Classifying Stack Overflow Posts on API Issues

Md Ahasanuzzaman†, Muhammad Asaduzzaman*, Chanchal K. Roy*, Kevin A. Schneider*
Queen's University†, University of Saskatchewan*, Canada
md.ahasanuzzaman@queensu.ca,{md.asad, chanchal.roy, kevin.schneider}@usask.ca

***Abstract*—The design and maintenance of APIs are complex tasks due to the constantly changing requirements of its users. Despite the efforts of its designers, APIs may suffer from a number of issues (such as incomplete or erroneous documentation, poor performance, and backward incompatibility). To maintain a healthy client base, API designers must learn these issues to fix them. Question answering sites, such as Stack Overflow (SO), has become a popular place for discussing API issues. These posts about API issues are invaluable to API designers, not only because they can help to learn more about the problem but also because they can facilitate learning the requirements of API users. However, the unstructured nature of posts and the abundance of non-issue posts make the task of detecting SO posts concerning API issues difficult and challenging.**

In this paper, we first develop a supervised learning approach using a Conditional Random Field (CRF), a statistical modeling method, to identify API issue-related sentences. We use the above information together with different features of posts and experience of users to build a technique, called *CAPS*, that can classify SO posts concerning API issues. Evaluation of *CAPS* using carefully curated SO posts on three popular API types reveals that the technique outperforms all three baseline approaches we consider in this study. We also conduct studies to test the generalizability of *CAPS* results and to understand the effects of different sources of information on it.

***Index Terms*—API Issue, unstructured data mining, text classification, feature extraction, Stack Overflow**

I. Introduction

Developers depend on frameworks and libraries for effective delivery of software in a timely manner. This is supported through Application Programming Interfaces (APIs) of those frameworks and libraries that provide access to the implemented functionality. For example, the Java Software Development Kit comes with thousands of components that the developers can reuse in their projects. This saves both development time and effort [1]. API designers must work hard to make their APIs accessible to its users. This not only ensures the business success of API providers/designers but also enables them to maintain a healthy satisfied user-base. Towards this goal, API designers need to provide development tools, documentation, and tutorials to support working with their APIs. Despite all these efforts, APIs may suffer from several issues. These include but are not limited to documentation error (including outdated or incomplete documentation), poor memory management, breaking changes that lead to backward incompatibility, and incompatibility of the APIs with underlying operating systems or other external libraries [2]. All these may lead to incorrect use of APIs, introduce bugs and security problems. The rapid changes in APIs does not give the designers much time to validate various changes and thus create confusion among its users [3]. It may also introduce faults in designing APIs, introduce usability issues, and ultimately leads to incorrect behaviour in applications using those APIs. API designers need to learn about these issues of using APIs to fix the problems and to find effective ways of informing developers about various API changes.

API related issues can be learned by mining bug repositories [4], newsgroups [5] and emails of developers [6]. However, they are not the only places to look for API issues. Nowadays, developers rely on online forums and question-answering sites to discuss about issues of APIs, ask questions and seek help from others. While many question answering sites (such as Yahoo Answers[1] and Quora[2]) allow users to ask questions on any topics they are interested in, Stack Overflow (SO) particularly focus on programming related questions. Thus, API issues discussed in SO are of great interest to API designers. However, extracting SO posts concerning API issues is a non-trivial task. This is mostly due to the presence of millions of questions, many of which are not related to API issues, and also due to the unstructured nature of the posts. Keyword searching is not an efficient solution to the problem because of the presence of a large number of *how-to* and *newbie* questions [7] that introduce a lot of noise. This motivates us to investigate the problem further.

We model the problem of identifying posts discussing API issues as a binary classification problem. Our goal is to separate these issue-related posts from the others. Towards this goal, we develop a supervised learning technique using Conditional Random Field (CRF) [8], that identifies API issue-sentences in a post. We not only collect features from the output of CRF but also combine that with different features of posts and user experience to build a classification technique, called *CAPS*. Evaluation of our technique using the corpus reveals that *CAPS* outperforms all three baseline approaches we consider in our study. Finally, we test the generalizability of *CAPS* and also study the effects of different sources of information on it.

To the best of our knowledge, the study most relevant to ours is that of Wang et al. [9]. They develop a mechanism to distill and rank SO posts that are likely to concern API related issues. They select those posts which are asked by the expert users as the candidate issue-related posts. While the technique is useful, it suffers from the problem of missing API issue-related posts that are not asked by experts. During our manual analysis,

[1]https://answers.yahoo.com/
[2]https://www.quora.com/

we found examples of several API issue-related questions that were asked by SO users with low reputation. For example, a low reputed user posted a question[3] in August, 2014 that is related to the unexpected behaviour in *JUnit* API. After six months, this was opened as a potential issue in the *JUnit* issue tracker labeled as *bug*[4].

Thus, our paper makes the following contributions.

