 11 times). Our hybrid parser works as follows: a) We split the code example into individual lines. For this paper, we focused only on Java code examples. Therefore, we use semi-colon as the line separator indicator. b) We attempt to parse each line using the ANTLR parser by feeding it the Java grammar provided by the ANTLR package. If the ANTLR parser throws an exception citing parsing error, we feed it to our Island Parser. c) For each line, we collect all the types as described above.

Inference of Code Types From Variables: An object instance of a code type declared in a separate post can be used in another post without any explicit mention of the code type. For example, consider the example:

```
Wrapper = mapper.readValue(jsonStr , Wrapper.class);
```

In this case, we associate the `mapper` object instance to the `ObjectMapper` type of Jackson, because the object instance was declared in a code snippet of a separate post (but in the same thread).

Generating Fully Qualified Names (FQNs). For each valid type detected in the parsing, we attempt to determine its fully qualified name by associating it to an import type in the same code example. Consider the following example:

```
import com.restfb.json.JsonObject;
JsonObject json = new JsonObject(jsonString);
```

We associate `JsonObject` to `com.restfb.json.JsonObject`. We only use the import types from the same code example to infer the FQNs based on the observations that a type name can be shared across

APIs. In Maven central, 981 distinct APIs have a class `JSONObject`.

(2) API Mentions. For each code snippet, we identify API mentions found in the same thread about which the code example may be provided. To detect API mentions, we apply both exact and fuzzy name matching techniques between API name in our database and each token in the textual contents of the forum post. We also apply similar name matching between each module name in the database and the texts in the forum post. Thus each such API mention can be linked to one or more APIs in our API database. We consider each such linked API as a potential API candidate and generate a *Mention Candidate List* (MCL) for each mention.

Snippet Mention Context. For each code snippet, we associate one or more detected API mentions based on the following proximity filters between the snippet and each API mention: **(P1)** Same Post Before: each mention found in the same post, but before the code snippet. For example, the mention 'gson' in ① of Figure 6.3 is placed in this bucket for ①. **(P2)** Same post After: each mention found in the same post, but after the code snippet. **(P3)** Same thread: all the mentions found in the title and in the question. For example, for the code snippet ⑤ in Figure 6.3, the mention 'jackson' is included in the 'same thread' context for the code snippet. Each mention is accompanied by a Mention Candidate List, i.e., a list of APIs from our database. Thus each code snippet can have one or more candidate APIs.

6.1.2.2 Association of a code example to an API

For a given code snippet with a list a APIs as a potential candidates, we associate the snippet to one of the APIs. First, we compute the *Type Similarity* between an API in the Mention Context and the code snippet, which determines the coverage of types in a snippet for a given API. Second, we attempt to associate the snippet to the API with the most Type similarity. If we have a tie there, we apply a number of filters to associate an API to the snippet (discussed below).

Filter 1. Type Similarity.

First, we create a bucket of distinct valid types for the code snippet. For a type resolved to a FQN, we only put the FQN in the bucket. For each valid types in the bucket, we search for its occurrence in each candidate API of the snippet. We compute the type similarity between a snippet s_i and a candidate c_i as follows:

$$\text{Type Similarity} = \frac{|\text{Types}(s_i) \bigcap \text{Types}(c_i)|}{|\text{Types}(s_i)|} \tag{6.1}$$

$\text{Types}(s_i)$ is the list of types for snippet s_i in bucket. $\text{Types}(c_i)$ is the list of the types in $\text{Types}(s_i)$ that were found in the types of the API[4]. When no types were matched between a candidate API and the snippet, we assign $\text{Types}(c_i) = \emptyset$, i.e., $|\text{Types}(s_i) \bigcap \text{Types}(c_i)| = 0$.

Filter 2. Association Heuristics.

We apply two types of heuristics to associate a code snippet to an API.

H1. Proximity Rules.

We apply the proximity filters in the following sequence: (P1) Same Post Before,

[4]We used the online code repository API of Maven Central (`https://search.maven.org/#api`)

If, by any change, you are in an application which already uses http://restfb.com/

```
import com.restfb.json.JsonObject;

...

JsonObject json = new JsonObject(jsonString);
json.get("title");
```

Figure 6.5: Example of linked presence filter [3]

(P2) Same Post After, (P3) Same Conversation, (P4) Same Thread. We stop the association process of a code snippet once we can associate an API to the code snippet. We apply the following rules under each filter:

– **F1. Maximum Type Similarity.** We associate the snippet to the API with the maximum type similarity. In case of more than one such APIs, we create a *hit list* by putting all those APIs in the list. Each entry is considered as a potential hit.

– **F2. Linked Presence.** We check if there is a URL to the resources of an API in the textual contents of the filter (e.g., all the texts preceding a snippet in a post will be considered for the 'Same Post Before' filter). If there is a URL and the API referred to by the URL is an API from the hit list, we associate the snippet to the API. Consider the example in Figure 6.5. We associate it to the com.restfb API, because the URL just above the code snippet refers to the homepage of the API and the API is in the hit-list of the snippet.

– **F3. Name Similarity** If there is an exact match between and API mention and the name of a candidate API in the hit list, we associate the snippet to the API. If not, for each mention in a given proximity filter for a code snippet, we compute the name similarity between the mention and each of the candidate APIs in the hit list as follows:$w = \frac{|\text{Tokens}(M) \cap \text{Tokens}(C)|}{|\text{Tokens}(M) \cup \text{Tokens}(C)|}$. If a mention shows similarity with an API in the hit list at least to a minimum threshold level, we associate the snippet to the API. For the current version of Opiner, we used a similarity threshold of 80%. In our future work, we will investigate the impact of the different similarity thresholds.

H2. Dependency Influence.

For the APIs in a hit list, we create a *dependency graph* by consulting the dependency list of each API in the database. Each node in the graph corresponds to an API from the hit list. An edge is established between two APIs in the graph, if one of the APIs has a dependency on the other API. From this graph, we find the 'Core' API with the maximum number of incoming edges, i.e., the API on which most of the other APIs in the hit list depend on. If there is just one 'Core' API, we assign the snippet to the.

H3. Coverage Rules.

We apply the coverage rule on a code snippet to which we cannot associate an API using the previous two heuristics (i.e., proximity and dependency). We developed the coverage rule based on two observations: (1) developers tend to refer to the same API types in many different forum posts, and (2) when an API type is well-known among developers, developers tend to refer to it in the code examples without mentioning the API in their posts (see for example [158]).

Our coverage rule operates by associating an API type to an API which is most frequently associated to the type in other forum posts. Given as input a code snippet S_i,

Table 6.3: Distribution of Code Snippets By APIs

API	Snippet	Avg	Top 10	Snippets Covered	Min	Max
175	8596	49.1	5766	67.1%	1	716

the rule works in the following steps:

(Step 1) For each type T_i in S_i, we determine how frequently T_i appeared in other already associated code snippets A. We create a matrix M as $C \times T$ where each row corresponds to an API C_i found in the already associated code snippets. Each column corresponds to an type T_i in S_i. We only add an API C_i in the matrix if at least one of the types in T was linked to the API in the already associated code snippets. Each co-ordinate in the matrix for a given API C_i and type T_i contains the frequency $freq(T_i, C_i)$. For example, if a type T_i is linked to C_i in five already resolved code snippets, we assign $freq(T_i, C_i) = 5$.

(Step 2) For the given input snippet S_i, we compute the coverage of each API C_i in the matrix M by computing the sum of all frequency values of all the types:

$$Coverage(C_i, S_i) = \sum_{i=1}^{n} freq(T_i, C_i) \tag{6.2}$$

(Step 3) We associate the code snippet S_i to the API C_i with the maximum coverage value. The proximity filter we associate to this rule is 'Global'.

In Table 6.3, we show the distribution of code snippets matched to APIs using our proposed algorithm. The 8596 code examples are associated with 175 distinct APIs. The majority of the code examples (67%) were associated to 10 of the most well-known APIs for JSON parsing for Java (e.g., org.json, com.google.code.gson, org.fasterxml.jackson). For example, 1498 code examples were associated to the API org.json and 1053 to the API com.google.code.gson and 802 to com.fasterxml.jackson. For a developer looking for quick but informed insights from the usage of an API, navigating through such huge volume of code examples could prove time-consuming and problematic. Therefore, the usage summaries in Opiner can be more useful to assist developers navigating the usage scenarios of popular APIs.

6.1.3 Generating API Usage Scenarios

Given as input a code snippet, we produce a usage scenario in Opiner by associating the following four items to the snippet: 1) An associated API, 2) A summary description of the snippet, 3) A one line title of the scenario, and 4) the positive and negative reactions of other developers towards the snippet.

We associate a given code snippet to one or more APIs using the API association technique described in the previous section. In this section, we explain the creation of the three other items: summary description, title, and reactions. The circle ④ in Figure 5.1 shows an example usage scenario from the API com.google.code.gson.

Summary Description. Given as input a code snippet and an associated API, we generate the short description of the snippet by picking selected sentences between the associated API and the code snippet in the same post as follows. **(Step 1)** We identify all the sentences between the API and the snippet from the post where the snippet was found. We sort the sentences in a bucket in the order they appeared. Therefore, the sentence that is the closest to the snippet is placed at the top. **(Step 2)** From this bucket, we start pulling the sentences from the top of the bucket until the number of sentences surpass a certain predefined window size. For the current version of Opiner, we used a window size of two to ensure brevity in the description. In our future work, we will investigate the impact of using different lengths in the summary description. **(Step 3)** We concatenate the sentences from the previous step to create the description.

Title. We assign a code snippet the title of the thread in Stack Overflow where the code snippet was found. Similar approach was followed by Subramanian et al. [35] in their Baker tool and Truede and Robillard [87] in their API insight tool. Both approaches annotate the Javadocs of a API type by linking selected Stack overflow posts that can be associated to the API type. As the title for each link, they used the title of the corresponding threads of the posts. In our future work, we will investigate the feasibility of shorter title (e.g., bigrams/trigrams). 5described properly by the thread title.

Reactions. We group reactions by their polarity, i.e., as positive and negative. We collect all the reactions to a given code example in a forum post as follows. If the code example was found in an answer post, we collect comments of other developers posted as replies to the answer in the comment section of the post. For the current version of Opiner, we do not consider the code examples found in a question post and only consider code examples found in the answers to the question, or in the comments posted as replies to the answer/question. Similar approach was previously adopted by Subramanian et al. [35] in their Baker tool. If the code example was found in a comment post, we first detect the corresponding answer/question post where the comment was found. We then collect the comments posted afterwards as part of the post. We detect individual sentences in the comments and sort those in time, with the most recent comment appearing at the top and its first sentence as the topmost. We detect sentiment in the sentences. To detect sentiments in the sentences, we used a rule-based algorithm, similar to what we did in our previous work [48]. We then group the sentences based on polarity into three buckets: positive, negative and others.

6.1.4 Summarizing API Usage Scenarios

Given as input all the mined usage scenarios of an API from forum posts, we design four different summarization algorithms to generate a consolidated, yet comprehensive overview of all the usage scenarios of the API:

- **Statistical:** Presents the usage statistics relevant to the scenarios using visualization.
- **Single Page:** Presents all the usage scenarios in a single page.
- **Type-Based:** Groups usage scenarios by the API types
- **Concept-Based:** Groups usage scenarios by similar concepts, each concept denoting specific API feature.

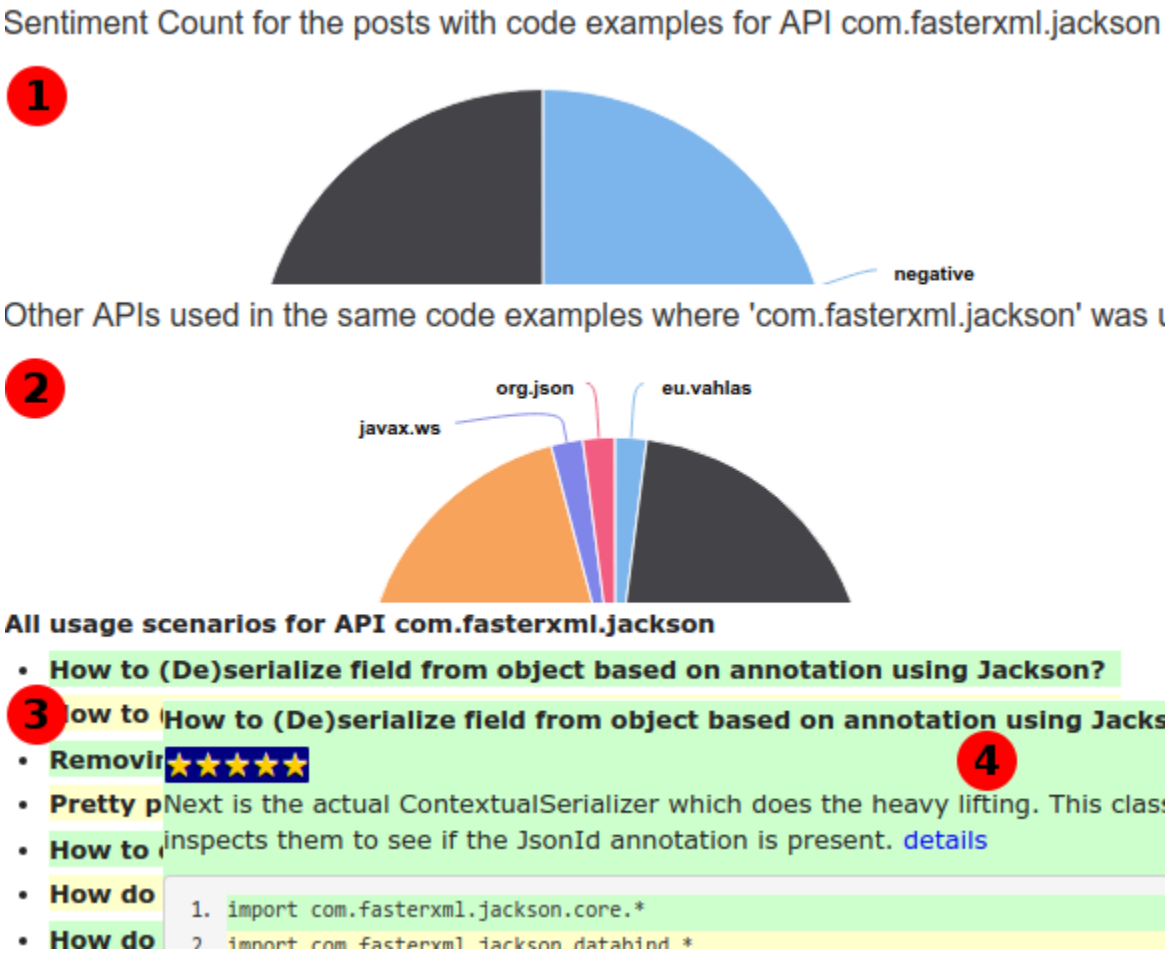

Figure 6.6: Statistical & single page summary for API Jackson

We describe the algorithms below.

6.1.4.1 Statistical Summary

In our work on API review summarization (Chapter 5), we observed that developers reported to have increased awareness about an API while seeing the visualized overview of the API reviews from developer forums. We take cues from our statistical API review summaries and produce two types of statistical summaries for every API in our dataset. The summaries are based on 1) the overall sentiments in the reactions to the usage scenarios of the API, and 2) the usage of other APIs in the same code examples of the API. In Figure 6.6, we show screenshots of the statistical summaries of the API com.google.code.gson in Opiner (①,②, and ④):

Sentiment Overview.

In Figure 6.6, ① and the star rating in ④ show the sentiment overview of an API. ① shows the overall distribution of positive and negative opinions in the forum posts where the code examples of an API were found. We only counted the positive and negative opinions in the subsequent replies to a post with a code example. For example, for an answer post, the comments

to the answer were considered as replies. In addition, for each usage scenario, we show a five-star sentiment rating of the scenario, adopted from the star rating schemes we proposed previously in Chapter 5. For every usage scenario of an API, we collect all the positive and negative opinions about the scenario, and compute the star rating as: $R = \frac{\#Positives \times 5}{\#Positives + \#Negatives}$.

Co-Used APIs.
For a given API, we show how many other APIs were used in the code examples where the API was used. For example, in Figure 6.6, ② shows that the API javax.ws was frequently used alongside the Jackson API in the same code examples. Because each API offers features focusing on specific development scenarios, such insights can be helpful to know which other APIs besides the given API a developer needs to learn to be able to properly utilize the API for an end-to-end development solution. We computed the pie chart ② as follows: (a) Identify all the other APIs discussed in the same code example. We did this by looking at imports and by matching the different types in the code example against all the APIs in the mention candidate lists associated with the code example. (b) Take a distinct count of API for each example, i.e., even if API javax.ws is used more than once in a code example for Jackson, we only record that javax.ws was used alongside Jackson API in the code example. Thus the bigger a pie in the chart, the more distinct code examples were found where the two APIs were used together.

6.1.4.2 Single Page Summary

In Figure 6.6, ③ and 4 show the single page summary of all usage scenarios of the API Jackson. This summary puts all the usage scenarios of an API in one single page, sorted by time, i.e., the most recent usage scenario is placed at the top.

Our single page summary format for an API was motivated by two observations:

Python Formal Documentation. A single page documentation format for an API is widely followed for Python APIs. In our survey of IBM developers (Chapter 3), we observed that the developers when mentioned about Python API documentation, it was only for the good examples and not a single one for the bad documentation experience.

Fragmentation. One of the top three presentation issues in the formal documentation, as we observed in our survey at IBM (Chapter 3), is the excessive number of fragmentation in the Javadocs, e.g., each class is described in separate pages, tutorial or getting started pages are divided into multiple pages, etc.

Our single page summary adheres to the Python formal documentation format. However, it differs from the Python documentation in two key areas:

On Demand Update. Our usage scenarios are generated automatically from the forum posts and thus can be automatically updated and easily maintained, thereby paving the way on demand generation of API documentation from developer forums [92].

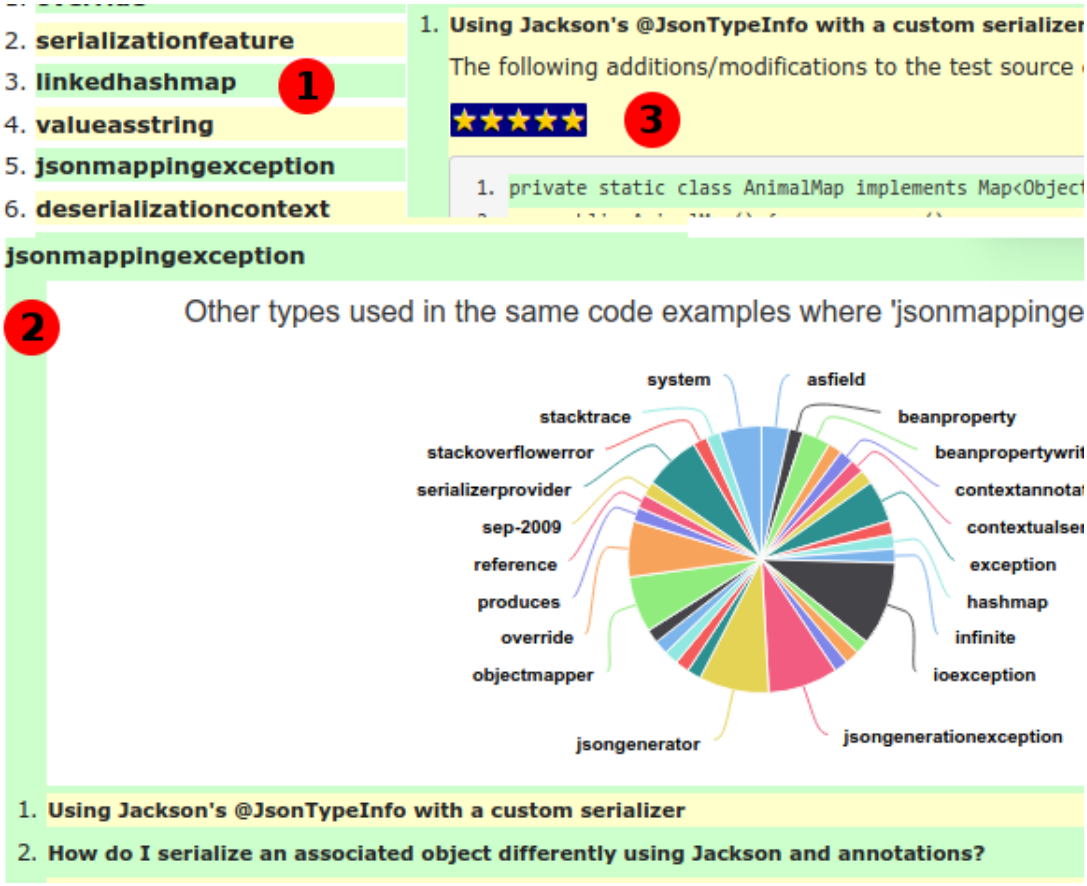

Figure 6.7: Type based summary for API Jackson

Recency. We sort the usage scenarios by time, so the most used scenario is always placed at the top. This helps prevent another documentation problem, the obsoleteness of the usage examples in the documentation (C4 in Section 3.3).

6.1.4.3 Type Based Summary

We aggregate all the usage scenarios per API type and present those in one single page in Opiner (see Figure 6.7). We produce this summary as follows: 1) We identify all the types in a code example associated with the API 2) We create a bucket for each type and put each code example in the bucket where the type was found. We present this bucket as a sorted list by time (most recent as the topmost) by labeling it as the name of the type. 3) For a given bucket corresponding to an API type, we also count how frequently other types of the same API were used in the bucket. We create a pie-chart that shows how frequently other types from the same API were co-used in the same code example where the given API type was used. 4) We then sort the buckets by time, i.e., the type that was used most recently would is placed at the top. Our type-based summary was produced based on the following observations:

Javadoc. Developers are familiar with the type-based (class, interface, annotation) description in the Javadocs and can leverage the javadocs through the structural connection among the

types as shown in the javadocs. Previous research showed that developers relied on structural information about code types while completing development tasks [159]

Structural Viewpoints. While structural viewpoints are required, too much structural information on how API types are connected to each other may be a blocker to get quick and actionable insights about an API. As we observed in our survey at IBM (Chapter 3), developers are not interested to see how an API type is hierarchically connected to other types, e.g., the java.object class is the super class of all other java class. Putting this information in every javadoc page is unnecessary. Rather developers wanted to see how an API type can be used based on examples and within the *context* of other types of the same API.

