Extracting Problematic API Features from Forum Discussions

Yingying Zhang
Department of Computer Science
Clarkson University
Potsdam, New York 13699
Email: yingzha@clarkson.edu

Daqing Hou
Department of Electrical and Computer Engineering
Clarkson University
Potsdam, New York 13699
Email: dhou@clarkson.edu

***Abstract*—Software engineering activities often produce large amounts of unstructured data. Useful information can be extracted from such data to facilitate software development activities, such as bug reports management and documentation provision. Online forums, in particular, contain extensive valuable information that can aid in software development. However, no work has been done to extract *problematic API features* from online forums. In this paper, we investigate ways to extract problematic API features that are discussed as a source of difficulty in each thread, using natural language processing and sentiment analysis techniques. Based on a preliminary manual analysis of the content of a discussion thread and a categorization of the role of each sentence therein, we decide to focus on a negative sentiment sentence and its close neighbors as a unit for extracting API features. We evaluate a set of candidate solutions by comparing tool-extracted problematic API design features with manually produced golden test data. Our best solution yields a precision of 89%. We have also investigated three potential applications for our feature extraction solution: (i) highlighting the negative sentence and its neighbors to help illustrate the main API feature; (ii) searching helpful online information using the extracted API feature as a query; (iii) summarizing the problematic features to reveal the "hot topics" in a forum.**

***Index Terms*—APIs; Online Forums; Design Feedback; Information Extraction; AWT/Swing.**

I. Introduction

Nowadays, software developers commonly use online forums to exchange information, ask questions, and seek help from each other. These forums are becoming rich repositories of valuable programming information that many experienced developers learn to revisit frequently. Information extraction techniques [13] have been investigated to extract significant information from these unstructured data to aid in such development activities as the provision of method descriptions [10] and the speed reading of bug reports [11], [9]. But three general problems still remain for API usage forums:

- *The difficulty of identifying API "hot topics."* Hot topics are problematic API features that are discussed frequently in a forum, thus representing the greatest concerns of API users. Knowing what are the hot topics for an API can be particularly valuable for the API support team as they can gain insights as to how to improve their API design, or to develop better tutorials. Given the large number of discussion threads, it is hard to tell what are the hot topics for a given API. Since it is not feasible to extract by hand API features discussed in each thread, an automated tool would make a significant difference.
- *The time-consuming nature of reading and understanding long threads.* Unfortunately, in a technical forum, a long thread tends to be common [1]. Due to the detail-oriented nature of technical discussions, multiple posts are often needed to help define or clarify a problem, provide background information, and illustrate and justify answers. Finding out ways to speed up the reading of such long threads has recently attracted interest in software engineering research [11], [9].
- *The challenge of finding relevant help in a timely manner.*

To help address these problems, in this paper, we explore an automated solution for extracting problematic API design features from forum threads. The key idea behind our proposed solution is that phrases extracted from a negative sentence and its neighboring sentences are likely to contain meaningful feature names. This is inspired by a preliminary manual analysis of a set of discussion threads, whereby we identify eight categories of sentences in a thread. In particular, we find that an API problem tends to be described in *negative sentences* using negative sentiment words and phrases, such as "invisible," "disappear," "unexpected," and "does not work." We also find that meaningful feature names are often located either directly in such negative sentences, or in their close neighbors. Therefore, we consider extracting API features from both the negative sentence and its neighbors. Through an empirical evaluation we confirm that the proposed negative sentences based solution is indeed more likely to contain problematic API features than several other alternative locations.

Utilizing sentiment analysis and natural language processing techniques, we have implemented our solution in a tool called *Haystack*. We have identified three subtree patterns in a sentence's parse tree that are likely to contain meaningful API feature names. Haystack uses an existing sentiment analysis tool to identify negative sentences, and a natural language parser to parse a sentence to identify the three desired subtree patterns contained therein.

[1]Our evaluation shows that on average, each thread contains 36 sentences (Section IV-A). To give the reader an idea how much text 36 sentences are equivalent to, there are about the same number of sentences in the first page of this paper. So it is justified to research how to speed up the reading.

The exploration of our feature extraction solution has been guided by considering three of its potential applications:

- Estimating the "hot topics" in a forum by summarizing the occurrence frequency of the identified API features,
- Highlighting the negative sentiment sentence and its neighbors, which can guide a reader's attention to the main properties of a thread and speed up reading, and
- Using the identified API features as queries to search for helpful online resources automatically.

To evaluate the precision of Haystack for feature extraction, we have manually extracted the problematic API features from a set of threads to create golden test data. In particular, the goal of our evaluation is to identify the best solution that achieves an ideal tradeoff between a good precision for API feature extraction and an acceptable amount of noise. We recommend our best solution that considers a negative sentence and its two preceding sentences and two follow-up sentences as a unit of extraction. This solution strikes a reasonable balance between precision (89%) and the amount of noise.

This work has made three contributions:

- A preliminary categorization of the sentences in an API discussion that leads to a proposal for API feature extraction based on a negative sentence and its neighbors.
- The implementation of the proposed solution in a tool called *Haystack* and its evaluation, which leads to the recommendation of a best solution that extracts API feature names from a negative sentence and its *four* surrounding neighbors. We validate that the proposed negative sentences based solution is indeed more likely to contain problematic API features than several other alternative approaches. Overall, the performance of Haystack is favorable for a practical tool.
- An initial investigation of three potential applications of the proposed feature extraction solution.

We have made our data and source code publicly available [2].

The remainder of this paper is organized as follows. Section II describes related work. Section III describes the design of our API feature extraction tool, Haystack. Section IV presents our evaluation of Haystack and its three potential applications. Finally, Section V concludes the paper.

