To Do or Not To Do: Distill Crowdsourced Negative Caveats to Augment API Documentation

Jing Li
School of Computer Science and Engineering, Nanyang Technological University, Singapore.
E-mail: jli030@e.ntu.edu.sg

Aixin Sun
School of Computer Science and Engineering, Nanyang Technological University, Singapore.
E-mail: axsun@ntu.edu.sg

Zhenchang Xing
College of Engineering and Computer Science, Australian National University, Australia.
E-mail: zhenchang.xing@anu.edu.au

Negative caveats of application programming interfaces (APIs) are about "how not to use an API," which are often absent from the official API documentation. When these caveats are overlooked, programming errors may emerge from misusing APIs, leading to heavy discussions on Q&A websites like Stack Overflow. If the overlooked caveats could be mined from these discussions, they would be beneficial for programmers to avoid misuse of APIs. However, it is challenging because the discussions are informal, redundant, and diverse. For this, for example, we propose DISCA, a novel approach for automatically DIStilling desirable API negative CAveats from unstructured Q&A discussions. Through sentence selection and prominent term clustering, DISCA ensures that distilled caveats are context-independent, prominent, semantically diverse, and nonredundant. Quantitative evaluation in our experiments shows that the proposed DISCA significantly outperforms four text-summarization techniques. We also show that the distilled API negative caveats could greatly augment API documentation through qualitative analysis.

Introduction

Application programming interfaces (APIs) are foundations of software development. To program with an API, developers need to know not only "how to use the API," but also "how *not* to use the API." Table 1 lists seven examples of "how not to use an API" extracted from Stack Overflow,[1] a Q&A website for topics in programming. We refer to such "how not to use an API" directives as API negative caveats.

API documentation is an important resource for developers to learn unfamiliar APIs (Kramer, 1999; Robillard & Deline, 2011; Dagenais & Robillard, 2012; Stylos, Faulring, Yang, & Myers, 2009). By providing important information about functionality, parameters, and use scenarios of an API, API documentation often does a good job at explaining "how to use an API" (Robillard & Deline, 2011; Subramanian, Inozemtseva, & Holmes, 2014). More often than not, API documentation does not mention any API negative caveats. Even when negative caveats are mentioned, they are often buried in the verbose descriptions of the API and can be barely noticed by developers.

Not mentioning API negative caveats is not always because API designers are reluctant to document negative caveats. We will use the examples in Table 1 to further illustrate this point. First, an API negative caveat is sometimes related to a broader context in which an API is used. For example, java.awt.event.ActionListener is often implemented as an inner class. According to Java language specification, an inner class cannot access nonfinal variables from the scope that contains the inner class. Second, an API negative caveat may be rooted in the API's design. For example, javax.swing.JTextArea is designed to display plain text only; as such, it does not support styled text. Third, API negative caveats may also emerge from practical use scenarios, which API designers are unable to foresee. As an example,

[1]https://stackoverflow.com/

Received July 28, 2017; revised April 12, 2018; accepted April 29, 2018

© 2018 ASIS&T • Published online August 9, 2018 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/asi.24067

TABLE 1. Examples of API negative caveats extracted from Stack Overflow.

API Types	API negative caveats	Post ID
java.util.HashMap	**Don't** use a HashMap if you are going to have multiple threads, use a ConcurrentHashMap instead.	15389917
	HashMap **doesn't** guarantee the order in which elements are returned.	10372667
javax.swing.JTextArea	The JTextArea **will not** scroll down with the text.	630030
	JTextArea **is not** a component designed for styled text.	9097995
java.awt.event.ActionListener	Inner classes, such as your ActionListener, **cannot** access nonfinal variables from the scope that contains it.	30689818
	Don't use an ActionListener to check when the button is selected.	19532759
javax.swing.text.html.ListView	You **cannot** use a ListView inside a ScrollView (any two views that have same orientation scrolling).	8551849

it is not expected that javax.swing.text.html.ListView to be used inside a ScrollView.

Nevertheless, once an API negative caveat has slipped a programmer's attention, it is very likely to result in unexpected programming errors. An effective way of seeking a solution is to post a question on Stack Overflow, and wait for suggestions from other developers. Often, the answers explicitly point out the overlooked API negative caveats and suggest ways to avoid such errors, as shown in Table 1.

Such Q&A discussions effectively document negative experiences emerging from overlooking API negative caveats in practice. They are also referred to as crowd documentation (Parnin, Treude, Grammel, & Storey, 2012), which generates a rich source of content that complements the official API documentation. If we could extract such crowd-sourced API negative caveats like those in Table 1, we may highlight the hard-to-notice negative caveats in API documentation or augment the API documentation with the missing negative caveats. Such augmentation would raise developer's caution to avoid misuse of APIs, or to help them fix errors caused by overlooking API negative caveats. However, these API negative caveats, present in crowd-sourced Q&A discussions, are informal, redundant, and are often related to different aspects of API use. How to effectively distill API negative caveats from unstructured Q&A discussions is a challenging task.

For this, for example, we present **Disca**, an approach for automatically distilling API negative caveats from large-scale unstructured Q&A discussions. To the best of our knowledge, our work is the first attempt to tackle the problem of negative uses of APIs. We formulate the problem as a text-summarization task and identify four desirable properties for the distilled API negative caveats: context-independence, prominence, semantic diversity, and semantic nonredundancy.

Given a set of programming-related sentences extracted from Stack Overflow discussions, **Disca** first selects a set of candidate sentences, that is, sentences mentioning a specific API with negative expressions. Then, **Disca** selects context-independent sentences that identify issues about an API use without referring to the discussion contexts. Next, **Disca** selects semantically diverse and nonredundant sentences that cover prominent domain-specific terms through a combination of techniques including relative entropy, term co-occurrence analysis, and set cover.

We conduct both quantitative and qualitative evaluations to demonstrate the effectiveness of **Disca**. For quantitative evaluation, we aim to answer the following research questions: RQ1: How much improvement can the **Disca** approach achieve over baseline methods? and RQ2: How effective is the proposed **Disca** approach in guaranteeing diversity over baseline methods? For RQ1, we compare the performance of **Disca** against four text-summarization techniques: eigenvector centrality of sentence graph (LexRank; Erkan & Radev, 2004), topic modeling (LDA, Blei, Ng, & Jordan, 2003), sentence clustering (KM MacQueen et al., 1967), and sentence diversification (MMR, Goldstein, Kantrowitz, Mittal, & Carbonell, 1999). We evaluate the performance of **Disca** and the baseline methods with two commonly used metrics: Recall-Oriented Understudy for Gisting Evaluation (ROUGE) and Normalized Discounted Cumulative Gain (nDCG). Our results show that **Disca** outperforms the four baselines by 10.60% to 22.47% for ROUGE and 17.63% to 42.87% for nDCG, respectively. For RQ2, we conduct an intermethod comparison, which compares the relative performance between one method and the other four methods using the Jackknifing procedure (Lin, 2004). The results of intermethod comparison show that the summaries of **Disca** are more diverse than the baseline methods.

For qualitative evaluation, we aim to answer the following research questions: RQ3: To what extent does the **Disca** approach miss the API negative caveats stated in official API documentation? RQ4: To what extent does the **Disca** approach augment the official API documentation? and RQ5: How important are the distilled API negative caveats by the **Disca** approach? For RQ3 and RQ4, we compare the negative caveats that are documented in the API documentation of 10 Java API types and the ones mined by **Disca**. The results show that official API documents mention only six negative caveats, while **Disca** distills 164 from Stack Overflow. These 164 negative caveats cover four out of the six negative caveats mentioned in API documentation (that is, two negative caveats missed). More important, **Disca** greatly

augments the official API documents of the 10 Java APIs with 146 correctly identified negative caveats out of 164. In order to answer RQ5, we conduct a case study to present the distilled API negative caveats of four Java API types. The results show that DISCA helps to reveal hard-to-notice API negative caveats. These distilled negative caveats are difficult to document by API designers and are hard to foresee, because they mainly emerge from misuse in practice.

Related Work

Given the commonality of crowd-generated content, our related work section is divided into three parts: works on crowdsourced knowledge in software development, App review opinion mining, and automatic text summarization.

Crowdsourced Knowledge in Software Development

Several studies have contributed effort on aiding developers in software development using crowdsourced knowledge. The crowdsourced knowledge was generally derived from two types of sources: code snippet and textual content. For crowdsourced code snippets, there are studies (Sahavechaphan & Claypool, 2006; Thummalapenta & Xie, 2007) investigating the problem of how to integrate code snippets in Integrated Development Environment (IDE) to recommend code examples during software development. For example, XSnippet (Sahavechaphan & Claypool, 2006), a context-sensitive code assistant framework, allows developers to query a sample repository for code snippets that are relevant to the programming task at hand. Other studies (Brandt, Dontcheva, Weskamp, & Klemmer, 2010; Kim, Lee, Hwang, & Kim, 2009; Zagalsky, Barzilay, & Yehudai, 2012) investigated the problem of how to build a new code search by utilizing source snippets on the web. In addition, a few studies focused on recovering the traceability of various software artifacts, such as the link between source code snippets and official API documentation (Kim et al., 2009; Subramanian et al., 2014) and the link between source code snippets and their learning resources (Dagenais & Robillard, 2012). Such existing studies on crowdsourced code snippet are based on code search engines and code static analysis. In contrast, DISCA is designed to distill negative caveats that are expressed in natural language.

