Automatic Summarization of API Reviews

Gias Uddin
School of Computer Science
McGill University, Montréal, QC, Canada
gias@cs.mcgill.ca

Foutse Khomh
SWAT Lab
Polytechnique Montréal, QC, Canada
foutse.khomh@polymtl.ca

***Abstract*—With the proliferation of online developer forums as informal documentation, developers often share their opinions about the APIs they use. However, given the plethora of opinions available for an API in various online developer forums, it can be challenging for a developer to make informed decisions about the APIs. While automatic summarization of opinions have been explored for other domains (e.g., cameras, cars), we found little research that investigates the benefits of summaries of public API reviews. In this paper, we present two algorithms (statistical and aspect-based) to summarize opinions about APIs. To investigate the usefulness of the techniques, we developed, Opiner, an online opinion summarization engine that presents summaries of opinions using both our proposed techniques and existing six off-the-shelf techniques. We investigated the usefulness of Opiner using two case studies, both involving professional software engineers. We found that developers were interested to use our proposed summaries much more frequently than other summaries (daily vs once a year) and that while combined with Stack Overflow, Opiner helped developers to make the right decision with more accuracy and confidence and in less time.**

***Index Terms*—Opinion mining, API informal documentation, opinion summaries, study, summary quality.**

I. Introduction

APIs (Application Programming Interfaces) offer interfaces to reusable software components. Modern-day rapid software development is often facilitated by the plethora of open-source APIs available for any given development task. The online development portal GitHub [1] now hosts more than 38 million public repositories. We can observe a radical increase from the 2.2 million active repositories hosted in GitHub in 2014.

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. In fact, we observed that more than 66% of Stack Overflow posts that are tagged "Java" and "Json" contain at least one positive or negative sentiment. Most of these (46%) posts also do not contain any code examples. The number of numerous APIs available and the sheer volume of opinions about any given API scattered across many different posts though pose a significant challenge to gain quick and digestible insights.

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 [2]. 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.

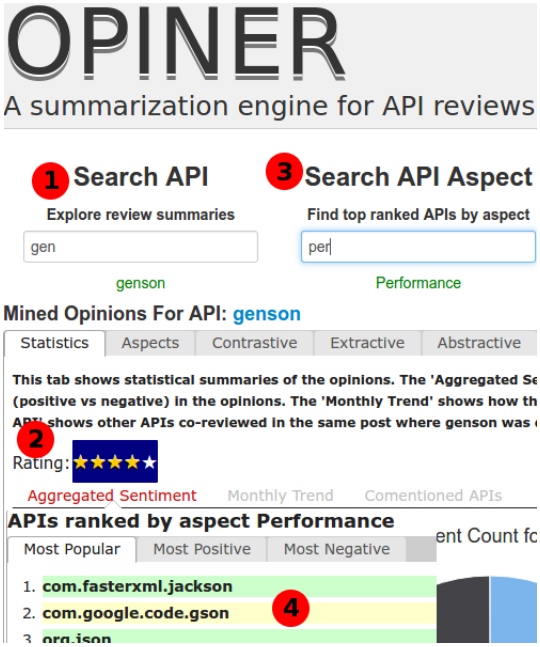

Fig. 1. Screenshots of Opiner API review search engine

We developed an online API review summarization engine, Opiner that given as input all the opinionated sentences about an API produces summaries of the reviews using our two proposed techniques: statistical and aspect-based summarization. In Figure 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 Opiner based on two research questions:

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

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.

This paper makes the following contributions:

1) **Opiner:** We present Opiner, our online API opinion summarization and search engine where developers can search for opinions about APIs.
2) **Evaluation:** We evaluated the effectiveness of the summaries and Opiner using two case studies.

II. Automatic Summarization of API Reviews

We investigated opinion summarization algorithms from the following major categories [2], [3]:

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 present algorithms to produce aspect-based and statistical summaries of API reviews (see Sections II-B,II-C). We leveraged off-the-shelf algorithms to produce extractive, abstractive, and topic-based summaries and implemented the algorithm proposed by Kim and Zhai [4] to produce contrastive summaries (see Section II-D).