II. Related Work

In [10], Panichella et al. propose a method to automatically mine source code descriptions from developer communications, such as emails and bug reports. The mined descriptions can help developers understand source code that otherwise lacks useful comments from the original developers, and be used as a starting point for source code re-documentation. Their method for extracting source code descriptions is based on measuring the textual similarity between each paragraph of text in developer communications, and the content of a method in the source code. In our work, we instead extract API feature names and negative sentiment sentences, with the ultimate goals to facilitate design improvement, the reading and browsing of long API discussion threads, and searching.

Software development activities often require a developer to peruse a substantial amount of text. Rastkar and Murphy [11] investigate whether it is possible to summarize software artifacts automatically and effectively so that developers could consult smaller summaries instead of entire artifacts. In particular, they investigate how well existing conversation-based classifiers can be used to summarize bug reports. They manually select and annotate 36 bug reports for training the classifiers. They ask graduate students to evaluate the generated summaries.

Lotufo, Malik, and Czarnecki [9] conduct another research on summarizing bug reports. Their summarization approach is based on a hypothetical model assuming that when pressed with time, the reader prefers to read the most important bug report sentences. Therefore, they propose to extract three categories of important sentences: sentences that are about frequently discussed topics, sentences that are evaluated or assessed by other sentences, and sentences that focus on the topics in the bug report's title and description. In addition to evaluation with standard measures, they also ask the developers who actually worked on the bugs to evaluate the summaries. In our project, we focus on the negative sentence and its close neighbors for extracting problematic API design features. One of the potential applications of our research is also highlighting the significant negative sentiment sentence and its close neighbors to facilitate the comprehension of long discussion threads. In this regard, these sentences can be considered a part of a summary. In addition, the extracted API design features can also be used to identify "hot topics" and search for pertinent help for the given API.

Our work is partially inspired by Hu and Liu's work on mining customer opinions about a product [6]. Such work can be broadly applied to mine opinions from reviews of movies, products, politicians, and almost anything, which can then be used to support Brand management (e.g., Windows 7), Polling (e.g., Obama), Purchase planning (e.g., Kindle) [3]. Hu and Liu's approach relies on automatically extracting from online reviews two key pieces of information, product features and customer opinions (positive or negative).

Our work is different Hu and Liu's in both aspects. We focus on not only nouns but also verbs and adjectives as potential feature words. Unlike simple products such as a camera, it is inadequate to consider only nouns or noun phrases for API feature names; verbs and adjectives are often also used in feature names, such as in "resize panel" and "visible." We therefore propose three subtree structures to extract features from the parse tree of a thread sentence that involve verbs, nouns or noun phrases, and adjectives. Moreover, to improve the quality of the extracted API feature names, we apply a filtering dictionary that consists of words from the official Java Swing tutorial. Lastly, for opinion mining, Hu and Liu has developed their own approach for mining customer sentiments. Their approach involves identifying adjectives that indicate either positive or negative orientation using WordNet. Instead,

[2]http://www.clarkson.edu/~dhou/projects/haystack2013.zip. All URLs last verified on 3/10/2013.

we use an off-the-shelf sentiment analysis solution [3], which is a more general approach than theirs in that it focuses on more than adjectives. We consider both the negative sentence and its neighbors in the discussion thread as a potential source for problemtic API features.

As electronic communications contain substantial knowledge about a software project, there has been substantial recent interest in improving the use and management of this information. In particular, classifiers have been applied to distinguish the different categories of information. Gottipati, Lo, and Jiang present an approach where tags automatically inferred for posts in software forum threads are used to find relevant answers in the forums [4]. Bacchelli et al. present an approach to classify email lines in five categories (i.e., text, junk, code, patch, and stack trace) so that one can subsequently apply appropriate ad hoc analysis techniques to process the lines in each category [2].

III. Design of Haystack

In our research, we apply sentiment analysis and natural language processing techniques to extract significant API features from a negative sentence and its neighbors. We have implemented an API feature extraction tool called *Haystack*. In this section, we start with a preliminary investigation of thread sentences that motivates our proposed technical solution for feature extraction. We then describe the three major components of Haystack: dictionary generation, negative sentence identification by sentiment analysis, and feature item extraction by parse-tree-based pattern matching.

A. Preliminary investigation of thread sentences

To extract significant API design features from online discussion threads, it is imperative for us to gain some insights into the structure and content of discussion threads. To this end, we have carefully studied 200 threads from the Java Swing Forum to better understand the structure and properties of a discussion thread. In the end, we have identified eight categories that can be used to categorize sentences in a discussion thread:

- The "Design-Goal" category includes sentences in which people describe requirements for the application they want to design or have developed, as in, for example, "I have made an application in Swing containing Grouplayout on the panel."
- The "How-To" category includes sentences in which people ask how to implement a concrete solution or how to use a particular API feature, as in, for example, "And first of all I don't know how to center the buttons."
- The "Question-of-Code" category involves sentences in which people ask questions about concrete code or API components, for example, as in "Does anyone know how Nimbus computes the button height."
- The "Neutral-Action" category includes sentences in which people describe what they have done or want to do in order to implement their design goal, as in, for example, "I'm trying to add a JTable to JScrollPane."
- The "Claims" category involves sentences in which people express their assertion about a certain program behavior, as in, for example, "This networking part is working very well."
- The "Neutral-Behavior" category involves sentences in which people describe the normal program runtime behavior, as in, for example, "As the JFrame is resized the JTextField has to move accordingly like any layout."
- In category "Question-of-Behavior," people ask questions about why a particular phenomenon has occurred, for example, "Why does this button fill the whole screen?".
- The "Negative-Behavior" category involves sentences that describe unexpected program behaviors. For example, the sentence "When I have a JPanel initialized with a null, the border does not work" expresses a negative sentiment about a particular program behavior (that the border does not work).