Another type of crowdsourced knowledge is textual content. Many studies developed tools to integrate Q&A resources into the IDE, such as Seahawk (Bacchelli, Ponzanelli, & Lanza, 2012), and Prompter (Ponzanelli, Bacchelli, & Lanza, 2013; Ponzanelli, Bavota, Di Penta, Oliveto, & Lanza, 2014). Studies also developed question answering systems to answer programming questions by leveraging official content and social context of software documentation (Li, Xing, Ye, & Zhao, 2016; Li, Sun, & Xing, 2018; Li, Xing, & Kabir, 2018). In addition, researchers have contributed their efforts for program comprehension by using software textual content (Ponzanelli et al., 2013; Treude, Barzilay, & Storey, 2011). These existing studies link or recommend crowdsourced knowledge from the point view of "how to use an API" at the post level or document level. In contrast, our work distills insights at the fine-grained sentence level from the point view of "how not to use an API." Recently, Treude and Robillard (2016) presented SISE, which automatically augments API documentation with insight sentences from Stack Overflow. This work is the closest to our work. However, in Treude and Robillard (2016), the authors trained a binary classifier with hand-coded features in a supervised manner and the solution does not consider the factors of redundancy, diversity, and negative expression in the summarization algorithm. In short, existing studies focus on general relevance of the recommended knowledge, while our work specifically focuses on negative insights related to API uses. To the best of our knowledge, no prior work has been done on negative uses of APIs.

App Review Opinion Mining

Our work summarizes the crowd-generated sentences with respect to APIs. It is similar to crowd-generated reviews for Apps. Miao, Li, and Zeng (2010) exploited the domain knowledge to assist product feature extraction and sentiment orientation identification from unstructured reviews. Wisniewski, Xu, Lipford, and Bello-Ogunu (2015) examined two prominent Facebook features that promote confidant disclosures: tagging and third-party applications. The results illustrate the complexity of the trade-off between privacy concerns, engaging with friends through tagging and Apps, and Facebook use. Gu and Kim (2015) presented SURMiner, which classifies reviews into five categories (that is, aspect evaluation, bug reports, feature requests, praise, and others) and extract aspects in sentences using a pattern-based parser. Chen, Lin, Hoi, Xiao, and Zhang (2014) developed AR-Miner, which helps App developers extract the most valuable information from raw user review data. Vu, Nguyen, Pham, and Nguyen (2015) proposed MARK, a keyword-based framework for semiautomated review summarization and visualization.

These existing works focused on opinion-aspect phrase extraction (Vu et al., 2016) and conducted sentiment analysis of opinion words (Gu & Kim, 2015; Serva, Senzer, Pollock, & Vijay-Shanker, 2015). Although API negative caveats are expressed in negative sentences, they state a neutral fact about API use rather than a polarity opinion. Thus, sentiment analysis followed by aspect extraction in the above work is generally not applicable for distilling API negative caveats from crowd-generated discussions.

Automatic Text Summarization

In recent years, there has been an explosion in the amount of text data, which need to be effectively summarized to be useful. Those existing approaches in general fall into two categories: extractive summarization and abstractive summarization. Extractive summarization methods select a few relevant sentences from the original document as a summary. Summary sentence selection therefore is a critical step

DOI: 10.1002/asi

in the extractive summarization process. Most previous shallow models estimate the salience of a sentence using predefined features, such as lexical chains (Barzilay & Elhadad, 1999), word co-occurrence (Matsuo & Ishizuka, 2004), and centrality (Erkan & Radev, 2004). Recently, many advanced models were developed to learn deep semantic features. For example, Cao et al. (2015) developed PriorSum, which applies enhanced convolutional neural networks to capture the summary prior features derived from length-variable phrases. The learned prior features are concatenated with document-dependent features for sentence ranking. Ren et al. (2017) proposed a neural extractive model, named contextual relation-based summarization, to take advantage of contextual relations among sentences so as to improve the performance of sentence regression.

Abstractive summarization methods produce a new concise text that includes words and phrases different from the ones in the source document. Structure-based approaches have been studied extensively, such as rule-based (Genest & Lapalme, 2012), ontology-based (Lee, Jian, & Huang, 2005), and template-based (Harabagiu & Lacatusu, 2002) approaches. Recently, semantic-based approaches were widely investigated. Bing et al. (2015) proposed an abstractive multidocument summarization framework that can construct new sentences by exploring more fine-grained syntactic units than sentences. Nallapati, Zhou, Santos, Gülçehre, and Xiang (2016) proposed an abstractive text summarization model using attentional encoder-decoder recurrent neural networks. Paulus, Xiong, and Socher (2017) proposed a neural network model with a novel intra-attention that attends over the input and continuously generated output separately. This model combines standard supervised word prediction and reinforcement learning for abstractive summarization. Tan, Wan, and Xiao (2017) proposed a graph-based attention mechanism in the sequence-to-sequence framework. This framework introduced a new hierarchical decoding algorithm with a reference mechanism to generate the abstractive summaries.

Although these existing studies leverage advanced neuro-linguistic programming (NLP) techniques to generate summaries, they require a great amount of training data. For this research problem, for example, there are no training data available. The advantage of our framework is that it is an unsupervised and a data-driven method.

Problem Definition

The raw input to our approach is a set of programming-related sentences extracted from online discussion. Such sentences can be easily obtained from Q&A websites such as Stack Overflow. Given an API type, for example, a class or an interface declared in Java SDK like java.util.HashMap, our task is to distill a small set of sentences as negative caveats related to the concerned API. The problem naturally well aligns with the objectives of an extractive text summarization task, in which the aim is to select salient information from a collection of documents. Thus, we formulate the problem of distilling desirable API negative caveats as a text-summarization task, through the following three definitions.

Definition 1 (candidate sentences)

Let S_{raw} be a set of programming-related sentences from online discussion, and let x be an API type. A candidate sentence for API type x is a sentence $s \in S_{raw}$ that mentions API x and contains negative expression(s). We denote the set of candidate sentences for API x as S_x.

Definition 2 (candidate API negative caveats)

Candidate API negative caveats is a set of sentences, denoted by $Cand_x \subseteq S_x$, after removing context-dependent sentences from candidate sentences.

Definition 3 (desirable API negative caveats)

Given an API type x, a set of desirable API negative caveats is a small subset of sentences, denoted by $\mathcal{A}_x \subseteq Cand_x$. $\mathcal{A}_x$ represents the semantically diverse and nonredundant sentences that cover the most prominent domain-specific terms related to the use issues of API type x.

Next, we detail the four desired properties: context-independence, prominence, semantic diversity, and nonredundancy.

Context-Independence

Sentences in a discussion often reference other part(s) of the discussion; for example, "The following for example addresses this question in some detail about HashMap." We consider such sentences context-dependent. The distilled API negative caveats should be context-independent, so that programmers know a negative caveat without having to refer to the original discussion. Having said that, the property of context-independence does not mean that programmers do not require any additional knowledge about the API to fully understand its negative caveats.

Prominence

Discussion about an API may cover many different aspects, not limited to negative caveats. Consider two sentences "HashMap essentially has O(1) performance" and "HashMap is not synchronized." The first sentence is about time complexity, while the second sentence is our main focus. More specifically, an API negative caveat is usually concerned about some domain-specific terms related to the API use, for instance, *multi-thread, synchronization, thread-safe, sort* for java.util.HashMap. Identifying such prominent domain-specific terms is crucial for important API use issues.

Semantic Diversity

An API type often has negative caveats related to different ways of using the API. For example, java.util.HashMap

DOI: 10.1002/asi

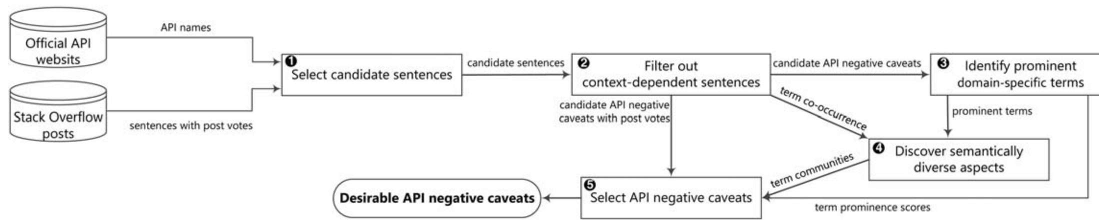

FIG. 1. Overview of the DISCA framework. The output of each step is stated on the corresponding edge.

does not support multi-thread and does not guarantee element order. The number of sentences with different aspects often varies greatly in discussion. This calls for a proper way of handling data imbalance to avoid getting sentences that are all for one aspect of an API type. Distilling semantically diverse sentences reveals a more complete picture of an API's negative caveats.

Semantic Nonredundancy

In a large volume of informal discussion, the same API negative caveat may be mentioned many times but in different wording. "A HashMap does not maintain an order" and "This is the property of HashMap where elements are not iterated in the same order in which they were inserted" express similar meanings but have low lexical similarity. In such cases, we would like to select sentences that convey richer information about the API use; for example, the second sentence in this example. At the same time, we should avoid selecting other sentences that are semantically redundant.

The Disca Approach

To solve the text-summarization problem as defined earlier, we propose DISCA (for DIStilling crowdsourced API negative CAveats), as shown in Figure 1.