- A supervised approach using Conditional Random Field (CRF) that can be used to identify API issue-related sentences in a post.
- A classifier that is created by combining the output of a CFR-based supervised learning technique, a diverse set of features from SO posts and the experience of SO users.
- An evaluation of our proposed technique against three other baseline approaches that consider different sources of information.
- A set of studies to test the generalizability of *CAPS* results and to discover the effects of different sources of information on it.

The remainder of the paper is organized as follows. Section II briefly describes previous work related to our study. Section III provides background of our work. We characterize the SO posts concerning API issues in Section IV. We describe our proposed technique in Section V. Section VI presents evaluation results. We discuss the key issues related to our study in Section VII. Section VIII summarizes threats to the validity of our work and Section IX concludes the paper.

II. Related Work

A number of research have been performed to characterize different facets of Stack Overflow. This includes question quality analysis [10], modeling difficulties of questions [11], low-quality post detection [12], topic distribution [13], patterns of asking and answering questions [7] and personality trait of users [14]. To facilitate developers, a number of recommendation systems are developed using SO data. For example, Bacchelli et al. [15] integrated crowd knowledge in the IDE by developing an Eclipse plugin, called *Seahawk*, that links relevant discussions to the source code. Ponzanelli et al. developed an Eclipse plugin, called *Prompter*, that can automatically retrieve relevant SO discussions by giving a context in the IDE [16]. Asaduzzaman et al. [17] conducted a qualitative study to categorize the unanswered questions. Correa and Sureka conducted an experimental study to analyze and predict the closed questions of SO [18]. In another study, they characterized the deleted questions in SO and build a predictive model to detect deleted questions at their creation time [19]. However, none focuses on the classification of API issue posts in SO. A study on SO addressed the detection of user issues and request types [20]. Their primary goal was to categorize the sentences in anomaly, how to, property and explanation categories using discourse analysis. While they focus on user issues, we focus on API issues in our work.

API learning difficulties and other issues have been investigated in many studies and the prime reasons are problematic features, API evolution and learning obstacles. Robillard [1] conducted a study of the obstacles faced by Microsoft developers when learning how to use APIs. Robillard and Deline [21] identified that inadequate API documentation and API structure are the top two obstacles in using APIs. Robbes et al. [22] conducted an empirical study on the actual incidence of the API changes and API deprecations, causing ripple effect in practice. The study shows that deprecation messages are not always helpful because of the absence of the guidelines and unclear instructions. Ho and Li [5] analyzed 172 discussions collected from a forum and identified a set of API learning obstacles. Zibran et al. [2] identified 22 factors as the API usability issues. Wang and Godfrey [23] analyzed Android and IOS developer questions on SO to detect API usage obstacles. However, the objectives of these studies are different from ours. While they focus on problems that cause API issues, our study focuses on detecting SO posts concerning API issues. Wang et al. [9] developed a methodology to recommend API design-related issues combining expert identification, topic mining and selection of late answered questions. However, their methodology only considers questions which are answered late and submitted by the expert users having more participation in SO. While the technique is useful, it may miss API issue-related posts that are not asked by expert users.

Conditional Random Fields (CRFs) [8] have been used in many natural language applications including parts-of-speech tagging and entity linking [24]. CRFs have also been used in extracting contexts and answers from online forums [25]. For example, Wang et al. [26] proposed a probabilistic model in the CRFs framework to predict the replying structure for a threaded online discussion. Raghavan et al. [27] extracted problem and resolution information from online forum discussions by formulating the problem as a sequence labeling task and proposed a method using CRFs. Instead of considering online forums, we consider the question answering site Stack Overflow in our study. The problem we address in this paper is also different from their studies.

TABLE I: Overview of the Corpus

API	Date	Questions	Answers	Sample Issues
Android	2008 - 2017	994237	1420973	2000
Jenkins	2008 - 2017	22782	26464	250
Neo4j	2008 - 2017	13434	16215	250
Total		1030453	1463652	2500

III. Background

A. Motivating Example

This section presents an example that shows the benefits of classifying SO posts concerning API issue. Although there are many other examples, due to space limitation we cannot discuss many others.

The example is about the issue of Android APIs. One of the Android developers filed an issue[5] about the design

[3]http://stackoverflow.com/questions/25436505/

[4]https://github.com/junit-team/junit4/issues/1083

[5]https://issuetracker.google.com/issues/36979732

problem of ControlFilter and DrawableContainer class (check Figure 1). However, SO users start discussing this issue almost two years before filing this issue by the developer in issue tracker. The Android developer detected this hidden issue with the help of SO discussions and therefore, he or she mentioned five different SO questions related to this issue. Other Android developers analyzed the discussions of those SO posts and using that knowledge they found a generalized solution just after two weeks of submitting the issue report. Eventually, this issue gets fixed almost two and a half year after the initial discussion in SO. This example indicates that SO posts concerning API issues can not only help API designers/developers to learn about API issues faster but can also help them to solve the problem. However, in the myriad of SO posts, it is very difficult for the API designers to find these issue posts. Therefore, a machine learning approach that can automatically classify issue-related posts will be useful for API designers.