In the process of analyzing and classifying these sentences, we find that the negative-behavior sentences are common in discussion threads. We further observe that the negative sentences are more likely to contain *problematic* API design features than the other seven categories. In addition, we find that sometimes the desired API features are not contained by the negative sentence itself but in its close neighbor sentences. Therefore, this motivates our investigation of extracting problematic API features from both the negative sentence and its neighbors. While our list of categories may not be complete, in future work, it can be used as a starting point for extracting other significant information from online discussions.

B. Generating a filtering dictionary for API feature names

We create a dictionary and use it as a filter to further improve the quality of the extracted candidate API feature names. This is because API discussions commonly contain misspellings and words that are otherwise irrelevant to API features. The dictionary is used to filter out such words from the candidate features that are initially extracted. By analyzing the content of sample threads and the features of the Java Swing API, we find that API features are commonly named with nouns, verbs, or adjectives. For example, a feature can be named with a noun such as "panel," a combination of a noun and a verb such as "resize panel," or with an adjective such as in "invisible panel." Therefore, the dictionary should be made to contain the common nouns, verbs, and adjectives that are used for an API.

In our case, we extract such words from the official Java Swing tutorial [3]. Specifically, we use the Stanford CoreNLP parser [7] [4] to parse the tutorial text to generate the Part-of-Speech tag [1] for each word in the tutorial. We then extract all the nouns, verbs, and adjectives. We further limit the dictionary words only to those that appear at least a minimum number of times in the tutorial (five in our case). In the end, our dictionary contains 1,638 words, which is used as a filter to restrict the extracted words to those that are related to the Swing API.

[3] http://docs.oracle.com/javase/tutorial/uiswing/index.html
[4] http://nlp.stanford.edu/software/lex-parser.shtml

TABLE I: Sample negative sentences and API feature words contained. Words or phrases that contribute to the negative sentiment are highlighted. Notice that not all negative sentences contain API feature words.

Negative Sentence	**API Feature Words**
I tried to resize jscrollpane, *but* it *doesnt work*.	resize jscrollpane
The *problem* is the graphic is very long and the scrollpane will not scroll.	long graphic, scrollpane scroll
I can scroll down the drop-down list, *but* the scrollbar is not fully visible, as the height of the drop-down list is *simply too* large.	visible scrollbar, large drop-down list
so now, im using a JDialog ... *but* it 's *not* showing up *properly*.	JDialog show
I *could not* find relevant API of the JFileChooser.	JFileChooser
I *cannot* change the length of the JProgressBar.	JProgressBar length
I have tried to figure out why that the events do not make it to the component *but* have *failed*.	event
But I'm *stuck* on one thing.	
Can someone pls help me figure out the *problem*?	

C. Identifying negative sentences via sentiment analysis

Although in general API feature names may appear in any sentence, we focus on negative sentiment sentences as the basis for our feature extraction. In our research, we define sentiment to be "a personal positive or negative feeling" [3]. Table I shows some examples of negative sentiment sentences [5]. The reason we focus on negative sentences is that API feature names that occur in such contexts are likely good indicators of potential problems in either API design or usage [6]. Such information can be fed back to the original API team to improve design, and provide better support for the API. It can also be used to create better recommendation tools to help the API users who are having problems with the API.

The goal of sentiment analysis is to classify each sentence into one of three categories: the positive and negative polarities, or neutral. Best results for sentiment analysis can be achieved by training classifiers, such as support vector machines, with a corpus of texts that have had its polarity previously annotated. Preparing the large amount of training data needed for such supervised machine learning, however, can be time-consuming. Furthermore, it is not our best interest to do it since we are more interested in solving the software engineering research problem rather than advancing the state of art in sentiment analysis. Luckily, Bo and Bhayani's Sentiment140 API is a free solution that allows us to completely avoid this effort. Their tool uses a training set composed of 800,000 Twitter messages with positive polarity and 800,000 with negative polarity. The key novelty of Sentiment140 is that they automatically annotate the training tweets as negative or positive polarity using the emoticons present in the comments [3]. Using this data, the authors have tested three classifiers for sentiment analysis, Naive Bayes, Support Vector Machines, and Maximum Entropy, and their results indicate that the performance of the three classifiers are similar [3]. Their current tool, which is available as a web service, uses Maximum Entropy [7].

D. Parsing and feature extraction

After sampling and analyzing some API discussions, we find that API feature names are commonly made of nouns, verbs, adjectives, or a combination of these three types of words. Moreover, because people often describe their problems or unexpected program behaviors in negative sentences before asking for help, these problematic features frequently appear in a negative sentiment sentence or its close neighbors. For example, the sentence "I tried to resize jscrollpane, but it doesnt work." is a negative sentence that contains two significant API feature words "resize jscrollpane" that refer to the key API problem that the particular discussion is about. Therefore, we decide to extract particular words and phrases that can be used to name API features from a negative sentence and its close neighbor sentences.

By manually analyzing the parse trees of some selected sentences, we have identified three basic tree patterns that can be used to generate meaningful API feature items. As shown in Table II, these three patterns specify syntactic structures where candidate API feature names can be extracted:

- SUB-VERB is intended to capture the subject-verb portion of a sentence. It represents any clause-level subtree structure that contains the required nouns and verbs;
- VERB-OBJ matches any phrase-level subtree structure that contains the required verbs and nouns;
- VERB-PPHRASE matches any phrase-level subtree structure that contains the required verbs, prepositions, and nouns.

The three patterns in Table II are specified using Tregex ("tree regular expressions") [8]. Tregex is a utility for matching patterns in parse trees based on tree relationships and regular expression matches on nodes [8]. After we parse a discussion thread and build a parse tree for each sentence using the Stanford CoreNLP parser [7], we use the Tregex API to identify subtrees from the generated parse trees that match the three patterns. From each identified matching subtree, we extract all leaf words that are both nouns, verbs, and adjectives and contained in our filtering dictionary. We call the set of

[5]Notice that the word `component` in the seventh sentence of Table I was filtered out by Haystack because it is considered a generic word that is unlikely to be part of any meaningful feature name in the Java Swing API. The validity of this design decision needs to be validated in future work.