Our type-based summary adheres to the javadoc documentation format of grouping usages by API types, but also offer the following improvements over the standard javadocs (to address the two observations mentioned above):

Usage Scenarios vs. API Methods. Rather than grouping each methods of an API for a given API type under the usage description of the type, we present all the usage scenarios of the API type in our type-based summary. This decision was made consciously based on the fact that such method information for a given API type can now be obtained easily through the auto-completion features shipped with any standard IDE (e.g., Eclipse Java).

Recency. The usage scenarios for each API type can be automatically refreshed whenever a new usage scenario is posted in the forum post.

See Alsos. Similar to 'see alsos' in the Javadocs for each API type where similar other API types are suggested, we also show other API types that are co-used mostly for a given API. Contrary to the manual process involved in those see alsos in the Javadocs, our co-used APIs are automatically inferred from the code examples.

Our type-based summaries differentiate from the following related work:

Live API documentation (Baker) [35]. The Baker tool provides a browser extension that can annotate a the pages of an API type in a Javadoc with the link of a forums post where a code example for the API type was found. In Opiner, we put all the API types in one single page. All the usage scenarios of a given API type are provided under a collapsible box under the type. The API type itself functions as toggle button, i.e., by clicking on it a user can either see all the code examples or collapse the view to hid the scenarios. In this way, we ensured that the user can see all the usage scenarios of the API types in one single page, thereby adhering the venerable single page documentation of Python documentation. Moreover, while Baker simply provides a link to a forum post where the code example is found, our tool Opiner shows a usage scenario. A usage scenario not only contains a code example, but also shows other relevant information, such as, short summary description and positive and negative opinions of other developers towards the code example. Such information thus can provide instant insight of the usefulness of a code example. For example, a code example with mostly negative opinions is unlikely to be useful and thus can be avoided.

API insight [87]. Treude and Robillard [87] proposed a tool to annotate the Javadocs of API types with key insights about them mined from Stack Overflow. Such insights are collected from the

textual contents of the forum posts where the the API types are discussed. It is not necessary for a code example of an API type to be present in the same forum post from where the insight is mined. In contrast, for a usage scenario of an API type in Opiner, all of the following five items are considered as required: 1) a code example 2) a summary description, 3) a title, and 4) positive and negative opinions. The positive and negative opinions in Opiner are collected from comments to a forum post (where a code example was found). Opiner's focus is suited to assist developers in their programming tasks by showing readily available code snippets of an API as well as in their instant decision process on whether a code snippet is a good fit.

6.1.4.4 Concept Based Summary

We detect *concepts* in the usage scenarios of an API, where a concept denotes a specific API feature that often requires the usage of a set of predefined API types. Our definition of a concept is adopted from our previous work [42]. We observed that developers in multiple unrelated software often used a pre-defined set of of the same API types to complete a development task. We denoted such a set of API types as a concept. We also observed patterns of concept usage. For example, to send or receive messages over HTTP using the HttpClient API [43], a developer would first need to establish an HTTP connection followed by the usage of the HttpGet or HttpPost method to send or receive message.

In Figure 6.8, we show an overview of the concept-based summary for the API Jackson in Opiner. In Opiner, each concept ① consists of one or more similar API usage scenarios. Each concept is titled as the title of the most representative usage scenario (discussed below). Each concept is provided a star rating as the overall sentiments towards all the usage scenarios under the concept. The concepts are sorted by time. The most representative usage scenario of a concept is placed at the top. Upon clicking on a each concept title, the most representative scenario for the concept is shown ②. Other relevant usage scenarios of the concept are grouped under a 'See Also' ③. Each usage scenario under the 'See Also' can be further explored ④. Each usage scenario is linked to the corresponding post in Stack Overflow where the code example was found (by clicking the *details* word after the description text of a scenario).

The detection of similar API usage scenarios as a *concept* and the presentation of those concepts using both description, sentiments, and a list of 'See Alsos' are motivated by our observation of two problems developers experience while seeking and analyzing API usage scenarios from developer forums: **(P1)** It can be difficult for a developer to find the relevant usage examples of an API amidst many different forum posts, leading them to ask the same/similar question repeatedly in the forum posts. **(P2)** A solution may be scattered across multiple usage scenarios in many different disconnected forum posts. In Chapter 1, we provided two motivating examples from Stack Overflow to demonstrate the two problem (see Figure 1.4 and Figure 1.5).

Given as input all the usage scenarios of an API, we produce the concept-based summary of the API in the following steps:

1) **Concept Detection.** We detect concepts as frequent itemsets. Each itemset corresponds to a list of API types.

2) **Concept Assignment.** We assign each usage scenario to the most likely frequent itemset.

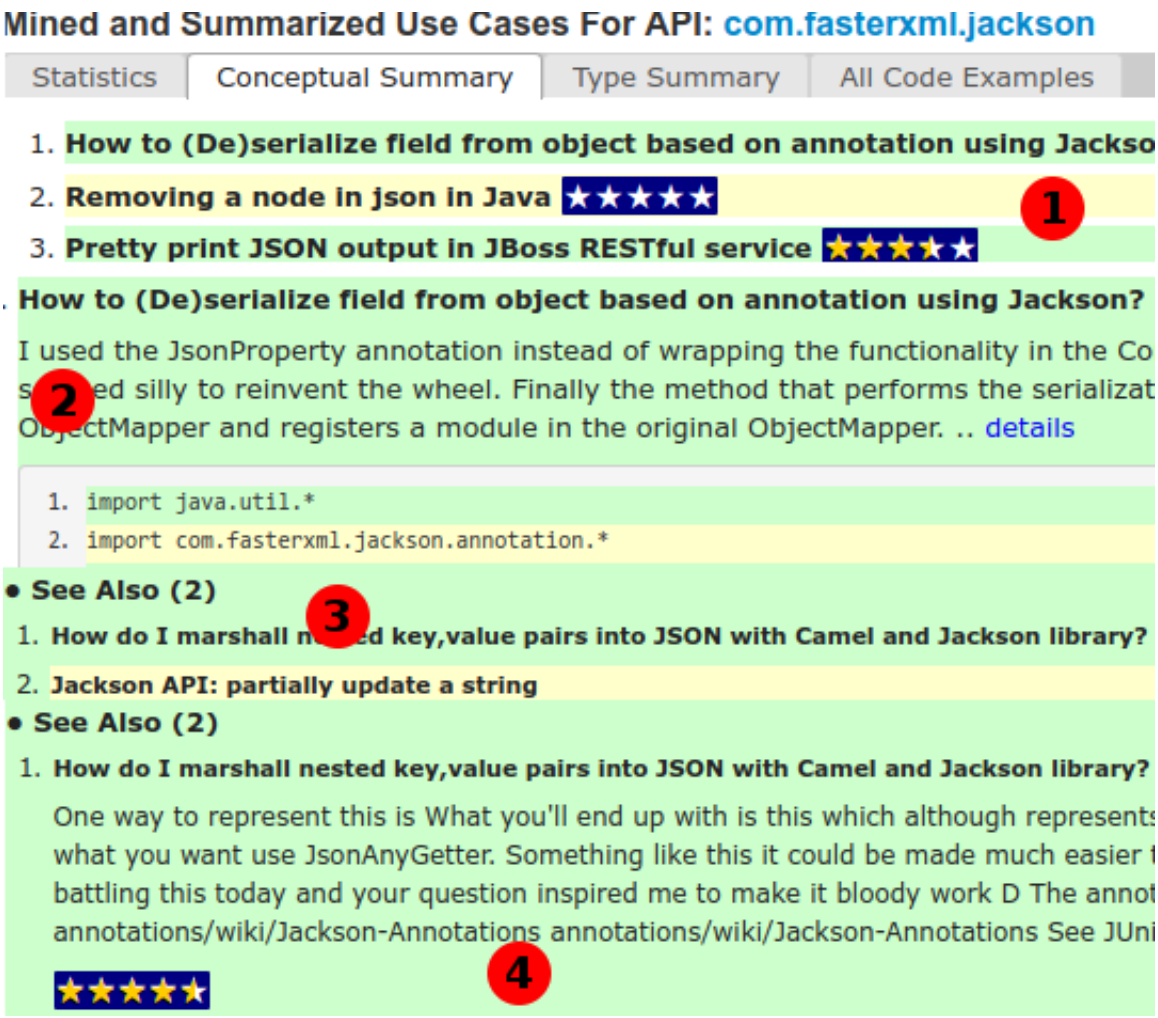

Figure 6.8: Concept based summary for API Jackson

3) **Concept and Usage Filtration.** We filter uninformative concepts and usage scenarios.

4) **Concept Fragmentation.** We divide usage scenarios under a concept into smaller buckets.

5) **Summary Generation.** We produce a summary by identifying representative usage scenario for each bucket.

We describe the steps below.

Step 1. Concept Detection.

We detect concepts as frequent itemsets from the code examples. Each item corresponds to a type of the API. We generate the frequent itemsets as follows. First, for each code example, we create a list with all the distinct types of the API found in the code example. Thus for each N code examples, we have N lists. Second, we apply the Frequent Itemset Mining [160] on the lists[5]. Frequent itsemset mining has been used in summarizing product features [115] and to find useful usage patterns from software repositories [57, 162]. Each frequent itemset is considered as a potential concept. Each itemset contains a list of API types (that were found as co-occurring frequently in the provided examples) and a support value (that shows many times that the itemset was found in the code example).

Step 2. Concept Assignment.

We assign each usage scenario to a concept as follows. First, we compute the similarity between a usage scenario and a concept as Similarity $= \frac{|\text{Types}(U) \bigcap \text{Types}(C)|}{|\text{Types}(U)|}$. We assign a usage scenario to the concept with the highest similarity. If more than one concept is found with the maximum similarity value, we assign the scenario to the concept with the maximum support.

Step 3. Concept and Usage Filtration.

We discard uninformative concepts with support less than a minimum threshold (we used 3 as originally used by Hu et al. [115] for product review summarization). We also discard the following usage scenarios as potentially uninformative: 1) **Length:** code examples with less than six lines. Code examples with less than six lines of code are normally considered as not informative enough (e.g., in code clone detection [163, 164]). 2) **Syntax:** Code examples with many syntax errors. We recorded syntax errors per line for each code examples during our parsing of the code snippets as part of our process to associate an API to a code snippet. If more than 50% of the lines of a code example had syntax errors, we discarded it as being of low quality.

Step 4. Concept Fragmentation.

For an API with many usage scenarios, a concept can be assigned a large number of usage scenarios. This normally happens for concepts with one or two API types, where the API types are among the most used types of the API (e.g., `ObjectMapper` in Jackson and `Gson` in com.google.code.gson). For example, the concept in Jackson with `ObjectMapper` as the only type, is assigned to more than 80 usage scenarios. While those usage scenarios may still offer similar features, they may differ among themselves when they correspond to different subtasks (e.g, one example can be using a subset of methods from the type, another using another

[5]We used the FPGrowth C implementation developed by Borget [161]

subset). We fragment those concepts into smaller clusters using two filters in the following order:

1) **Cloned Similarity.** First, we apply the NiCad clone detection tool [163] on all of the code examples associated to an itemset for an API. We use NiCad 3, which detects near-miss clones. We set set a minimum of 60% similarity and a minimum block size of five lines in the code example. We put each obtained clone pairs in separate buckets C_{clone}.

2) **Topic Similarity.** For the rest of the code examples in a concept that are not found as potential clones, we apply the topic modeling algorithm, Latent Dirichlet Allocation (LDA) [102] to find code examples with similar topic. LDA has been used in software engineering to find discussion topics and trends in Stack Overflow [103]. We apply LDA as follows:

(i) We collect all the titles from the usage scenarios. Recall that, the title of a usage scenario in Opiner is the title of the thread where the usage scenario was found.

(ii) We create a list of distinct titles from all the titles.

(iii) We tokenize each title and remove the stopwords.

(iv) We apply LDA on the produced tokens. The optimal number of topics is determined based on the topic coherence measure [104].[6]

(v) For each topic produced, we create a separate bucket for it C_{Topic}.

Step 5. Summary Generation

We create a list of buckets as $B = \{C_{Clone}, C_{Topic}, C_{Rest}\}$. C_{Rest} corresponds to the concepts that did not needed to be fragmented, C_{Clone} and C_{Topic} are the buckets generated during the concept fragmentation. Each bucket is considered as a concept in the Opiner UI.

In Opiner UI, we present each bucket as a list of four items: R, S, O, T. Here R corresponds to a usage scenario among all the scenarios in the bucket that we considered as the most representative of all the usage scenarios grouped under a given concept (discussed below). S corresponds to the rest of the usage scenarios in the bucket that we wrap under a 'See Also' sub-list for the bucket. O is the overall star rating for the bucket (discussed below). T is the title of the concept. We describe the fields below.

$F1$. **Representative Scenario.** Given as input all the usage scenarios of a given bucket, we determine one of the scenarios as the most representative of all as follows: (a) We sort the scenarios first by time and then by line. (b) From this sorted list, we pick the usage scenario that is the most recent in time. (c) If more than one scenario was found with the most recent date time, we pick the one with the maximum line size among those with the most recent date time.

$F2$. **See Alsos.** We put the rest of the scenarios under a 'See Also' list. We sort the scenarios in the 'See Also' list for a given concept based on time, by putting the most recent code example as the topmost.

$F3$. **Bucket Star Rating.** We compute the overall star rating of the concept by taking into account all the positive and negative reactions towards the usage scenarios.

$F4$. **Bucket Title.** We assign to the bucket the title of the most representative usage scenario.

In the Opiner UI, we sort the buckets as follows: (a) We assign to each bucket the date time of its representative scenario (b) We sort the buckets based on the date time in descending order, i.e., the most recent bucket is placed at the top.

[6] We used gensim [105] with c_v coherence to produce topics.

6.2 API Association Accuracy (RQ1)

In this section, we report the performance of our association algorithm to link APIs to a given code example. First, we construct an evaluation corpus. Second, we report the performance of Opiner's code example to API association algorithm using the corpus. We use the following four performance measures to assess the performance of the algorithm: precision (P), recall(R), F-measure ($F1$), and Accuracy (A).

$$P = \frac{TP}{TP + FP},\ R = \frac{TP}{TP + FN},\ F1 = 2 * \frac{P * R}{P + R},$$
$$A = \frac{TP + TN}{TP + FP + TN + FN}$$

$TP =$ Nb. of true positives, and $FN =$ Nb. false negatives.

6.2.1 Evaluation Corpus

The evaluation corpus consists randomly selected 730 code examples from our entire dataset. During our concept summary generation process, we discard code examples with less than five lines of code. However, the association process should be independent of the summary requirements, and thus should work well for any code example in general. We consider that this assumption is important to ensure that we can investigate code examples of any length in our summaries in future. Therefore, we assessed the accuracy of our algorithm to associate an API to a code example based on a set 730 code examples that was randomly samples from *all* the 13422 the code examples that we found in our entire dataset.

Out of the 730 code examples in our evaluation corpus, 375 code examples were sampled from the list of 8589 valid code snippets and 355 from the list of 4826 code examples that were labeled as invalid by the *invalid code detection* component of our framework. Recall from Section 6.1 that our algorithm to detect invalid code example was based on the rules originally proposed by Dagenais and Robillard [59]. An invalid code in Opiner is blob of text that does not have any code (e.g., an XML or JSON block). Note that a valid code can still have an XML or JSON string. A valid code can also have typos (e.g., in class or variable names). We observed many well-received (e.g., upvoted) code examples to have typos in Stack Overflow. The size of each subset (i.e., valid and invalid samples) is determined to capture a statistically significant snapshot of our entire dataset of code examples (with a 95% confidence level and 5% confidence interval). Each of the code examples in the evaluation corpus was manually validated by three independent coders. For each code example, the final decision is taken based on the majority votes out of the three coders.

Coding Process. A total of three coders contributed to the manual validation process. Among the three coders, the first coder is Gias Uddin (the author of this thesis) and the second coder was Prof. Foutse Khomh. The third coder is a graduate student at the University of Saskatchewan. The third coder is not a co-author of this paper. The third coder was not involved in any stages of this project. The first coder did all the tagging. The other two coders each assessed approximately 11% of the 730 code examples in the validation dataset (total 80 examples). The manual validation process

Table 6.4: Analysis of agreement among the coders to validate the association of APIs to code examples (Using Recal3 [5])

	Kappa (Pairwise)	Fleiss'	Percent	Krippendorff's α
Overall	96.6	96.5	99.35	96.55
Valid	93.2	93	98.7	93.1
Discarded	100	100	100	100

Table 6.5: Performance of the association of code examples to APIs

Overall	Precision	Recall	F1 Score	Accuracy
Overall + Discarded	88.3	98.2	93	93.2
Associated	88.3	100	93.8	88.3
By Filter				
SamePostBefore	96.8	100	98.7	96.8
SamePostAfter	100	100	100	100
SameThread	100	100	100	100
Global	87.4	100	93.2	87.4
MaxHit	94.7	100	97.3	94.7
Link	100	100	100	100
Name	100	100	100	100
Coverage	85.5	100	92.3	85.5

involved the following steps in order:

Step 1. **Agreement Set.** The first coder randomly selected 80 code examples out of the validation dataset of 730 code examples: 50 from the valid code examples and 30 from the code examples that were tagged as invalid by the algorithm. We refer to this list of 80 code examples as the agreement set, because we used this set to assess the agreement among the three coders on the labels produced produced by coders. The relative proportion of valid and invalid code examples was determined based on the presence of valid and invalid code examples in the entire dataset.

Step 2. **Initiation.** The first coder removed his labels from the agreement set and created a spreadsheet with two tabs, 50 valid examples in one tab named as 'valid' and 30 invalid examples in another tab named as 'invalid'. The first coder then sent the spreadsheet to the other two coders by email. To avoid bias, the coders did not have any personal or on-line communication during their assessment of the agreement set.

Step 3. **Agreement Calculation.** Once all the coders completed their labeling of the validation set, we computed the agreement among the coders using the online Recal3 calculator [5]. We used the following four metrics from the calculator to assess the agreement: a) Average pairwise percent agreement, b) Fleiss' Kappa [165], c) Average pairwise Cohen Kappa [64], and d) Krippendorff's

α (nominal) [166] In Table 6.4, we show the level of agreements among the coders using the four metrics. The overall agreement among the coders was near perfect: pairwise Cohen κ was 96.6%, the Fleiss κ was 96.5%, the percent agreement was 99.35% and Krippendorff's α was 96.5%

Step 4. **Agreement Assessment.** For the invalid code examples, all the coders agreed with each other on each of the assessment of the 30 code examples. For the valid code examples, the third coder disagreed with the other two coders for only one code example:

```
@JsonCreatorResponseWrapper(  @JsonProperty(             ) ResponseType1 response1,  @JsonProperty(
                  ) ResponseType2 response2,  @JsonProperty(             ) ResponseType3 response3,  ...)
```

The algorithm associated this one line code example to the API Jackson, which the third coder ruled as an error because the code was *too short* to make an informed decision. The code example was taken from the Stack Overflow thread [167] which was titled as "Jackson Polymorphic Deserialisation...". Therefore, the code example is indeed related to the Jackson API. We thus took the decisions of the first two coders on this disagreement, i.e., the algorithm was correct to associate the code example to the API Jackson.

Since the agreement level among the coders was near perfect, we considered that if any of the coders does further manual labeling, there is little chance that those labels would introduce any subjective bias in the assessment. The first coder then finished all the labeling of the rest of the code examples (i.e., 650 code examples). The manual validation dataset with the assessment of the coders are shared in our online appendix [168].

6.2.2 Results

In Table 6.5, we show the performance of the association classifier. The overall precision is 88.2%, with a recall of 98.2%, and an accuracy of 93.2%. For the code examples that are considered as valid, our algorithm associates each to at least an API. For such associations, the precision is 88.3%, with a recall of 100%, and an accuracy of 88.3%.

Almost one-third of the misclassified associations happened due to the code example either being written in programming languages other than Java or the code example actually being invalid. For those should be invalid code examples, the off-the-shelf invalid code detection component in our framework erroneously passed those as valid code examples and thus Opiner tried to associate those to an API. For example, the following is a JavaScript code snippet.

```
var jsonData = {...};
$.ajax({type :       ,    dataType :       ,    ...}
```

The invalid code detection component erroneously considered it as valid, beacuse it has multiple lines ending with a semi-colon and it was similar brackets used to enclose type declaration in Java. In our future work, we will improve the detection approach of invalid code examples.

Among the other misclassifications, five occurred due to the code examples being very short. A common problem with a short code example is the lack of sufficient API types in the example to make an informed decision. Such problem exacerbates when an API type name is generic or it is shared among multiple APIs (e.g., JSONArray is a common API type name shared among almost all Java JSON APIs).

Misclassifications also occurred due to the API mention detector not being able to detect all the API mentions in a forum post. For example, the following code example from [169] was erroneously assigned to the com.google.code.gson API. However, the correct association would be the com.google.gwt API. The forum post (answer id 20374750) contained the mentions of both the Google GSON and the com.google.gwt API. However, com.google.gwt was mentioned using an acronym GWT and the API mention detector missed it.

```
String serializeToJson(Test test) {
AutoBean<Test> bean = AutoBeanUtils.getAutoBean(test);
return AutoBeanCodex.encode(bean).getPayload();
}
Test deserializeFromJson(String json) {
AutoBean<Test> bean = AutoBeanCodex.decode(myFactory, Test.class, json);
return bean.as();
}
```

Opiner showed near-perfect accuracy when an API is mentioned in the same post or thread where a code example was found and the association context generation process could successfully include the API mention to the association context of the code example. Opiner could have achieved a better performance, had we included all the APIs in our database into the association contexts of each code example. This approach, though, could be prone to scalability issues, due mainly to the querying of the code types against the entire database.