Input Data

The input data to DISCA include: a set of APIs identified by their full qualified names and a set of sentences from crowdsourced discussions. In this work, we focus on classes and interfaces defined in Java SDK as the API types of interest.[2] Sentences are extracted from Stack Overflow, considering its popularity among programmers and the volume of the data. More specifically, sentences are extracted for Stack Overflow posts that are tagged with Java. The sentences are cleaned as in many other studies (Ponzetto & Strube, 2007; Robillard & Chhetri, 2015). We preserve the textual content by removing HTML tags, and we remove long code snippets enclosed in *<pre >< code >* but keep short code elements in *<code >*. To help determine the quality of the sentences, we attach post votes to sentences based on the number of votes received by its original post. Voting is a function offered by Stack Overflow where the community users could up- and down-vote for an answer based on its quality. The post votes in this study are the sum of up-votes and down-votes. Higher post votes mean higher quality.

Based on our observations on Stack Overflow, we remove three types of sentences that are unlikely to discuss API negative caveats. First, a discussion thread consists of a question and several answers. The sentences from question are removed, for example "ListView adapter with HashMap isn't displaying correctly," because these sentences are more likely to discuss programming problems rather than the cause of the problems. Our approach only considers the sentences from answers. Second, the interrogative sentences in answers are removed, for example "Have you overridden the keySet() method in your HashMap?"; these sentences pose questions rather than give solutions. Third, we remove opinion-based sentences with subjective opinions in answers, such as "I'm not sure...," "I do not think...," etc.

Selecting Candidate Sentences

Given an API type and a set of sentences, the first step of our approach is to select a set of negative sentences that mention the API type. The selected sentences are treated as candidate sentences from which API negative caveats will be distilled.

Selecting sentences that mention an API. Because we want to distill API negative caveats of an API, the candidate sentences should mention the given API. This task is often referred to as named entity recognition (Tjong, Kim, Sang, & De Meulder, 2003) and entity linking (Shen, Wang, & Han, 2015). As entity recognition and linking are not the key focus of this work, we adopt a name-matching strategy to select sentences that mention a given API. More specifically, we use a software-specific tokenizer (Ye, Xing, Foo, Ang, et al., 2016) to tokenize the sentences. This tokenizer preserves the integrity of code-like tokens like java.util.-HashMap and the sentence structure. If a token in a sentence matches the full or partial name of an API, the sentence is considered mentioning the API. Although this strategy is simple, it has shown effectiveness (precision 0.92 and recall 0.97) in several studies (Treude & Robillard, 2016; Ye, Xing, Foo, Li, & Kapre, 2016; Bacchelli, Cleve, Lanza, & Mocci, 2011; Rahman, Roy, & Lo, 2016) for recognizing mentions of APIs with distinct orthographic features.

When selecting candidate sentences, variations of API mentions have to be taken into account. Informal discussions

[2]https://docs.oracle.com/javase/8/docs/api/overview-summary.html

DOI: 10.1002/asi

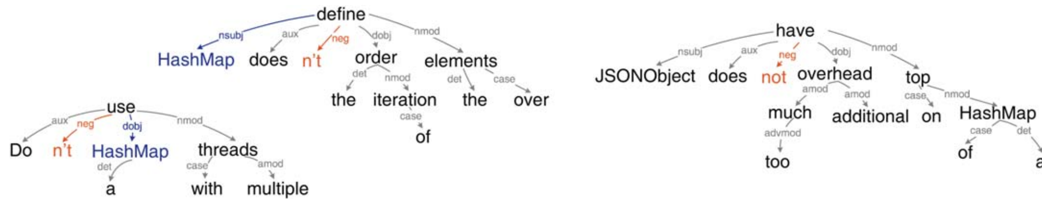

FIG. 2. Three examples of dependency parsing for sentences: "*HashMap doesn't define the order of iteration over the elements*," "*Don't use a HashMap with multiple threads*," and "*JSONObject does not have too much additional overhead on top of a HashMap*." Note that the two examples on the left-hand side are selected as candidate sentences in this study. [Color figure can be viewed at wileyonlinelibrary.com]

on social platforms (for example, Stack Overflow) are contributed by millions of users with diverse technical and linguistic backgrounds (Ye, Xing, Foo, Li, & Kapre, 2016; Chen, Xing, & Wang, 2017). Such informal discussions are full of misspellings and synonyms (Beyer & Pinzger, 2016; Ye, Xing, Foo, Li, & Kapre, 2016; Chen, Xing, & Wang, 2017). Consequently, the same API is often mentioned in many different forms intentionally or accidentally. For example, the mentions of "HashMap" include "hash map," "hashmaps," and "hash-map." We resort to the software-specific synonym thesaurus (C. Chen et al., 2017) to match API-mention variations. This synonym thesaurus documents commonly seen misspellings and synonyms mined from Stack Overflow.

Selecting sentences with negative expressions. API negative caveats are usually expressed in negative sentences, that is, sentences containing negative expressions. To this end, we use a dependency parse tree to detect negative sentences. The dependency parse tree provides a representation of grammatical relations between words in a sentence.[3] It is a directed graph where nodes represent words and edges represent syntactic roles, for example, *nsubj*: nominal subject, *aux*: auxiliary, *det*: determiner, etc. Among these syntactic roles, we can use negation modifier (that is, *neg*) to detect negative expressions. Figure 2 illustrates three examples dependency parse trees produced by Stanford Parser.[4] Syntactic roles of negation, that is, *neg(define, n't)*, *neg(use, n't)*, and *neg(have, not)* can be detected from these two examples (highlighted in orange in Figure 2).

To ensure the negative expressions are on APIs, we select only negative sentences whose subject or object is a given API. For example, both sentences "JSONObject does not have too much additional overhead on top of a HashMap" and "HashMap doesn't define the order of iteration over the elements" are negative sentences and both mention HashMap. Only the second sentence is selected as a candidate sentence for HashMap because the negative expression is on the API. More specifically, a given API must exist in *nsubj* or *dobj* syntactic role in a sentence, as highlighted in blue in Figure 2.

[3]http://universaldependencies.org/en/dep/all.html
[4]http://nlp.stanford.edu/software/lex-parser.html

Filtering Out Context-Dependent Sentences

Context-dependent sentences are less meaningful without referring to the original discussion where the sentences appear. We remove context-dependent sentences from the candidate sentences, based on a set of predefined sentence patterns.[5] The patterns are defined from our observation made on the data. The first category of patterns removes sentences that reference code snippets in the discussion, such as "An equivalently synchronized HashMap can be obtained by... some code...." The second category removes sentences that reference to demonstrative pronoun (for example, "do this," "like this," "this won't," etc.); for example, "If you are trying to do this in a single thread, I would recommend HashMap." The third category removes sentences that reference another sentence in the discussion (for example, "see the next step," "the following," etc.); for example, "The following for example addresses this question in some detail: HashMap requires a better hashCode()." We refer to the rest of context-independent sentences as candidate API negative caveats, denoted by $Cand_x$.

Identifying Prominent Domain-Specific Terms

An API negative caveat is usually concerned with domain-specific terms related to the particular API use. Identifying prominent terms in candidate API negative caveats helps to distill frequently overlooked but important API use issues.

Inspired by Park, Patwardhan, Visweswariah, and Gates (2008) and C. Chen et al. (2017), we identify prominent terms by contrasting term frequency of a term in candidate API negative caveats and its frequency in background corpus. Recall that $Cand_x$ represents the set of candidate API negative caveats for API type x. For the sentences in $Cand_x$, we build a term (unigram) vocabulary V_x after removing stop words and performing word stemming. For a term $t \in V_x$, we use relative entropy to weight its prominence: $w(t)=p(t)\log\frac{p(t)}{q(t)}$, where $p(t)$ is the probability of observing t in $Cand_x$ and $q(t)$ is probability of observing t in all Stack Overflow posts that are tagged with the corresponding programming language, that is, Java in our setting. Based on the term weight, we select the top-k ($k = 100$ in this work)

[5]See the full list of defined patterns at http://128.199.241.136/disca/appendix

DOI: 10.1002/asi

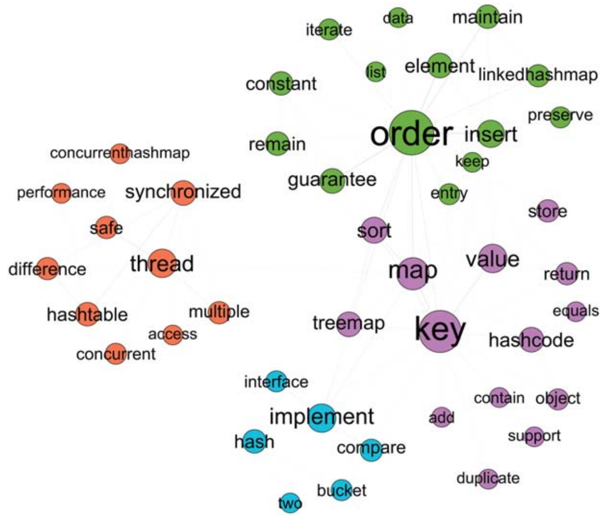

FIG. 3. Term communities shown in different colors identified from the term co-occurrence graph of java.util.HashMap. [Color figure can be viewed at wileyonlinelibrary.com]

ranked terms as the prominent domain-specific terms in candidate API negative caveats. Note that the setting of k may not significantly affect the results, as the prominent terms will be grouped to semantic aspects; to be discussed next.