In Section II-A, we introduce the API review dataset that we used to produce the summaries. We then leverage the dataset to describe our summarization techniques.

A. 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: (1) **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. (2) **Diverse Opinions.** We observed diverse opinions associated to the competing opinions from the different stakeholders (both API users and authors). (3) **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.

TABLE I
Statistics of the dataset (A = Answers, C = Comments)

Threads	Posts	A	C	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

In Table I we show descriptive statistics of the dataset. There were 22,733 posts from 3048 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. 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 [5]. From this corpus, we identified all of the Java APIs that were mentioned in the posts. Our API database consists of the Java official APIs and the open source Java APIs listed in the two software portals Ohloh [6] and Maven central [7].[1] 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 in the absence of any guidelines available to consider otherwise. In total, our API database contains 62,444 distinct Java APIs. All of the APIs (11,576) hosted in Maven central are for Java. From Ohloh, we only included the Java APIs (50,863) out of the total crawled projects (712,663). We considered a project in Ohloh as a Java API if its main programming language was Java.

We collected the opinionated sentences about APIs using a technique previously developed by Uddin an Khomh [8]. 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 [9] and the Sentiment Orientation (SO) algorithm [10].
3) **Detection of API names and hyperlinks** in the forum texts and the **association of APIs to opinionated sentences** based on a set of heuristics.

In Table II, we present summary statistics of the opinionated sentences detected in the dataset. Overall 415 distinct APIs were found. While the average number of opinionated sentences per API was 37.66, it was 2066 for the top five most

[1] We crawled Maven in March 2014 and Ohloh in Dec 2013.

TABLE II
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

reviewed APIs. 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.

B. 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"* [3]. In Figure 2, ② shows such a summary for API `jackson`. We propose the following statistical summaries for API reviews:

1) **Star rating.** Provides an overall sentiment representation towards an API 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 visualize other APIs that were mentioned in the same post where an API was reviewed. Such insights can help find competing APIs.

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 2, ① shows the calcuated five-star rating for `Jackson`.

Our approach is similar to Blair-Goldensohn et al. [11] 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.

2) Sentiment Trend: We produce monthly aggregated sentiments for each API by grouping total positive and negative opinions towards the API. In ③ of Figure 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.

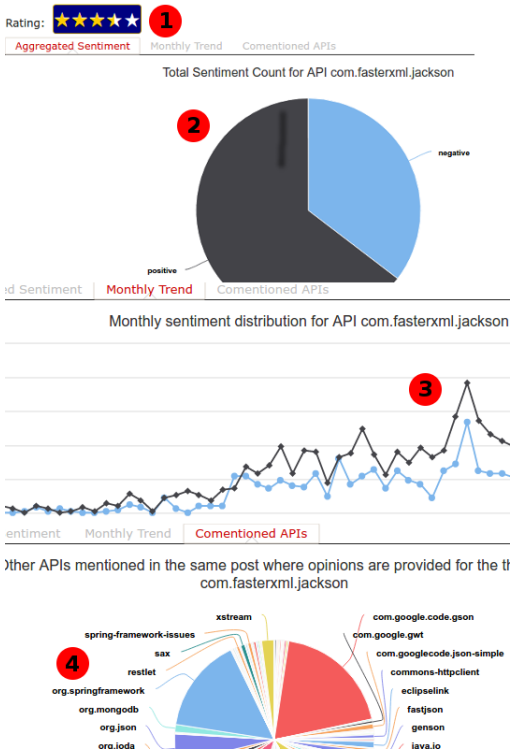

Fig. 2. Statistical summarization of opinions for API `Jackson`.

3) Co-Reviewed APIs: In Figure 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 identify the corresponding posts where the sentence was found. (2) We detect 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 group the positive and negative opinions in 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.