[6]From Table I, we can see that the term *API feature* can refer to a UI component (e.g., JFileChooser), an interaction possibility between a user and the interface (e.g., invisible scrollbar), or implementing a specific behavior of a UI component (e.g., resizing a JScrollPane).

[7]http://help.sentiment140.com/api

[8]http://nlp.stanford.edu/software/tregex.shtml

TABLE II: Three subtree patterns and example sentences, for localizing API feature words from a parse tree for a sentence. The Tregex expressions /S.?/, /VB.?/ and /NN.?/ specify tree nodes whose tags start with S (for several kinds of clauses), VB (for verbs), and NN (for nouns), respectively. VP is the short name for Verb Phrases, NP Noun Phrases, and PP Propositional Phrases. A<B means that tree node A *immediately* dominates B, whereas A<<B means that A dominates B. Although each example sentence may match more than one pattern, for clarity, only one is shown.

Pattern Name	Tree Regular Expression
SUB-VERB	`/S.?/<(NP<</NN.?/)<(VP<</VB.?/)`
	1. Problem is when I runs it JTextArea is missing! 2. But this column is not sorting like numbers. 3. My JTree is inside a JScrollPane. the problem that I have now is that the text in the JTextPane completely ignores the size of my JScrollPane and the words no longer wrap.
VERB-OBJ	`VP</VB.?/<(NP<</NN.?/)`
	1. Please tell me how to add the scrollpane. 2. May I know how can I disable focus traversal caused by enter key on jtable? 3. All, I have a problem with resizing the JFrame at runtime.
VERB-PPHRASE	`VP</VB.?/<(PP<</NN.?/)`
	1. I have to undo and redo of path and arrows to them. 2. What are the changes I have to do to sort like number or Double. 3. Made button and text box added to panel and then added to frame, and all is ok (did all in code, not in vs editor).

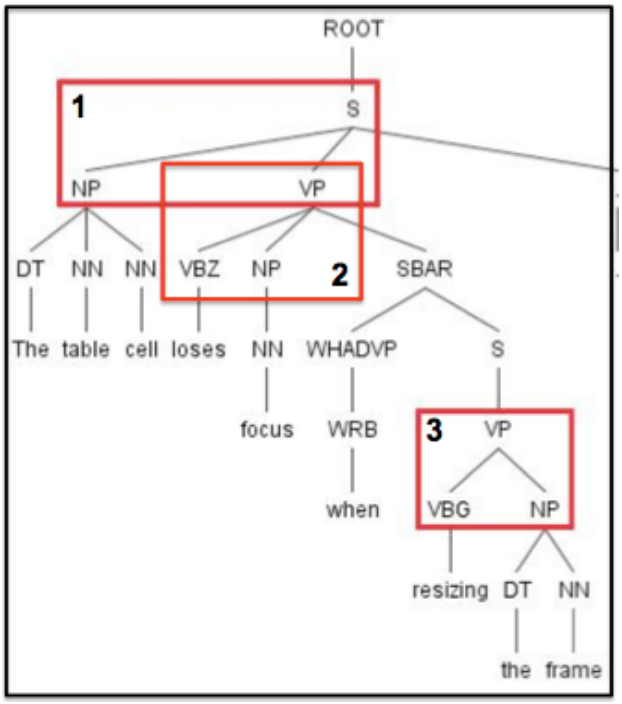

Fig. 1: Three Tregex pattern matchings highlighted on the parse tree of "The table cell loses focus when resizing the frame." The first matching is a SUB-VERB, producing an API feature item "table cell lose focus;" the second one is a VERB-OBJ, producing a feature item "lose focus;" the third is also a VERB-OBJ, producing a feature item "resize frame."

words extracted from each matched subtree *a feature item*. Table II shows one feature item for each example sentence.

Figure 1 depicts an example to illustrate pattern matching on a parse tree. This particular example contains three pattern matchings but produces only two feature items because one of the three feature items is completely subsumed by another.

IV. Evaluation

To implement the API design feature extraction tool described in Section III, some additional research questions need to be answered first:

RQ1: How many, and which, neighbor sentences should the API feature extraction tool consider processing, in order to strike an ideal balance between producing useful information and still reducing the amount of noise to an acceptable level?

RQ2: What are the characteristics of the ideal solution? What causes its imprecisions? How well do the three proposed subtree patterns work for extracting API feature names?

RQ3: How well does the ideal solution support the three potential applications? Specifically, to what extent can the estimated "hot topics" approximate the actual? How effective can highlighting the negative sentence and its neighbors be in speeding up reading and saving developer time? How effective can it be when using extracted feature words to search online?

We've chosen to gather our test data from the Swing Forum (Section IV-A) as it is a very active forum; as of March 10, 2013, it contains 47,276 discussion threads with 212,796 messages. An advantage of choosing the Swing Forum is that we can leverage the considerable prior research that we have conducted with the forum, e.g., [5], [12], so we can have greater confidence with the quality of our data.

Based on our evaluation in Section IV-B, we recommend a solution that extracts API features from a negative sentence and its two preceding sentences and two follow-up sentences.

A. Data collection and manual preparation of golden test data

To evaluate our feature extraction solution, we collected the texts of 928 threads from Oracle's Java Swing Forum [9]. More specifically, we randomly selected 928 threads and stored their

[9]https://forums.oracle.com/forums/forum.jspa?forumID=950

TABLE III: A sample post and the feature words that each of four methods M0, P2, F2, and B2 extracts from each sentence. Sentences are marked relative to the negative one (N). The manually extracted feature is "change jprogressbar length." M0: a method that processes only the negative sentence itself; P2: a method that processes the negative sentence and its two preceding sentences; F2: a method that processes the negative sentence and its two follow-up sentences; B2: a method that processes the negative sentence and its two preceding sentences and two follow-up sentences. ×: non-matching; ✓: matching.