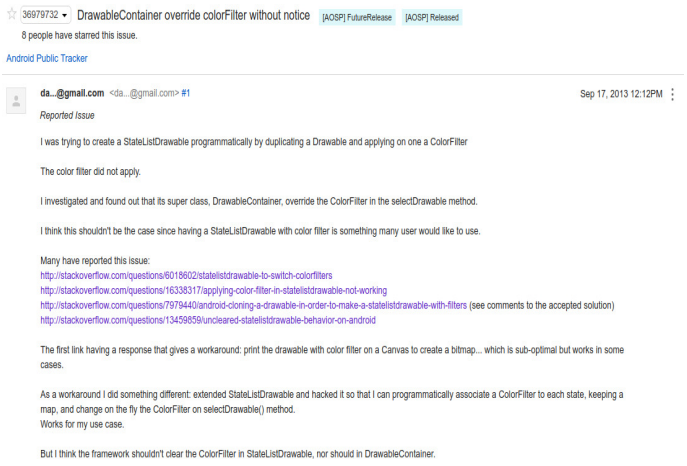
B. Corpus Creation

To determine what constitutes API-related issues, we use API usability factors discussed by Zibran et al. [2]. Although they presented a number of usability factors, in this paper we focus on only five of them. These are missing features, documentation, memory management, correctness, and backward compatibility. We consider any issues related to these factors as API-related issues.

We selected posts covering three different types of APIs based on several criteria. First, we chose those APIs that are popular, diverse in nature and have active user bases. Second, we need to identify API issue-related posts to train a classifier. We can make our decision with confidence for only those APIs that we are familiar with. Finally, we wanted APIs that have different sizes, and have varying number of SO posts. Table I shows the types of APIs we considered in this study. Among these API types, the largest is the Android[6]. It allows developers to create applications and games for mobile devices. Jenkins[7] is a continuous integration and continuous delivery application. Neo4j[8] APIs provide access to scalable graph databases.

We collected the latest September, 2017 SO data dump from the Stack Exchange Data Dump[9]. This data includes the publicly available history of question and answer posts, tags, votes on the posts, and the reputation of the users from August 2008 to September 2017. We downloaded the four files (i.e. posts, users, votes and tags) which were more than 80GB in total size. Finding issue-related posts was not easy because SO does not support identifying API issue-related posts. We found that in the SO community, active users provide links to issue trackers in the answer or comment sections of a post. These are valid posts concerning API issues. We traversed each of the SO posts and extract the link part with the tag

[6]https://developer.android.com
[7]https://jenkins.io
[8]https://neo4j.com
[9]https://archive.org/details/stackexchange



(a) Issue report

(b) Stack Overflow post

Fig. 1: Stack Overflow posts added in Android issue tracker

"$\langle a \rangle$ .. $\langle /a \rangle$". Then we checked whether the link contained particular issue tracker address or not. We only considered those posts for investigating where the links pointed to issue trackers or the issue tracker pointed SO posts in their issue description. Finally, we conducted a manual study to ensure that all these posts were related to the issue and after that we selected 2,500 SO posts for three different APIs. These selected posts were used for validating the effectiveness of our proposed technique for classifying API issue-related posts.

IV. Characteristic of Issue-Related SO Posts

We analyzed the corpus to understand the characteristics of SO posts concerning API issues. We analyzed the reputation of those who asked the question of the issue post and also those who provided the accepted answer. This is to understand whether reputation has any connection in asking API issue-related questions or answering them. We also investigated the time duration between posting the question and submitting the

TABLE II: Result of Reputation Analysis (IQ: Issue Questioner, IA: Issue Answerer, IAA: Issue Accepted Answerer, AQ: All Questioner, AA: All Answerer, AAA: All Accepted Answerer)

Reputation	IQ (%)	IA (%)	IAA (%)	AQ (%)	AA (%)	AAA (%)
<100	15.49	15.82	4.84	41.67	12.80	6.71
100 - 1000	31.56	28.86	19.13	33.03	27.14	22.31
1000 - 10000	40.69	38.42	43.68	22.01	39.23	41.35
10000 - 50000	9.86	12.56	20.42	3.01	14.92	20.19
>50000	2.42	4.35	11.92	0.27	5.88	9.42

answer in SO to see whether issue-related questions take more time to get an answer or not. We also performed a topic model analysis of issue posts to understand the frequently discussed topics.

A. Reputation

For this analysis, we categorize users into five different groups based on their reputation level. Table II shows the percentage of the API issue-related questions which were asked by users with different reputation categories. We find that the largest number of questions were asked by users (40.69%) having reputation between 1000 and 10000. When we consider all the questions that are asked by SO users with the reputation level 1000-10000, we find that only 22.01% of them are in this reputation level but the number is almost double for issue questioners (40.69%).