C. Aspect-Based Summaries of API Reviews

Aspect-based summarization [11], [12] 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 be different depending on the domains. Thus the detection of domain specific aspects is the first step towards aspect-based opinion summarization [13]. 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 leverage the fact that similar to other domains [11], 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 [11] and we present techniques to automatically detect those in Section II-C1.
2) **Dynamic aspect detection:** We observe that certain aspects can be more common in an API or a group of APIs (e.g., 'object conversion' for JSON parsing APIs vs 'applet size' for APIs to design user interfaces). The detection of these *dynamic* aspects requires techniques different from the detection of static aspects (see Section II-C2)
3) **Summarizing opinions for each aspect:** For each API, we produce a consolidated view by grouping the reviews under the aspects and by presenting different summarized views of the reviews under each aspect (see Section II-C3).

1) Static Aspect Detection: In a previous study, Uddin et al. [8] surveyed software developers and found that developers prefer to see opinions about the following API aspects in the forum posts: (1) Performance: How well does the API perform? (2) Usability: How usable is the API? (3) Security: How secure is the API? (4) Documentation: How good is the documentation? (5) Compatibility: Does the usage depends on other API? (6) Portability: Can the API be used in different platforms? (7) Community: How is the support around the community? (8) Legal: What are licensing requirements? (9) Bug: Is the API buggy? (10) Only Sentiment: Opinions without specifying any aspect. (11) Others: Opinions about other aspects. We developed a supervised classifier to detect each aspect to account for the fact that more than one aspect can be discussed in one opinionated sentence. We used four performance measures to assess the performance of the classifiers: 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.

We report the performance of our aspect detection component using a dataset previously labeled by Uddin and Khomh [8]. The benchmark consisted of 4,594 manually labeled sentences from 1,338 Stack Overflow posts. The threads were selected from 18 tags representing nine distinct domains (two tags for each domain).

Candidate Classifiers. 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.[2]

TABLE III
PERFORMANCE OF STATIC ASPECT DETECTORS

		Precision		Recall		F1 Score		Accuracy	
Aspect	**N**	**A**	**S**	**A**	**S**	**A**	**S**	**A**	**S**
Performance	B	0.78	0.10	0.46	0.16	0.56	0.13	0.95	0.01
Usability	B	0.53	0.05	0.75	0.10	0.62	0.06	0.71	0.04
Security	U	0.78	0.27	0.58	0.17	0.60	0.16	0.97	0.06
Community	U	0.40	0.32	0.24	0.22	0.26	0.20	0.97	0.01
Compatibility	T	0.50	0.50	0.08	0.09	0.13	0.14	0.98	0.00
Portability	U	0.63	0.21	0.63	0.22	0.61	0.19	0.99	0.01
Documentation	B	0.59	0.18	0.43	0.17	0.49	0.18	0.95	0.02
Bug	U	0.57	0.16	0.50	0.16	0.51	0.12	0.96	0.01
Legal	U	0.70	0.28	0.46	0.18	0.52	0.19	0.99	0.00
OnlySentiment	B	0.61	0.14	0.43	0.14	0.50	0.14	0.94	0.02
Others	U	0.61	0.07	0.67	0.06	0.64	0.06	0.71	0.05

N = Ngram, U = Unigram, B = Bigram, T = Trigram, A = Average, S = Stdev

We applied the SVM-based classification steps as recommended by Hsu et al. [15] 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.[3] As recommended by Chawla [16], 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.

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 [17]. (3) As recommended by Hsu et al. [15], 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 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 nine times - 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.

[2]We used the SGDClassifier of Scikit [14]

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

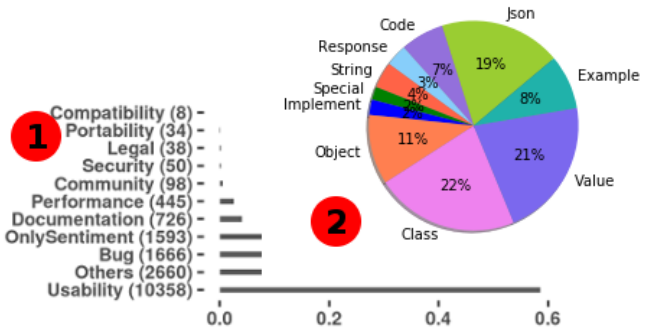

Fig. 3. The distribution of dynamic aspects in the dataset

In Table III, 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. 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.