Sample Post	**M0**	**P2**	**F2**	**B2**
(N-2) I've added a JProgressBar in to a JFrame. (N-1) It's working fine. **(N)** But I'm stuck on one thing. (N+1) I cannot change its length. (N+2) It's use a same length even I've change the length(width) of the JFrame. (N+3) How can I change that. (N+4) I've read the doc, but didn't found the way to change it's size. (N+5) Please help me.	stick	add jprogressbar jframe stick	stick change length change jframe length width	add jprogressbar jframe stick change length change jframe length width
separate	×	×	×	×
merged	×	×	×	✓

links in a file. We then wrote a program to automatically load the HTML code for each thread from the forum. Observing that the Swing Forum regularly puts the content of each post in a thread between two special HTML tags, * * and * *, we were able to extract the text of each post and store it separately for further processing. There are currently 4,997 posts in our data.

We parsed each post using the Stanford CoreNLP parser. The parser recognized a total of 33,249 sentences in the 928 threads. A thread may contain a minimum of 2, and a maximum of 169, sentences. On average, a thread may contain 36 sentences.

Using the Sentiment140 API, we found that there are 549 threads containing at least one negative sentiment sentence. There are a total of 1,652 negative sentences in the 549 threads, which is about 5% of all sentences in the 928 threads. A thread may contain a minimum of 1, and a maximum of 17, negative sentences. On average, a thread may contain 3 negative sentences.

To produce the golden test data needed for evaluating our proposed feature extraction solution, we manually analyzed a random subset of 131 out of the 549 threads that contain negative sentences [10]. We determined that our focus should be on questions and discussions that are related to the use of the Swing API, so we excluded and considered ten of the threads out of scope because they are about designing specific applications rather than using the API. For each of the remaining 121 threads, we manually extracted one API feature that had been the core difficulty of the original discussion. The words that appear in the name of such an API feature are compared with the feature words generated by our feature extraction solution for evaluation purpose (see Section IV-B for details). Notice that our manual extraction is *extractive*, not *abstractive*, in that we have always picked API words that have already appeared in the thread as our manual answer. The first author created the answer for each of the 121 cases, and the second author reviewed them multiple rounds for accuracy. All of our data have been archived and made available, so independent reviews by third-parties are possible.

[10]Due to time constraint, we were only able to finish 131 of the 549 discussions. We intend to expand our golden data set in future work.

B. Balancing between precision and noise

As explained in Section III, our feature extraction tool Haystack extracts feature items from a negative sentiment sentence and its close neighboring sentences. As shown by the last two examples in Table I, the reason why we consider close neighbors for feature extraction is that a negative sentence itself may not always contain the desired feature name. In this section, we set out to answer two important questions: How many and which neighbors should we consider? Is this approach better than others in terms of extracting problematic API features?

The number of neighbors considered matters because one of the potential application of Haystack is to highlight the negative sentence and its close neighbors to speed up comprehension. Although using more neighboring sentences will increase the chance of locating the correct API feature name, using too many of them can be counter-productive because the reader now is forced to read more sentences. Therefore, we need to strike a fine balance between finding useful information and reducing noise. Moreover, the location of a neighbor sentence considered also matters. For example, it may appear intuitively correct to assume that people might start with describing program behavior in a few sentences, while using the API feature names, *before* they complain about a problem negatively. Therefore, we should consider the negative sentence and its preceding neighbors. On the other hand, our experience with reading API discussions shows that API feature names may also appear in follow-up sentences.

To identify the best solution with regards to both the number and location of neighbors, we have tested 16 solutions against our golden test data of the 121 threads (Table IV). In general, a thread may contain multiple units, each of which consists of a negative sentence and its neighbors considered. As illustrated by the example shown in Table III, for each unit, Haystack may produce multiple feature items.

We have considered two ways for testing the correctness of the extracted feature. One is to compare the manual answer with the set of words *merged* from all feature items in the unit. The other is to compare against individual feature items *separately*. In both cases, we consider the comparison a match if the manual answer is contained in the set of words extracted for representing an API feature. The particular example in Table III illustrates that merging feature words can make a difference in matching. If a thread contains at least one match, we consider that Haystack has worked correctly once, for that thread. The precision for each solution is defined to be the percentage of threads that Haystack has handled correctly.

As shown in Table IV, the solution that considers only the negative sentence itself (M0) has a significantly lower precision than those also considering neighbors. Moreover, solutions that consider *only* preceding (Pn) or follow-up (Fn) sentences are inferior to those that consider both directions. On the whole, we are in favor of the solution B2, which is shown in italic in Table IV. B2 extracts API feature names from a negative sentence and its two preceding sentences and two follow-up sentences. Although B3, B4, and B5 all have higher precisions than B2, the increase appears to be minor (only 3%) and starts to plateau as the number of neighbors increases by two.

To validate that our recommended B2 is indeed a better solution, we have compared it with three alternative locations in a thread for feature extraction: (i) thread titles, (ii) the first five sentences, and (iii) randomly selected five sentences. The respective precisions are 61% (74 matches), 41% (50 matches), and 20% (24 matches on average for ten trials). Therefore, we can conclude that our proposed solution, in particular due to the use of sentiment analysis, is more likely to extract problematic API features than the three alternatives.