Discovering Semantically Diverse Aspects

A group of semantically-related terms together reveal a semantic aspect of API uses, for example, (*thread, synchronization, safe*) for the issue of using java.util.HashMap in multi-thread settings, (*key, hashcode, equal*) for the element uniqueness issue of java.util.HashMap, and (*order, insert, iterate*) for the element ordering issue of java.util.HashMap. Clustering semantically related prominent terms helps to discover semantic aspects of an API, which in turn help to distill semantically diverse API negative caveats.

Semantic relatedness between terms can be discovered from term co-occurrence in sentences (Hua, Wang, Wang, Zheng, & Zhou, 2015; Lund & Burgess, 1996). To capture semantic relatedness between all prominent terms, we construct a term co-occurrence graph, where nodes are prominent terms and edges reflect the frequencies of term co-occurrences in candidate API negative caveats. An edge is added between two terms if their co-occurrence frequency is above a threshold. To discover the different aspects of an API, we cluster prominent terms in the graph into a set of disjoint term communities. In particular, we use the Louvain method (Blondel, Guillaume, Lambiotte, & Lefebvre, 2008). It iteratively optimizes local communities until global modularity no longer improves. Figure 3 shows the community detection results for prominent terms of API java.util.HashMap in our evaluation. This graph is constructed from the top-100 prominent terms in the candidate API negative caveats, and the edge co-occurrence frequency threshold is set to 3. The node size is proportional to the degree centrality of the node in the graph. Observe that the detected term communities are semantically diverse (highlighted in different colors in Figure 3). Each term community represents one key semantic aspect of java.util.HashMap, including comparator implementation, element order, key/hashcode, and multiple threads.

Selecting API Negative Caveats

The final step of **Disca** is to select sentences to represent each term community discovered in the earlier step. We select sentences based on three intuitions: (i) prominence: the selected sentences should be as prominent as possible; (ii) quality: sentences should be of high quality, preferred from highly voted answer posts; and (iii) nonredundancy: the selected sentences should minimize redundant information.

Based on the three intuitions, we formulate the selection of desirable API negative caveats as a weighted *set cover* problem. Given an API type x, let $T_x=\{t_1, t_2, \ldots, t_N\}$ be a set of N prominent terms in its term co-occurrence graph. Assume M term-communities are detected in the term co-occurrence graph, that is, $\Phi_x=\{C_1, C_2, \ldots C_M\}$ $(C_i \cap C_j=\emptyset)$. For a term-community $C_m \in \Phi_x$, there are K prominent terms in this community, that is, $C_m=\{t_1^m, t_2^m, \ldots, t_K^m\}$, where the superscript m indicates the m-th community, $t_K^m \in T_x$ and $\cup C_m=T_x$. Recall that $Cand_x$ is a set of candidate API negative caveats for API type x. For each sentence $s_i \in C\,and_x$, we represent s_i as a set of prominent terms in the sentence, that is, $s_i=\{t_1^i, t_2^i, \ldots, t_W^i\}$ where the superscript i indicates the i-th sentence and $t_W^i \in T_x$. The two sentences in $Cand_x$ may have overlapping terms. Each sentence s_i has a $cost(s_i)$. The goal is to find a set cover $\mathcal{A}_x \subseteq C\,and_x$ of minimal total sentence weight to cover all terms in T_x, that is,

$$\begin{aligned} &Minimize \sum_{s_i \in \mathcal{A}_x} cost(s_i) \\ &Subject\,to \bigcup_{s_i \in \mathcal{A}_x} s_i=T_x \end{aligned} \tag{1}$$

The $costs(s_i)$ is computed from two parts: (i) average value of the prominence score of terms in sentence s_i by $w(t)$ defined earlier, denoted by $prom(s_i)$, and (ii) post score $post(s_i)$, the user votes of the post that contains sentence s_i. These two scores are normalized independently based on their corresponding maximum and minimal values. Then $costs(s_i)$ is a linear combination of the two scores.

$$cost(s_i)=-\alpha \cdot prom(s_i)-\beta \cdot post(s_i) \tag{2}$$

where α and β are the coefficients and $\alpha+\beta=1$. The intuitive interpretation of Equation (2) is that the selected sentences should be as prominent as possible and have been up-voted by many users. The two coefficients control the trade-off between prominence and quality of sentences (set equal in this work).

DOI: 10.1002/asi

Algorithm 1: Greedy Selection Algorithm

```
Input: Candidate API negative caveats Cand_x; prominent terms T_x; term communities Φ_x
Output: A_x: Map of API negative caveasts for term communities
1  A_x = ∅;
2  foreach C_m in Φ_x do                                    // iterate through communites
3  |  ψ_m = ∅ ;                                             // selected sentences for C_m
4  |  while ψ_m ≠ C_m do
5  |  |  foreach s_i in Cand_x and s_i ⋂ C_m ≠ ∅ do    //  s_i ⋂ C_m = ∅ ⇔ s_i is completely redundant
6  |  |  |  Compute cost(s_i) by Equation 2;
7  |  |  |  Compute p(s_i) = cost(s_i) · |s_i \ ψ_m| ;  // update contribution of s_i for current loop
8  |  |  Select s_j ∈ Cand_x where p(s_j) has the minimal value;
9  |  |  ψ_m ← ψ_m ⋃ {s_j} ;                                  // select s_j
10 |  |  Cand_x ← Cand_x \ {s_j} ;                            // remove s_j from Cand_x
11 |  A_x ← A_x ⋃ {map⟨C_m, ψ_m⟩} ;                           // map⟨key, value⟩
12 return A_x
```

Term community: {thread, multiple, safe, synchronized, access, difference, hashtable, concurrent, concurrenthashmap, performance}.

⇓

1. HashMap is not thread safe for concurrent access.
2. The important difference between HashTable and HashMap is performance, since HashMap is not synchronized it perform better than HashTable.
3. Don't use a HashMap if you are going to have multiple threads, use a ConcurrentHashMap instead.

FIG. 4. Selected API negative caveats for the term community "*multiple thread synchronized*".

The weighted set cover problem in Equation (1) is NP-hard (Aho & Hopcroft, 1974). But there is a polynomial time greedy approximate algorithm, which provides a $O(\log n)$ approximate solution (Blondel et al., 2008). Algorithm 1 shows the steps of this greedy approximate algorithm for selecting a set of representative sentences from candidate API negative caveats to satisfy Equation (1). Instead of selecting sentences to cover the term set T_x as a whole, we use a divide-and-conquer strategy that selects sentences for one randomly selected term community at a time (Lines 2–3). This divide-and-conquer strategy, together with nonredundant sentence selection mechanism (Line 7), ensures the semantic diversity of the selected sentences, even though the mentions of API negative caveats related to different semantic aspects of an API are imbalanced.

For a term community C_m, the inner loop (Lines 4–10) continues until the union of prominent terms in the selected sentences ψ_m, covers all terms in C_m. The notation $|s_i \setminus T(\psi_m)|$ (Line 7) denotes the number of terms in s_i that are not in the selected sentences ψ_m. The notation $p(s_i)$ is the production of $cost(s_i)$ and the number of newly added terms if s_i is selected (Line 7). The intuition is that the more new terms brought in by selecting sentence s_i, the more likely the sentence will be selected. A sentence once selected is removed from candidate negative caveats set $Cand_x$ (Line 10). The algorithm returns the map of the selected sentences for each term community as desirable API negative caveats for API x.

Consider the "multiple thread synchronized" term community for java.util.HashMap (in orange color) in Figure 3. Figure 4 shows the API negative caveats selected by Algorithm 1 for this term community. Observe that all three negative caveats are related to *multi thread* and *synchronization* issues of HashMap, and there is no redundancy. Instead, the selected sentences together provide complementary information for better understanding the issues, compared to the sentence "this implementation is not synchronized" in the official API documentation of HashMap. The second and third sentences even provide alternative APIs that are not mentioned in the official documentation.

Evaluation

Our evaluation aims to answer the following five research questions:

- RQ1: How much improvement can the DISCA approach achieve over baseline methods?
- RQ2: How effective is the proposed DISCA approach in guaranteeing diversity over the baseline methods?
- RQ3: To what extent does the DISCA approach miss the API negative caveats stated in official API documentation?
- RQ4: To what extent does the DISCA approach augment the official API documentation?
- RQ5: How important are the distilled API negative caveats by the DISCA approach?

DOI: 10.1002/asi

TABLE 2. The statistics of Java API types used in evaluations.

API types	Mention frequency	$Cand_x$	# Term communities	**Disca**
java.util.ArrayList	55,802	689	4	18
javax.swing.JFrame	27,468	302	6	17
java.lang.NullPointerException	20,079	133	4	24
javax.xml.bind.JAXB	14,445	191	6	16
java.io.IOException	7,223	97	4	18
java.awt.event.ActionListener	7,014	82	5	17
java.sql.ResultSet	6,948	94	6	14
java.text.SimpleDateFormat	6,585	136	5	22
java.math.BigDecimal	6,568	78	3	12
java.nio.ByteBuffer	3,193	26	3	6

Note. "$Cand_x$" and "**Disca**" denote number of candidate API negative caveats and number of negative caveats mined by **Disca**, respectively.