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'. 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 III, 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. In the future we will investigate this issue with different settings, e.g., using more labeled data.

In Figure 3, (1) shows the distribution of the static aspects in our dataset. Some aspects are 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', because 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 'Others'. The sentences belonging to the 'Others' category contain opinions about API aspects not covered by the static aspects. We apply our dynamic aspect detection on the opinions labeled as 'Others'.

2) Dynamic Aspect Detection: We detect dynamic aspects on the 15% of the sentences in our dataset that were labeled *exclusively* as 'Others'. Our dynamic aspect detection algorithm is adopted from similar techniques used in other domains, e.g., electronic products [10], and local service reviews (e.g., hotels/restaurants [11]).

Our approach consists of the following three steps:

1) **Keyword Identification**: We find *representative* keywords and phrases in the sentences labeled as 'Others'.
2) **Pruning**: We discard keywords/phrases based on two filters. The remaining items are considered as dynamic aspects.
3) **Assignment**: We assign each sentence to a dynamic aspect.

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

(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. [10] and BlairGoldensohn et al. [11] who used frequent itemset mining, we use the keyphrase detection process used in the Textrank algorithm. Textrank uses the Google Page rank algorithm to find representative keyphrases in a given dataset.[4] The page rank algorithm has been successfully applied in software engineering research [18]. We detect the keyphrases as followed: (1) We tokenize the sentences (2) We remove the stopwords (3) We detect Parts of Speech (4) We discard any words that are not Nouns (5) We create 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. (7) From this ranking, we keep the top 5% words as keywords. In our future work, we will investigate the algorithm with different percentages of keywords. We then merge two or more keywords into one if they appeared together in the sentences.

(2) **Pruning:** Hu et al. [10] and BlairGoldensohn et al. [11] observed that, the frequent keywords by themselves can still be too many to produce a concise list of aspects. We thus apply two filters on the keywords to remove the keywords that are not widely found across different APIs. Both of the filters were originally proposed by Hu et al. [10] We then compute the TF-IDF score of the remaining keywords and phrases in all the sentences labeled as 'Others' and rank those based on score (i.e., a keyword with a higher score is more representative).

(3) **Assignment:** We assign each sentence in the 'Others' dataset to a keyword as identified in the previous step. If a sentence is not labeled as any of the keywords/phrases, we consider it as a 'general comment'. We label each such sentence as discussing about an API 'feature'. For a sentence labeled as more than one dynamic aspect, we assign it to the

[4]We used the Page rank implementation from Python networkx library

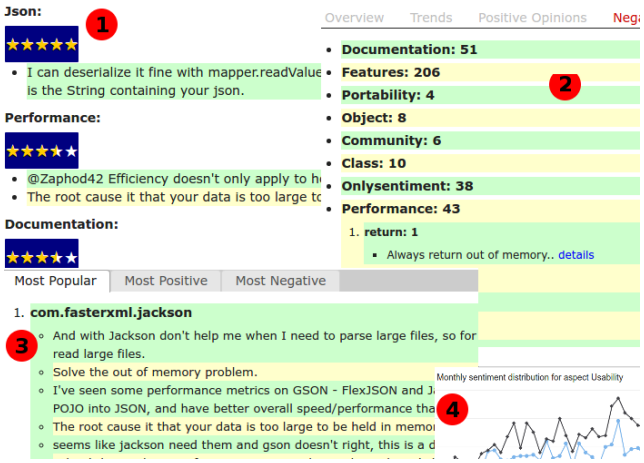

Fig. 4. Screenshot of the aspect-based summarizer

.

one with the highest TF-IDF score. In Figure 3, ② shows the distribution of the dynamic aspects (except the general comments) in the dataset.