Thus, we see that experienced users ask more issue-related questions than the other users. We find less participation of novice users (< 100) in the issue-related posts (15.49% issue-related questions were asked by novice users) compared to the overall posts. However, the number is not insignificant. In fact, around 46% of issue-related questions were asked by users without high reputation. We also observe similar patterns in reputation of those users who provided the accepted and the first answer.

B. Time Duration

We investigated following two kinds of time duration: 1) duration between the post creation and submission of the accepted answer and 2) duration between the post creation and submission of the first answer. Figure 2 shows the results of our analysis. We can see that almost 60% of all SO posts get the first and the accepted answer within one hour of posting the question. However, only 25% and 33% of the issue-related posts get the accepted answer and the first answer (respectively) within the first hour. This is an indication that issue-related questions are harder to answer than all the questions in SO. Besides, 26.68% of issue-related questions took 10 to 30 days to get the accepted answer. Withing the same time duration, 21.15% of issue-related questions received the first answer. However, only 7.5% of all questions get either accepted or the first answer within this time period. This shows that issue-related questions take considerably long time to get the accepted answer or the first answer.

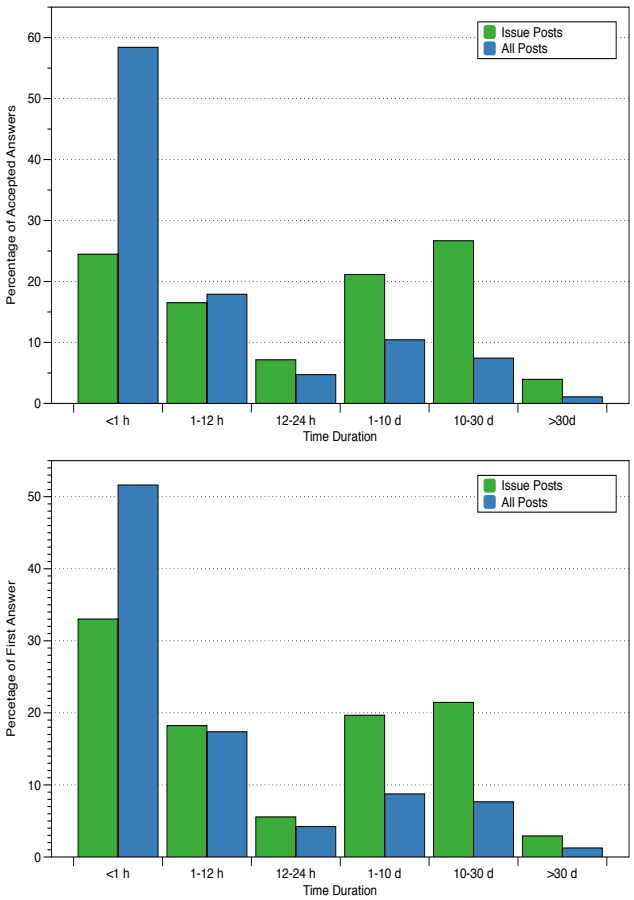


Fig. 2: Time duration analysis of issue-related posts

C. Topic Distribution Analysis

We ran the LDA model [28], an unsupervised learning method to generate topic word distribution for our corpus of API issue posts using the tool MALLET [29]. We trained the model for 200 iterations with 40 topics. Table III shows the sample of topics discovered. The top words of T1 are "image", "bitmap", "imageview", "audio", etc., which are related to the image and media. T4 discusses with the emulator or version related problem. T7 and T8 discuss with the error in debugging, thread or compilation. Topic 9 discusses the layout and design of the Android and the last topic is about string related discussion.

V. Proposed Technique

This section presents our proposed technique for **c**lassifying **API** issue **p**osts in **S**tack Overflow, called *CAPS*. We consider the problem as a binary classification task that requires to generate a model consisting features of API issue-related posts. The model is used to train a classification method that classifies any SO posts into two classes, issue and non-issue. To avoid bias we need to train the method using equal number of issue and non-issue posts.

Our proposed technique consists of the following steps. The first step is the **Sentence Extraction and Text Transformation**. The textual content of a post is interleaved with HTML