C. Reasons for imprecision

As shown in Table IV, our best solution B2 fails to produce a correct API feature name in 13, or 11%, of all 121 golden test cases. We have manually investigated these 13 cases and identified four reasons that have caused Haystack to produce incorrect feature names:

- The Stanford CoreNLP parser (7 of 13): English parsers have varying degrees of tolerance toward improper grammatical structures. As a result, in the presence of irregular sentences, they may produce 'incorrect' PoS tags that fail to match any of the three subtree patterns that we have considered. For example, Haystack fails to recognize the words "combobox" and "cell editor" from the sentence "Then I am setting combobox (with label (image icon) rendered) cell editor."; the parser mistakenly recognizes the portion starting at "combobox" as a clause. Another minor issue is that the parser treats a single dot as a sentence, causing an incorrect number of neighbors being used. This is not a problem of the parser per se, and can be easily avoided by adjusting the Haystack implementation.

TABLE IV: Precisions evaluated for the 16 extraction solutions that use different number of neighboring sentences at different locations. All solutions are measured using the 121 threads for which we have manually identified the API features. M0: a method that processes only the negative sentence itself; Pn: a method that processes the negative sentence and its n preceding sentences; Fn: a method that processes the negative sentence and its n follow-up sentences; Bn: a method that processes the negative sentence and its n preceding sentences and n follow-up sentences.

Solution	**#Matched Cases (out of 121)**	
	merged	**separate**
M0	44 (36%)	38 (31%)
P1	74 (61%)	60 (50%)
P2	90 (74%)	74 (61%)
P3	97(80%)	75 (62%)
P4	97(80%)	75 (62%)
P5	97(80%)	76 (63%)
F1	62 (51%)	52 (43%)
F2	68 (56%)	58 (48%)
F3	68 (56%)	58 (48%)
F4	69 (57%)	58 (48%)
F5	70(58%)	58 (48%)
B1	89 (74%)	74 (61%)
B2	*108 (89%)*	*93 (77%)*
B3	110 (91%)	93 (77%)
B4	110 (91%)	93 (77%)
B5	110 (91%)	94 (78%)

- Haystack design (3 of 13): Sometimes sentences containing significant API feature words are beyond two sentences away from the negative sentence. As a result, Haystack fails to extract the relevant API feature names from such sentences. An example can be found at the URL below [11].
- The Sentiment140 API (2 of 13) is not perfect: It may miss a negative sentiment sentence that contains the desired API feature names. An example can be found at the URL below [12].
- The thread does not contain the desired API feature words (1 of 13). This is often because the discussants have not explicitly discussed the relevant API feature in the discussion. Since Haystack is extractive, it fails to find such features. An example can be found at the URL below [13].

D. Feature items extracted from each unit

One potential application for Haystack is to use the extracted API feature words as query to search for pertinent help on the Internet. To inform the investigation of this potential application, we have collected statistics for feature items that are extracted from all the units in the 549 threads that contain at least one negative sentiment sentence:

- A total of 5,745 feature items are extracted from the 549 threads that contain negative sentences, of which 1,771 (31%) is due to the SUB-VERB tree pattern, 2,235 (39%)

[11] https://forums.oracle.com/forums/thread.jspa?threadID=2272230
[12] https://forums.oracle.com/forums/thread.jspa?threadID=1355791
[13] https://forums.oracle.com/forums/thread.jspa?threadID=1363932

TABLE V: Statistics for feature items extracted from each unit within the 549 threads that contains a negative sentence and its four surrounding sentences.

Solution	Min	Max	Average
M0	0	7	1
P1	0	10	2
P2	0	12	2
P3	0	14	2
P4	0	15	2
P5	0	18	3
F1	0	11	1
F2	1	11	1
F3	0	12	2
F4	0	15	2
F5	0	17	2
B1	0	14	2
B2	0	16	3
B3	0	17	3
B4	0	23	3
B5	0	26	4

the VERB-OBJ pattern, and 1,739 (30%) the VERB-PPHRASE pattern.

- Furthermore, Table V depicts the min/max/average numbers of feature items that each five-sentence unit contains. So, on average, our best solution B2 produces 3 feature items for each unit.
- The min/max/average numbers of words contained by each feature item are 1, 8, and 2, respectively. So, on average, our best solution B2 produces 6 words that can be used as a query (3 feature items times 2 words per feature item).

In this case, we extract feature items by utilizing the best solution we have identified in Section IV-B. That is, each unit consists of one negative sentence as well as its two preceding and two follow-up sentences.

To find out the effect that each of the three tree patterns has for extracting correct feature names, we have also counted how many times each pattern have produced a feature item that matches the manual answer. By processing each unit from the 121 threads in our golden test data, where a unit consists of a negative sentence plus its two preceding and two follow-up sentences, we find that the SUB-VERB tree pattern has produced a matching feature item 28 times, VERB-OBJ 40 times, and VERB-PPHRASE 29 times. These numbers indicate that there are cases where the same feature items have been extracted by using multiple patterns (because the numbers sum up to 97, more than than the number 93 that is shown in the cell at the intersection of the *B2* row and **separate** column in Table IV). This also indicates that all three patterns are useful and necessary.

E. Potential application 1: Highlighting negative sentences and its close neighbors for speed reading

Developers are busy professionals that have to comprehend a large amount of text on a daily basis. Others have looked into the possibility of automatically creating significantly shorter summaries (up to 25% of the original text) of a lengthy document, such as a bug report, to save developers time in finding the right piece of information needed for performing their task [11], [9]. These summaries can be either highlighted in context or extracted out for independent display for perusal by a developer.

Based on the evaluation shown in Table IV, we have recommended the solution B2 for extracting API feature names from API discussions. The precision of B2 is determined to be 89%. Therefore, this feature extraction solution can identify significant API design features fairly accurately. This also implies that the information that this tool extracts could be used to help readers to discern the core problem in an API discussion more quickly. This will in general provide a more efficient and effective way for them to manage the information contained in the discussions and to help each other.