3) Aspect-Based Summarizing of Opinions: We produce the following summaries based on the aspects: 1) Overview, 2) Nested aspect views, and 3) Comparative by aspect. We explain the techniques behind each summaries below:

(1) **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 4, 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 aspect for the API. Such trends can be useful to compare how its different aspects have been reviewed over time.

(2) **Nested 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 observe that even after this grouping, some of the aspects contain hundreds of opinions (e.g., Usability). To address this, we further categorize the opinions under each aspect into sub-categories. We denote those as nested-aspect. We detect the nested aspects as follows. (a) We collect all the Stack Overflow tags associated to the threads in the dataset. (b) For each sentence under a given aspect of an API, we label each opinion as a tag, if the opinion contains 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 covers the most number of opinions within the aspect for the API. (d) For an opinion that can not be labeled as any of the tags, we label it as 'general'. In Figure 4, the second circle(②) shows a screenshot of the nested views of the negative opinions of the API `Jackson`. By clicking the 'details' link the user is taken to the corresponding post in Stack Overflow where the opinion was provided.

(3) **Comparative By Aspect** While all the above three summarizations take 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 $\frac{P_A}{P_C}$, and negativity $\frac{N_A}{N_C}$. P_A is the total number of positive opinions about an API, and P_C is the total number of positive opinions about all APIs. In Figure 4, 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.

D. Summarization Algorithms Adopted For API Reviews

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

1) Topic-based Summarization: In a topic-based opinion summarization [19]–[23], 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 [19], [20], phrase level [21] and sentence level [22], [23]. For each API in our dataset, we summarize the reviews by applying LDA (Latent Dirichlet Allocation) [24] once for the positive opinions and once for the negative opinions. We apply standard practices, e.g., optimal number of topic detection (we use the technique proposed by Arun et al. [25]). To determine the representativeness of topics, we use the topic *coherence* measure as proposed by Röder et al. [26]. For each topic, we produce a description by taking the top ten words of the topic.

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 [4].

3) Extractive Summarization: With the extractive summarization techniques, a summary of the input documents is made by selecting *representative* sentences from the original documents. Extractive summaries are the most prevalent for text summarization across domains (e.g., news, blogs). For each API, we apply three extractive summarization algorithms: Luhn [27], Lexrank [28], and Textrank [29]. Luhn is the oldest summarization algorithms, while Texrank and Lexrank are among the most recent. Using each algorithm, we produce two summaries for each API, one for positives and one for negatives. For each API, we produce summaries containing 1% of the inputs or 10 opinionated sentences (whicever 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 [30].

4) Abstractive Summarization: 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. We produce abstractive summary for an API once for positive sentences and once for negative sentences using Opinosis [31], a domain-independent abstractive opinion summarization engine.

III. 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 [32], [33]. In this section, we describe the design and results of the study.

A. 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 preliminary survey of software developers of their preference to use summaries of API reviews in assisting development tasks [8].

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 proven effective both in the Academia and Industry [34]. 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.

B. 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 Mechnical turks and Freelancer.com have been gaining popularity to conduct studies in empirical software engineering research due to the availability of efficient, knowledgable 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.

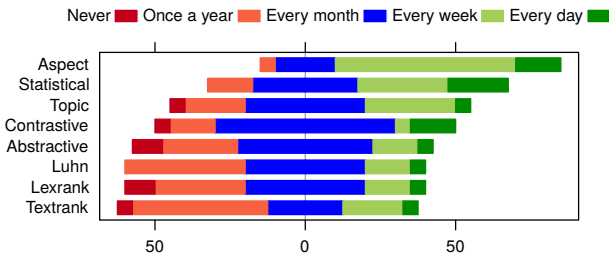

Fig. 5. Developers' preference of usage of summaries

TABLE IV
IMPACT OF THE SUMMARIES BASED ON THE SCENARIOS

Task	Option	S	A	T	C	B	L	P	U
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

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.