TABLE III: Topic Distribution Analysis

Topic	Top Topic Words
T1	intent, image, bitmap, imageview, bitmapfactory, media, findviewbyid, bundle, onactivityresult, audio
T2	response, json, jsonobject, request, url, session, jsonarray, httppost, asynctask, progressdialog
T3	layout_width, wrap_cont, textview, linearlayout, fill_par, relativelayout, gravition, edittext, layout_grav
T4	ndk, device, debug, platform, fail, command, source, target, emul, version
T5	android, item, style, drawable, color, anim, tabhost, parent, res, layoutparameter
T6	intent, context, void, android, class, string, notif, overrid, log, message
T7	com, dalvikvm, android, debug, app, thread, freed, method, error, activitymanagement
T8	com, android, compile, org, gradle, class, support, google, app, error
T9	android, mediaplay, video, player, image, screen, drawable, videoview, text, png
T10	string, null, connect, log, ioexception, printstacktrace, return, inputstream, buffer, fileinputstream

tags. Thus, we need to parse those tags to get the textual content of the post. Besides, we also include text transformation mechanism for successfully extracting sentences. The second step is the **Issue Sentence Identification**. We argue that issue-related sentences are valuable for our classification task because they provide important hints for deciding whether a post is API issue-related or not. Thus, we develop a supervised learning approach using Conditional Random Field (CRF), a statistical modeling method, to classify API issue-related sentences. The third step is the **Discriminative Classifier Generation**. In this step, we generate a set of feature values considering textual content, structural properties, user experience and issue sentences (if any) of posts. This leads to the development of a classification model. The model is used to train our machine learning method. We also describe how the trained machine learning method can be used for classifying issue posts.

A. Sentence Extraction and Text Transformation

This step is responsible for extracting textual content, code examples, and stack traces from each SO post that are interleaved with HTML tags. We also extract sentences from SO posts which are essential to train CRFs. We use Stanford Parser [30] in order to extract sentences. This parser relies on sentence ending characters to find the boundary of a sentence. When we analyze the HTML data, we find that many text units are not terminated by an ending symbol or are splitted by structural elements (i.e., code snippets). To overcome these, we remove the code snippets and inject punctuations as sentence ending symbol. We extract each code examples and map them with unique id. We find that SO users add code examples using $\langle pre\rangle\langle code\rangle \ldots \langle/code\rangle\langle/pre\rangle$ tag. Whenever the $\langle pre\rangle\langle code\rangle$ part of the above tag is not ended with a sentence ending symbol and the next word starts with an upper case letter, we inject a period. This solves the problem of splitted text units due to the insertion of code snippet. Figure 3 shows above scenarios that are taken from SO posts. The result is shown in Figure 4, after applying code collapsing and removing html tags. Here, a period is added to indicate the end of the sentence where it meets the above criteria. We also separate exception

```
<p>I am not using Fragments, still there is a reference of
FragmentManager. If any body can throw some light on some
hidden facts to avoid this type of issue:</p>
<pre><code>
java.lang.IllegalStateException: . . .
at android.app.FragmentManagerImpl.checkStateLoss . . .
at android.app.FragmentManagerImpl . . .(FragmentManager.java:399)
. . . . . </code></pre>
```

```
<p>I already tried a </p>
<pre><code>webView.destroy();
webView = null;
</code></pre>
<p>in onDestroy() of my activity, but that doesn't help much.</p>
```

Fig. 3: Text before removing code snippet

```
I am not using Fragments, still there is a reference of
FragmentManager. If any body can throw some light on some
hidden facts to avoid this type of issue: PROBLEM_CODE.
```

```
I already tried a NORMAL_CODE in onDestroy() of my activity, but
that doesn't help much.
```

Fig. 4: Text after removing code snippet

code (i.e., having stack traces or log information) from the normal code. If we find match of the content in $\langle code\rangle\langle/code\rangle$ with a regular expression (see Figure 5), we consider them as problematic source code (i.e., stack traces).

We remove the content and place the single word **PROBLEM_CODE**. Otherwise, if the content does not match the regular expression, we place the word **NORMAL_CODE** after removing the code snippet. Besides, we apply naming convention in Java in order to detect the API elements (i.e. method and class name). If we find a word having the first letter of each internal word capitalized and does not contain the opening and closing brackets, we consider them as a class. However, if the first letter is lowercase and maintain a camel case convention, we consider them as a method. Next, we remove the word in the text and replace with the word CLASS or METHOD. This generalized information help classifiers to learn better.

B. Issue Sentence Identification

In this step, we propose a supervised learning approach using Conditional Random Fields (CRFs) [8] for identifying issue-related sentences. We consider detection of issue sentences as a sequence labeling task because of the availability

1. at (.)* ? \\ ([A-Z] [a-z0-9A-Z]*.java:[0-9]*
2. (V|D|I|W|E|F|INFO|DEBUG|WARN|FATAL|VERBOSE|ERROR)\\/(.)*?\\(
3. (Exception:|Error:)(.)*?\\([A-Z][a-z0-9A-Z]*\\.java:[0-9]*\\)

Fig. 5: Regular expression for extracting problematic code

of the contextual information in a SO post. An issue-related post should contain one or more issue-related sentences. We first introduce the CRF model and later describe its application to classify issue and non-issue sentences.