In our dataset, 88% of the association occurred based on the 'Global' filter, followed by the 'same post before' (9.3%), 'same post after' (1.4%), and 'same thread' (1.1%) filters. The Global filter was based on the coverage rule which assigns a code example to the API with the maximum type match based on the other filters. This filter was the most useful because many API names were not mentioned in the textual contents at all and developers assumed that other developers would automatically infer that information. For example, the API org.json is the default pre-packaged API in the Android SDKs. Therefore, unless explicitly mentioned otherwise, developers just assumed that the Json parsing example for Android would refer to the org.json API. In Table 6.5, we show the performance of the association resolver that is based on the proximity filters. While the 'Global' filter has the most coverage, it also has the lowest precision among all. This mainly happened for code examples with one or two lines and with generic type names. For example, 'JsonObject' is a common type name among the APIs offering Json Parsing. If a code example only contained a reference to this type and nothing else, our coverage rule would associate the code example to the org.json API. This was wrong for the following code example [158] which was referring to the com.google.gwt API:

```
json_string =
JSONObject json_data  = JSONParser.parseLenient(json_string);
JSONObject data = json_data.get(      ).isObject();
```

Our algorithm achieved similar performance with regards to the other state of the art code example resolver techniques, notably Baker [35] (precision 97%, recall 83%) and Recodoc [59] (precision 96%, recall 96%). A direct comparison with Recodoc was not possible because Recodoc uses a brute-force technique by comparing a code example against all the APIs in the database. This means that for every code type found in a code example, we would need to search for it in every single API in the database. Such querying was clearly not scalable for our current infrastructure, which systematically queries the online Maven central database by using a list of only the APIs that are mentioned within the association context of a code example. The tool Baker and its associated dataset were not available. With our current infrastructure, the reproduction of Baker or enhance-

ment of Recodoc to improve the performance is not possible due to the following reasons: (1) Both Baker and Recodoc require syntactically correct code examples. As we explain in Section 6.1, many code examples in our dataset do not have fully correct syntax, yet they are well-received as useful in the forum posts. For our usage summaries, we did not want to lose such potentially valuable usage scenarios. (2) Baker and Recodoc rely on the matching of both API types and methods. Opiner only relies on API types due mainly to our observation that API types are more reliable indicator to trace an API (method names can be very generic, e.g., get, set, etc.). Similar findings were also observed by Baker and Recodoc. (3) The current infrastructure of Opiner relies on the online code repository of Maven central to match API types during the association process. The Maven central engine only allows queries by API types and not by API methods.

Opiner can associate a valid code example to an API with 88.3% precision and 98.2% recall. The misclassifications are due mainly to the greedy approach in Opiner to associate any code example to an API, including one line code example to often syntactically incorrect snippets which are upvoted in the forum posts. Thus, Opiner achieves higher recall than Baker and Recodoc, but the precision drops due to this greedy approach.

6.3 Effectiveness of Opiner (RQ2)

Because the goal of the automatic summarization of APIs usage scenarios in Opiner is to assist developers finding the right solutions to their coding tasks relatively easily than other resources, we conducted a user study involving programmers. We assessed the usefulness of Opiner to assist programmers in their coding tasks. In this section, we describe the design of the study in Section 6.3.1), the study participants in Section 6.3.2, and metrics used in the data analysis in Section 6.3.3. We then present the results in Sections 6.3.4 and 6.3.5.

6.3.1 Study Design

The *goal* of this study was to analyze the *effectiveness* of Opiner to assist in coding tasks. The *objects* were the APIs and their usage summaries in Opiner, Stack Overflow, and Official documentation. The *subjects* were the participants who each completed four coding tasks. Our study consisted of two parts:

P1 - Effectiveness Analysis
Assessment of Opiner's effectiveness to support coding tasks.

P2 - Industrial Adoption
Opportunities of Industrial adoption of Opiner.

We describe the designs of the two parts in our study below.

6.3.1.1 Effectiveness Analysis (P1)

Analysis Dimensions. Because developers often refer to API documentation (formal and informal) to assist them in completing tasks correctly and quickly [34], we compared the usefulness of Opiner over the other development resources (Stack Overflow, API formal documentation, and everything). We assessed the effectiveness of each resource to assist in the coding tasks along three dimensions:

Correctness ***of the provided coding solution***

We considered a solution as correct if it adhered to the provided specification and showed functional correctness [170]. Because each coding solution in our study for a given task would require the usage of a given API, we define correctness based on two assessments: whether the correct API was used, and whether the required API elements were used.

Time ***taken to complete a coding task***

We defined the time spent to complete a coding task using a given development resource as the total time took for a participant (1) to read the task description, (2) to consult the given development resolution for a solution and (3) to write the solution in the provided text box of the solution.

Effort ***spent to to complete a coding task***

We defined the effort spent to complete a coding task using a given development resource as how hard the participant had to work to complete the task. We analyzed the effort using the NASA TLX index [171] which uses six measures to determine the effort spent to complete a task. TLX computes the workload of a given task by averaging the ratings provided for six dimensions: mental demands, physical demands, temporal demands, own performance, effort, and frustration.

Design of the study. Our design was a between-subject design [172], with four different groups of participants, each participant completing four tasks in four different settings. Our between-subject design allowed us to avoid repetition by using a different group of subjects for each treatment. To avoid potential bias in the coding tasks (see Section 6.6), we enforced the homogeneity of the groups by ensuring that: 1) no group entirely contained participants that were only professional developers or only students, 2) no group entirely contained participants from a specific geographic location and–or academic institution, 3) each participant completed the tasks assigned to him independently and without consulting with others 4) each group had same number of four coding tasks 5) each group had exposure to all four development settings as part of the four development tasks. The use of balanced groups simplified and enhanced the statistical analysis of the collected data.

Coding Tasks. The four tasks are described inTable 6.6. Each task was required to be completed using a pre-selected API. Thus for the four tasks, each participant needed to use four different APIs: Jackson [13], Gson [14], Xstream [173], and Spring framework [174]. Jackson and Gson are two of the most popular Java APIs for JSON parsing [175]. Spring is one of the most popular web framework in Java [176] and XStream is well-regarded for its efficient adherence to the JAXB principles [177], i.e., XML to JSON conversion and vice versa. For each task, we prepared four settings:

Settings Used to Evaluate the Coding Tasks

SO - Stack Overflow.
Complete the task only with the help of Stack Overflow.

DO - Official Documentation.
Complete the task only with the help of API official documentation.

OP - Opiner.
Complete the task only with the help of Opiner.

EV - Everything.
Complete the task with any resources available (i.e., SO, DO and Opiner).

The participants were divided into four groups (G1, G2, G3, G4). Each participant in a group was asked to complete the four tasks. Each participant in a group completed the tasks in the order and settings shown in Table 6.7. To ensure that the participants used the correct development resource for a given API in a given development setting, we added the links to those resources for the API in the task description. For example, for the task TJ and the setting SO, we provided the following link to query Stack Overflow using Jackson as a tag: `https://stackoverflow.com/questions/tagged/jackson`. For the task TG and the setting DO, we provided the following link to the official documentation page of the Google GSON API: `https://github.com/google/gson/blob/master/UserGuide.md`. For the task TX and the setting PO, we provided a link to the summaries of the API XStream in the Opiner website `http://sentimin.soccerlab.polymtl.ca:38080/opinereval/code/get/xstream/`. For the task TS with the setting EV, we provided three links, i.e., one from Opiner (as above), one from Stack Overflow (as above), an one from API formal documentation (as above).

Task Selection Rationale. The cues of the four coding tasks were taken from Stack Overflow. The solution to each task spanned over two posts. The two posts are from two different threads in Stack Overflow. Thus the developers could search in Stack Overflow to find the solutions. However, that would require them searching posts from multiple threads in Stack Overflow. All of those tasks are common using the four APIs. Each post related to the tasks was viewed and upvoted more than hundred times in Stack Overflow. To ensure that each development resource was treated with *equal fairness* during the completion of the development tasks, we also made sure that each task could be completed using any of the development resources, i.e., the solution to each task could be found in any of the resources at a given time, without the need to rely on the other resources.

Coding Guide. A seven-page coding guide was produced to explain the study requirements (e.g., the guide for Group G1: `https://goo.gl/ccJMeY`). Before each participant was invited to complete the study, he had to read the entire coding guide. Each participant was encouraged to ask questions to clarify the study details before and during the study. To respond to the questions the participants communicated with the first author over Skype. Each participant was already familiar with formal and informal documentation resources. To ensure a fair comparison of the different resources used to complete the tasks, each participant was given a brief demo of Opiner before the beginning of the study. This was done by giving them an access to the Opiner web site.

Data Collection Process. The study was performed in a Google Form, where participation was by invitation only. Four versions of the form were generated, each corresponding to one group. Each

Table 6.6: Overview of coding tasks

Task	API	Description
TJ	Jackson	Write a method that takes as input a Java Object and deserializes it to Json, using the Jackson annotation features that can handle custom names in Java object during deserialization.
TG	GSON	Write a method that takes as input a JSON string and converts it into a Java Object. The conversion should be flexible enough to handle unknown types in the JSON fields using generics.
TX	Xstream	Write a method that takes as input a XML string and converts it into a JSON object. The solution should support aliasing of the different fields in the XML string to ensure readability
TS	Spring	Write a method that takes as input a JSON response and converts it into a Java object. The response should adhere to strict JSON character encoding (e.g., UTF-8).

Table 6.7: Distribution of coding tasks per group per setting

↓ Group | Setting →	StackOverflow (SO)	Javadoc (DO)	Opiner (OP)	Everything (EV)
Group G1	TJ (Jackson)	TG (Gson)	TX (XStream)	TS (Spring)
Group G2	TS (Spring)	TJ (Jackson)	TG (Gson)	TX (XStream)
Group G3	TX (XStream)	TS (Spring)	TJ (Jackson)	TG (Gson)
Group G4	TG (Gson)	TX (XStream)	TS (Spring)	TJ (Jackson)

group was given access to one version of the form representing the group. An offline copy of each form is provided in our online appendix [168]. The developers were asked to write the solution to each coding task in a text box reserved for the task. The developers were encouraged to use any IDE that they are familiar with, to code the solution to the tasks. Before starting the study, each participant was asked to put his email address and to report his expertise based on the following three questions:

Profession the current profession of the participant

Experience software development experience in years

Activity whether or not the participant was actively involved in software development during the study

Before starting each task, a participant was asked to mark down the beginning time. After completing a solution the participant was again asked to mark the time of completion of the task. The participant was encouraged to not take any break during the completion of the task (after he marked the starting time of the task). To avoid fatigue, each participant was encouraged to take a short break between two tasks. Besides completing each coding task, each participant was also asked to assess the complexity and effort required for each task, using the NASA Task Load Index (TLX) [171], which assesses the subjective workload of subjects. After completing each task, we asked each subject to provide their self-reported effort on the completed task through the official NASA TLX log engine at `nasatlx.com`. Each subject was given a login ID, an experiment ID and task IDs, which they used to log their effort estimation for each task, under the different settings.

Variables. The main independent variable was the development resource that participants use to find solutions for their coding tasks. The other independent variables were participants' development experience and profession. Dependent variables were the values of the different metrics we analyzed across the three analysis dimensions (correctness, time, and effort).

Hypotheses. We introduce the following null and alternate hypotheses to evaluate the performance of the participants along three analysis dimensions (correctness, time, and effort) while completing coding tasks in the four different settings.

H_0 The primary null hypothesis is that there was no difference across the three analysis dimensions among the participants while using Opiner, Stack Overflow, API formal documents, and any development resources to complete the coding tasks.

H_1 An alternative hypothesis to H_0 is that there was statistically significant different among the participants across the three analysis dimensions while using Opiner, Stack Overflow, API formal documents, and any development resources to complete the coding tasks.

6.3.1.2 Industrial Adoption (P2)

After a participants completed the tasks, we asked the following three questions to capture his experience and opinion about whether and how Opiner could be adopted into his daily development tasks:

Q1. Usage Would you use Opiner in your development tasks?

Q2. Usability How usable is Opiner?

Q3. Improvement How would you like Opiner to be improved?

The first question (Usage) had two parts. The first part contained the above usage question with three options (yes, no, maybe). The second part required respondent to justify his selection from above three options. His answer was recorded in a text box. The participants were asked to write as much as they could for the last two questions (Usability and Improvement) in two text boxes. The three questions were adopted from our previous study assessing the potential industrial adoption of API review summaries (Chapter 5).

6.3.2 Participants

The coding tasks were completed by a total 34 participants. Each of the three groups (G1-G3) had eight participants. The group G4 had nine participants. Among the 34 participants, 18 were recruited through the online professional social network site, Freelancer.com. The rest of the participants (16) were recruited from four universities, two in Canada (University of Saskatchewan and Polytechnique Montreal) and two in Bangladesh (Bangladesh University of Engineering & Technology and Khulna University). Sites like Amazon Mechanical turks and Freelancer.com have been gaining popularity to conduct studies in empirical software engineering research due to the availability of efficient, knowledgeable and experienced software engineers. In our study, we only recruited a freelancer if he had professional software development experience in Java. Each freelancer was remunerated with $20, which was a modest sum given the volume of the work.

Among the 16 participants recruited from the universities, eight reported their profession as students, two as graduate researchers, and six as software engineers and student. Among the 18 freelancers, one was a business data analyst, four were team leads, and the rest were software developers. Among the 34 participants 88.2% were actively involved in software development (94.4% among the freelancers and 81.3% among the university participants). Each participant had a background in computer science and software engineering.

The number of years of experience of the participants in software development ranged between less than one year to more than 10 years: three (all students) with less than one year of experience, nine between one and two, 12 between three and six, four between seven and 10 and the rest (nine) had more than 10 years of experience. Among the four participants that were not actively involved in daily development activities, one was a business analyst (a freelancer) and three were students (university participants). The business data analyst had between three and six years of development experience in Java. The diversity in the participant occupation offered us insights into whether and how Opiner was useful to all participants in general.

6.3.3 Study Data Analysis

We computed the dependent variables (i.e., Correctness, Time, and Effort) of our study as follows:

1) **Correctness.** To check the correctness of the coding solution for a coding task, we used the following process: a) We identified the correct API elements (types, methods) used for the coding task. b) We matched how many of those API elements were found in the coding solution and in what order. c) We quantified the correctness of the coding solution using the following equation:

$$\text{Correctness} = \frac{|\text{API Elements Found}|}{|\text{API Elements Expected}|} \tag{6.3}$$

An API element can be either an API type (class, annotation) or an API method. Intuitively, a complete solution should have all the required API elements expected for the solution. We discarded the following types of solutions: a) **Duplicates.** Cases where the solution of one task was copied into the solution of another task. We identified this by seeing the same solution copy pasted for the two tasks. Whenever this happened, we discarded the second solution. b) **Links.** Cases where developers only provided links to an online resource without providing a solution for the task. We discarded such solutions. c) **Wrong API.** Cases where developers provided the solution using an API that was not given to them. We discarded such solutions.

2) **Time.** We computed the time taken to develop solutions for each task, by taking the difference between the start and the end time reported for the task by the participant. Because the time spent was self-reported, it was prone to errors (some participants failed to record their time correctly). To remove erroneous entries, we discarded the following type of reported time:a) reported times that were less than two minutes. It takes time to read the description of a task and to write it down, and it is simply not possible to do all such activities within a minute. b) reported times that were more than 90 minutes for a given task. For example, we discarded one time that was reported as 1,410 minutes, i.e., almost 24 hours. Clearly, a participant cannot be awake for 24 hours to complete one coding task. This happened in only a few cases.

3) **Effort.** We used the TLX metrics values as reported by the participants. We analyzed the following five dimensions in the TLX metrics for each task under each setting:

– ***Frustration Level.*** How annoyed versus complacent the developer felt during the coding of the task?

– ***Mental Demand.*** How much mental and perceptual activity was required?

– ***Temporal Demand.*** How much time pressure did the participant feel during the coding of the solution?

– ***Physical Demand.*** How much physical activity was required.

– ***Overall Performance.*** How satisfied was the participant with his performance?

Each dimension was reported in a 100-points range with 5-point steps. A TLX 'effort' score is automatically computed as a task load index by combining all the ratings provided by a participant.

Because the provided TLX scores were based on the judgment of the participants, they are prone to subjective bias. Detecting outliers and removing those as noise from such ordinal data is a standard statistical process [178]. By following Tukey, we only considered values between the following two ranges as valid: a) Lower limit: First quartile - 1.5 * IQR b) Upper limit: Third quartile + 1.5 * IQR Here IQR stands for 'Inter quartile range', which is calculated as: IQR $= Q3 - Q1$. Q3 stands for the third and Q1 for the first quartile.

For each setting, we accepted or refuted the null hypothesis using the non-parametric Mann Whitney U test and Cliff's delta effect size [179]. The Mann Whitney U test measures whether two independent distributions have equally large values. For each hypothesis, we used a 95% confidence level (i.e., p = 0.05). We used Cliff's delta to measure the differences in size between the two distributions. For a given metric (correctness, time, and effort), the input to the non-parametric test case is the list of values for each setting coming from all the tasks, i.e., a matrix with four columns (OP, DO, SO, EV). Each column contains metric values from all tasks for the setting. The outputs are pair-wise adjusted p values and cliff's delta values for each pair of settings. Therefore, for the four settings (OP, SO, DO, and EV), we have four pairs:

1) OP vs DO: Opiner vs API formal documentation
2) OP vs SO: Opiner vs Stack Overflow
3) OP Vs EV: Opiner vs Everything else
4) DO VS SO: API formal documentation vs Stack Overflow

We refute or accept the null hypothesis based on the p-values from the first three pairs where Opiner is compared, because we compare the effectiveness of Opiner against the other documentation resources. The two other probable pairs (DO vs EV and EV vs SO) are not reported, because the EV setting also contained both of the documentation resources.

Impact Analysis. For a given metric, we further sought to understand the impact of each task on the testing of the hypothesis. For a non-parametric test, the minimum size of sample should be minimum 20 [180]. For a given task and given setting, we have at most eight or nine participants, with each participant contributing to one sample. Therefore, we adopted the following approach to analyze the impact of each task. For each metric, we applied the following steps:1) we picked a task (say TG) and removed the values from the matrix 2) we tested the hypothesis, and name it (except TG). 3) we repeated the above two steps for each task. The acceptance or rejection of a null hypothesis does not depend on the impact analysis.

6.3.4 Results of Opiner's Effectiveness Analysis (P1)

In this section, we present the results of the study. A total of 135 coding solutions were provided by the 34 participants. From there, we discarded 14 as invalid solutions (i.e., link/wrong API). Out of the 135 reported time, we discarded 23 as being spurious. 24 participants completed the TLX metrics (providing 96 entries in total), with each setting having six participants each.