C. Results

In Table IV, we show the percentage of the ratings for each summary for the four tasks (Selection, Documentation, Presentation, Authoring). In Figure 5, 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 V, we show

TABLE V
RATIONALE BASED ON THE SCENARIOS (ONLY COMPLETELY AGREED AND AGREED STATISTICS ARE PRESENTED FOR BREVITY)

T	Criteria	Agree	S	A	T	C	B	L	P	U
S	Rightness	👍👍	**55**	**65**	25	15	5	10	20	25
E		👍	30	20	15	**55**	35	40	25	25
L	Relevant	👍👍	**65**	**60**	25	10	5	15	25	10
E		👍	25	20	20	**55**	40	30	25	40
C	Usable	👍👍	40	**60**	25	30	10	15	25	10
T		👍	45	20	30	30	30	35	35	45
D	Complete	👍👍	**55**	**60**	35	15	15	20	30	30
O		👍	15	20	20	45	35	30	20	20
C	Readable	👍👍	**60**	**65**	35	30	20	20	20	20
U		👍	20	30	35	35	45	35	40	35
P	Concise	👍👍	**60**	**65**	30	15	10	10	25	15
R		👍	25	20	25	**50**	40	35	20	35
E	Recency	👍👍	**65**	**70**	35	15	25	20	25	25
S		👍	15	10	20	30	20	30	25	20
A	Diversity	👍👍	**60**	**75**	25	15	15	10	25	15
W		👍	30	10	40	**60**	40	55	50	60
A	Recency	👍👍	**55**	**70**	30	10	15	10	15	10
R		👍	15	10	25	40	45	40	30	30
A	Highlight	👍👍	**50**	**70**	25	15	10	15	25	15
U		👍	25	15	35	**50**	**50**	**50**	35	45
T	Theme	👍👍	25	**65**	20	20	10	15	20	15
H		👍	**50**	20	25	35	40	30	25	30

👍👍= Completely Agree, U = Luhn, L = Lexrank,
S = Statistics, A = Aspect, T = Topic, B = Abstract, P = Textrank

the percentages of the ratings for each criteria under each development task. We discuss the rationales below.

1) Selection: 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."*

2) Documentation: 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."*

3) Presentation: 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"*.

4) Awareness: 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 developers who are not involved in daily programming activities.

5) Authoring: 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."*

IV. Effectiveness Analysis of Opiner (RQ3)

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.

A. 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.

(T1) The participants were asked to make a selection out of two APIs (GSON and org.json) based on two criteria: a) Usability, and b) Licensing usage. The right answer was GSON. The licensing agreement of the API org.json is generally considered not safe for Industrial usage [35].

(T2) The participants were asked to make a selection out of two APIs (Jackson and json-lib) based on two criteria:a) Performance and b) 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: (1) **Selection**: Their choice of API (2) **Confidence**: How confident they were while making the selection. We used a five-point scale: Fully confident (value 5) - Fully unsure (1). (3) **Rationale**: The reason of their selection in one or more sentences. 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:

TABLE VI
Progression of learning from Stack Overflow to Opiner

T	Tool	Correctness	Conversion	Confidence	Time
1	SO	20.0%	–	4.3	12.3
	SO + Opiner	100%	100%	4.5	7.5
2	SO	66.7%	–	2.7	18.6
	SO + Opiner	100%	100%	4.6	6.8

- **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.

B. 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.

C. 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.

D. Results

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 VI, 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.

Industrial Adoption. All developers answered to the three questions. We summarize major themes below.

1) 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."*

2) 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."*

3) How would you improve Opiner: Moving forward, the participants offered a number of features that would like to see in Opiner:1) **Enhanced contrastive**: *"Focus on contrastive view, ...instead of its technical qualities (e.g. why use one vs another)."* 2) **Documentation**: *"A brief description regarding what 'Aspects', 'Contrastive', 'Extractive', 'Abstractive' mean would be helpful"* 3) **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."* 4) **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."*

V. 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 [8]. 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 [36]. 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.