1) Conditional Random Field: CRFs are undirected and discriminative graphical models trained to maximize the conditional probability [8]. They are sequential version of the logistic regression and a log-linear model for sequential labeling. Linear chain is a common special-case graph structure, which corresponds to a finite state machine and is suitable for sequence labeling. A linear chain CRF compute the probability of label sequence give an observation sequence, assuming that the current label depends only on the previous label and observation, as given below:

$$P(Y|X,W) = \frac{1}{Z(X)} exp(\sum_{t=1}^{T} \sum_{k} w_k f_k(y_{t-1}, y_t, t) \quad (1)$$

where, $Y = y_1, y_2, y_3, ..., y_T$ denote the label sequence and $X = x_1, x_2, x_3, ..., x_T$ denote the input sequence, $f_k(.)$ denote the k^{th} feature function which is often binary-valued, but can be real-valued, w_k denote the weight of the k^{th} feature function. $Z(X)$ is the normalization constant that makes the probability of all state sequences sum to one, defined as follows:

$$Z(X) = \sum_{y \subset Y^L} exp(\sum_{t=1}^{T} \sum_{k} w_k f_k(y_{t-1}, y_t, t) \quad (2)$$

where, Y^L is the set of all label sequences.

Inference to the most probable labeling sequence given the observation sequence, can be efficiently calculated by dynamic programing using the Viterbi Algorithm in the following way:

$$Y^* = argmax_Y P(Y|X,W) \quad (3)$$

CRFs have many advantages over other generative models. One of the important advantages is that, a wide variety of arbitrary number of independent and non-independent features computed from the observation state can be used along with observation for labeling task because there is no constraint that feature components and observation should be independent of each other.

2) CRF Training: We use manually annotated issue sentences for the training of CRF. We use sentence-level labeling for generating the data set and consider two labels. If we find a sentence containing information about the API issue, we label it as an *issue*. Otherwise, we use the *non-issue* to label the sentence. Table IV shows examples of manual annotations at a sentence-level. In order to train the CRF, we select textual and structural features of sentences from the posts. We consider the following features:

Words: In order to have a better contextual information, we consider the words of a sentence as features for CRF. However, we do not perform any stop word removal or stemming operations. The reason behind this is that frequent words can be representative of a class. Furthermore, stemming operation can hamper the contextual information. One of the advantages of CRFs is that they easily afford the use of arbitrary features of input. Therefore, the number of features in CRF is not fixed and it varies with the sentence containing different number of words.

Part-of-Speech (POS): Part-of-speech (POS) tags are extracted from the sentence to include additional information of the grammatical structure and category of words of a sentence. We used Stanford NLP Part-Of-Speech Tagger [31] to extract the information.

Sentiment Information: When we annotate the data, we find that most of the API issue-related sentences express negative sentiment. We used SentiWordNet 3.0 [32] in order to capture the sentiment information. We calculate sentiment value of each of the words in the sentence and average the total sum. Using a threshold value over the total sum, we consider the following sentiments of the sentence: very_positive, positive, neutral, negative and very_negative.

Normalize Position in post: Most of the API issue-related sentences are expressed in the beginning and in the middle of a post. We consider this trend as a feature to improve the learning of CRF. We normalize the position of the sentence in a post and using threshold values we generate one of three following features: BEGIN, MIDDLE and END.

C. Discriminative Classifier Generation

1) Feature Collection: In this section, we briefly describe the set of features we collected for each post to train the classifier. We selected these features by analyzing the issue-related posts. Each SO post contains a title, a question, zero or more answers and comments. Thus, we consider features for the title, the question and the set of answers associated to that question. In SO, novice users typically ask more how-to questions and participate less in question answering. Bajaj et al. [33] found that majority of the accepted answers are provided by users with high reputation. Thus, the reputation of questioners and answerers can help detecting issue-related posts. Reputation indicates the expertise of a contributor in SO. Thus, we include reputation of questioners and answerers as features. We also consider three other features that can also serve as a proxy for reputation. These features fall under the experience category. A SO post can have a number of answerers. Thus, we consider the answerer who has the maximum reputation. In addition, we also consider two features whose values are calculated by detecting issue sentences in question body using the CRF. Table V summarizes the five sets of features we selected for the classification model.

2) Training and Testing the Classifier: To train the classifier we need to generate a training data set. For each post, we collect all the feature values as described in the previous section. These act as predictors or independent variables. The target or response variable indicates whether the post is API issue-related or not. Thus, the response variable has two classes. We use logistic regression to derive our classification model. Logistic regression is a discriminative classification model that operates on the real valued vector input. It is also a probabilistic classifier that given a test post generates a class

TABLE IV: Examples of issue and non-issue sentences in post questions

PostId	Sentence	Label
5796611	So, should this really be considered a ”bug”, since we are officially,advised to use Activity.getApplication() and yet it doesn’t function as,advertised	issue
12803797	It seems that the Android documentation about layout aliases is incorrect, and certainly appears inconsistent	issue
12389115	So i added my project an AsyncTask class that i wrote a while ago for quick,testing purposes but it is causing memory leak errors	issue
6218143	If anyone knows of a good Android book that deals with this please let me know	non-issue
11014953	I want to provide user credentials from an Android application to the API, get the user logged in, and then have all subsequent API calls pre-authenticated	non-issue
3264383	What is the difference between Service, Async Task & Thread	non-issue