As discussed in Section IV-A, on average, each discussion thread contains 36 sentences and 3 negative sentences. The B2 solution extracts five sentences for each negative sentence unit, hence 15 out of 36 sentences, or 42% of the original thread, with a precision of 89%. The B1 solution, on the other hand, extracts 9 sentences, or 14% of the original thread, with a lower precision of 74%. As an extreme example, the longest thread in our corpus, titled "subclass DefaultTableCellRenderer" [14], contains 15 posts, 169 sentences, and 14 negative sentences. For this thread, the B2 solution would extract 70 sentences, or 41% of the original content.

The Haystack approach essentially highlights sentences that describe the problems. By contrast, a summary of a thread would describe not only the problems but also the solutions. Nonetheless, we believe that the Haystack approach can complement these summary-based work [11], [9].

F. Potential application 2: Search using extracted feature items as queries

In addition to highlighting each unit of a negative sentence and its close neighbors to illustrate and explicate the key problematic API feature in a thread, the feature items extracted from each unit can also be used as a query to search for pertinent help on the Internet. This potential application can help collect from elsewhere related online information for similar problems. Such additional information will at least complement the current discussion in some ways. Because such a query is run automatically, an advantage over manual search is that it will always be updated with the latest available information. This approach will benefit not only the original question publisher but also other future visitors to the forum.

To test this potential application, we conducted an experiment by searching on the Internet and analyzing the content behind the returned links. In this experiment, we obtained the set of API feature words from the first unit for each of the 108 matching cases for B2 (Table IV). We then used the set of words as a query to search in Google. For each query, we manually went through the list of returned links to identify the first helpful link that can be used to solve the problem discussed in the original thread. We recorded both the link and

[14]https://forums.oracle.com/forums/thread.jspa?threadID=2138325

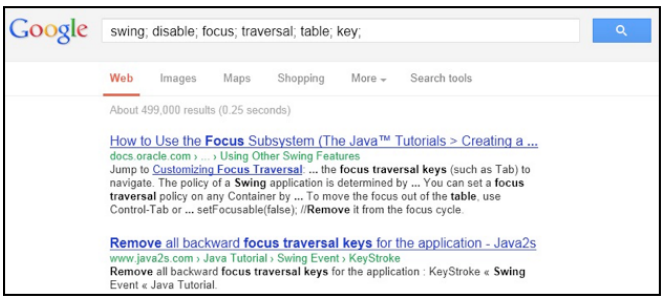

Fig. 2: Google search with query words taken from two feature items, "disable focus traversal" and "key table." The keyword "swing" is added to improve search quality. Notice that Haystack has substituted the word "table" for "JTable".

the rank of the first useful link. Both authors have reviewed the recorded ranks for accuracy. We have also made the data available for further third-party analysis.

In what follows, we show a post taken from one thread to illustrate our manual analysis process. Notice that we have marked the negative sentence with (N). We have also underlined the feature words extracted by Haystack.

> (N) I really need a help here. (N+1) I have been trying to figure it out for two days, but I still don't know how to do this. (N+2) May I know how can I disable focus traversal caused by ENTER key on jTable? (N+3) I have a jTable with only one column and multiple rows, and everytime I pressed the enter key, the focus will always traverse to the next row. (N+4) May I know how can I disable it?

Figure 2 depicts the Google query that we used for this example as well as the first two links that Google returned. Notice that in an attempt to unify synonyms, Haystack has substituted the word "table" for "JTable". Since all of our test threads are about the Java Swing API, we have added a common key word "Swing" to all queries so as to improve the quality of the search results. The first link in Figure 2 pointed to the official tutorial on focus management, which we considered relevant to the problem discussed in the original post. Therefore, we recorded a rank of 1 for this case.

After analyzing all of the 108 cases using the above approach, we find that using our approach, on average a user can locate a helpful link by perusing no more than the top three links returned by Google. Although our experiment is still preliminary in scale and rigor to be conclusive, this result is encouraging enough to justify further in-depth investigation.

G. Potential application 3: Estimating API "hot topics"

As shown in Table III, our feature extraction solution may extract feature items from each sentence in a negative-sentence unit. Given that our recommended solution B2 has achieved a precision of 89% (Table IV), one is naturally left to wonder whether the occurrence frequencies of the extracted feature items can be used to estimate the trend in a forum to illustrate what kinds of API problems are discussed most frequently and what are the "hot topics" in a forum. These "hot topics" can be especially meaningful to the API support team as feedback; they can check what kinds of API problems their users have encountered most frequently, and where and how they can improve the usability of their API to help the users more effectively.

TABLE VI: Top 10 features produced by Haystack versus through manual analysis. The features are based on the golden test data of 121 threads. Features that are unique to each case are shown in italic. Features that an experienced user of the API indeed considers problematic are marked up with *.

Haystack	**Manual**
Button	Table
Display*	Display
Table*	Button
Frame	Image
Panel*	Panel
Image*	Size
Size*	*Scrollpane*
Filechooser*	Dialog
Dialog*	Filechooser
Label	*Event*

To evaluate the potential of estimating API "hot topics" by means of counting the occurrence frequencies of the feature items that are produced by Haystack, we compare the top 10 common features produced by Haystack with the top 10 produced by manually analyzing the 121 threads in our golden test set. The features are shown in Table VI in descending order of occurrence frequencies. The idea is that if these two sets overlap significantly, we could have greater trust in the Haystack-generated "hot topics." As shown in Table VI, this indeed seems to be the case; the two lists share eight common features out of ten, indicating that the accuracy of the estimated "hot topics" is fairly good. As an additional evaluation, after the first author produced Table VI, the second author, who has more experienced with the Swing API, confirmed that seven of the ten features are indeed problematic for the average users. The three exceptions are "Frame," "Label," and "Button," which could also just be the most used topics and not necessarily the most problematic areas in the Swing toolkit. More details about the experiment can be found in [14].