VI. Related Work

Related work can broadly be divided into (1) Analysis of developer forums; and (2) Summarization of software artifacts.

Analysis of Developer Forums. Online developer forums have been studied extensively, to find dominant discussion topics [37], [38], to analyze the quality of posts [39]–[44], to analyze developer profiles [45], [46], or to determine the influence of badges in Stack Overflow [47]. Several tools have been developed, such as autocomment assistance [48], collaborative problem solving [49], [50], and tag prediction [51].

Summarization in Software Engineering. Natural language summaries have been investigated for software documentation [52]–[55] and source code [56]–[59]. Murphy [60] proposed two techniques to produce structural summary of source code. Storey et al. [61] analyzed the impact of tags and annotations in source code. The summarization of source code element names (types, methods) has been investigated [56], [57], [62]. A more recent work proposed automatic documentation via the source code summarization of method context [58]. The selection and presentation of source code summaries have been explored through developer interviews [63] and eye tracking experiment [64]. Our findings confirm that developers also require API review summaries.

VII. 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 paper, we have presented opiner, our summarization engine for API reviews. Using Opiner, developrs 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. In our future work, we plan to extend Opiner to integrate opinions from multiple forums and to integrate continuous learning modules into the summaries to learn and improve from developers' feedbacks.

REFERENCES

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

[2] B. Liu, *Sentiment Analysis and Subjectivity*, 2nd ed. Boca Raton, FL: CRC Press, Taylor and Francis Group, 2010.

[3] H. D. Kim, K. Ganesan, P. Sondhi, and C. Zhai, "Comprehensive review of opinion summarization," University of Illinois at Urbana-Champaign, Tech. Rep., 2011.

[4] H. D. Kim and C. Zhai, "Generating comparative summaries of contradictory opinions in text," in *Proceedings of the 18th ACM conference on Information and knowledge management*, 2009, pp. 385–394.

[5] StackOverflow, *A better Java JSON library?*, http://stackoverflow.com/q/338586/, 2008.

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

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

[8] G. Uddin and F. Khomh, "Mining aspects in API reviews," Tech. Rep., May 2017. [Online]. Available: http://swat.polymtl.ca/data/opinionvalue-technical-report.pdf

[9] M. Thelwall, K. Buckley, G. Paltoglou, D. Cai, and A. Kappas, "Sentiment in short strength detection informal text," *Journal of the American Society for Information Science and Technology*, vol. 61, no. 12, pp. 2544–2558, 2010.

[10] M. Hu and B. Liu, "Mining and summarizing customer reviews," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004, pp. 168–177.

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

[12] M. Hu and B. Liu, "Mining and summarizing customer reviews," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004, pp. 168–177.

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

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

[15] C. wei Hsu, C. chung Chang, and C. jen Lin, "A practical guide to support vector classification," p. 16, 2010.

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

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

[18] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usages," in *Proc. 33rd Intl. Conf. on Software Engineering*, 2011, pp. 111–120.

[19] I. Titov and R. McDonald, "Modeling online reviews with multi-grain topic models," in *In Proceedings of the 17th international conference on World Wide Web*, 2008, pp. 111–120.

[20] A.-M. Popescu and O. 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*, 2005, pp. 339–346.

[21] Y. Lu, C. Zhai, and N. Sundaresan, "Rated aspect summarization of short comments," in *In Proceedings of the 18th international conference on World wide web*, 2009, pp. 131–140.

[22] Q. Mei, X. Ling, M. Wondra, H. Su, and C. Zhai, "Topic sentiment mixture: modeling facets and opinions in weblogs," in *In Proceedings of the 18th international conference on World wide web*, 2007, pp. 171–180.

[23] L.-W. Ku, L.-Y. Lee, and H.-H. Chen, "Opinion extraction, summarization and tracking in news and blog corpora," in *Proceedings of AAAI-2006 Spring Symposium on Computational Approaches to Analyzing Weblogs*, 2006, p. 8.