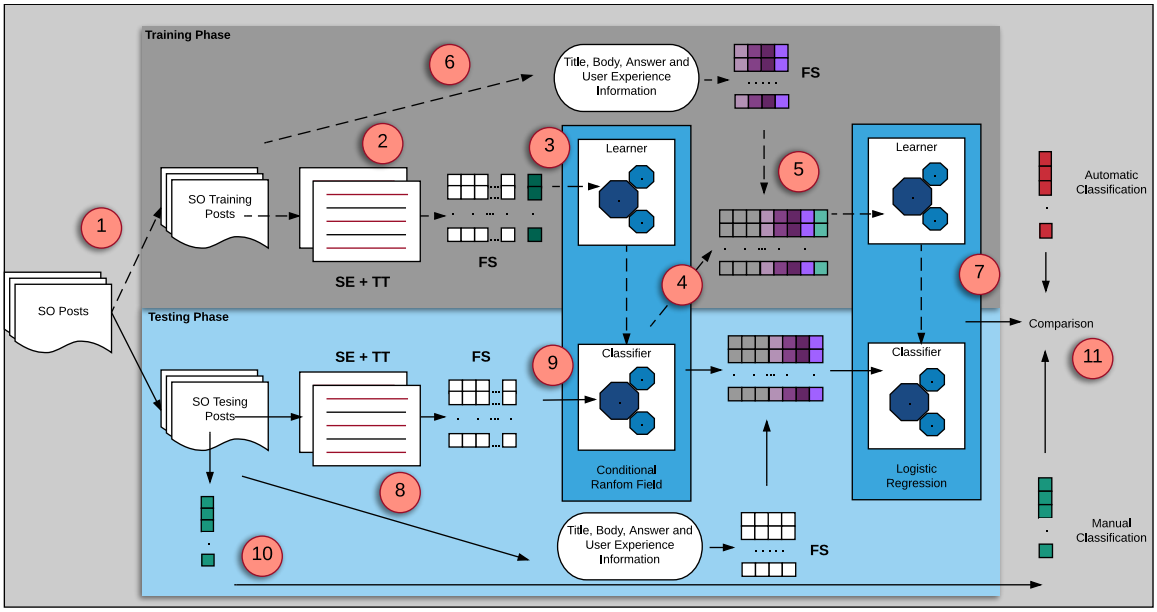


Fig. 6: Overview of our proposed technique (SE = Sentence Extraction, TT = Text Transformation and FS = Feature Selection)

probability value. This values indicates the likelihood of the post belonging to that class.

Figure 6 explains the training and the testing phases of *CAPS*. We first create a corpus containing manually classified SO posts. We select these SO posts for the training of *CAPS* (Point 1). We extract sentences and perform text transformation (Point 2). We then determine the feature values for each post (Point 3) (check Table V). We also train the CRF (Point 4) classifier. We generate title, question body, answer and experience level features for each post (Point 6). We use all these and CRF features to train our classification model. Point 7 shows the actual output of the training. To avoid any biases, we use the same number of issue and non-issue-related posts to train our classification model. To test *CAPS*, we also need a manually classified SO posts. For each test post, we determine the title, question body, answer and experience level features (Point 8). We also collect the CRF features (Point 9). These five sets of feature values act as an input to our classifier build in the training phase. A total of 18 features are extracted including the post features and CRF features. Then, the 18 dimensional vector fed into the previously trained classification model for testing. The classifier then tell us whether the test post is API issue-related post or not. Recall that the posts in our data set are already manually classified. We collect the results of manual classification (Point 10) and compare the results with that of the classifier (Point 11).

VI. Evaluation

We evaluate *CAPS* in two different ways. First, we compare the technique with three baseline approaches. Second, we compare the technique with the work of Wang et al. [9]. The following section describes each of the experiment in detail.

A. Comparison with Baseline Approaches

We compare *CAPS* with three baseline approaches that address the same problem using different heuristics and sources of information. These are a text classification technique, a CRF-based technique and a reputation-based technique.

1) Description: We consider a machine learning-based text classification technique because it has shown great promise in various problems in software engineering [6], [34]. To determine whether issue-related sentences can solely be used for classifying issue-related posts, we include the CRF-based technique in this study. Finally, we consider a reputation-based technique to determine the usefulness of reputation for solving the classification problem. We briefly describe each of the technique as follows.