Experimental Settings

Data collection. From the official Java 8.0 website, we obtain 4,240 Java API types. We collect all Stack Overflow posts tagged with Java from the March 2016 data dump as the general corpus. Among the posts, 1,081,439 sentences mention at least one Java API type. For the top 10 most frequently mentioned Java packages, we choose the top 1 frequently mentioned API type in each package in our evaluation. Reported in Table 2 (the first three column), the 10 API types have a wide range of mention frequency (MF) and candidate API negative caveats ($Cand_x$) ranging from tens to hundreds of sentences. We made our data set publicly available.[6]

Parameter setting for Disca. The configuration of **Disca** is based on the performance of a development set (java.util.-HashMap). Accordingly, for each of the 10 API types, we use the top-100 prominent terms in its candidate API negative caveats to construct the term co-occurrence graph with term co-occurrence frequency being set at 3. The resulting term co-occurrence graph has 3 to 6 term-communities. The α and β parameters for $cost(s_i)$ are set to 0.5.

Baseline methods. Our approach is a data-driven approach and is unsupervised, because there are no training data to learn from. This limits our choices of baseline methods, as many recent summarization methods adopt supervised learning and require training data (see Automatic Text Summarization). Thus, we compare **Disca** with four classical text-summarization methods. All methods take candidate API negative caveats for an API type as input, and independently select a subset of sentences as summaries.

- LexRank: This method selects important sentences based on the concept of eigenvector centrality in a graph representation of sentences (Erkan & Radev, 2004). Cosine similarity is used to calculate the similarity between two sentences.
- LDA: This method represents each sentence in a vector space using Latent Dirichlet Allocation (LDA) topic model (Blei et al., 2003). For each topic, the sentence with the maximum probability is selected as an API negative caveat.
- KM: This method represents each sentence with a TF-IDF vector and performs a k-means algorithm (MacQueen et al., 1967) to cluster the sentences, then chooses the centroids in clusters as API negative caveats.
- MMR: This method iteratively selects API negative caveats with the maximal marginal relevance that measures novelty and diversity of the selected sentences (Goldstein et al., 1999).

Evaluation metrics. We use two evaluation metrics, namely, ROUGE and nDCG.

- ROUGE measures the quality of a summary by counting the unit overlaps between a machine-generated summary and a set of gold standard summaries. ROUGE-N is the n-gram recall computed as follows:

$$ROUGE-N=\frac{\sum_{S\in ref}\sum_{gram_n\in S}Count_{match}(gram_n)}{\sum_{S\in ref}\sum_{gram_n\in S}Count(gram_n)} \quad (3)$$

where n represents the length of the n-gram, and ref is the set of the gold standard summaries.

In our evaluation, we used the ROUGE toolkit (Lin, 2004) (v. 1.5.5) with ROUGE-1 (unigram-based) and ROUGE-2 (bigram-based). We also use ROUGE-SU4 that measures unigram recall and skip-bigram recall with maximum skip distance of 4. These three ROUGE measures have been shown to be able to identify the machine-generated summary that is the most correlated with human summaries (Lin & Hovy, 2003; Ganesan, Zhai, & Han, 2010).

- nDCG measures the performance of a ranked list based on graded relevance levels. The main idea of *nDCG* is that the more relevant items should be ranked higher than those less relevant items. It is computed as follows:

$$nDCG@k=\frac{DCG@k}{IDCG}=\frac{1}{IDCG}\sum_{i=1}^{k}\frac{2^{rel_i}-1}{\log_2(i+1)} \quad (4)$$

$DCG@k$ is the Discounted Cumulative Gain accumulated at a particular rank position k. The idea of $DCG@k$ is that highly relevant documents appearing lower in ranking

[6]https://github.com/IRNLPCoder/CaveatDataSet

results should be penalized. *nDCG@k* is the normalized discounted cumulative gain, with respect to *IDCG*, which is the discounted cumulative gain of the ideal ordering of all instances. rel_i is the relevance score of the *i*-th element in the ranked list. LexRank and MMR output a ranked list of sentences. For LDA and KM, we rank the centroids based on cluster size. In **Disca**, Algorithm 1 ranks the API negative caveats for a term community. A ranked list of API negative caveats across communities is then obtained by ranking all term-communities based on the highest degree centrality of the communities. For the relevance judgment, the relevance level of each negative caveat is defined as the number of annotators who select the sentence. For example, the relevance of a negative caveat is 3 if all three annotators select this sentence in their gold standard summary and the relevance is 0 if no annotator selects it.

Gold standard generation. To make a fair comparison, selecting the same number of words or sentences is commonly used in the comparison of text-summarization methods (Wang, Zhu, Li, & Gong, 2012; Ganesan et al., 2010). In our experiments, for each API type in Table 2 we use each of the four baseline methods to select the same number of API negative caveats as **Disca** selects. Then we mix the selected sentences of the five methods for human annotation. The annotators do not know which sentences are from which methods. We recruit three annotators who all have more than 4 years of programming experience in Java and are familiar with the 10 Java types in Table 2. Because there are five methods, we ask each annotator to select 20% of sentences from the mixed sentences, based on the following criteria: (i) the selected sentences should cover prominent and diverse topics, and (ii) the selected sentences should be informative and context-independent.

Overall, 25.9%, 19.2%, 14.5%, 20.3%, and 20.1% of the selected sentences by annotators are from **Disca**, LexRank, LDA, KM, and MMR, respectively. We consider the selected sentences by the three annotators as three independent gold standard summaries for an API type, because the evaluation metric ROUGE can handle multiple gold standard summaries.

Interannotator agreement. The Jackknifing procedure (Lin, 2004; Ganesan et al., 2010) is widely used to estimate average human performance from multiple reference summaries. Thus, we use the Jackknifing procedure to quantitatively assess the interannotator agreement. With this procedure, the ROUGE scores are computed over K sets of $K-1$ reference summaries. That is, each human summary is evaluated against the remaining $K-1$ gold standard summaries, and the average ROUGE scores are computed as reported in Table 3. Observe from the table that the average scores of ROUGE-1, ROUGE-2, and ROUGE-SU4 are 0.7571, 0.6325, and 0.6363, respectively. The largest standard deviation is about 0.0136, which indicates that the ROUGE scores of different annotators are close to the mean.

TABLE 3. Agreement between annotators, by evaluating the summary of one annotator against the summaries of the other two annotators on three ROUGE measures.

Annotator	ROUGE-1	ROUGE-2	ROUGE-SU4
Human A	0.7478	0.6169	0.6231
Human B	0.7609	0.6393	0.6410
Human C	0.7626	0.6414	0.6447
Average	0.7571 ± 0.0081	0.6325 ± 0.0136	0.6363 ± 0.0116

In short, we can see that the annotators have good agreement among themselves.

Quantitative Evaluation

RQ1: How much improvement can the Disca approach achieve over the four baseline methods? Motivation: A novel approach, **Disca,** is proposed in this work to distill API negative caveats from a large amount of unstructured data. Experimental Settings lists four classical text-summarization methods. We would like to investigate whether the proposed **Disca** performs better than the baseline methods in terms of ROUGE and nDCG.

Approach: We have generated gold standard summaries for the Java API types in Table 2, where annotators achieved a good agreement among them. Given gold standard summaries, we compare the performance of **Disca** with the four baselines. Moreover, we apply the paired *t*-test to test the statistical significance of the improvements between **Disca** and baseline methods.

Results: We first report the ROUGE and nDCG scores for the negative caveats produced by each method against the gold standard summaries. Table 4 reports the ROUGE scores of the five methods, and Figure 5 plots nDCG values of these methods at different rank positions. We also list ROUGE-1, ROUGE-2, and ROUGE-SU4 for the 10 Java API types using **Disca** in Table 5. From the results, we made the following observations.

First, **Disca** achieves the best performance against the four baseline methods in terms of all ROUGE scores and all nDCG values. In terms of ROUGE scores, **Disca** achieves 22.47%, 12.25%, 10.66%, and 10.60% improvements over LDA, KM, LexRank, and MMR, respectively. For nDCG, **Disca** achieves 42.87%, 17.65%, 20.01%, and 17.63%

TABLE 4. Performance of the five methods on ROUGE measures.

Method	ROUGE-1	ROUGE-2	ROUGE-SU4
LDA	0.5473	0.3202	0.3550
KM	0.6026	0.3498	0.3836
LexRank	0.6045	0.3555	0.3923
MMR	0.6097	0.3583	0.3868
Disca	**0.6269***	**0.4152***	**0.4374***

Note. The best performance is highlighted in bold face.

*The improvements made by Disca over the best baseline is statistically significant under paired *t*-test with $p \leq .05$.

DOI: 10.1002/asi

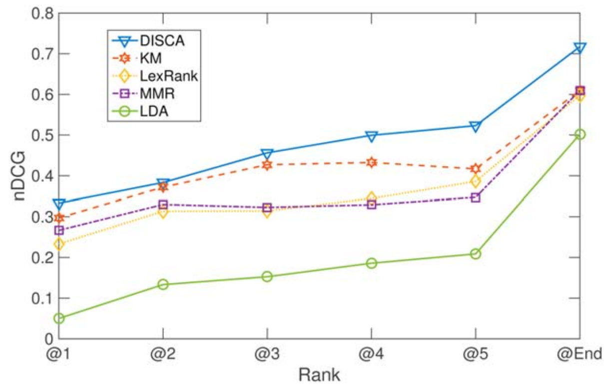

FIG. 5. nDCG values of the five methods at different ranking positions {1, 2, 3, 4, 5, all}. [Color figure can be viewed at wileyonlinelibrary.com]

improvements over the four methods. The improvements are statistically significant for the three kinds of ROUGE scores and nDCG under paired *t*-test with $p \leq .05$. As shown in Figure 5, the higher the nDCG, the better the ranking result. With our approach, the most relevant caveats for an API are ranked at top-most positions leading to a higher nDCG. We attribute this to the fact that **Disca** takes context-independence, prominence, semantic diversity, and semantic nonredundancy into account when selecting desirable API negative caveats.