Table 6.8 shows descriptive statistics about the completed coding tasks along the three dimensions. The effort calculation is based on the TLX effort index as calculated by the TLX software

Table 6.8: Summary statistics of the correctness, time and effort spent in the coding tasks

	Correctness				**Time**					**Effort**				
	Avg	**Stdev**	**Median**	**Perfect**	**Avg**	**Stdev**	**Median**	**Min**	**Max**	**Avg**	**Stdev**	**Median**	**Min**	**Max**
SO	0.46	0.45	0.33	33%	22.3	11.5	20	3	44	55.8	22.2	55	5	99
DO	0.5	0.44	0.33	33%	23.7	12.6	21	5	58	63.9	18.9	64	24	99
OP	**0.62**	0.39	**0.73**	37%	**18.6**	12.5	**15**	2	**55**	**45.8**	26.6	**55**	9	98
EV	0.55	0.41	0.6	33%	19.4	14.3	14.5	2	55	54.8	25.5	62.5	4	94

Table 6.9: Results of the Wilcoxon test and Cliff's effect size for the pair-wise comparisons of the usage of development resources

		Overall		Except TG		Except TJ		Except TX		Except TS	
Metric	**Comparison**	$p-value$	δ	$p-value$	δ	$p-value$	δ	$p-value$	δ	$p-value$	δ
Correctness	**OP vs DO**	0.289	-0.293	**0.049**	-0.397	0.983	-0.075	0.183	-0.317	0.485	-0.256
	OP vs SO	0.186	-0.323	**0.009**	-0.488	0.549	-0.171	0.417	-0.238	0.450	-0.282
	OP vs EV	0.542	-0.240	0.229	-0.287	0.834	-0.047	0.844	-0.150	0.450	-0.264
	DO vs SO	0.721	-0.038	0.445	-0.004	0.481	0.112	0.737	-0.055	0.836	0.034
Time	**OP vs DO**	0.109	0.248	0.214	0.217	**0.037**	0.369	0.271	0.200	0.278	0.203
	OP vs SO	0.195	0.210	0.338	0.188	0.169	0.252	0.165	0.256	0.527	0.123
	OP vs EV	0.975	0.006	0.579	-0.099	0.343	0.167	0.869	0.031	0.764	-0.057
	DO vs SO	0.838	-0.202	0.816	-0.310	0.608	-0.175	0.573	-0.227	0.657	-0.110
Effort	**OP vs DO**	**0.050**	-0.330	**0.031**	0.227	**0.030**	0.178	0.048	0.210	0.174	0.155
	OP vs SO	0.297	-0.177	0.113	0.124	0.139	0.060	0.519	-0.028	0.616	-0.005
	OP vs EV	0.288	-0.181	0.516	-0.254	0.067	0.060	0.206	-0.354	1.000	-0.138
	DO vs SO	0.353	0.158	0.488	-0.304	0.635	-0.454	0.206	-0.560	0.447	-0.357

The bold numbers show statistically significant p-values

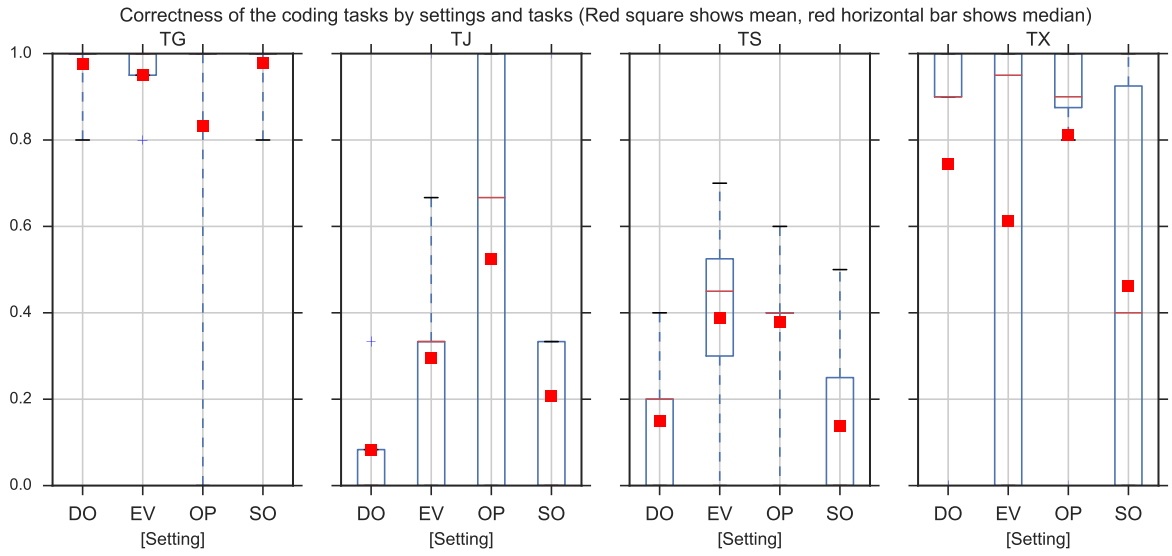

Figure 6.9: Correctness of the completed solutions by tasks across the settings. The red horizontal bar shows the median

based on the reported five dimensions. As we see in Table 6.8, while using Opiner, the participants on average coded with more correctness (62.3%), spent less time (18.6 minutes on average per coding task) and less effort (45.8). The correctness of the provided solutions were the lowest when participants used only Stack Overflow; showing the difficulties that the participants faced to piece together a solution from multiple forum threads. Participants on average spent the highest amount of time and effort per coding solution when using only the formal documentation. Since the formal documentation did have all the solutions, it was a matter of finding and piecing those solutions together. This longer time and effort highlights the need for a better presentation format of the contents of official documentations, as originally identified by Uddin and Robillard [1]. Participants showed better performance when using Opiner only, than when using every available resources (which included Opiner, formal documentation and Stack Overflow) during the coding tasks. This was perhaps due to the fact that the participants were more familiar with formal and informal documentation than Opiner. Thus, they may have started exploring those resources before resorting to Opiner for their coding solution when they were allowed to use all the resources. Also, the participants reported in the questionnaire that they were familiar with Stack Overflow and the official documentations, but that they needed time to learn Opiner. Several participants (including professional freelancers) mentioned that they found the coding tasks to be difficult. This difficulty contributed to the lower number of participants providing completely accurate solution.

In Table 6.9, we show the results of the non-parametric tests. The differences in efforts spent between Opiner and API official documentation were statistically significant. For the other settings and metrics, the differences are not statistically significant. We thus reject the null hypothesis between Opiner and API official documentation for efforts spent, and cannot reject it for others. Therefore, while Opiner improved overall the productivity of the participants (along the three metrics), there is a room for future improvement. We now discuss the results in more details below.

6.3.4.1 Correctness

In Figure 6.9, we show the correctness across the four settings using four boxplots, one for each coding task. The red squares in each box shows the mean (μ) of accuracy. The red horizontal line for each box shows the median (M) of accuracy. For example, for the task TJ, the mean in accuracy using Opiner was 0.52 while the median was 0.67.

The participants achieved almost perfect accuracy in the coding for the task TG, using both SO and DO. They showed the lowest accuracy ($\mu = 0.83$) using Opiner. For this task, the API was the Google Gson API for which the official documentation is well-regarded by developers [181]. The entire solution to the task can be found in the 'Getting Started' page of the API. The 'Getting Started' official documentation page of Gson follows the format of the single page documentation champion by the Python community. Therefore, finding the right information for this task using the formal documentation was very easy for participants; which explains why participants had the highest accuracy using formal documentation for this task. For the setting 'SO', participants also performed better than when using only Opiner. This is because the participants were able to find the right post in Stack Overflow with keywords like (generic, json conversion). Five participants provided perfect answer using Opiner for this task. Compared to Stack Overflow and official documentation, the search for this solution in Opiner for this task was non-trivial. The participants needed to look at more than one summaries in Opiner. Instead of providing the solution, one participant simply wrote *"no solution found. There is no example code provided by Opiner for GSON"*. We considered that as a wrong answer (i.e., accuracy = 0).

For two of the other three tasks (TJ, and TX), the participants showed the most accuracy in their coding while using Opiner. For these tasks, the search for the solution in Stack Overflow and official documentation was not trivial as the task TG. Instead, the search through the summaries in Opiner was more useful. For example, for the task TJ, the solution could be found under one concept that grouped four relevant usage scenarios. For the task TS, while the participants showed the maximum accuracy ($\mu = 0.39, M = 0.45$), Opiner was the second best with a slightly lower mean accuracy of 0.38 and a median of 0.4. The participants showed the lowest accuracy for this task among the four tasks. This task involved the usage of the Spring framework in Java. The usage of the API for this task required the participants to find the right annotation types in Spring that can be used to inject customized values into JSON inputs in a Java class. Two of the participants using Opiner for TS were students, both had the lowest accuracy among all the participants using Opiner for the task. One of the freelancers mentioned that he found this task to be quite hard (he had the lowest accuracy among all the freelancers for this task using Opiner). While Spring framework is widely used in the Industry, it is a big framework and thus learning how to use it takes time and effort (e.g., 'why is learning spring framework so hard' [182]). Thus, when discoverability of a solution is easier in Opiner compared to the other two resources, developers showed more accuracy using Opiner. Because the concepts in Opiner are produced automatically, these finding offer promises towards the generation of a better API documentation through the API usage summaries.

> Opiner usage summaries can help developers code with more accuracy than both formal and informal API documentation. Opiner usage summaries with relevant code examples can assist developers finding the right solution.

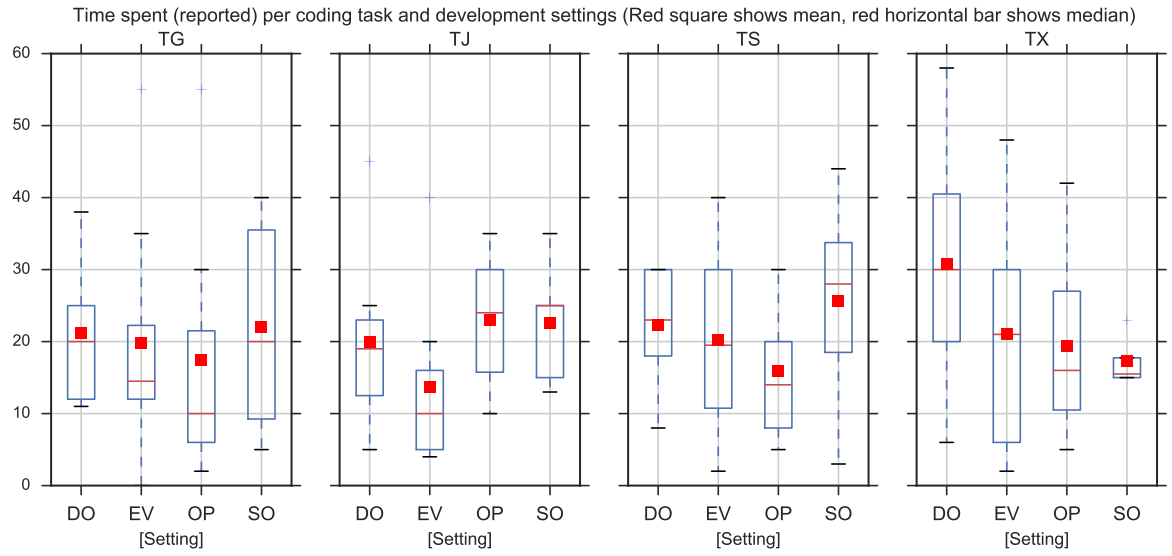

Figure 6.10: Time spent by the developers to complete the solutions for the tasks. The red horizontal bar denotes the median.

6.3.4.2 Time

In Figure 6.10, we show the distribution of the time taken to complete the four tasks across the four settings. The participants spent less time while using Opiner, than the other two resources for the two tasks (TG, TS). For the task TX, participants spent on average 19.4 minutes using Opiner compared to 17.3 minutes while using Stack Overflow (the lowest completion time). Out of all tasks, the participants spent the longest time (on average) while completing the task TJ across all the settings. However, as we observed from Figure 6.9, participants achieved the best accuracy on tasks TJ and TX when using Opiner. The task TJ involved the API Jackson. This task required participants to use the annotation features in Jackson to deserialize a Java object to JSON. Understanding the annotation feature in Jackson can be a non-trivial task and it may not appeal to everyone (ref. the blog post 'Jackson without annotation' [183]). Therefore, it may take more time for a developer to be familiar with this feature. For the task TG, even though participants spent the lowest time while using Opiner. However, that number is low due to the one participant who complained that he could not find a solution using Opiner. He only spent four minutes on this task (that involved reading the task description, opening Opiner, and searching for the solution in Opiner). While using the formal documentation, participants spent the most amount of time for the task TX, but they also showed the second best accuracy among all the four tasks while using formal documentation. Therefore, it may be that participants who coded with more accuracy were more cautious to explore the resources. Thus it took them a longer time. Nevertheless, it is encouraging that participants spent the lowest time while using Opiner for the two tasks (TS and TX) and still achieved the highest accuracy among the four settings.

> Using Opiner, developers can spend less time, while still achieving higher accuracy than when using the formal documentation or Stack Overflow. Hence, Opiner can offer a considerable boost on productivity.

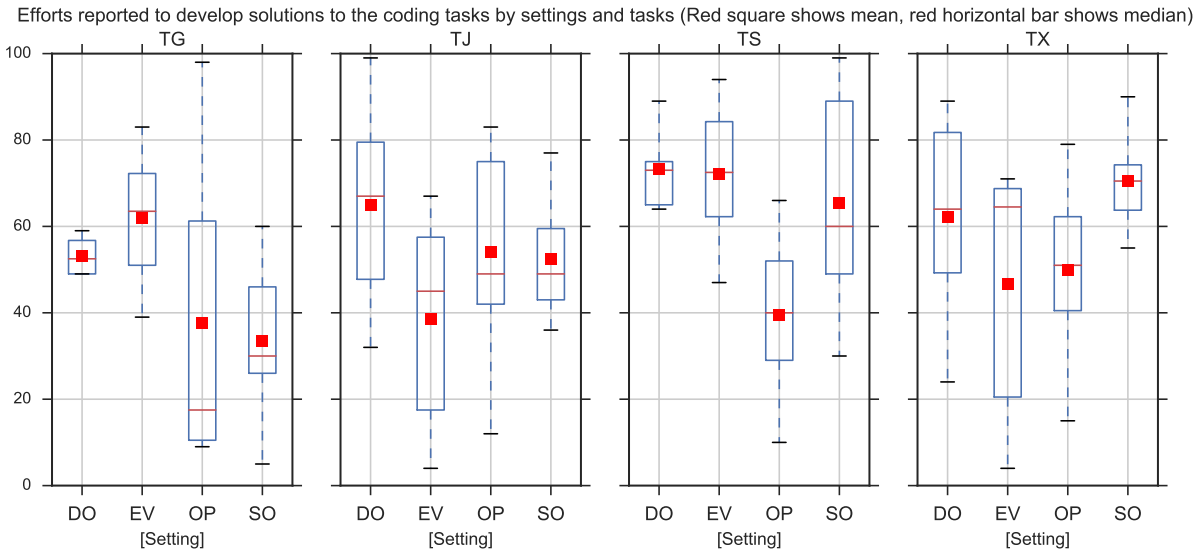

Figure 6.11: Effort required to complete the tasks. The red horizontal bar denotes the median

6.3.4.3 Effort

In Figure 6.11, we show the distributions of the efforts (reported) spent by the participants to complete the coding tasks. The participants spent the least amount of effort while using Opiner for one task (TS). For two tasks (TG, TX), Opiner was the second best. For the other task (TJ), developers spent more effort when using the official documentation in comparison to when they used Opiner only. The overall effort across all the tasks was the lowest for Opiner, across the four settings (see Table 6.8). Task TJ required using the Jackson API. The official documentation of Jackson is considered to be complex [184]. The summaries in Opiner did lessen the effort by more than 10% on average. For TJ, participants spent almost similar effort while using Opiner and Stack Overflow. However, their developed code had more accuracy while using Opiner. For the task TS, developers need to use the Spring Framework. Opiner considerably outperformed the other settings for this task (less than 40% for Opiner to more than 70% for formal documentation).

In Figures 6.12, 6.13, we show the five TLX dimensions as reported by participants for each coding tasks under the four settings. Participants expressed the lowest frustration while using Opiner for two tasks (TS, TX). Participants, however, felt more satisfaction about their coding solution while using Opiner for three task (TJ, TS, TX). For the task TG, developers showed the lowest frustration and best satisfaction while using the formal documentation. The solution to this task was easily found in the 'Getting Started' page of the Gson formal documentation. Recall that, participants had the best accuracy using the formal documentation for this task. The participants, however, felt more time pressure while completing the task using Opiner (Figure 6.13). This is perhaps due to their relative less familiarity with the Opiner interface and their expectation of finding a quicker solution while using Opiner.

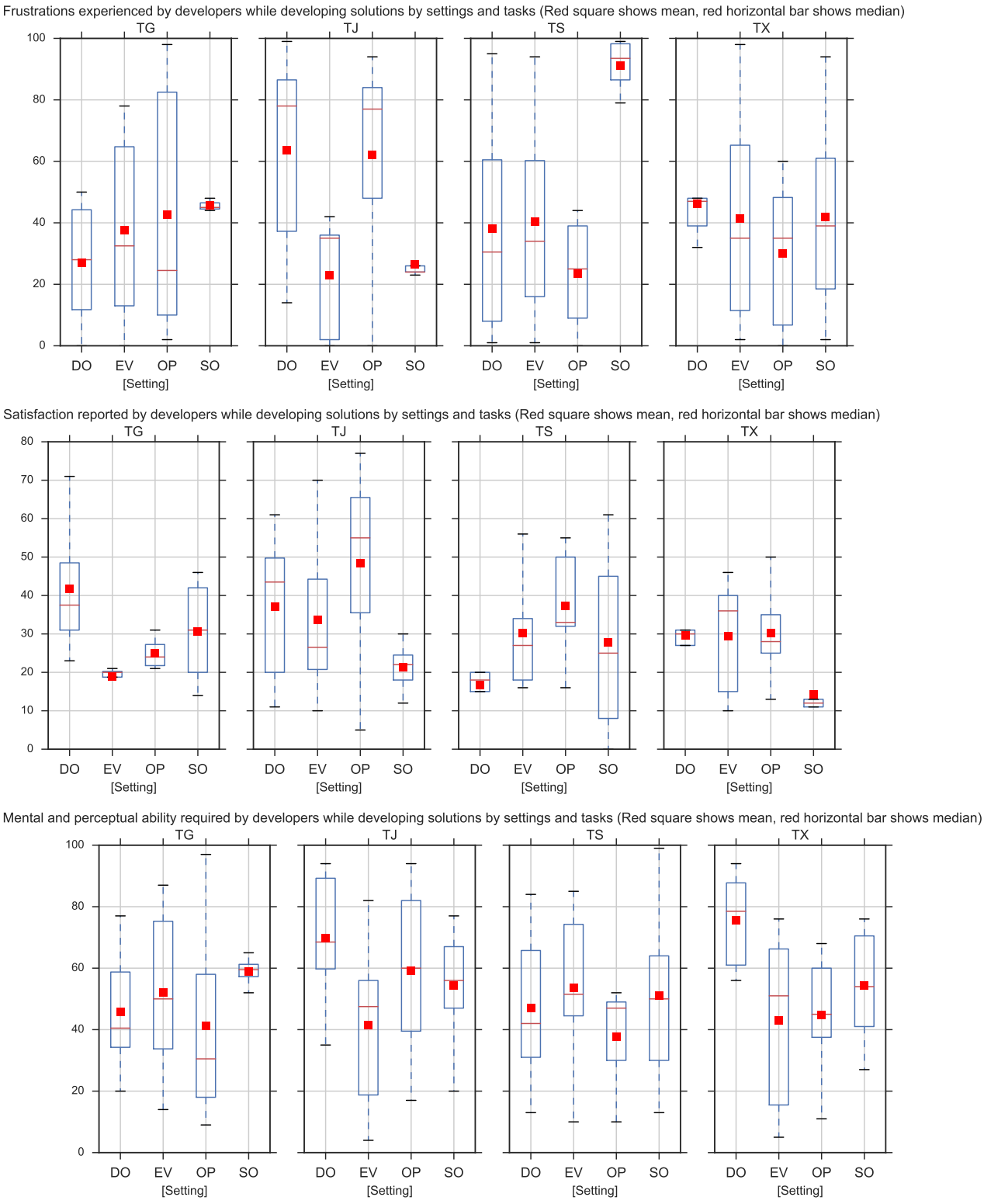

Figure 6.12: Frustration, mental effort, and satisfaction reported developers while completing the tasks. The red horizontal bar denotes the median

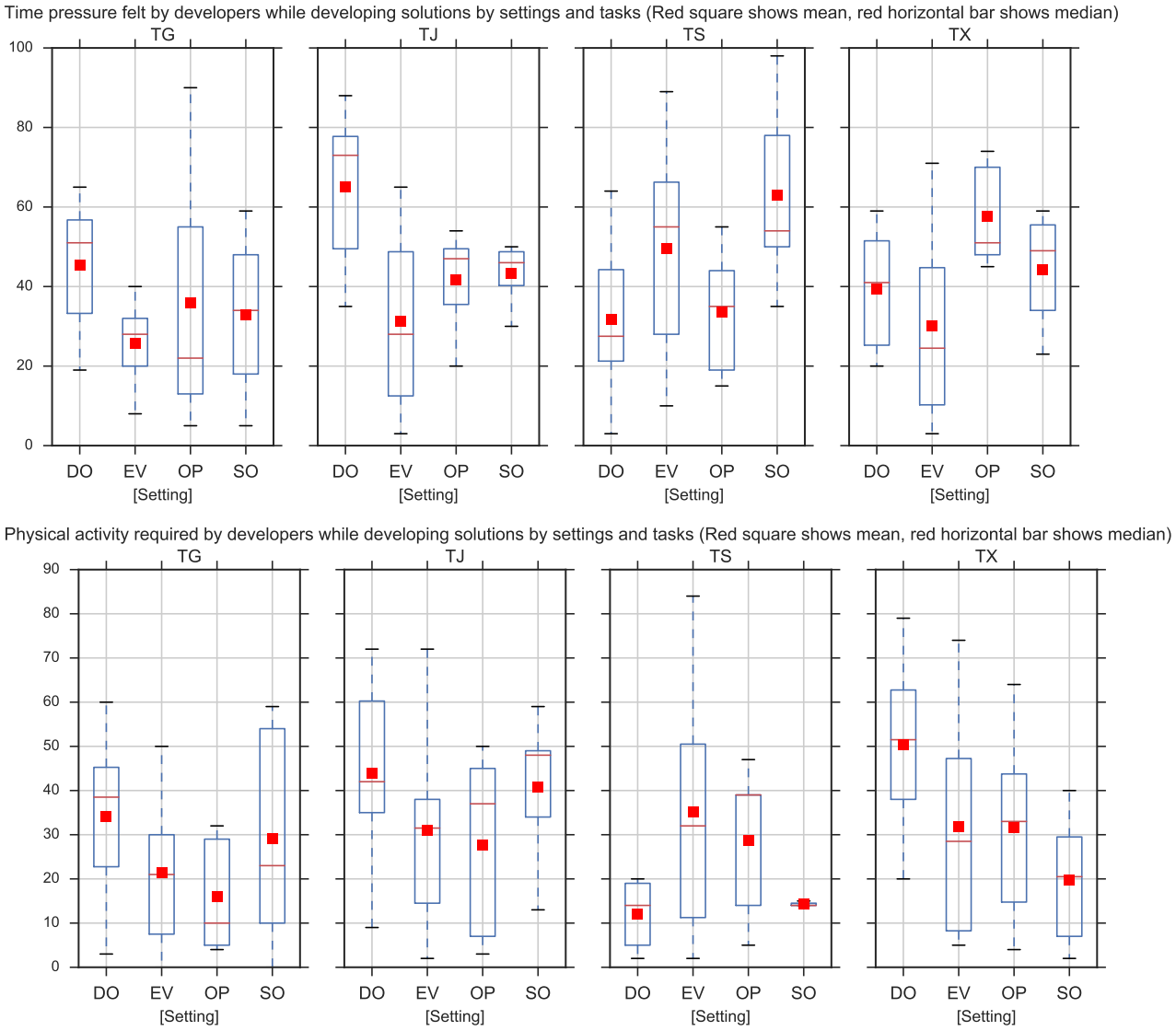

Figure 6.13: Time pressure felt and physical effort spent by developers to complete the tasks. The red horizontal bar denotes the median

The overall effort spent was the lowest while using Opiner across all the development resources. Developers reported more satisfaction when working with Opiner. The difference in the reported effort was statistically significant between Opiner and formal API documentation.

6.3.5 Comments on Industrial Adoption of Opiner (P2)

Out of the 34 respondents that completed the coding tasks, 31 (91.2%) responded to our three questions about industrial adoption. We summarize major themes from their responses below.