In estimating the "hot topics," we have encountered several issues that require further investigation in future:

- One issue in automatically producing the estimation is dealing with synonyms. To produce a better estimation, we manually identify and normalize synonyms in a post-processing step. For example, the feature words "display," "appear," "show," and "see" are all about displaying a widget on screen. Hence we normalize them by the same word "display."
- Another issue is related to the ability to precisely pinpoint the target of the negative sentiment. For example, although "Frame" and "Label" are included as problematic features in Table VI, this is not because they are a major source of difficulty (actually they are not), but simply that they are used in the context where a problem occurs.

An improved sentiment analysis that is specific about the target of the sentiment negativity would help eliminate such popular "features" and improve the accuracy of the estimated "hot topics."

- The third issue is handling the containment relationship between features. For example, we have broken down a meaningful feature name produced by our manual analysis, such as "use mediatracker with jprogressbar," into two words, "mediatracker" and "jprogressbar," in order to produce better match.

H. Threats

Our golden test set consists of only 121 threads, for a single API. More data and additional APIs are needed in order to obtain more reliable and general conclusions.

Another threat that could affect the validity of the study is the manual extraction of API features performed by one of the authors. Specifically, since we know how the approach works, there is a possibility that we may be biased when identifying the API features in a discussion thread. An external validator would help improve the reliability of our findings in this regard. The online publication of our data and source code has made such a third-party validation possible.

The set of predefined subtree patterns can be enlarged to identify more API features. In this regard, our current work should be viewed mainly as demonstrating feasibility for the proposed approach. Currently, we define three subtree patterns for capturing the common expression structures that we have identified from manually analyzing the collected threads. This solution may miss some special subtree patterns that are not as popular as these three subtree patterns.

Our study measures only precision. Measuring recall is harder as it would require substantial efforts in creating the golden test data (for example, we would have to annotate all sentences for their sentiment polarity). Nonetheless, it would be useful to measure recall in future work.

Lastly, although our focus in this paper is on evaluating the proposed feature extraction and its three potential applications, user studies are needed for more serious evaluation to substantiate the benefits expected from the developed techniques, which we would like to save as future work.

V. Conclusion and future work

Software libraries and APIs are important productivity tools, but they are difficult to use due to their extensive content and rich details. As a result, developers have widely used online software forums to exchange information and seek help to solve problems in using APIs. Tens of thousands of discussions have been archived for popular frameworks. Therefore, tools that can help developers effectively manage these data can potentially have a major impact on development productivity.

In this paper, we have investigated an approach for automatically extracting problematic API features from online discussions. Based on our evaluation, we recommend one solution that treats each negative sentiment sentence and its four surrounding neighboring sentences as one unit of extraction. We have also investigated three potential applications of the recommended solution, i.e., highlighting, searching, and estimating API "hot topics." We conclude that the investigated approach is promising to become a useful developer tool.

More work is needed to expand the range of applications for Haystack and to improve its performance. Firstly, Haystack depends on some open source APIs that have influenced its performance. For example, the Stanford CoreNLP parser cannot parse all sentences correctly, and the Sentiment140 API sometimes fails to identify all the negative sentences in a thread. Haystack will perform better if these problems can be resolved. Related, improved sentiment analysis algorithms and/or deeper, semantics-oriented analysis of natural language text could lead to a much better feature extraction tool. In particular, it would be interesting to test how much precision can be improved if we replace the Sentiment140 API with a machine learning classifier that is trained directly with sentences used in forum posts. Furthermore, the potential applications of Haystack can be fully implemented and tested with real users in the future. Lastly, future work should also evaluate the proposed approach on additional APIs in order to demonstrate generality.

Acknowledgment

The authors are grateful for the very thorough and detailed comments from the three anonymous reviewers, which have helped improve the presentation of this paper.

References

[1] S. Abney, "Part-of-speech tagging and partial parsing," in *Corpus-Based Methods in Language and Speech*. Kluwer Academic Publishers, 1996, pp. 118–136.

[2] A. Bacchelli, T. D. Sasso, M. D'Ambros, and M. Lanza, "Content classification of development emails," in *ICSE*, 2012, pp. 375–385.

[3] A. Go, R. Bhayani, and L. Huang, "Twitter sentiment classification using distant supervision," Stanford University, Tech. Rep., 2009.

[4] S. Gottipati, D. Lo, and J. Jiang, "Finding relevant answers in software forums," in *ASE*, 2011, pp. 323–332.

[5] D. Hou and L. Li, "Obstacles in using frameworks and APIs: An exploratory study of programmers' newsgroup discussions," in *ICPC*, 2011, pp. 91–100.

[6] M. Hu and B. Liu, "Mining and summarizing customer reviews," in *KDD*, 2004, pp. 168–177.

[7] D. Klein and C. D. Manning, "Accurate unlexicalized parsing," in *ACL*, 2003, pp. 423–430.

[8] R. Levy and G. Andrew, "Tregex and Tsurgeon: tools for querying and manipulating tree data structures," in *LREC 2006*, 2006.

[9] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the 'hurried' bug report reading process to summarize bug reports," in *ICSM*, 2012, pp. 430–439.

[10] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora, "Mining source code descriptions from developer communications," in *ICPC*, june 2012, pp. 63 –72.

[11] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *ICSE*, 2010, pp. 505–514.

[12] C. R. Rupakheti and D. Hou, "Evaluating forum discussions to inform the design of an API critic," in *ICPC*, 2012, pp. 53–62.

[13] S. Sarawagi, "Information extraction," *Foundations and Trends in Databases*, vol. 1, no. 3, pp. 261–377, 2008.

[14] Y. Zhang, "Where do people stumble? Automatically detecting API features that have troubled users," Master's thesis, Clarkson University, Potsdam NY, March 2013.