Second, LDA yields the worst performance in terms of ROUGE scores and nDCG. A challenge in using LDA is to set an appropriate number of topics. In this evaluation, the number of topics is based on the number of API negative caveats that **Disca** distills. From the results of LDA, we note that having too many topics results in the extracted topics to be similar to each other. On the other hand, too few topics makes the extracted topics less meaningful or noninterpretable. Unfortunately, without prior knowledge of the distribution of candidate API negative caveats, it is difficult to set the right topic number. According to the ROUGE scores in Table 4, MMR outperforms KM. That is, the summaries of MMR are closer to gold standard summaries than KM without considering the relevance ranking of selected sentences. However, when considering the relevance ranking of the selected sentences, Figure 5 shows that KM outperforms MMR in nDCG values. This is because the sentences selected by KM are ranked by cluster size that reflects the prominence of the selected sentences. The performance of LexRank is comparable to that of MMR, because LexRank takes into account both relevance ranking and diversity. Recall that LexRank first ranks candidate API negative caveats in a graph model based on sentence similarity. Then it uses a greedy algorithm to select diverse sentences. The main issue of LexRank and MMR is that both are based on sentence-level lexical similarity, which cannot distinguish lexically different but semantically redundant sentences.

TABLE 5. ROUGE scores obtained by Disca on different Java API types.

API types	ROUGE-1	ROUGE-2	ROUGE-SU4
java.util.ArrayList	0.6618	0.4669	0.4855
javax.swing.JFrame	0.6658	0.4033	0.4311
java.lang.NullPointerException	0.6605	0.3761	0.4247
javax.xml.bind.JAXB	0.7296	0.5153	0.5304
java.io.IOException	0.4733	0.2639	0.2848
java.awt.event.ActionListener	0.6587	0.3985	0.4311
java.sql.ResultSet	0.6130	0.3933	0.4128
java.text.SimpleDateFormat	0.5072	0.3302	0.3465
java.math.BigDecimal	0.6188	0.4558	0.4709
java.nio.ByteBuffer	0.6831	0.5512	0.5580
Average	0.6269	0.4152	0.4374

Third, although the performance of **Disca** varies for different API types in Table 5, **Disca** outperforms all baseline methods for all API types.[7] We used Pearson's correlation test and the results show that there is no correlation between the number of mentions of a type and the ROUGE score obtained by **Disca**. Among the 10 API types, **Disca** has the best performance for javax.xml.bind.JAXB and the worst performance for java.io.IOException. For javax.xml.bind.-JAXB, most candidate API negative caveats discuss issues related to "*XML document*," "*unmarshal*" and "*marshal*." These repetitive discussions have more n-gram overlap, which leads to higher ROUGE scores. For java.io.IOException, it is a common IO exception class that can be thrown in many different scenarios. As such, the sentences that discuss IOException have the least level of overlap, which leads to lower ROUGE score.

RQ2: How effective is the proposed Disca approach in guaranteeing diversity over the four baseline methods?. Motivation: As shown in Discovering Semantically Diverse Aspects, **Disca** discovers semantically diverse aspects using a graph clustering technique. Given gold standard summaries, RQ1 investigated the overall performance of **Disca** and the baseline methods. We would like to confirm the ability of **Disca** in guaranteeing diversity over the baseline methods.

Approach: To answer this research question, we conduct an intermethod comparison, which compares the relative performance between one method and the other four methods using the Jackknifing procedure (Lin, 2004). That is, we treat the summary of one method as a machine-generated summary, and the summaries of the other four methods as reference summaries. Then we measure the performance of each combination in terms of ROUGE score.

Results: Table 6 reports the results of the intermethod comparison. First, LexRank and MMR outperform LDA and KM. This is in line with our previous analysis that LexRank and MMR take into account both relevance ranking and diversity, while LDA and KM do not. Second, the table shows that **Disca** achieves the best performance in all ROUGE scores when the summaries of the other four baseline methods are used as reference summaries. In contrast, the performance of the other four baseline methods is poorer when the summary of **Disca** is used as a reference summary.

[7]Detailed results not reported for the interests of page space.

DOI: 10.1002/asi

TABLE 6. Results of intermethod comparison, based on ROUGE measures.

Method	ROUGE-1	ROUGE-2	ROUGE-SU4
LDA	0.5368	0.2617	0.3051
KM	0.5419	0.2718	0.3142
MMR	0.5724	0.3073	0.3537
LexRank	0.5859	0.31357	0.3569
DISCA	**0.6098***	**0.3431***	**0.3839***

Note. For each method in first column, the summaries of the other four methods are treated as its reference summaries. *The improvements made by **DISCA** over the best baseline is statistically significant using paired t-test with $p \leq .05$.

These results indicate that none of the baseline methods can well cover API negative caveats that **DISCA** distills, but **DISCA** covers theirs. We attribute this to the fact that **DISCA** can distinguish lexically different but semantically redundant– sentences (see Discovering Semantically Diverse Aspects), while LexRank and MMR only consider sentence-level lexical similarity. In short, the summaries of **DISCA** are more diverse than the baseline methods.

Qualitative Analysis

RQ3: To what extent does the DISCA approach miss the API negative caveats stated in official API documentation? Motivation: Some API negative caveats are documented when API designers wrote software documentation. The distilled caveats by **DISCA** are from crowd-generated Q&A discussions. We are interested to know how many caveats stated in official API documentation the proposed **DISCA** may miss.

Approach: To answer this research question, we compare the API negative caveats mentioned in official API documentation and those distilled by **DISCA**. We recruited two developers to read the official documentation of the 10 Java API types to annotate the API negative caveats mentioned in these documents. If a distilled API negative caveat and a mentioned API negative caveat both discuss the same aspect of a given API type, we manually judge that they match each other. For inconsistent judgments, the two developers reached a consensus through discussion.

TABLE 7. The three columns show the numbers of negative caveats that are mentioned in official documentation, distilled by **DISCA**, and the matched between the two.

API types	Mentioned	Matched	DISCA
java.util.ArrayList	1	1	18
javax.swing.JFrame	1	0	17
java.lang.NullPointerException	0	0	24
javax.xml.bind.JAXB	0	–	16
java.io.IOException	0	–	18
java.awt.event.ActionListener	0	–	17
java.sql.ResultSet	1	1	14
java.text.SimpleDateFormat	2	2	22
java.math.BigDecimal	1	0	12
java.nio.ByteBuffer	0	–	6
Sum	6	4	164

Result: Table 7 reports the results of this comparative study. The three columns show the numbers of negative caveats that are mentioned in official documentation, distilled by **DISCA**, and the matched ones. Observe that only 5 out of 10 official documentation mention negative caveats and in total six negative caveats are mentioned. **DISCA** manages to identify four out of the six mentioned negative caveats. One missed negative caveat is about rounding behavior of java.-math.BigDecimal class.[8] The other missed negative caveat is from javax.swing.JFrame.[9] By checking our data set, we find that none of the candidate API negative caveats for Big-Decimal mention "rounding mode" or relevant concepts; the same observation holds for JFrame. Searching Stack Overflow website using queries "*BigDecimal rounding mode*" and "*JFrame serialized objects*" results in 105 and 45 posts, respectively. We did not find any negative sentences discussing the two issues in the search results. The results suggest that the two missed API negative caveats have not been well discussed on Stack Overflow.

RQ4: To what extent does the DISCA approach augment the official API documentation? Motivation: RQ3 reveals that two out of six negative caveats were missed by **DISCA**. However, little is known about how many of the negative caveats distilled by **DISCA** are false-positive instances. Moreover, we would like to investigate to what extent the **DISCA** approach augments the official API documentation.

Approach: We recruited two developers to examine the distilled negative caveats to find the false positives. The annotation was done individually by the two developers and for inconsistent judgments, the two developers reached a consensus through discussion.