6.3.5.1 Would you use Opiner in your daily development tasks?

Out of the 31 respondents, 87.1% showed interest in using Opiner for their daily development tasks (45.2% with a 'yes' and 41.9% with a 'maybe').

14 participants (45.2%) decided that they would use Opiner in their daily development tasks because of analytics based on sentiment analysis and machine learning, usability, and richness of contents. According to one participant *"Opiner is a splendid tool to use. It provides great features and functionalities to solve different tasks so that i will use it"*. We present the major themes with selected quotes from the participants below.

- **Analytics**

  Machine Learning. *"Given there are improvements made in the AI used by Opiner, there is a definitely a boost in the levels of productivity while using Opiner to solve a problem."*

  Usage Statistics. *"coding snippet and statistics regarding API are useful and helpful for developer"*

  Sentiment Analysis. *"In depth knowledge plus the filtered result from Opiner can easily increase the productivity of daily development tasks, as it recommends most recently asked questions from Stack Overflow with the quick glimpse of the positive and negative feedback."*

- ***Usability***

  Simplicity. *"Opiner summarizes all what I want in Stack Overflow in simple way"*

  Ease of Use. *"I will use Opiner for my daily tasks because I can easily get necessary code examples from Opiner."*

- ***Search***

  One stop. *"I will use Opiner for my daily tasks because I can easily get necessary code examples from Opiner."*

  Issue search. *"I develop and run into issues daily, so will search on Opiner to find out details."*

  Awareness. *"During my interaction with the tool, I experienced its usefulness. It not only answers the questions but also helps to keep track of API behavior."*

Time. *"Reduction in time needed for research is a big advantage."*

Content. *"The data available is presented in a good format, we can find more information related to the content we want"*, *"It's helpful tool that I can use to get overview on APIs."*

Four participants (12.9%) considered Opiner as redundant or insignificant to the already available resources (i.e., Stack Overflow, API official documentation). They were reluctant to use Opiner. Out of those four participants, three were graduate students and one was a project manager.

Insignificant. *"Though Opiner has nicer interface and more features to identify API better than other similar systems [Stack Overflow, Google], but those systems have much larger user base."*

Redundant. *"I already have everything available from google search. . . "*

13 Participants (41.9%) were not sure about using Opiner on a daily basis, because for some (1) it was new to them and they wanted to explore it more before making a decision, the others (2) would like to use it but not on a daily basis

– **Wanted to explore Opiner before making a decision**

Integration with Google. *"Most of my responses are found through google search from stack overflow. Unless I can get similar results from Opiner without going into their site. Won't be able to use it on a daily basis."*

Developer Expertise. *"For new APIs, I will first go through the official documentation first to be used to it and gain its insights. For the APIs which I am in intermediate level or for the problems or questions or limitations or optimizations issues that are hindering my development task, I would surely check Opiner first. "*

Lack of Familiarity. *"I am a new user of Opiner. So, at this moment I am not so sure about using it for my daily development tasks."*,

"Already accustomed to Stack Overflow and other sites I get information from using google search."

– **Wanted to use Opiner, but not on a daily basis**

Occasionally. *"If i get some problem in API usage during my development process, then at least once I will use Opiner"*.

In Future *"May be in future I will use Opiner for finding solutions and related issues of an API"*. (by one student participant)

More than 88% (45.2% with a 'yes') of the participants responded that they may consider using Opiner in their daily development activities. The participants liked the fact that the tool combined machine learning techniques and sentiment analysis to cluster code examples.

6.3.5.2 How usable is Opiner?

31 out of the 34 participants (93.5%) considered Opiner as usable for being a single platform to provide insights about API usage and being focused towards a targeted audience

Big data. *"it provides huge data in a presentable format"*

Targeted audience. *"It is usable for specific scenarios and if one knows the apis very well."*

Single platform *"Opiner is useful in the sense it is providing information about API's at a single platform. One don't have to search over the internet and compare by themselves. Opiner does that task"*

Search. *"Helps in finding answers easily and in presentable format,more options"*

Diversity. *"Opiner is usable from different perspectives: daily development, viewing monthly usage etc."*

Adaptation. *"Difficult in the beginning but fast when you know how to use."*

Expertise. *"Yes, it is. It's usability depends on the stage of API and experience or progress of developer/development."*

Statistics. *"I think the best part of Opiner are the statistics."*

Among the two who did not consider it usable, one was not entirely negative, rather he was expecting Opiner to be more usable than Stack Overflow, *"Its almost as good as stack over flow for problem solutions."*

> More than 93% of the participants considered Opiner as usable. They liked the visual representation based on statistics and the appeal of the summaries to cater to diverse development needs in a single platform.

6.3.5.3 How would you improve Opiner?

Developers mentioned the following aspects of Opiner for future improvement:

User guidelines. *"Improve the usability a bit. Providing a help guide and using more generic terms for the different tabs would help the novice set of users that are currently new to the world of programming."*

Performance. *"Also there is a lot of graphic content, that slows down the loading of the webpage. Opiner should be optimized for speed and productivity first and later for how it looks."*

Consolidated Views *"By opening the positive and negative reactions and the topics while clicking the charts. Integrating the documentation side by side."*

Integration. *"Integrating the documentation side by side."*

Multiple Forums. *"People are looking many web sites available in net to get the details what they are looking for. But Opiner completely depends on Stack Overflow. You must look for data in other sites as well."*

Content. *"There are some specific problems (very less) to which the solution is not provided. "*

Better Description. *"subjects/summary names can be renamed and make more relevant to question type summary can be specific to topic of API."*

Search filters. *"Finding the answers for specific questions through search criteria... "*

Collaborative Platform. *"by adding some explanation over the one of Stack Overflow, and add comment space."*

Some developers simply wanted Opiner to be a production ready software, e.g., *" so, it would be my habit issue but if Opiner could be presented to me as a very furnished product (even incomplete), I will surely consider it. E.g. I know DataBricks is not complete yet for cloud distributed tasks, but I like it at least for learning purpose cause they are offering me a neat and clean platform."*

> The most pressing suggestion was to include usage summaries from multiple development forums, as opposed to only Stack Overflow. Such requirement was also expressed by the participants of our previous study on API review summaries [48], which highlights the need for a consolidated viewpoint, to get quick and actionable insights from the large, disconnected and heterogeneous developer forums online.

6.4 Informativeness of the Four Summarization Techniques in Opiner (RQ3)

Because participants in the coding study effectively consulted the four types of usage summaries in Opiner to complete their coding tasks and expressed positive views towards the Industrial adoption of Opiner, it would be useful to know whether and how the different summaries in Opiner are useful in diverse development scenarios besides assisting developers in their coding tasks. Hence, we performed another study in a manner similar to previous evaluation efforts on software artifact summarization [40, 48, 143].

6.4.1 Study Design

Our *goal* was to judge the *usefulness* of a given summary. The *objects* were the different summaries produced for a given API and the *subjects* were the participants who rated each summary. The *contexts* were five development tasks. The five tasks were designed based on a similar study previously conducted in Chapter 5.

Each task was described using a hypothetical development scenario where the participant was asked to judge the summaries through the lens of a software engineering professional. Persona based usability studies have been proven effective both in Academia and Industry [144]. We briefly describe the tasks below.

T1. Selection. (Can the usage summaries help you to select this API?) The persona was a 'Software Architect' who was tasked with making decision on the selection of an API given the usage summaries produced for the API in Opiner. The participants were asked to consider the following decision criteria in their answers: the summary (C1) contained all the *right* information, (C2) was *relevant* for selection, and (C3) was *usable*.

T2. Documentation. (Can the summaries help to create documentation for the API?) The persona was a 'Technical API Documentation Writer' who was tasked with the writing of the documentation of an API by taking into accounts the usage summaries of the API in Opiner. The decision criteria on whether and how the different summaries in Opiner could be useful for such a task were: (C1) the *completeness* of the information, and (C2) the *readability* of the summaries.

T3. Presentation. (Can the summaries help you to justify your selection of the API?) The persona was a development team lead who was tasked with the creation of a presentation by using the usage summaries in Opiner to justify the selection of an API that his team has recently started using. The decision criteria for justification in the presentation were: (C1) the *conciseness* of the information and (C2) *recency* of the provided scenarios. .

T4. Awareness. (Can the summaries help you to be aware of the changes in the API?) The persona was a software developer who was interested in staying aware of his favorite API. To stay aware, he was offered to leverage the usage summaries in Opiner to quickly gain insights. His decision criteria to leverage Opiner usage summaries were: (C1) the *diversity* of provided scenarios and (C2) the *recency* of the provided scenarios.

T5. Authoring. (Can the summaries help you to author an API to improve its features?) the persona was an API author who was tasked with the creation of a new API by gaining insights into the strengths and weakness of API by analyzing the summaries of the API in Opiner. The decision criteria were: (C1) the *Strength and Weakness highlighted* in Opiner and (C2) the presence of *diverse scenarios*.

We assessed the ratings of the three tasks (Selection, Documentation, Presentation) using a 3-point Likert scale (the summary does not miss any info, misses some info, misses all the info). For the task (Authoring), we used a 4-point scale (Fully helpful, partially helpful, Partially Unhelpful, Fully unhelpful). For the task (Awareness), we asked participants how frequently they would like to use the summary (never, once a year, every month, every week, every day). Each of the decision criteria under a task was ranked using a five-point Likert scale (Completely Agree – Completely Disagree). For the task, Authoring, we further asked the participants a question on whether he had decided to author a new API as a competitor to the API whose summaries he had analyzed in Opiner. The options were: Yes, No, Maybe. The API he was asked to investigate in Opiner was the Jackson API, one of the most popular Java APIs for JSON parsing. The survey was conducted in an online Google form.

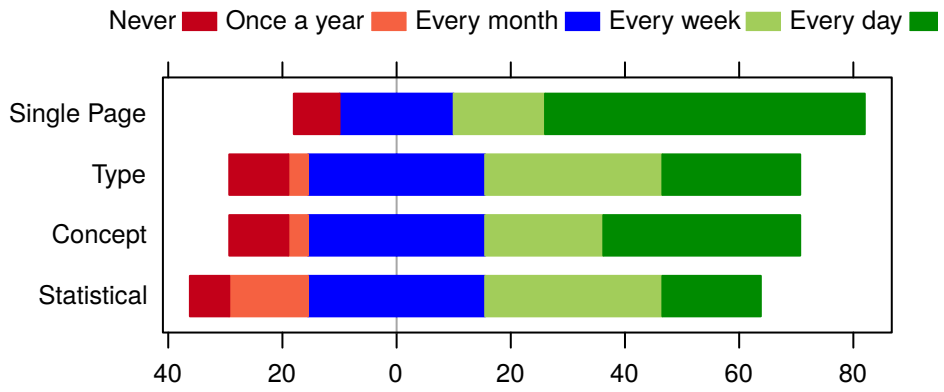

Figure 6.14: Developers' preference of usage of summaries

6.4.2 Participants

We asked the same group of participants that participated in our coding study (Section 6.3). 31 of those 34 participants responded to the survey. Once a participant completed the coding tasks and the survey on Industrial adoption of Opiner, he was invited to participate in this survey. To avoid fatigue due to the participation in the coding study, we asked each participant to respond to the survey at any time after their completion of the coding tasks.

6.4.3 Results

In Table 6.10, we show the percentage of the ratings for each usage summary algorithm for the four development scenarios (Selection, Documentation, Presentation, and Authoring). In Figure 6.14, we show the percentage of participants showing their preference towards the consultation of Opiner's usage summaries to stay aware about an API (awareness task).

For each of four development scenarios (Selection, Documentation, Presentation, and Authoring), more than 50% of the participants considered that the usage summaries in Opiner did not miss any information for their given tasks (except 48.3% for Presentation in Single Page Summary). The statistical summary was more favorable for presentation, and the concept-based summary was more useful for selecting an API to use. For documentation, all of the single, concept, and type-based summaries recorded equal preferences. We now discuss the responses of the participants for each development scenario in detail below.

Table 6.10: Impact of the summaries based on the scenarios

Scenario	Rating Option	Statistical Summary	Single Page Summary	Concept Based Summary	Type Based Summary
Selection	**Not Misses Any Info**	62.1	69.0	**72.4**	69.0
	Misses Some Info	31.0	17.2	27.6	24.1
	Misses Some Info	6.9	17.2	0.0	6.9
Document	**Not Misses Any Info**	51.7	**65.5**	**65.5**	**65.5**
	Misses Some Info	27.6	20.7	34.5	27.6
	Misses All Info	20.7	13.8	0.0	6.9
Presentation	**Not Misses Any Info**	**65.5**	48.3	58.6	58.6
	Misses Some Info	31.0	31.0	34.5	34.5
	Misses All Info	3.4	20.7	6.9	6.9
Authoring	**Fully Helpful**	48.3	**55.2**	41.4	31.0
	Partially Helpful	20.7	20.7	48.3	41.4
	Partially Unhelpful	6.9	3.4	3.4	3.4
	Fully Unhelpful	3.4	0.0	0.0	6.9
	Neutral	20.7	6.9	6.9	17.2

6.4.3.1 Selection

The concept-based summaries were considered as the most complete (72.4%) while making a decision on an API. Both the type-based and single page summaries were slightly behind (69%). The participants liked the incorporation of sentiment analysis into the usage summaries:

Concept and Reaction. The incorporation of reactions as positive and negative opinions in the usage summaries were considered as useful. According to one participant: *"Conceptual summary is the most useful of all. Inexperienced developers can select code snippet based on positive or negative reactions while the experienced developers can compare the code and go for the best one. They will be most benefited from the "All code examples summary" too."*

Statistics and Sentiment. However, just the presence of the reactions were not considered as enough for selection of an API when coding tasks were involved: *"Statistical summary just shows the negative and positive views, it is not useful to view the working examples or code. Conceptual summary groups together the common API features, it helped me to find the common examples in same place. Type based summary is also ok but had to dig in to find the description. All code examples provided the full code example, so it was good. I liked the link to details"*.

Type-based Summary and Rating. Type-based summary and the sentiment-based start rating was favored by many of the participants to assist in their selection task: *"I would probably make my choice thanks to the type based summary. As I want to use an API in a precise purpose (implement JSON features), the type based summary would give me all other types used with the API dealing with JSON and also code examples, so I would have a subsequent idea on how to roughly use the API in the system knowing the contents of the system source code and types that we use in it. Plus, I would have the opinions of Stack Overflow users about the best practicals to have using this API with the stars rate system"*

The single page summary was considered as too verbose for this task by some participants: *"It can be used to quickly look for some outstanding problem/solution for choosing this API and also quickly jump to detail page in stack overflow. But like type based summary, the information may be too bulky. Also this lacks "Getting Started" or "Setup" or "Prerequisite" sections which will be helpful for selection of new API."*

6.4.3.2 Documentation

The participants equally liked (65.5%) the three summaries (Concept-based, Type-Based, and Single Page) for the documentation task. Similar to the selection task, for this task also, all participants considered that concept-based summaries did not miss any information. According to one participant *""For Documentation purpose , Conceptual Summary is more readable than other two summary.""*. This preference towards concept-based summaries underlines the importance of previous research efforts in API usage pattern mining [185]. The participants provided the following strong and weak points of the top three summaries (concept, single page, and type-based) from documentation perspectives:

Concept-based + *completeness,* + *readability*

Completeness *"the code snippets are fairly complete. The give me the impression that you can just copy and paste and the code will work.""*

"Conceptual summary is important to generate different kind of ideas and thoughts to complete different kind of task such as serialization, deserialization, format specification, mapping with jackson API. This summary is most useful to me because of the availability of resource."

Conceptual Summary presented the usage with rating, I would chose the answer with highest rating cause it shows that this particular answer is correct on the given scenario, it also has example plus more related answer.

Readability *"For Documentation purpose , Conceptual Summary is more readable than other two summary."*

Type-bsaed + *completeness,* - *readability*

Completeness *"Type summary helps by directing different kind of operations type for to solve problems and also imply the importance of those operations with stats. So type summary is helpful to give instruction to accomplish particular task."*

Readability *"type based summary should have been more clear"*

Single page - *completeness,* + *readability*

Completeness *"I can't tell if it tells me everything I need to know about the API or not."*

Readability *"But the most readable for this task is the code examples summary, because we immediately know what we are talking about and that is where instinctively I would go if I wanted to copy code examples"*

The participants considered that the statistical summaries need to be enhanced with more summaries *"The statistical summary is fine but a time series visualization for an API usage over time could be very helpful. Because, it may be an important topic to know whether an API is still in use/popular etc."* and that the current visualizations are not good: *"Statistical summary is of no use for documentation."*

6.4.3.3 Presentation

The statistical summaries were favored by the most participants (65.5%), followed by concept-based summaries (58.6%). According to one participant, the combination of ratings and examples is the key: *"If I was the team lead, statistical summary would help to decide me to view the users positive and negative reaction over the selection API, I would definitely take this into consideration. Conceptual summary would help me to create a presentation based on the examples and the ratings."*

One participant ranked *"Considering conciseness and recency as decision criteria, the most important summaries are conceptual summary, statistical summary, all code summary and type based summary respectively. Conceptual summary consist outstanding, recent features/trends/developments in the API along with the sentiments and other related examples. A team lead will have*

already dove into some technical details and these insights will be helpful for presentation. The statistical summary helps to summarize the popularity of the API and other co-existing APIs. But it would be more helpful if there was comparisons among competitive APIs as well. The team lead should should this information one by one. The code examples doesn't provide concise information but since they are arranged in order of recency, new trends or questions about the API can be looked for. The type based summary may be a bit bulky. But if the team lead decides to highlight specific type and go in depth about it, then this summary will be useful."

6.4.3.4 Awareness

In Figure 6.14, we show how developers considered the usefulness of each summary to stay aware of API usage. Each summary was selected at least by five respondents to be used daily (maximum 15 for single page summary followed by 10 for concept-based summary). Each type of summary was considered as necessary by a participant: *"There are different scenario to which all these summary needs to be aware every week, because it misses some of the requirement related to the development."*. The combination of statistical and conceptual summaries was considered as useful by more than one participant: *"The statistical summary is very helpful in this scenario. API implementation evolves over time. Many APIs get changes and improvement which may effect the positive and negative sentiments. The conceptual summary also helps to identify how and when an API got improvement. There might be lot of information in the type based summary making the track for changes difficult."*

6.4.3.5 Authoring

When asked about whether the summaries and reactions of the Jackson API in Opiner showed enough weakness of the API that the developers would like to author a competing API, 48.4% responded with a 'No', 35.5% with a 'Yes' and 16.1% with Maybe. All participants considered that the combination of code examples with reactions helped them making the decision.

No *"In order to author a new API, I would like to understand how the market of developers is reacting to the current API. More negative responses would indicate that there is dissatisfaction among the developers and there is a need to create a new API for which I would look into the negative responses in the conceptual summaries. Given that I can see that there is a positive trend for jackson Api from statistical summaries and there are no or very few negative responses from the users among the eight summaries that I had selected. I feel that I would not author a new API to compete Jackson, simply because it still works and the developers are content, which I could identify from the statistical summaries."*

Yes *"The negative sentiments in statistical summary indicate that many think it might not be best tool for JSON manipulation or there are other alternatives. It also indicate that an improved API can be derive for the same purpose. The conceptual summary help to identify the weaknesses. "See Also" section in conceptual summary shows the competing library which helps to compare the merits and demerits of the API with other similar APIs."*

Maybe *"It is tough to decide to author a new API to compete this API . The summaries which are provided are good to learn. I think these will helpful for me in future to use override*

, searilizationfeature , deserialization context , visibility , finding , iterator , typereference , feature , generator etc type features to create or enhance new API . Opiner code tool might be helpful for take decision about because it provides various summaries to learn. I think as I get more experience I get better decision for authorship."

Besides using Opiner's usage summaries to code solutions for development tasks, developers found usefulness of Opiner's usage scenarios in diverse other development scenarios, ranging from learning an API, to writing its documentation, etc. The majority of the developers were interested in using Opiner in a daily or weekly basis.

6.5 Usefulness of Opiner Usage Summaries over API Documentation (RQ4)

Because the purpose of developing Opiner was to add benefits over the amazing usefulness Stack Overflow provides to the developers and to address the shortcomings in API official documentation, we sought to seek the usefulness of Opiner over both API formal and informal documents by asking a a series of questions to the same 31 participants who rated the different summaries in Opiner.

6.5.1 Study Design

The *goal* of this study was to analyze whether Opiner can offer benefits over the API documentation and how Opiner can be complement the existing API documents. The *objects* were the APIs and their usage summaries in Opiner, Stack Overflow, and Official documentation. The *subjects* were the participants who each responded to the questions.

We answer two research questions:

- **RQ4.1 Benefits over Formal Documentation.** How do the collected API usage scenarios offer improvements over formal documentation?
- **RQ4.2 Benefits Over Informal Documentation.** How do the collected API usage scenarios offer improvements over informal documentation?

RQ4.1 How do the collected API usage scenarios offer improvements over formal documentation?

We asked three questions to each developer based on her experience of using Opiner:

Improvements. For the tasks that you completed, did the summaries produced by Opiner offer improvements over formal documentation?

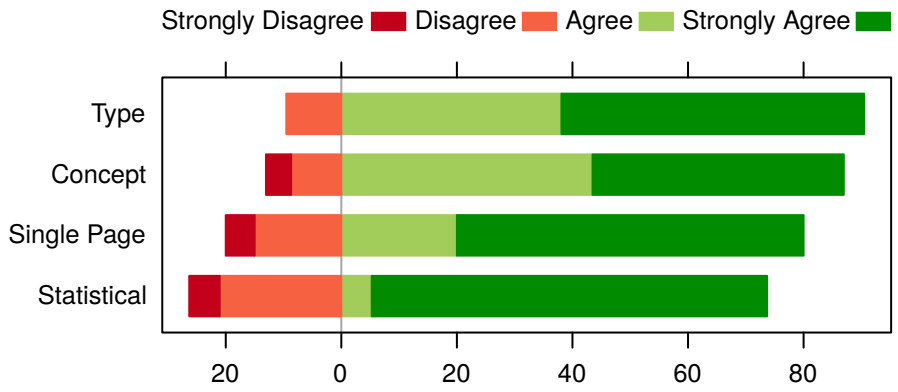

Figure 6.15: Developers' preference of Opiner summaries over formal documentation

Complementarity. For the tasks that you completed, how would you complement formal API documentation with Opiner results? How would you envision for Opiner to complete the formal API documentation.