[24] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, no. 4-5, pp. 993–1022, 2003.

[25] R. Arun, V. Suresh, C. E. V. Madhavan, and M. N. N. Murthy, "On Finding the Natural Number of Topics with Latent Dirichlet Allocation: Some Observations," in *Proceedings of Conference on Knowledge Discovery and Data Mining*, 2010, pp. 391–402.

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

[27] H. P. Luhn, "The automatic creation of literature abstracts," *IBM Journal of Research and Development*, vol. 2, no. 2, pp. 159–165, 1958.

[28] G. Erkan and D. R. Radev, "Lexrank: graph-based lexical centrality as salience in text summarization," *Journal of Artificial Intelligence Research*, vol. 22, no. 1, pp. 457–479, 2004.

[29] R. Mihalcea and P. Tarau, "Textrank: Bringing order into texts," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2004, pp. 404–411.

[30] Y. Mehdad, A. Stent, K. Thadani, D. Radev, Y. Billawala, and K. Buchner, "Extractive summarization under strict length constraints," in *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, N. C. C. Chair), K. Choukri, T. Declerck, S. Goggi, M. Grobelnik, B. Maegaard, J. Mariani, H. Mazo, A. Moreno, J. Odijk, and S. Piperidis, Eds. Paris, France: European Language Resources Association (ELRA), may 2016.

[31] K. Ganesan, C. Zhai, and J. Han, "Opinosis: a graph-based approach to abstractive summarization of highly redundant opinions," in *Proceedings of the 23rd International Conference on Computational Linguistics*, 2010, pp. 340–348.

[32] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, 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*, 2013, pp. 23–32.

[33] G. Sridhara, E. Hill, D. Muppaneni, L. 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*, 2010, pp. 43–52.

[34] S. Mulder and Z. Yaar, *The User Is Always Right: A Practical Guide to Creating and Using Personas for the Web*, 1st ed. New Riders, 2006.

[35] T. Ortolo, "Json license considered harmful," https://tanguy.ortolo.eu/blog/article46/json-license, 2012.

[36] *Automatic Summarization of API reviews*, https://github.com/anon-ase2017/opinereval, 12 May 2017 (last accessed).

[37] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," *Empirical Software Engineering*, pp. 1–31, 2012.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[52] G. M. Sarah Rastkar, Gail C. Murphy, "Automatic summarization of bug reports," *IEEE Trans. Software Eng*, vol. 40, no. 4, pp. 366–380, 2014.

[53] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Softawre*, vol. 23, no. 4, pp. 76–83.

[54] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE Trans. Soft. Eng.*, vol. 32, no. 12, 2006.

[55] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions." *ACM Trans. Softw. Eng. Methodol*, vol. 20, no. 3, pp. 952–970.

[56] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proc. 25th IEEE/ACM international conference on Automated software engineering*, 2010, pp. 43–52.

[57] L. Moreno, J. Aponte, G. Sridhara, M. 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)*, 2013, pp. 23–32.

[58] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 21st IEEE International Conference on Program Comprehension*, 2014, pp. 279–290.

[59] L. Guerrouj, D. Bourque, and P. 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*, 2015, pp. 639–642.

[60] G. Murphy, *Lightweight Structural Summarization as an Aid to Software Evolution*. University of Washington.

[61] M.-A. D. Storey, L.-T. Cheng, R. I. Bull, and P. C. Rigby, "Shared waypoints and social tagging to support collaboration in software development," in *In Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, 2006, pp. 195–198.

[62] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *In Proceedings of the 32nd International Conference on Software Engineering*, 2010, pp. 223–226.

[63] A. T. T. Ying and M. P. Robillard, "Selection and presentation practices for code example summarization," in *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 460–471.

[64] P. Rodeghero, C. McMillan, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proc. 36th International Conference on Software Engineering*, 2014, pp. 390–401.