Results: For the 10 API types, **DISCA** distills 164 negative caveats related to 46 semantic aspects of the 10 API types. Recall that each term-community is considered a semantic aspect of an API, and it may have several complementary API negative caveats (see Figure 4). The annotation finds that there are only 18 false-positive instances. There are two main reasons for false-positive instances. First, **DISCA** distills seven programming exceptions as API negative caveats. For example, one of the false positive instances is "*Exception: IOException is not compatible with throws clause in Plants.eat()*." This is an exception related to the implementation of a specific program, Plants.eat(), but not the API IOException. Second, 11 false-positive instances are context-dependent sentences that our sentence filtering patterns fail to filter out. For example, in the sentence "you cannot attach an ActionListener without having to rewrite the controller and the view," "the controller" and "the view"

[8]If no rounding mode is specified and the exact result cannot be represented, an exception is thrown. https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html

[9]Serialized objects of this class will not be compatible with future Swing releases. https://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html

DOI: 10.1002/asi

TABLE 8. Examples of API negative caveats mined by **Disca**.

java.util.HashMap
(1.1) HashMap **does not** provide any guarantees of order among its entries.
(1.2) HashMap **is not** thread safe for concurrent access.
(1.3) HashMap **does not** need keys to be Comparable but still implements Map interface.
(1.4) HashMap **cannot** store multiple values for the same key.
javax.swing.JFrame
(2.1) **Don't** extend a JFrame, but instead create a local JFrame variable and use it.
(2.2) JFrame **isn't** focusable JComponent, you would need to use focusable contianer for example, JPanel.
(2.3) JFrame **does not** extend JComponent and **does not** have a paintComponent method.
(2.4) You **shouldn't** set a JFrame visible until all the components have been added.
(2.5) Calling validate on a top-level component (JWindow, JDialog, JFrame) **will not** necessarily resize that component.
(2.6) **Don't** call JFrame#setSize(..) on JFrame rather just call JFrame#pack() before setting JFrame visible.
java.awt.event.ActionListener
(3.1) You **can't** add an ActionListener to a JFrame, it does not function like a button and so has no action listeners.
(3.2) An ActionListener **can't** be added to a JPanel, as a JPanel itself does not lend itself to create what is considered to be "actions."
(3.3) **Don't** implement ActionListener in top classes, use anonymous classes or private classes instead.
(3.4) **Don't** implement single ActionListener for multiple components.
(3.5) An ActionListener **cannot** distinguish states on it's own, it simply responds to a user input.
java.math.BigDecimal
(4.1) BigDecimal **is not** a primitive type.
(4.2) BigDecimal **cannot** support numbers that cannot be written as a fixed length decimal, for example, 1/3.
(4.3) Unlike Integer and Double, BigDecimal **does not** participate in autoboxing.

refer to other parts of the discussion. As such, this sentence is hard to understand without additional context. Although there is a small percentage (about 11%) of false-positive instances, **Disca** distills 146 correct API negative caveats that can drastically augment the 10 official API documents.

RQ5: How important are the distilled API negative caveats by the Disca approach? Motivation: RQ4 reveals that **Disca** greatly augments the official API documentation. However, readers may not have an intuitive impression for these distilled caveats. We now show the importance of the distilled API negative caveats through intuitive examples.

Approach: To answer this research question, we conducted a case study to present the distilled API negative caveats of four Java API types. We selected three API types in Table 2: javax.swing.JFrame, java.awt.event.ActionListener, and java.math.BigDecimal, which are mentioned frequently, moderately, and relatively infrequently. We also include java.util.HashMap, which has been used to illustrate our approach throughout our discussion. For the four API types, **Disca** detects 4, 6, 5, and 3 term-communities, respectively. For each term-community, we list the top-1 ranked API negative caveat as a representative caveat in Table 8.

Results: For java.util.HashMap, its long official documentation mentions three negative caveats related to element order, multiple threads synchronization, and comparable element. Only the sentence for multiple threads is in bold text. Compared with the lengthy official documentation, the API negative caveats (1.1), (1.2), and (1.3) in Table 8 show that **Disca** helps to reveal hard-to-notice API negative caveats. Furthermore, **Disca** augments the official documentation with caveat (1.4) about multiple values for the same key. This may seem natural, but is often overlooked. Making it explicit provides an important reminder for novice developers.

For javax.swing.JFrame, all the mined API negative caveats by **Disca** do not exist in its Javadoc. Caveats (2.1) and (2.2) caution users about not extending JFrame and JFrame being not focusable. Moreover, they give alternative solutions at the same time. Caveat (2.3) emphasizes that JFrame has no paintComponent method. Caveats (2.4), (2.5), and (2.6) are on the issues related to setting JFrame visible, validating JFrame, and setting JFrame size, respectively. These API negative caveats are difficult to document by API designers, because they mainly emerge from misuse in practice.

For java.awt.event.ActionListener, its Javadoc does not mention any API negative caveats. **Disca** distills five negative caveats. Caveats (3.1) and (3.2) not only caution users that they cannot add ActionListener to JFrame or JPanel but also explain the reason behind. Caveats (3.2), (3.3), and (3.5) are good but implicit coding practices when implementing ActionListener. It is infeasible for API designers to take all these aspects into account when documenting API, because these API negative caveats can only be accumulated in practice.

For java.math.BigDecimal, although **Disca** does not find API negative caveat regarding round mode of BigDecimal, it finds three other negative caveats. Caveat (4.1) warns developers that "*BigDecimal is not a primitive type*." Similarly, caveats (4.2) and (4.3) provide two important use cautions about "*fixed length decimal*" and "*autoboxing*." These to-be-avoided use contexts are hard to foresee.

Threats to Validity

This section outlines potential threats to the validity of this study.

DOI: 10.1002/asi

Use of Data

All APIs investigated in our evaluation are Java JDK APIs, and our evaluation uses only Stack Overflow discussions. The framework of DISCA makes no specific assumptions about APIs and discussion data. Therefore, it is generally applicable for other programming languages or third-party libraries or other Q&A websites, which we leave as our future work. Our quantitative evaluation is based on gold standard summaries generated by human annotators. They could be biased. But our interannotator agreement analysis suggests that the gold standard summaries used in the evaluation are acceptable.

Coverage of Candidate Sentences

First, DISCA currently uses a simple name-matching strategy to select candidate sentences. Entity linking approaches (Ji, Sun, Cong, & Han, 2016; Ye, Xing, Foo, Li, & Kapre, 2016; Moro, Raganato, & Navigli, 2014) based on machine learning could improve the performance of DISCA, because these approaches can better handle API-mention variations and thus provide more candidate sentences for selection.

Second, based on our observation, important information about a caveat mostly appears in a single sentence on Stack Overflow. DISCA may miss some multiple-sentences caveats because DISCA is designed at the single sentence level. However, it is infeasible to get the number of multiple-sentence caveats without manual annotation of the data. We show one example here: "Iterator returned by HashMap are fail-fast while Enumeration returned by the HashTable are fail-safe. Fail-safe is relevant from the context of iterators. If an iterator has been created on a collection object and some other thread tries to modify the collection object 'structurall,' a concurrent modification exception will be thrown." These three sentences are about the caveat of "thread safe." Although our approach will not extract these three sentences, it will extract "HashMap is not thread safe for concurrent access" from many other single-sentence candidates for this caveat.

Third, the sentence selection process of DISCA may result in missing some caveats because of some strict restrictions (for example, removing subjective opinions in section Input Data, selecting explicitly negative sentences in section Selecting Candidate Sentences, selecting prominent terms in section Identifying Prominent Domain-Specific Terms). Given the thousands of sentences to be examined during the sentence selection process, annotating intermediate results and analyzing influence factors requires much human effort, if even feasible. As many sentences are available for important caveats, our proposed approach is able to pick up the representative sentences even though some sentences are missed.

Conclusion and Future Work

This research identifies a new task of distilling crowdsourced API negative caveats from a large volume of programming-related discussions. We present an effective text-summarization approach to distilling context-independent, prominent, semantically diverse, and nonredundant API negative caveats. Our approach significantly outperforms other text-summarization methods, including the methods that are based on eigenvector centrality of sentence graph, topic modeling, sentence clustering, and sentence diversification. Furthermore, our approach greatly augments official API documentation with crowdsourced API negative caveats and explanations, as well as suggestions (for example, alternative APIs) for solving API use issues. We are developing web applications that can push distilled API negative caveats when developers read API documents. On the other hand, through this, we demonstrate how to cast a domain-specific problem into an interesting text summarization problem, and how to work on every single step in this data-driven framework to achieve the desired result. Our proposed solution opens a way to better support programmers leveraging official API documentation and social discussions in a Q&A website.

As a part of future study, we will mine programming errors related to API negative caveats to develop semantic search systems that can provide direct answers to such errors caused by overlooking API negative caveats.

References

Aho, A.V., & Hopcroft, J.E. (1974). The design and analysis of computer algorithms. Chennai, India: Pearson Education India.

Bacchelli, A., Cleve, A., Lanza, M., & Mocci, A. (2011). Extracting structured data from natural language documents with island parsing. In *Proceedings of the International Conference on Automated Software Engineering* (pp. 476–479).

Bacchelli, A., Ponzanelli, L., & Lanza, M. (2012). Harnessing stack overflow for the ide. In *Proceedings of the ACM Recommender Systems* (pp. 26–30).

Barzilay, R., & Elhadad, M. (1999). Using lexical chains for text summarization. Advances in Automatic Text Summarization, pp. 111–121.

Beyer, S., & Pinzger, M. (2016). Grouping android tag synonyms on stack overflow. In *Proceedings of the International Conference on Mining Software Repositories* (pp. 430–440).

Bing, L., et al. (2015). Abstractive multi-document summarization via phrase selection and merging. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics* (pp. 1587–1597).

Blei, D.M., Ng, A.Y., & Jordan, M.I. (2003). Latent Dirichlet Allocation. Journal of Machine Learning Research, 3(Jan), 993–1022.

Blondel, V.D., Guillaume, J.-L., Lambiotte, R., & Lefebvre, E. (2008). Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment, 2008(10), P10008.

Brandt, J., Dontcheva, M., Weskamp, M., & Klemmer, S.R. (2010). Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 513–522).

Cao, Z., et al. (2015). Learning summary prior representation for extractive summarization. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics* (pp. 829–833).