The first question had a five-point Likert scale (Strongly Agree - Strongly Disagree). The second question had two parts, first part with a five-point likert scale and the second part with a open-ended question box.

RQ4.2 How do the collected API usage scenarios offer improvements over informal documentation?

We asked two questions to each developer based on his experience of using Opiner:

Improvements. How do Opiner summaries offer improvements over the Stack Overflow contents for API usage?

Complementarity. How would you envision Opiner to complement the Stack Overflow contents about API usages in your daily development activities?

6.5.2 Results

In this section, we discuss the results of the study.

RQ3.1 How do the collected usage scenarios offer improvements over formal documentation?

In Figure 6.15, we show the responses of the participants to the question of whether in their four development tasks Opiner offered improvements over formal documentation and if so which summary. The participants mostly (more than 90%) agreed that the Opiner usage summaries did indeed offer improvements over formal documentation. Because formal documentation in Java are mainly javadocs with a single page view for each class, the type-based summary was the most relevant for such documentation. Therefore, it is intuitive that participants thought that the Opiner type-based summaries were a considerable upgrade over the formal javadocs. The concept-based summaries while not present in any formal documentation, came in second position, just slightly below the type-based summary, showing the needs for recommendations based on clustering of similar API usage scenarios, in the formal documentation. When we asked the same participants how they would like Opiner to introduce such benefits to the formal documentation, a majority of developers participants (79%) wanted Opiner results to be integrated into the formal documentation, rather than replacing the formal documentation (65.5%) altogether with the summaries of Opiner. The developers offered the following suggestions for complementing the formal documentation with Opiner.

Augmenting Informal Viewpoints *"The formal API documentations are complete and are perfectly tuned now a days to suit the naivest developer. One thing that these miss on are the problems and discussion forum, for which Stack Overflow was created. Opiner could complement the formal documentation, if the conceptual summaries are incorporated within the API documentations."*

Sentiment Presence *"Statistical summary would help me to find the users decisions about the API. I would chose conceptual summary and over other summaries rather than formal documentation because these provides the usage example with positive and negative reaction."*

Uptodateness *"Formal documentation will almost always be the starting point for new APIs. But as the APIs start to grow, Opiner will serve as the most useful tool to find correct solutions to the problems in less time, become familiar with the trends and compare different alternatives."*

More than 85% of the respondents agreed that Opiner usage summaries offer improvements over formal API documentation to assist in their development activities. The respondents liked how Opiner combined sentiment analysis with code examples to produce usage summaries and considered that formal API documentation can be more effective with Opiner's summaries integrated into the API formal documents. As complimentary to the formal documents, type-based summaries are considered as the most viable candidate, followed by concept-based and single page summaries.

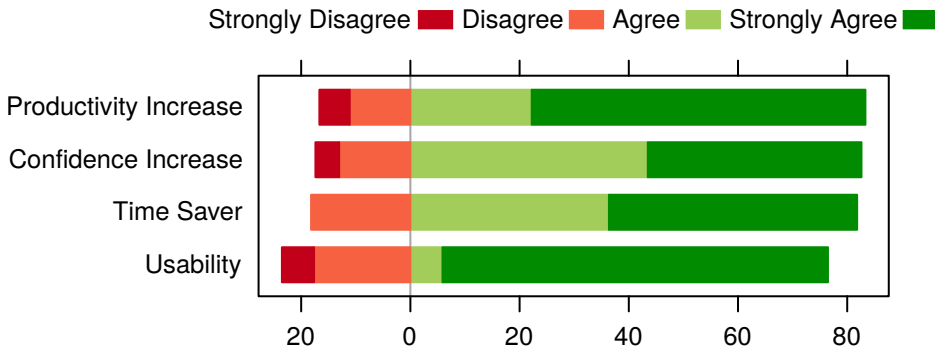

Figure 6.16: Benefits of Opiner over informal documentation

RQ3.2 How do the collected usage scenarios offer improvements over informal documentation?

In Figure 6.16, we show how developers responded to our questions on whether in their coding tasks Opiner offered benefits over the informal documentation. More than 80% of them agreed that Opiner summaries offered increases in productivity, confidence, and saved time over the informal documentation. The developers offered suggestions on how Opiner can complement informal documentations:

Targeted summaries *"there are some sections like conceptual summary and code example would be useful for developer on daily basis"*

Learnability *"Just for learning any new API, it is great. Because, the most challenging thing I faced in tech based domains is the collection of my vocabulary. To search or ask something we need to first know the proper question. If we do not know with what terms I would ask something, then no only a tool but not even a human can answer me. "*

Sentiment *"Opiner has more feature set to help the developers find out the better API for a task. The evaluation and nice presentation of positive and negative sentiments help them decide an API."*

Time *"It is quicker to find solution in Opiner since the subject is well covered and useful information is collected."*

More than 80% of the developers agreed that Opiner summaries offered benefits over informal API documents by facilitating increase in developer productivity, confidence and time spent. The developers asked for better search and filtering capabilities in the tool to find and discover usage scenarios quickly and easily.

6.6 Threats to Validity

We now discuss the threats to validity of our study following common guidelines for empirical studies [172].

Construct validity threats concern the relation between theory and observations. In our study, they could be due to measurement errors. In fact, the accuracy of the algorithm used to associate code examples to APIs has an impact on our results. In RQ1, we assessed the performance of this algorithm and found that the algorithm achieves a 92% precision and 99% recall. Which is similar to the performance of other state of the art code example resolver techniques, e.g., Baker [35] and Recodoc [59]. To assess the performance of participants, we used time and percentages of correct answers, which are two measures that could be affected by external factors, such as fatigue. To mitigate the effect of fatigue, we asked participants to take breaks between the tasks. We also used TLX to measure the effort of participants. TLX is by its nature subjective. However, it is a mature tool that has been used in multiple empirical studies, e.g., [186].

Internal validity threats relevant in this study are learning, selection, and diffusion threats. Learning threats do not affect our study since we used a between-subject design. Each participant performed different tasks in different settings. Diffusion threats do not impact our study because our participants did not know each other, and therefore could not discuss about the experiments.

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our study. The anonymized survey responses are provided in our online appendix [168].

6.7 Summary

The rapid growth of online developer forum posts encourages developers to harness the knowledge of the crowd in their daily development activities. The enormous volume of such insights distilled into the API code examples in the forum posts presents challenges to the developers to gain quick and actionable insights into the usage of an API. In this chapter, we present a summarization framework to automatically mine and summarize API usage scenarios from developer forum posts. We implemented the framework in our tool Opiner. In multiple case studies involving Opiner API usage summaries, we found that

1) Opiner can associate a code example to the right API with up to 88.3% precision and 98% recall.

2) Developers spent less time and effort while coding solutions to development tasks using Opiner compared to formal and informal documentation. Moreover, the accuracy of their solutions is higher using Opiner.

3) More than 80% of developers considered that Opiner offer improvements over both formal and informal API documentation, and wished to use Opiner in their daily or weekly development activities.

Chapter 7

Related Research

This thesis builds on the previous work related to the programming studies on how developers select and use APIs, the tools and techniques to assist developers in such development needs, and the techniques related to the automatic mining and summarization of software engineering artifacts.

7.1 How developers learn to select and use APIs

Robillard and DeLine [19] conducted a survey and a series of qualitative interviews of software developers at Microsoft to understand how developers learn APIs. The study identified that the most severe obstacles developers faced while learning new APIs were related to the official documentation of the APIs. With API documentation, the developers cited the lack of code examples and the absence of task-oriented description of the API usage as some major blockers to use APIs.

Carroll et al. [20] observed developers while using traditional documentation to learn an API (e.g., Javadoc, manual). They found that the learning of the developers was often interrupted by their self-initiated problem-solving tasks that they undertook during their navigation of the documentation. During such unsupervised exploration, they observed that developers ignored groups and entire sections of a documentation that they deemed not necessary for their development task in hand. Unsurprisingly, such unsupervised exploration often led to mistakes. They conjectured that traditional API documentation is not designed to support such active way of developers' learning. To support developers' learning of APIs from documentation, they designed a new type of API documentation, called as the *minimal manual*, that is task-oriented and that helps the users resolve errors [20, 187–189].

Shull et al. [190] conducted a study of 43 participants to compare the effectiveness of example-based documentation with hierarchy-based documentation. The example-based documentation contained example applications that were found within the framework that the participants used. The hierarchy-based documentation is similar to the Javadocs, where the classes were presented. Participants were specifically taught on how to use the hierarchy-based documentation. However, they abandoned it in favor of the example applications based documentation, because it took them longer to start writing application using the hierarchy-based documentation.

Forward and Lethbridge [191] conducted a survey of 48 software developers and managers. The goal was to understand the perceived importance of software documentation and the required tools and techniques to maintain those. The participants considered the content (information and use of examples) and organization (index, sections) of the documents as the most important, as well as the recency and availability of such information.

De Souza et al. [39] studied the relative importance of the various documentation artifacts using two surveys of software developers. They identified 24 artifacts based on the responses from software developers, such as, source code, comments, data model, test plan, gloassary, etc. More than 92% of the participants considered source code to be most important in the documentation, followed by comments (78.2%) and logical data model (74.6%).

Nykaza et al. [192] assessed the needs on the desired and required content of API documentation in a software organization. They found similar needs among junior programmers with extensive knowledge of a domain and the experienced programmers with no knowledge of the domain. The common needs among them pertained to the availability of code examples that are easy to understand and used for a given task. The developers also preferred self-contained sections in API documentation, as opposed to a manual that is fragmented and long, and needed to be completed from the start to finish before writing the code.

The advent of cloud-based software development has popularized the adoption of Web APIs, such as, the Google Map APIs, API mashups in the ProgrammableWeb, and so on. Tian et al. [193] conducted an exploratory study to learn about the features offered by the Web APIs via the ProgrammableWeb portal and the type of contents supported in the documentation of those APIs. They observed that such Web APIs offer diverse development scenarios, e.g., text mining, business analysis, and so on. They found that the documentation of the APIs offer insights into different knowledge types, e.g., the underlying intent of the API features, the step-by-step guides, and so on.

During our user studies (in Section 6.3) involving both professional software developers and students, we found that the developers were more productive while using the usage summaries in Opiner. In a way, the summaries in Opiner are task-based in that they correspond to specific development problems the developers discussed in Stack Overflow. We can corroborate to the findings of Forward and Lethbridge [191] and De Souza et al. [39] that developers prefer the quality of the contents in the API documentation the most, in multiple observations: 1) Our survey with the developers at IBM showed that developers were more concerned about the documentation contents and that problems in the contents may even force them to abandon the usage of an API. 2) Our surveys of developers from Stack Overflow and Github (in Chapter 2) and the evaluation of the API review summaries in Opiner (in Chapter 5) show that developers consider the summaries as a form of API documentation. We also observed that the summaries can assist them in diverse development needs, such as, selection of an API amidst choices, improving a feature, fixing a bug, and so on.

7.2 Content Categorization in Software Engineering

Maalej and Robillard [41] analyzed the nature of knowledge patterns that exist in the Java and .NET official documentation. They identified a taxonomy of 12 knowledge types, such as, functionality and behavior, concepts, structure, directives and so on. To assess the prevalence of the knowledge types, 17 coders used the taxonomy and rated a total of 5574 documentation units. Among the coded knowledge types, functionality was found as the most frequent, followed by non-info (e.g., an uninformative boilerplate text). In a subsequent study, Robillard and Chhetri [194] categorized the text fragments in the API official documentation as of three types: (1) Indispensable, (2) Valuable, or (3) Neither. They built a recommendation tool that given as input an API official documentation unit, can automatically categorize the unit as one of the three types.

Chen and Xing [195] developed a technique to group similar technological terms in Stack Overflow. They considered tags associated to the questions in Stack Overflow as *technologies for computer programming*. The identified the technology associations based on the co-occurrence of the tags in Stack Overflow. They used association rule mining and community network graphs to mine the co-occurred tags. From the mined association rules, they construct a 'Technology Associative

Network' (TAN). In a TAN, each node corresponds to a tag and the edge between two tags is determined based on the association rules. In a subsequent work, Chen et al. [196] developed a thesaurus of software-specific terms by mining Stack Overflow. Their thesaurus can be used to find similar conceptual terms in the textual contents of forum posts (e.g., multi-threaded vs multi-core). In a previous study, Chen et al. [197] also developed a technique to automatically translate an Stack Overflow post in English to Chinese. Zou and Hou [198] developed a tool, LDAAnalyzer, that using LDA, can cluster Stack overflow posts with similar topics.

Bacchelli et al. [199] investigated island parsing and machine learning classifiers to automatically categorize the contents of development emails at the line level into the following categories: 1) Textual contents, 2) Stack traces, 3) Patch, 4) Source code, and 5) Irrelevant. They experienced high accuracy in the classification when island parsing was fused with the machine learning classifiers.

Hou and Mo [95] proposed techniques to automatically categorize API discussions in the online Oracle Java Swing forums. First, they manually labeled 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. [96] proposed an algorithm to improve accuracy of the automatic categorization.

Zhang and Hou [98] 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 [200] investigated the design of an automated engine that can advise the use of an API based on the formal semantics of the API.

Thung et al. [201] proposed a semi-supervised technique to automatically classify software defects into different categories, such as, symptoms, root causes, and so on. The algorithm takes a small number of labeled defects and iterates over the rest of the unlabeled defects until all the defects are labeled. For each iteration, the algorithm adds the defects to the labeled dataset for which it had high confidence. The technique showed a precision of 0.651 with a recall of 0.669.

Sharma et al. [202] proposed a technique, NIRMAL to automatically identify tweets related to software. The technique is trained on a corpus of Stack Overflow posts. From this training a language model is learned, which is then applied to a large number of tweets. A manual analysis of 200 tweets showed that the technique can achieve an accuracy between 0.695 and 0.900.

Similar to Maalej and Robillard [41], in our study at IBM (Chapter 3), we also observed that developers consider excessive structural information about a Java class in the Javadoc as something that is not useful, i.e., a non-info. Similar to the above content categorization approaches, in this dissertation, we also developed automatic categorization approaches to gain insights from API discussions. Our categorization approaches differ from the above techniques as follows:

- We categorize the opinions about APIs by 11 API aspects, such as, performance, usability, other general features, and so on (see Chapter 4). We also identify *dynamic aspects* in the sentences labeled as 'other general features'. Unlike Hou and Mo [95], 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 [95].

- The negative opinions we mine about APIs can be considered as denoting to *problematic API features*. However, unlike Zhang and Hou [98], 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.

7.3 API Artifact Traceability Recovery

Hayes et al. [203] used three IR algorithms (LSI, VSM, and VSM with thesaurus) to establish links between a high and low-level requirement descriptions. Lormans et al. [204] used LSI to find relationships between requirements, tests, and design documents. Baysal et al. [205] correlated emails in mailing lists and software releases by linking emails with the source code. Types and variable names in the source code were matched against natural language queries to assist in feature location [206–208]. Given as input a bug report, Hipikat [209] finds relevant source code and other artifacts (e.g., another bug report). Subsequent techniques linked a bug fix report to its related code changes [210, 211], or detected duplicate bug reports [84].

Recodoc [59] resolves code terms in the formal documentation of a project to its exact corresponding element in the code of the project. Baker [35] 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 [60] resolves Java code terms in forum textual contents of posts using island grammars [157]. Bacchelli et al. [61] developed Miler to determine whether a development email of a project contains the mention of a given source code element. They compared information retrieval (IR) techniques (LSI [208] and VSM [107]) against lightweight techniques based on regular expressions. Prior to Miler, LSI was also used by Marcus et al. [208], and VSM was compared with a probabilistic IR model by Antoniol et al. [212]. Tools and techniques have been proposed to leverage code traceability techniques, e.g., linking software artifacts where a code term is found [213], associating development emails with the source code in developers' IDE [214], recommending posts in Stack Overflow relevant to a given code context in the IDE [215].

Parnin et al. [216] investigated API classes discussed in Stack Overflow using heuristics based on exact matching of classes names with words in posts (title, body, snippets, etc.). Using a similar approach, Kavaler et al. [217] 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 [218, 219] investigated the relationship between API changes and developer discussions.

Opiner extends to this line of work by proposing a heuristic-based technique to associate a code example in forum posts to one or more APIs.

7.4 API Usage Scenario Extraction

To the best of our knowledge, no existing API usage scenario extraction techniques from informal documents have been proposed prior to our thesis (Section 6.1). However, a significant body of

research extracted API usage scenarios from API repositories. The techniques used both static and dynamic analyses [147–150].

Our API usage concept detection technique (Section 6.1) is influenced by our previous work [42, 220], where we investigated the automatic detection of the usage concepts of an API from the version histories of client software using the API. Given as input the different versions of an API and the version history of a client software where the API was used, we identified the change sets and the API types and methods in the version history. The identification technique utilized static code analysis and it was built on top of two techniques, SemDiff [221] and Partial Programming Analysis (PPA) [222]. We transformed each change set into a list. A list consisted of the elements (types and methods) used from the API in the change set. Given as input all such lists of an API, we clustered the API elements that are used mostly together. We then linked each change set to a cluster. Each cluster was considered as an API concept. Each cluster was then assigned a timestamp based on the commit time of the change set. We then identified patterns of cluster sequence for an API. We observed that more than one such patterns can be found across the client software. For example, while using the Apache HttpClient API [223], developers first implemented the concept to established a connection to a HTTP server and then implemented another concept to send/receive HTTP messages using the connection.

A considerable number of research efforts has been devoted to utilize dynamic analysis to detect usage patterns in a software. Safyallah and Sartipi [150] applied sequential pattern mining algorithm on the execution traces of the application software. The execution traces are collected by running a set of relevant task scenarios, which is called as a feature-specific scenario set. For example, for a drawing software all the tasks related to the zooming ability of the software can be considered as a feature-specific scenario set. Important execution patterns for a given feature are retrieved from all the execution traces of the feature by identifying traces that are shared among the different traces. Lo et al. [148] designed an algorithm to find closed iterative patterns by analyzing the execution traces of an application. A pattern is closed if it represents a sequence of events in the traces and the sequence does not have any super-sequence. A pattern is iterative, if it can be observed multiple times in a trace. Lo and Maoz [147] proposed a technique to mine specifications of hierarchical scenarios. A *hierarchy* is defined by a partial order/sequence of the elements (e.g., method, class, package) used in a software system. Once such a partial order is defined, *zoom in* and *zoom out* features are automatically mined. For example, given as input an execution pattern, a zoom out feature will show the super set of the pattern. In a later work, Lo and Maoz [224] proposed a technique to automatically mine both specification as a sequence of method calls and the values that can be passed to the methods. Fahland et al. [149] proposed a technique to mine branching-time scenarios from the execution traces. A branch can be a conditional execution point in the software. Depending on the conditions different execution traces can be obtained. The technique was able to automatically find alternative behaviors of the software.

Our findings in this thesis show that API usage concepts can also be mined from informal documents and that can be useful to summarize API usage scenarios.

7.5 Sentiment Analysis in Software Engineering

Ortu et al. [51] observed weak correlation between the politeness of the developers in the comments and the time to fix the issue in Jira, i.e., bullies are not more productive than others in a software development team. Mântylà et al. [52] correlated VAD (Valence, Arousal, Dominance) scores [225] in Jira issues with the loss of productivity and burn-out in software engineering teams. They found that the increases in issue's priority correlate with increases in Arousal. Pletea et al. [226] found that security-related discussions in GitHub contained more negative comments. Guzman et al. [227] found that GitHub projects written in Java have more negative comments as well as the comments posted on Monday, while the developers in a distributed team are more positive.

Guzman and Bruegge [54] summarized emotions expressed across collaboration artifacts in a software team (bug reports, etc.) using LDA [102] and sentiment analysis. The team leads found the summaries to be useful, but less informative. Murgia et al. [55] labelled comments from Jira issues using Parrot's framework [22]. Jongeling et al. [122] compared four sentiment tools on comments posted in Jira: SentiStrength [114], Alchemy [228], Stanford NLP [126], and NLTK [229]. While NLTK and SentiStrength showed better accuracy, there were little agreements between the two tools. Novielli et al. [56] analyzed the sentiment scores from the StentiStrength in Stack Overflow posts. They observed that developers express emotions towards the technology, not towards other developers. They found that the tool was unable to detect domain-dependent sentiment words (e.g., bug).

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. Indeed, in Opiner, we developed a sentiment detection algorithm to detect opinionated sentences about APIs. As we discussed in Section 4.5, our technique relies on both domain specific sentiment vocabularies and two sentiment detection algorithms to offer improved performance over the cross-domain sentiment classifiers.

7.6 Summarization in Software Engineering

Natural language summaries have been investigated for software documentation [62, 230–232]. Most recently, McBurney et al. [233] proposed a technique to generate automatic documentation via the summarization of method context. The notion of context is very important for developers because it helps understand why code elements exist and the rationale behind their usage in the software [7, 234, 235].

Natural language summaries have been investigated for source code [86, 233, 236, 237]. Murphy [238] proposed two techniques to produce structural summary of source code. Storey et al. [239] analyzed the impact of tags and annotations in source code. The summarization of source code element names (types, methods) has been investigated by leveraging both structural and lexical information of the source code [240], by focusing on contents and call graphs of Java classes [237], based on the identifier and variable names in methods [236]. The selection and presentation of source code summaries have been explored through developer interviews [38], as well as eye tracking experiment [241].

Sorbo et al. [242, 243] developed SURF, a summarizer engine for user reviews for mobile apps.