Chen, C., Xing, Z., & Wang, X. (2017). Unsupervised software-specific morphological forms inference from informal discussions. In *Proceedings of the International Conference on Software Engineering*.

Chen, N., Lin, J., Hoi, S. C., Xiao, X., & Zhang, B. (2014). Ar-miner: mining informative reviews for developers from mobile app

DOI: 10.1002/asi

marketplace. In *Proceedings of the International Conference on Software Engineering* (pp. 767–778).

Dagenais, B., & Robillard, M. P. (2012). Recovering traceability links between an API and its learning resources. In *Proceedings of the International Conference on Software Engineering* (pp. 47–57).

Erkan, G., & Radev, D.R. (2004). Lexrank: Graph-based lexical centrality as salience in text summarization. Journal of Artificial Intelligence Research, 22, 457–479.

Ganesan, K., Zhai, C., & Han, J. (2010). Opinosis: a graph-based approach to abstractive summarization of highly redundant opinions. In *Proceedings of COLING* (pp. 340–348).

Genest, P.-E., & Lapalme, G. (2012). Fully abstractive approach to guided summarization. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short for Examples-Volume 2* (pp. 354–358).

Goldstein, J., Kantrowitz, M., Mittal, V., & Carbonell, J. (1999). Summarizing text documents: sentence selection and evaluation metrics. In *Proceedings of the Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 121–128).

Gu, X., & Kim, S. (2015)." what parts of your apps are loved by users?"(t). In *Proceedings of the International Conference on Automated Software Engineering* (pp. 760–770).

Harabagiu, S.M., & Lacatusu, F. (2002). Generating single and multi-document summaries with gistexter. In *Proceedings of the Document Understanding Conferences* (pp. 11–12).

Hua, W., et al. (2015). Short text understanding through lexical-semantic analysis. In *Proceedings of the International Conference on Data Engineering* (pp. 495–506).

Ji, Z., Sun, A., Cong, G., & Han, J. (2016). Joint recognition and linking of fine-grained locations from tweets. In *Proceedings of WWW* (pp. 1271–1281).

Kim, J., Lee, S., Hwang, S.-w., & Kim, S. (2009). Adding examples into java documents. In *Proceedings of the International Conference on Automated Software Engineering* (pp. 540–544).

Kramer, D. (1999). API documentation from source code comments: a case study of javadoc. In *Proceedings of SIGDOC* (pp. 147–153).

Lee, C.-S., Jian, Z.-W., & Huang, L.-K. (2005). A fuzzy ontology and its application to news summarization. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 35(5), 859–880.

Li, J., Sun, A., & Xing, Z. (2018). Learning to answer programming questions with software documentation through social context embedding. Information Sciences, 448–449, 36–52.

Li, J., Xing, Z., & Kabir, A. (2018). Leveraging official content and social context to recommend software documentation. IEEE Transactions on Services Computing, doi: 10.1109/TSC.2018.2812729.

Li, J., Xing, Z., Ye, D., & Zhao, X. (2016). From discussion to wisdom: web resource recommendation for hyperlinks in stack overflow. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (pp. 1127–1133).

Lin, C.-Y. (2004). Rouge: A package for automatic evaluation of summaries. In *Proceedings of ACL-04 Workshop* (Vol. 8).

Lin, C.-Y., & Hovy, E. (2003). Automatic evaluation of summaries using n-gram co-occurrence statistics. In *Proceedings of NAACL HLT* (pp. 71–78).

Lund, K., & Burgess, C. (1996). Producing high-dimensional semantic spaces from lexical co-occurrence. Behavior Research Methods, Instruments, & Computers, 28(2), 203–208.

MacQueen, J., et al. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability* (Vol. 1, pp. 281–297).

Matsuo, Y., & Ishizuka, M. (2004). Keyword extraction from a single document using word co-occurrence statistical information. In Proccedings of the International Journal on Artificial Intelligence Tools, 13(01), 157–169.

Miao, Q., Li, Q., & Zeng, D. (2010). Fine-grained opinion mining by integrating multiple review sources. Journal of the Association for Information Science and Technology, 61(11), 2288–2299.

Moro, A., Raganato, A., & Navigli, R. (2014). Entity linking meets word sense disambiguation: a unified approach. Transactions of the Association for Computational Linguistics, 2, 231–244.

Nallapati, R., Zhou, B., Santos, C.N. dos, Gülçehre, Ç., & Xiang, B. (2016). Abstractive text summarization using sequence-to-sequence rnns and beyond. In *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning, CoNLL 2016, Berlin, Germany, August 11–12, 2016* (pp. 280–290).

Park, Y., Patwardhan, S., Visweswariah, K., & Gates, S.C. (2008). An empirical analysis of word error rate and keyword error rate. In *Proceedings of Interspeech* (pp. 2070–2073).

Parnin, C., Treude, C., Grammel, L., & Storey, M.-A. (2012). Crowd documentation: Exploring the coverage and the dynamics of API discussions on stack overflow. Georgia Institute of Technology, Technical Reports.

Paulus, R., Xiong, C., & Socher, R. (2017). A deep reinforced model for abstractive summarization. CoRR*, abs/1705.04304*.

Ponzanelli, L., Bacchelli, A., & Lanza, M. (2013). Leveraging crowd knowledge for software comprehension and development. In *Proceedings of the European Conference on Software Maintenance and Reengineering* (pp. 57–66).

Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., & Lanza, M. (2014). Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the International Conference on Mining Software Repositories* (pp. 102–111).

Ponzetto, S.P., & Strube, M. (2007). Deriving a large scale taxonomy from Wikipedia. In *Proceedings of AAAI* (Vol. 7, pp. 1440–1445).

Rahman, M.M., Roy, C.K., & Lo, D. (2016). Rack: Automatic API Recommendation Using Crowdsourced Knowledge. In *Proceedings Of The International Conference on Software Analysis, Evolution, and Reengineering* (Vol. 1, pp. 349–359).

Ren, P., et al. (2017). Leveraging contextual sentence relations for extractive summarization using a neural attention model. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 95–104). New York: ACM.

Robillard, M. P., & Chhetri, Y.B. (2015). Recommending reference API documentation. Empirical Software Engineering, 20(6), 1558–1586.

Robillard, M.P., & Deline, R. (2011). A field study of API learning obstacles. Empirical Software Engineering, 16(6), 703–732.

Sahavechaphan, N., & Claypool, K. (2006). Xsnippet: Mining for sample code. ACM Sigplan Notices, 41(10), 413–430.

Serva, R., Senzer, Z.R., Pollock, L., & Vijay-Shanker, K. (2015). Automatically mining negative code examples from software developer Q & A forums. In *Proceedings of the International Conference on Automated Software Engineering* (pp. 115–122).

Shen, W., Wang, J., & Han, J. (2015). Entity linking with a knowledge base: Issues, techniques, and solutions. IEEE Transactions on Knowledge and Data Engineering, 27(2), 443–460.

Stylos, J., Faulring, A., Yang, Z., & Myers, B.A. (2009). Improving API documentation using API usage information. In *Proceedings of VL/HCC* (pp. 119–126).

Subramanian, S., Inozemtseva, L., & Holmes, R. (2014). Live API documentation. In *Proceedings of the International Conference on Software Engineering* (pp. 643–652).

Tan, J., Wan, X., & Xiao, J. (2017). Abstractive document summarization with a graph-based attentional neural model. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics* (pp. 1171–1181).

Thummalapenta, S., & Xie, T. (2007). Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering* (pp. 204–213).

Tjong Kim Sang, E.F., & De Meulder, F. (2003). Introduction to the CONLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of HLT-NAACL* (pp. 142–147).

Treude, C., Barzilay, O., & Storey, M.-A. (2011). How do programmers ask and answer questions on the web?: Nier track. In *Proceedings of the International Conference on Software Engineering* (pp. 804–807).

Treude, C., & Robillard, M.P. (2016). Augmenting API documentation with insights from stack overflow. In *Proceedings of the International Conference on Software Engineering* (pp. 392–403).

Vu, P.M., Nguyen, T.T., Pham, H.V., & Nguyen, T.T. (2015). Mining user opinions in mobile app reviews: A keyword-based approach (t). In *Proceedings of the International Conference on Automated Software Engineering* (pp. 749–759).

Vu, P.M., Pham, H.V., Nguyen, T.T., et al. (2016). Phrase-based extraction of user opinions in mobile app reviews. In *Proceedings of the International Conference on Automated Software Engineering* (pp. 726–731).

Wang, D., Zhu, S., Li, T., & Gong, Y. (2012). Comparative document summarization via discriminative sentence selection. ACM Transactions on Knowledge Discovery from Data, 6(3), 12.

Wisniewski, P., Xu, H., Lipford, H., & Bello-Ogunu, E. (2015). Facebook apps and tagging: The trade-off between personal privacy and engaging with friends. Journal of the Association for Information Science and Technology, 66(9), 1883–1896.

Ye, D., et al. (2016). Software-specific named entity recognition in software engineering social content. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering* (Vol. 1, pp. 90–101).

Ye, D., Xing, Z., Foo, C.Y., Li, J., & Kapre, N. (2016). Learning to extract API mentions from informal natural language discussions. In *Proceedings of the International Conference on Software Maintenance and Evolution* (pp. 389–399).

Zagalsky, A., Barzilay, O., & Yehudai, A. (2012). Example overflow: Using social media for code recommendation. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering* (pp. 38–42).

DOI: 10.1002/asi