The tool analyzes and classifies reviews for a mobile app and synthesizes action-related change tasks from the reviews to inform the app developer. Carrẽno and Winbladh [244] used topic modeling on the positive and negative comments for three different android mobile applications and showed that the topics can be used to change and create new requirements that *"better represent user needs"*. Chen et al. [245] mined *informative* reviews for developers from the Android mobile app reviews. First, they removed the noisy and irrelevant ones. Second, they grouped the informative reviews automatically by using topic modeling. Third, they presented a review ranking scheme to further prioritize the informative reviews. Fourth, they presented the the most "informative" reviews via a visualization approach.

Bowen et al. [246] proposes AnswerBot that given as input as the description of a task recommends a number of relevant posts from Stack Overflow. The technique consists of three major steps. First, it finds questions relevant to the task. Second, it finds paragraphs from the answers to the questions that may be useful to complete the task. Third, it summarizes those answer paragraphs to offer a quick insight into the potential solutions to the task.

Our findings in this dissertation confirmed that developers also require different types of API review and usage scenario summaries.

7.7 Leveraging Crowd-Sourced API Code Examples

Zilberstein and Yahav [247] proposed an approach to leverage code examples from forum posts to assist in the auto-code completion while using an IDE. They built Codota, a tool that given as input a code context from the IDE where a developer is programming, automatically searches forum posts to find the most relevant code examples that can be relevant to the code context. While the focus of both Opiner and Codota remains the same, i.e., assisting developers in their development activities, Opiner offers the following notable differences from Codota:

Summary Vs Targeted Recommendation. Opiner focuses on the automatic mining and summarization of API usage scenarios. Codota offer targeted recommendations that depend on the context of the coding tasks of a developer in the IDE.

Documentation Vs. IDE. Opiner's focus is to offer improvements over developers' usage of formal and informal API documentation, by producing automated summaries of usage scenarios from forum posts. Codota is a real-time assistant that aim to facilitate auto code-completion in the developers' IDE.

Diversity. Opiner's usage scenarios can not only assist developers in their coding tasks, but also in other diverse development scenarios, e.g., their selection of an API as well as the production of documentation.

Sentiments. Opiner leverages sentiment analysis to offer insights to developers on whether a given usage scenario is well-regarded or not in the forum posts. To the best of our knowledge, Codota does not offer such feature.

Statistics. Opiner offers visualized representation of API usage in the summaries to facilitate quick insights into the API usage, which our study participants considered to be very useful

for their learning and usage of the APIs. To the best of our knowledge, Codota does not offer such feature.

Chapter 8

Conclusions

Opinions can shape the perception and decisions of developers related to the selection and usage of APIs. The plethora of open-source APIs and the advent of developer forums have influenced developers to publicly discuss their experiences and share opinions about APIs.

To understand how and why developer seek for opinions about APIs in online developer forums and the challenges they face while analyzing such information, we conducted two surveys involving a total of 178 software developers from Stack Overflow and Github. We found that developers seek for opinions about APIs to support diverse development needs, such as, selection of an API amidst choices, fixing or enhancement of a software feature, and so on. We also found that developers analyze opinions to determine how and whether an API usage scenario should/can be used. To understand why API formal documents fail to meet API usage needs, we conducted two more surveys at IBM. We found 10 common documentation problems that persist in API formal documentation. To compensate for the shortcomings of API formal documents, usage scenarios posted in the developer forums can be used. The developers considered the development of tools as necessary to analyze the plethora of such opinions and usage scenarios posted in the developer forums.

We designed a framework to automatically mine and summarize API reviews and usage scenarios from developer forums. We developed, Opiner, our web-based search and summarization engine for API reviews and usage scenarios. In multiple user studies, we found that developers were able to make more correct API selection using Opiner review summaries, and were able to write more correct code in less time and effort using Opiner usage scenario summaries.

In the following, we summarize the major contributions of this dissertation in Section 8.1, and discuss the future research opportunities based on the research conducted in this thesis in Section 8.2. We conclude the thesis in Section 8.3.

8.1 Contributions

We started the thesis with the design and analysis of two surveys of a total of 178 software developers from Stack Overflow and Github. The goals of the survey were to understand the developers' needs to seek opinions about APIs and to determine their needs for tool support to analyze the opinions. Through an exploratory analysis of the open-ended and closed questions, we found that developers consider opinions about APIs as a source of API documentation and that they use the opinions for diverse development needs, such as, selection of an API amidst choices, learning how to use an API feature or determining the effectiveness of a provided code example in the forum posts, and so on. The design of the surveys and the analysis of the results form our first research contribution.

As we observed in the above two surveys, developers considered API usage scenarios posted in the developer forums as an important source to learn and use an API. They also considered that they use the combination of code examples and the opinions posted in the forum posts to validate usage examples posted in the API formal documentation. Therefore, the API usage scenarios posted in

the forum posts and the opinions can be used to compensate for the shortcomings in the API formal documentation. To properly understand how such shortcomings can be addressed, we first need to understand what specific problems developers face while using API formal documentation. In Chapter 3, we presented the design and results of two surveys involving 323 software developers from IBM. We conducted the study to understand the common problems that developers face while using formal API documentation. We found 10 common documentation problems that persist in the API formal documentation, which may force a developer to decide not to use the API. The design of the survey and the analysis of the results form the second contribution of this thesis.

With a view to understand how opinions about APIs are discussed, we produced a dataset with manual labels from a total of 11 human coders (Chapter 4). The benchmark contains a total of 4522 sentences from 1338 posts from Stack Overflow. Each sentence is labeled as (1) the API aspect that is discussed in the sentence and (2) the overall polarity (positive, negative, neutral) expressed in 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 in the opinions. The dataset and the case study form the third contribution of this thesis. We present a suite of techniques to automatically detect API aspects in API reviews, and to mine opinions about APIs from the reviews. The algorithms and their evaluation form the fourth contribution of this thesis.

In Chapter 5, we investigate a suite of algorithms to summarize opinions about APIs. We designed the statistical and aspect-based summaries by taking into account the specific contextual nature of software engineering into those summaries. We discuss the results of two user studies that we conducted to analyze the effectiveness of the summaries to assist developers in their development needs. The design of the statistical and aspect-based summarization algorithms and their evaluation form the fifth contribution of the thesis. We take cues from the study of documentation problems to design the API usage scenario summaries in Opiner. In Chapter 6, we present a framework to automatically mine and summarize usage scenarios about APIs. We evaluate the usage summaries in Opiner using three user studies. The design and development of the algorithms and their evaluation form the sixth and final contribution of this thesis.

8.2 Future Work

In this section, we discuss immediate future research opportunities based both on the technical and theoretical refinements of Opiner and the general research direction of the summarization of API artifacts.

Multi-faceted analysis of API opinions

As we observed, the analysis of API-related opinions is often multi-faceted, correlating both technical (e.g., API features) and social aspects (e.g., community engagement). By understanding how opinions guide developers' decisions, we can design tools that can better support their use and development of APIs. Such analysis can add another dimension to the understanding of team productivity through emotion analysis [51,52]. The research in this dissertation and the tool, Opiner, can be considered as a first step towards understanding this phenomenon. The benefits of Opiner to assist developers in diverse development needs during our user studies are encouraging. A finer-

grained analysis of the opinions can shed more light into the usefulness of Opiner and about API reviews in general. We are currently extending Opiner to mine and summarize diverse emotions expressed in the forum posts, such as, joy, anger, politeness, etc. Specifically, we aim to understand how the six emotions defined in Parrot's framework [22] are prevalent in API reviews, and how they impact the decision making process of the developers, and whether there is a correlation between certain emotions and sentiments expressed in the developer forums. For example, Ortu et al. [51] analyzed emotions expressed in Jira issues and found that bullies are not necessarily more productive than others. We would like to extend this analysis on API reviews posted in forum posts, such as, are bullies regarded more positively or negatively in the forum posts?

Enabling Situational Relevance Analysis with API Aspects

In our survey of Stack Overflow and Github developers to understand the role of opinions about APIs, we observed that developers face difficulties to determine the situational relevancy of a provided opinion. Such problems can stem from their development needs. For example, a developer seeking to use a code example about an API as posted in the forum post, can analyze the opinions posted about the API and the code example to decide whether he should use the code example. In our usage scenario summaries about APIs, we aimed to assist developers by mining and summarizing API usage summaries that combine both code example and sentiment analysis. We found that the participants in our user studies appreciated this approach. We can further assist the developers by categorizing the opinions by the API aspects (e.g., performance, usability). We can also explore the feasibility of integrating Opiner usage and review summaries directly into the IDE. One possible way to achieve this would be to integrate the Opiner results into the Codota tool [247]. Codota is a tool that given a code context from the IDE where a developer is programming, automatically searches forum posts to find most relevant code examples that can be relevant to the code context (see Section 7.7 in Chapter 7).

Mining Multiple Developer Forums

The current version of Opiner shows opinions and usage scenarios about APIs mined from Stack Overflow. In our user study to learn about the Industrial adoption opportunities of Opiner, one of the participants mentioned that *"Another thing that could improve Opiner is if it mined from other websites too. I think it would give an even better opinion and I would be tempted to use it more than Google then."*. One of the other participants (a software architect) asked after the study about the feasibility of having all the data mined and summarized from Stack Overflow in Opiner. Such large-scale analysis of reviews and usage summaries about APIs can provide us interesting insights into how an API is discussed across the forums. This can also pave the way for interesting task-centric extensions of Opiner, such as, searching for API information based on a task description (e.g., give me the top 10 APIs that offer object conversion in Java). A number of research initiatives can be investigated, such as, searching software documentation using domain-specific queries [248], automatic generation of answer summaries for a given task [249], and so on.

Opinion Quality Analysis

In our surveys of Stack Overflow and Github developers on how and why developer seek and analyze opinions about APIs (Chapter 2), the developers expressed their concerns about the trustworthiness of the provided opinions, in particular those that contain strong bias. Since by the definition an "opinion" is a "personal view, belief, judgement, attitude" [250], all opinions are biased. However, developers associate *biased* opinions with noise defining them as the ones not being supported by facts (e.g., links to the documentation, code snippets, appropriate rationale). While the detection of spam (e.g., intentionally biased opinion) is an active research area [24], it was notable that developers are mainly concerned about the *unintentional* bias that they associate with the user experience. Developers are also concerned about the overall writing quality. A number of exciting research avenues can be explored in this direction, such as, the design of a theoretical framework to define the quality of the opinions about APIs, the development of a new breed of recommender systems that can potentially warn users of any potential bias in an opinion about API in the developer forums, and so on.

Modelling opinion evolution

Since frameworks and libraries change their APIs [251], opinions about APIs evolve too. The evolution can be influenced by diverse factors, such as, a problematic feature is fixed, a new API version is released with more features, and so on. In our surveys of Stack Overflow and Github developers on how and why developer seek and analyze opinions about APIs (Chapter 2), the developers raise their needs to track the opinions about APIs across multiple versions of an API. One potential way to achieve this would be to track an API version in the provided opinions. Another way would be to pair contrastive pairs of opinions about an API. For this second approach, we experimented with the contrastive viewpoint algorithm in Opiner in two ways:

1) We produce a contrastive summary of all the opinions about an API (see Section 5.4)

2) We further cluster those opinions by aspects of an API and then identified the contrastive pairs of opinions (see Section 5.3).

One possible weakness of the above two approaches is that the contrastive pairs do not distinguish the API versions in the opinion pairs. Unlike other domains (e.g., camera or car), the problem is more prevalent to the domain of APIs in that APIs can be updated and released in multiple versions and quite rapidly. The understanding of the semantic differences between opinions and the linking of those opinions to the evolution of APIs can be useful to assist the developers more precisely in their selection of APIs, e.g., which version of an API is more stable?

Feedback Adapted Summarization

Finally, the mining and summarization of reviews and usage scenarios in Opiner (in chaps4-6) are based on a number of supervised and rule-based techniques. The accuracy of the techniques can be improved by collecting feedback from the users during their usage of the system (using an online

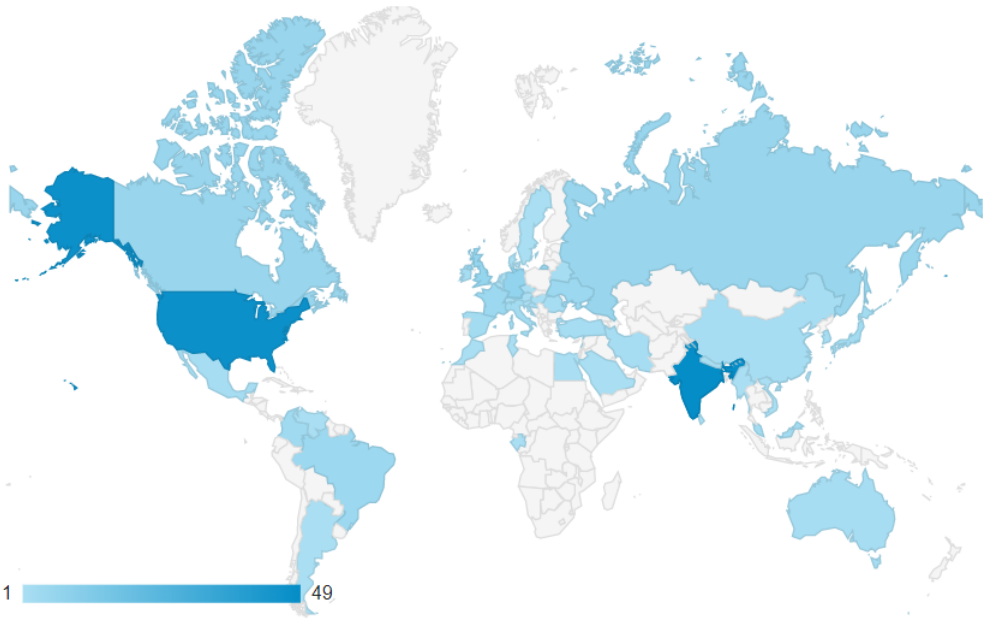

Figure 8.1: Screenshots from the Google Analytics of the visits of Opiner's website by countries between 30 October 2017 and 7 March 2018

feedback button). The feedback thus can be automatically fed to an online learner, such that, the techniques can be improved adaptively (e.g., using the relevance feedback algorithm [107]).

8.3 Closing Remarks

API reviews and usage scenarios are sought-after, and are widely available in the developer forums. Our surveys of software developers show how and why such reviews and usage scenarios can be useful to support diverse development needs, and how API official documentation may not be sufficient enough to suite those needs. We show that the plethora of such reviews and usage scenarios can pose problems to the developers to make quick, but informed insights about APIs. Our proposed techniques and the tool Opiner were found as useful by the participants, showing that there is an opportunity to assist developers by advancing research in this direction.

Since its online deployment on October 30, 2017, Opiner has been accessed almost everyday by hundreds of users from 49 countries across all the continents (except Antarctica) (Figure 8.1).

Bibliography

[1] Gias Uddin and Martin P. Robillard. How API documentation fails. *IEEE Softawre*, 32(4):76–83, 2015.

[2] Hyun Duk Kim, Kavita Ganesan, Parikshit Sondhi, and Chengxiang Zhai. Comprehensive review of opinion summarization. Technical report, University of Illinois at Urbana-Champaign, 2011.

[3] StackOverflow. *Converting JSON to Java*. `http://stackoverflow.com/q/1688099/`, 2009.

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

[5] Deen Freelon. ReCal3: Reliability for 3+ coders. `http://dfreelon.org/utils/recalfront/recal3/`, 2017.

[6] Ekwa Duala-Ekoko and Martin P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. In *Proc. 25th Euro Conf. Object-Oriented Prog.*, pages 79–104, 2011.

[7] Ekwa Duala-Ekoko and Martin P. Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proc. 34th IEEE/ACM International Conference on Software Engineering*, pages 266–276, 2012.

[8] github.com. *Github*. `https://github.com/`, 2017.

[9] Ohloh.net. `www.ohloh.net`, 2013.

[10] Sourceforge.net. `http://sourceforge.net/`, 2017.

[11] freecode.com. *Freecode*. `http://freecode.com/`, 2017.

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

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

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

[15] StackExchange. *Data Explorer*. `https://data.stackexchange.com/stackoverflow/queries`.

[16] StackOverflow. *Tags*. `https://stackoverflow.com/tags`.

[17] Lars Magne Ingebrigtsen. *Gmane*. `http://gmane.org/`.

[18] Reddit. `https://www.reddit.com/`.

[19] Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[20] John M. Carroll, Penny L. Smith-Kerker, James R. Ford, and Sandra A. Mazur-Rimetz. The minimal manual. *Journal of Human-Computer Interaction*, 3(2):123–153, 1987.

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

[22] W. Gerrod Parrott. *Emotions in Social Psychology*. Psychology Press, 2001.

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

[24] Bing Liu. *Sentiment Analysis and Subjectivity*. CRC Press, Taylor and Francis Group, Boca Raton, FL, 2nd edition, 2010.

[25] Minqing Hu and Bing Liu. Mining and summarizing customer reviews. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 168–177, 2004.

[26] Sasha Blair-Goldensohn, Kerry Hannan, Ryan McDonald, Tyler Neylon, George A. Reis, and Jeff Reyner. Building a sentiment summarizer for local search reviews. In *WWW Workshop on NLP in the Information Explosion Era*, page 10, 2008.

[27] Hyun Duk Kim and ChengXiang Zhai. Generating comparative summaries of contradictory opinions in text. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 385–394, 2009.

[28] H. P. Luhn. The automatic creation of literature abstracts. *IBM Journal of Research and Development*, 2(2):159–165, 1958.

[29] Rada Mihalcea and Paul Tarau. Textrank: Bringing order into texts. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 404–411, 2004.

[30] K. kireyev. Using latent semantic analysis for extractive summarization. In *Proc. of Text Analysis Coneference*, pages 84–89.

[31] Yashar Mehdad, Amanda Stent, Kapil Thadani, Dragomir Radev, Youssef Billawala, and Karolina Buchner. Extractive summarization under strict length constraints. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, Paris, France, may 2016.

[32] Kavita Ganesan, ChengXiang Zhai, and Jiawei Han. Opinosis: a graph-based approach to abstractive summarization of highly redundant opinions. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 340–348, 2010.

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

[34] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Prompter: Turning the IDE into a self-confident programming assistant. *Empirical Software Engineering*, 21(5):2190–2231.

[35] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *Proc. 36th International Conference on Software Engineering*, page 10, 2014.

[36] Stack Overflow. *Parsing to class with GSON*. `https://stackoverflow.com/questions/47134527`, 2017. Last accessed on 6 November 2017.

[37] StackOverflow. *How do I ask a good question?* `https://stackoverflow.com/help/how-to-ask`, 2017.

[38] Annie T. T. Ying and Martin P. Robillard. Selection and presentation practices for code example summarization. In *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 460–471, 2014.

[39] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75, 2005.

[40] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for Java classes. In *Proceedings of the 21st IEEE International Conference on Program Comprehension*, pages 23–32, 2013.

[41] Walid Maalej and Martin P. Robillard. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, 2013.

[42] Gias Uddin, Barthélémy Dagenais, and Martin P. Robillard. Temporal analysis of API usage concepts. In *Proc. 34th IEEE/ACM Intl. Conf. on Software Engineering*, pages 804–814, 2012.

[43] Apache. *HttpClient 3.1 Tutorial*. `http://hc.apache.org/httpclient-legacy/index.html`.

[44] Canada Post. *Developer Program*. `https://www.canadapost.ca/cpotools/apps/drc/home?execution=e1s1`, 2017.

[45] Barthélémy Dagenais. *Analysis and Recommendations for Developer Learning Resources*. PhD in Computer Sscience, McGill University, 2012.

[46] Gias Uddin, Olga Baysal, Latifa Guerrouj, and Foutse Khomh. Understanding how and why developers seek and analyze API-related opinions. *IEEE Transactions on Software Engineering*, page 27, 2017. Under review.

[47] Gias Uddin and Foutse Khomh. Automatic opinion mining from API reviews. *IEEE Transactions on Software Engineering*, page 34, 2017. Under review.

[48] Gias Uddin and Foutse Khomh. Automatic summarization of API reviews. In *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 159–170, 2017.

[49] Gias Uddin and Foutse Khomh. Opiner: A search and summarization engine for API reviews. In *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 978–983, 2017.

[50] Gias Uddin, Foutse Khomh, and Chanchal K Roy. Automatic summarization of crowd-sourced API usage scenarios. *IEEE Trans. Soft. Eng.*, page 35, 2017. Under Review.

[51] Marco Ortu, Bram Adams, Giuseppe Destefanis, Parastou Tourani, Michele Marchesi, and Roberto 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.

[52] Mika Mântylà, Bram Adams, Giuseppe Destefanis, Daniel Graziotin, and Marco 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.

[53] Emitza Guzman and Walid 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.

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

[55] Alessandro Murgia, Parastou Tourani, Bram Adams, and Marco 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.

[56] Nicole Novielli, Fabio Calefato, and Filippo 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.

[57] Thomas Zimmermann, Peter Weiβgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

[58] Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd IEEE/ACM International Conference on Software Engineering*, page 167–176, 2000.

[59] Barthélémy Dagenais and Martin 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.

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

[61] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. In *32nd International Conference on Software Engineering*, pages 375–384, 2010.

[62] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Soft. Eng.*, 32(12), 2006.

[63] M.B. Miles and A.M. Huberman. *Qualitative Data Analysis: An Expanded Sourcebook*. SAGE Publications, 1994.

[64] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[65] Deen Freelon. ReCal2: Reliability for 2 coders. `http://dfreelon.org/utils/recalfront/recal2/`, 2016.

[66] Gias Uddin, Olga Baysal, Latifa Guerrouj, and Foutse Khomh. *Understanding How and Why Developers Seek and Analyze API-related Opinions (Online Appendix)*. `https://github.com/giasuddin/SurveyAPIReview`, 1 December 2017 (last accessed).

[67] Christoph Treude, Fernando Figueira Filho, and Uirá Kulesza. Summarizing and measuring development activity. In *Proc. 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 625–636, 2015.

[68] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Soft. Eng.*, 30(9):721–734, 2002.

[69] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *Proc. 2013 International Conference on Social Computing*, pages 188–195, 2013.

[70] Bogdan Vasilescu. *How does SO calculate email hash?* `https://meta.stackexchange.com/questions/135104/how-does-so-calculate-email-hash`, 2012.

[71] Canada Telecommunications Commission. *Canadas Anti-Spam Legislation*. `http://crtc.gc.ca/eng/internet/anti.htm`, 2017.

[72] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. Improving developer participation rates in surveys. In *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE, 2013.

[73] Richard Nisbett and Timothy Wilson. The halo effect: Evidence for unconscious alteration of judgments. *Journal of Personality and Social Psychology*, 35:250–256, 4 1977.

[74] Phil Edwards, Ian Roberts, Mike Clarke, Carolyn DiGuiseppi, Sarah Pratap, Reinhard Wentz, and Irene Kwan. Increasing response rates to postal questionnaires: systematic review. *BMJ*, 324(1183):9, 5 2002.

[75] Jeannine M. James and Richard Bolstein. Large monetary incentives and their effect on mail survey response rates. *Public Opinion Quarterly*, 56, 12 1992.

[76] William A Scott. Reliability of content analysis: The case of nominal scale coding. *Public Opinion Quarterly*, 19(3):321–325, 1955.

[77] Klaus Krippendorff. Reliability in content analysis: Some common misconceptions and recommendations. *Human Communication Research*, 30:411–433, 2004.

[78] Mary Joyce. *Picking the best intercoder reliability statistic for your digital activism content analysis*. http://digital-activism.org/2013/05/picking-the-best-intercoder-reliability-statistic-for-your-digital-activism-content-analysis/, 2013.

[79] Abhishek Sharma, Yuan Tian, Agus Sulistya, David Lo, and Aiko Fallas Yamashita. Harnessing twitter to support serendipitous learning of developers. In *Proc. IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, page 5, 2017.

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

[81] Christian Bird and Thomas Zimmermann. Assessing the value of branches with what-if analysis. In *Proc. ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page Article 45, 2012.

[82] Stack Overflow. *Sunsetting Documentation*. `https://meta.stackoverflow.com/questions/354217/sunsetting-documentation`, 2017.

[83] Kevin Lerman, Sasha Blair-Goldensohn, and Ryan McDonald. Sentiment summarization: Evaluating and learning user preferences. In *12th Conference of the European Chapter of the Association for Computational Linguistics*, pages 514–522, 2009.

[84] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 70–79, 2012.

[85] Hazeline U. Asuncion, Arthur U. Asuncion, and Richard N. Tylor. Software traceability with topic modeling. In *Proc. 32nd Intl. Conf. Software Engineering*, pages 95–104, 2010.

[86] Latifa Guerrouj, David Bourque, and Peter C. Rigby. Leveraging informal documentation to summarize classes and methods in context. In *Proceedings of the 37th International Conference on Software Engineering (ICSE) - Volume 2*, pages 639–642, 2015.

[87] Christoph Treude and Martin P. Robillard. Augmenting API documentation with insights from stack overflow. In *Proc. 38th International Conference on Software Engineering*, pages 392–403, 2016.

[88] Audris Mockus and James D. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *Proc. 24th International Conference on Software Engineering*, pages 503–512, 2002.

[89] Benjamin V. Hanrahan, Gregorio Convertino, and Les Nelson. Modeling problem difficulty and expertise in stackoverflow. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work Companion*, CSCW '12, pages 91–94, 2012.

[90] Fatemeh Riahi, Zainab Zolaktaf, Mahdi Shafiei, and Evangelos Milios. Finding expert users in community question answering. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, pages 791–798, 2012.

[91] Barthélémy Dagenais and Martin 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.

[92] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurelio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vasquez, Gail C. Murphy, Laura Morenox David Shepherd, and Edmund Wong. On-demand developer documentation. In *Proc. 33rd IEEE International Conference on Software Maintenance and Evolution New Idea and Emerging Results*, page 5, 2017.

[93] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example? a study of programming Q&A in stackoverflow. In *Proc. 28th IEEE Intl. Conf. on Software Maintenance*, page 10, 2012.

[94] Apache. *Version Numbering*. http://apr.apache.org/versioning.html, 2013.

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

[96] Bo Zhou, Xin Xia, David Lo, Cong Tian, and Xinyu Wang. Towards more accurate content categorization of API discussions. In *22nd International Conference on Program Comprehension*, pages 95–105, 2014.

[97] Chunyang Chen, Sa Gao, and Zhenchang Xing. Mining analogical libraries in q&a discussions – incorporating relational and categorical knowledge into word embedding. In *23rd International Conference on Software Analysis, Evolution, and Reengineering*, pages 338–348, 2016.

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

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

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

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

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

[103] Anton Barua, Stephen W. Thomas, and Ahemd E. Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, pages 1–31, 2012.

[104] Michael Röder, Andreas Both, and Alexander 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.

[105] Radim Řehůřek and Petr 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.

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

[107] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *An Introduction to Information Retrieval*. Cambridge Uni Press, 2009.

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

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

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

[111] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16(1):321–357, January 2002.

[112] Sida Wang and Christopher 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.

[113] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram 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.

[114] Mike Thelwall, Kevan Buckley, Georgios Paltoglou, Di Cai, and Arvid Kappas. Sentiment in short strength detection informal text. *Journal of the American Society for Information Science and Technology*, 61(12):2544–2558, 2010.

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

[116] Theresa Wilson, Janyce Wiebe, and Paul 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.

[117] Arup 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.

[118] Andrea Esuli and Fabrizio 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.

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

[120] Vasileios Hatzivassiloglou and Janyce 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.

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

[122] Robbert Jongeling, Subhajit Datta, and Alexandar 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.

[123] Robbert Jongeling, Proshanta Sarkar, Subhajit Datta, and Alexander Serebrenik. On negative results when using sentiment analysis tools for software engineering research. *Journal Empirical Software Engineering*, 22(5):2543–2584, 2017.

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

[125] Fabio Calefato, Filippo Lanubile, Federico Maiorano, and Nicole Novielli. Sentiment polarity detection for software development. *Journal Empirical Software Engineering*, pages 2543–2584, 2017.

[126] Richard Socher, Alex Perelygin, Jean Wu, Christopher Manning, Andrew Ng, and Jason Chuang. Recursive models for semantic compositionality over a sentiment treebank. In *Proc. Conference on Empirical Methods in Natural Language Processing (EMNLP)*, page 12, 2013.

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

[128] Gias Uddin and Foutse Khomh. *Automatic Opinion Mining from API reviews (online appendix)*. `https://github.com/giasuddin/OpinionValueTSE`, 1 Dec 2017 (last accessed).

[129] Gias Uddin and Martin P. Robillard. Resolving API mentions in informal documents. Technical report, Sept 2017.

[130] Stack Overflow. *JSON and XML comparison*. `https://stackoverflow.com/questions/4862310`, 2017.

[131] Stack Overflow. *A better Java JSON library*. `http://stackoverflow.com/questions/338586/`, 2010. (last accessed in 2014).

[132] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: Finding relevant functions and their usages. In *Proc. 33rd Intl. Conf. on Software Engineering*, pages 111–120, 2011.

[133] NetworkX developers. *NetworkX: Software for complex networks*. `https://networkx.github.io/`, 2017.

[134] Ivan Titov and Ryan McDonald. Modeling online reviews with multi-grain topic models. In *In Proceedings of the 17th international conference on World Wide Web*, pages 111–120, 2008.

[135] Ana-Maria Popescu and Oren Etzioni. Extracting product features and opinions from reviews. In *In Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 339–346, 2005.

[136] Yue Lu, ChengXiang Zhai, and Neel Sundaresan. Rated aspect summarization of short comments. In *In Proceedings of the 18th international conference on World wide web*, pages 131–140, 2009.

[137] Qiaozhu Mei, Xu Ling, Matthew Wondra, Hang Su, and ChengXiang Zhai. Topic sentiment mixture: modeling facets and opinions in weblogs. In *In Proceedings of the 18th international conference on World wide web*, pages 171–180, 2007.

[138] Lun-Wei Ku, Li-Ying Lee, and Hsin-Hsi Chen. Opinion extraction, summarization and tracking in news and blog corpora. In *Proceedings of AAAI-2006 Spring Symposium on Computational Approaches to Analyzing Weblogs*, page 8, 2006.

[139] R. Arun, V. Suresh, C. E. Veni Madhavan, and M. N. Narasimha Murthy. On finding the natural number of topics with latent dirichlet allocation: Some observations. In *Proceedings of Conference on Knowledge Discovery and Data Mining*, pages 391–402, 2010.

[140] Adrien Guille and Pavel Soriano. Tom: A library for topic modeling and browsing. In *Proceedings of Actes de la conference francaise sur Extraction et la Gestion des Connaissances*, pages 451–456, 2016.

[141] Gunes Erkan and Dragomir R. Radev. Lexrank: graph-based lexical centrality as salience in text summarization. *Journal of Artificial Intelligence Research*, 22(1):457–479, 2004.

[142] Mahak Gambhir and Vishal Gupta. Recent automatic text summarization techniques: a survey. *Journal of Artificial Intelligence Review*, 47(1):1–66, January 2017.

[143] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

[144] Steve Mulder and Ziv Yaar. *The User Is Always Right: A Practical Guide to Creating and Using Personas for the Web*. New Riders, 1st edition, 2006.

[145] Tanguy Ortolo. Json license considered harmful. `https://tanguy.ortolo.eu/blog/article46/json-license`, 2012.

[146] Gias Uddin and Foutse Khomh. *Automatic Summarization of API reviews (online appendix)*. `https://github.com/giasuddin/OpinerEVALAppendix`, 1 December 2017 (last accessed).

[147] David Lo and Shahar Maoz. Mining hierarchical scenario-based specifications. In *Proc. IEEE/ACM international conference on Automated software engineering*, pages 359–396, 2009.

[148] David Lo, Siau-Cheng Khoo, and Chao Liu. Efficient mining of iterative patterns for software specification discovery. In *Proc. 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 460–469, 2007.

[149] Dirk Fahland, David Lo, and Shahar Maoz. Mining branching-time scenarios. In *Proc. IEEE/ACM international conference on Automated software engineering*, pages 443–453, 2013.

[150] Hossein Safyallah and Kamran Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Proc. 14th IEEE International Conference on Program Comprehension*, pages 84–88, 2006.

[151] Brian Fox. *Now Available: Central download statistics for OSS projects*. `http://blog.sonatype.com/2010/12/now-available-central-download-statistics-for-oss-projects/`, 2017.

[152] Maven Central. *API Guide*. `https://search.maven.org/#api`, 2017.

[153] StackOverflow. *How to parse JSON*. `http://stackoverflow.com/q/2591098/`, 2015.

[154] StackOverflow. *How to tell Jackson to ignore a field during serialization if its value is null?* `http://stackoverflow.com/q/11757487/`, 2015.

[155] StackOverflow. *Jackson with JSON: Unrecognized field, not marked as ignorable*. `http://stackoverflow.com/q/4486787/`, 2010.

[156] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 1st edition, 2007.

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

[158] Stack Overflow. *Name/value pair loop of JSON Object with Java & JSNI*. `http://stackoverflow.com/questions/7141650/`, 2010.

[159] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[160] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfl Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. 7th International Conference on Database Theory*, pages 398–416, 1999.

[161] Christian Borgelt. Frequent item set mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(6):429–445, 2012.

[162] Nicolas Bettenburg and Andrew Begel. Deciphering the story of software development through frequent pattern mining. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1197–1200. IEEE Press, 2013.

[163] Chanchal K Roy and James R Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proc. 16th IEEE International Conference on Program Comprehension*, pages 172–181, 2008.

[164] Jeffrey Svajlenko and Chanchal K. Roy. Evaluating clone detection tools with bigclonebench. In *Proc. 31st IEEE International Conference on Software Maintenance and Evolution*, pages 131–140, 2015.

[165] Joseph L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

[166] Klaus Krippendorff. *Content Analysis: An Introduction to Its Methodology*. Sage Publications, 2nd edition, 2003.

[167] StackOverflow. *Jackson Polymorphic Deserialisation (where the class depends on the JSON key)*. `http://stackoverflow.com/q/13715753`, 2010.

[168] *Automatic Summarization of Crowd-Sourced API Usage Scenarios (Online Appendix)*. `https://github.com/opario/OpinerUsageSummTse`, 4 Sept 2017 (last accessed).

[169] Stack Overflow. *Converting JSON to Hashmap<String, POJO> using GWT*. `https://stackoverflow.com/questions/20374351`, 2017.

[170] Douglas D. Dunlop and Victor R. Basili. A comparative analysis of functional correctness. *ACM Computing Surveys*, 14(2):229–244, 1982.

[171] S. G. Hart and L. E. Stavenland. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. pages 139–183, 1988.

[172] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[173] Joe Walnes. *Xstream*. `http://x-stream.github.io/`, 2017.

[174] Pivotal Software. *Spring Framework*. `https://spring.io/`, 2017.

[175] StackOverflow. *Jackson Vs. GSON*. `http://stackoverflow.com/q/2378402/`, 2013.

[176] fryan. *Language Framework Popularity: A Look at Java, June 2017*. `http://redmonk.com/fryan/2017/06/22/language-framework-popularity-a-look-at-java-june-2017/`, 2017.

[177] StackOverflow. *XStream or Simple*. `http://stackoverflow.com/q/1558087/`, 2017.

[178] John W. Tukey. *Exploratory Data Analysis*. Pearson, 1st edition, 1977.

[179] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman and Hall, 5 edition, 2011.

[180] Scipy. *Mann Whitney U Test*. `https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.stats.mannwhitneyu.html`, 2017.

[181] StackOverflow. *Separate JSON string into variable in android*. `http://stackoverflow.com/q/18599591/`, 2013.

[182] Quora. *Why is learning Spring Framework so hard?* `https://www.quora.com/Why-is-learning-Spring-Framework-so-hard`, 2017.

[183] Manos Nikolaidis. *Jackson without annotations*. `https://manosnikolaidis.wordpress.com/2015/08/25/jackson-without-annotations/`, 2017.

[184] StackOverflow. *Jackson Vs. GSON*. `http://stackoverflow.com/q/2378402/`, 2013.

[185] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, page 28, 2012.

[186] M. Abbes, F. Khomh, Y. G. Gueheneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 181–190, 2011.

[187] Ian Cai. *Framework Documentation: How to document object-oriented frameworks. An Empirical Study*. PhD in Computer Sscience, University of Illinois at Urbana-Champaign, 2000.

[188] Mary Beth Rosson, John M. Carrol, and Rachel K.E. Bellamy. Smalltalk scaffolding: a case study of minimalist instruction. In *Proc. ACM SIGCHI Conf. on Human Factors in Computing Systems*, pages 423–430, 1990.

[189] Hans Van Der Maij. A critical assessment of the minimalist approach to documentation. In *Proc. 10th ACM SIGDOC International Conference on Systems Documentation*, pages 7–17, 1992.

[190] Forrest Shull, Filippo Lanubile, and Victor R. Basili. Investigating reading techniques for object-oriented framework learning. *IEEE Transactions on Software Engineering*, 26(11):1101–1118, 2000.

[191] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *Proc. ACM Symposium on Document Engineering*, pages 26–33, 2002.

[192] Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. What programmers really want: Results of a needs assessment for SDK documentation. In *Proc. 20th Annual International Conference on Computer Documentation*, pages 133–141, 2002.

[193] Yuan Tian, Pavneet Singh Kochhar, and David Lo. An exploratory study of functionality and learning resources of web apis on programmableweb. In *Proc. 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 202–207, 2017.

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

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

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

[197] Guibin Chen, Chunyang Chen, Zhenchang Xing, and Bowen 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.

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

[199] Alberto Bacchelli, Tommaso Dal Sasso, Marco D'Ambros, and Michele Lanza. Content classification of development emails. In *Proc. 34th IEEE/ACM International Conference on Software Engineering*, pages 375–385, 2012.

[200] Chandan R. Rupakheti and Daqing Hou. Evaluating forum discussions to inform the design of an API critic. In *20th International Conference on Program Comprehension*, pages 53–62, 2012.

[201] Ferdian Thung, Xuan-Bach D. Le, and David Lo. Active semi-supervised defect categorization. In *Proc. IEEE 23rd International Conference on Program Comprehension*, pages 60–70, 2015.

[202] Abhishek Sharma, Yuan Tian, and David Lo. Nirmal: Automatic identification of software relevant tweets leveraging language model. In *Proc. 22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 449–458, 2015.

[203] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[204] Marco Lormans and Arie van Deursen. Can lsi help reconstructing requirements traceability in design and test? In *Proc. 10th European Conference on Software Maintenance and Reengineering*, pages 47–56, 2006.

[205] Olga Baysal and Andrew J. Malton. Correlating social interactions to release history during software evolution. In *Proc. Fourth International Workshop on Mining Software Repositories*, page 7, 2007.

[206] Denys Poshyvanyk and Andrian Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proc. 15th Intl. Conf. Program Comprehension*, pages 37–48, 2007.

[207] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[208] Andrian Marcus and Jonathan 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.

[209] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

[210] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: recovering links between bugs and changes. In *Proc. 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 15–25, 2011.

[211] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Multi-layered approach for recovering links between bug reports and fixes. In *Proc. ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[212] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

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

[214] Alberto Bacchelli, Michele Lanza, and Vitezslav 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.

[215] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele 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.

[216] 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.

[217] David Kavaler, Daryl Posnett, Clint Gibler, Hao Chen, Premkumar Devanbu, and Vladimir 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.

[218] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys 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.

[219] 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.

[220] Gias Uddin, Barthélémy Dagenais, and Martin P. Robillard. Analyzing temporal API usage patterns. In *Proc. 26th IEEE/ACM Intl. Conf. on Automated Software Engineering*, pages 456–459, 2011.

[221] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. 30th Intl. Conf. Soft. Eng.*, 2008.

[222] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for Java programs. In *Proc. 23rd Conf. on Object-Oriented Prog. Systems, Languages and Applications*, pages 313–328, 2008.

[223] Apache. *HttpClient 3.1 Tutorial*. `http://hc.apache.org/httpclient-legacy/index.html`.

[224] David Lo and Shahar Maoz. Scenario-based and value-based specification mining: better together. In *Proc. IEEE/ACM international conference on Automated software engineering*, pages 387–396, 2010.

[225] Amy Beth Warriner, Victor Kuperman, and Marc Brysbaert. Norms of valence, arousal, and dominance for 13,915 english lemmas. *Behavior Research Methods*, 45(4):1191–1207, 2013.

[226] Daniel Pletea, Bogdan Vasilescu, and Alexander 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.

[227] Emitza Guzman, David Azócar, and Yang 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.

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

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

[230] Gabriel Murray Sarah Rastkar, Gail C. Murphy. Automatic summarization of bug reports. *IEEE Trans. Software Eng*, 40(4):366–380, 2014.

[231] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the Eclipse IDE? *IEEE Softawre*, 23(4):76–83.

[232] John Anvik and Gail C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Trans. Softw. Eng. Methodol*, 20(3):952–970.

[233] Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 21st IEEE International Conference on Program Comprehension*, pages 279–290, 2014.

[234] Jonathan Sillito and Gail C. Murphy Kris De Volder. Asking and answering questions during a program change task. *Journal of IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[235] Susan Elliott Sim, Charles L A Clarke, and Richard Craig Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings of the 6th International Workshop on Program Comprehension*, page 180, 1998.

[236] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proc. 25th IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

[237] L. Moreno, J. Aponte, G. Sridhara, Marcus A., L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC'13)*, pages 23–32, 2013.

[238] Gail C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. University of Washington, 1996.

[239] R. Ian Bull Peter C. Rigby Margaret-Anne D. Storey, Li-Te Cheng. Shared waypoints and social tagging to support collaboration in software development. In *CSCW*, pages 195–198.

[240] A. Marcus S. Haiduc, J. Aponte. Supporting program comprehension with source code summarization. In *In Proceedings of the 32nd International Conference on Software Engineering*, pages 223–226, 2010.

[241] Paige Rodeghero, Collin McMillan, Collin McMillan, Nigel Bosch, and Sidney D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proc. 36th International Conference on Software Engineering*, pages 390–401, 2014.

[242] Andrea Di Sorbo, Sebastiano Panichella, Carol V. Alexandru, Junji Shimagaki, Corrado A. Visaggio, Gerardo Canfora, and Harald C. Gall. What would users change in my app? summarizing app reviews for recommending software changes. In *IEEE/ACM 39th International Conference on Software Engineering Companion*, pages 499–510, 2016.

[243] Andrea Di Sorbo, Sebastiano Panichella, Carol V. Alexandru, Corrado A. Visaggio, and Gerardo Canfora. Surf: Summarizer of user reviews feedback. In *IEEE/ACM 39th International Conference on Software Engineering Companion*, pages 53–62, 2017.

[244] Laura V. Galvis Carreno and Kristina Winbladh. Analysis of user comments: an approach for software requirements evolution. In *Proceedings of the 35th International Conference on Software Engineering*, pages 582 – 591, 2013.

[245] Ning Chen, Jialiu Lin, Steven C. H. Hoi, Xiaokui Xiao, and Boshen 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.

[246] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. Answerbot: automated generation of answer summary to developers' technical questions. In *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 706–716, 2017.

[247] Meital Zilberstein and Eran Yahav. Leveraging a corpus of natural language descriptions for program similarity. In *Proc. 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 197–211, 2016.

[248] Christoph Treude, Martin P. Robillard, and Barthélémy Dagenais. Extracting development tasks to navigate software documentation. *IEEE Transactions on Software Engineering*, 41(6):565–581, 2015.

[249] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. AnswerBot: automated generation of answer summary to developers technical questions. In *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 706–716, 2017.

[250] Dictionary.com. Opinion. `http://www.dictionary.com/browse/opinion`, 2016.

[251] Danny Dig and Ralph Johnson. How do APIs evolve? a story of refactoring. *J. Softw. Maint. Evol.*, 18(2):83–107, mar 2006.