TABLE V: Summary of features

Feature	Description
	Title Features
titleIssueWord	True if the title contains the words 'bug', 'issue', 'error' or 'exception'
titleWHwords	True if the title starts with 'How', 'Where' or 'What'
titleWordCountNorm	Normalization value of number of words in a title
	Body Features
bodyProblemCode	True if there exists a stacktrace or log information
bodyIssueLink	True if there is a link in the question and contains the word 'issue' or 'bug'
bodyNegativeSentiWordRatio	The ratio of negative sentiment word to the total number of word in the post
bodySentenceNorm	The normalize value of sentence in the body
	CRF features
issueSentenceCount	Total number of issue sentences in the post body
issueSentenceRatio	The ratio of issue-related sentences to total number of sentences in the body
	Answer Features
answerPresent	True if there is at least one answer
answerIssueLink	True if there is a link in the answer and contains the word 'issue' or 'bug'
durationPostAnswer	The time duration between the post created and first accepted answer
answerIssueWord	True if the answer contains 'bug', 'issue' 'exception' or 'error'
	Experience Features
questionerReputation	SO reputation of the questioner
answererReputation	The highest SO reputation among the answerers
questionerExperience	The total number of questions asked and answered by the questioner
answererExperience	The total number of accepted answer posted by the answerer
questionerQualityPost	The difference between the total number of upVotes and downVotes of the questioner

a) Text Classification Technique: In our study, we consider a machine learning-based text classification technique that automatically learns from training data. In our case, the data comes from SO posts. Our technique is statistical because we provide manually labeled API issue and non-issue-related posts to learn each class. Automatic text classification has been found effective in various problem areas that deal with large amount of textual content. Examples include but are not limited to content classification of developer emails [6], separating features from bug reports, discovering tutorial section that explain a given API type [34]. Typically, a text classification technique generates features using terms appearing in documents. The documents are modeled as vectors of features and these features values are determined from the frequency of those terms in documents. In our case, documents are SO posts and features are the set of words appearing in those posts.

- **Term Selection:** We consider each post as a bag of words. For each post, we collect any words by tokenizing the title, question body, and answers including any code fragments in them. We neither perform any stop word removal nor apply any stemming. This is because frequently appearing words or words that derive from the same root word can representative of a class [6]. Depending on the posts size, the set of words can be very large. Instead of considering all terms as features that can lead to overfitting problem, we consider a subset of terms as features using frequency-based feature selection technique. The technique performs often well when many thousands of features are considered.
- **Machine Learning Method:** We select the logistic regression classifier that learns from the training data and performs the classification. Logistic regression is a discriminative classification model that operates on the real valued vector input. Despite the simplicity, logistic regression has been found effective in text classification tasks. Details of the technique can be found elsewhere [35].

b) CRF-based Technique: Conditional Random Fields (CRFs) are statistical modeling methods. We use CRF to detect API issue-related sentences. We hypothesize that if a post contains such sentences, it is an API issue post. To validate the hypothesis, we make the following change to allow CRF to classify issue-related posts. We train CRF using manually validated API issue sentences. Given a test post, we apply CRF to its textual content to detect issue sentences. We classify the post as API issue-related if CRF identify any issue sentences in it.

c) Reputation-based Technique: We also implement another technique that uses reputation of SO users to classify a post into issue category. The basic idea is that if a post question is asked or answered by a user with high reputation, it is likely to be an issue post. We include the technique to verify to what extent the claim can be supported by empirical study. However, it is difficult to define the term high reputation. Thus, we determine the average reputation of SO users participated in the posts of target APIs and consider anything above the average value as the high reputation. For a test post, if the reputation of the questioner or any of its answerers is greater than the average reputation, we classify the posts into issue category.

2) Evaluation Metrics: We evaluate our results using *precision*, *recall* and *F-measure* which have been used in a number of previous studies [34]. We compare our generated ground truth with automatically generated classification. The correctly classified posts have been considered as *true positive* and the post incorrectly classified as belonging to the class have been considered as *false positive*. The post incorrectly labeled belonging to other class have been computed as the *false negative*. Thus, *precision*, *recall* and *F-measure* can be computed as the following way:

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

TABLE VI: Evaluation Results

API Name	Technique	Issue			Non-issue		
		Precision	Recall	F-measure	Precision	Recall	F-measure
Android	Reputation	0.63	0.43	0.51	0.27	0.25	0.26
	CRF	0.46	0.83	0.59	0.30	0.34	0.32
	Text Classification	0.54	0.56	0.54	0.49	0.51	0.49
	CAPS	0.95	0.71	0.81	0.76	0.95	0.84
Neo4j	Reputation	0.38	0.25	0.30	0.32	0.31	0.31
	CRF	0.28	0.45	0.34	0.36	0.51	0.42
	Text Classification	0.50	0.59	0.54	0.64	0.55	0.59
	CAPS	0.95	0.75	0.83	0.83	0.95	0.88
Jenkins	Reputation	0.43	0.36	0.39	0.32	0.31	0.31
	CRF	0.42	0.62	0.50	0.39	0.52	0.45
	Text Classification	0.49	0.50	0.49	0.50	0.59	0.54
	CAPS	0.92	0.71	0.80	0.73	0.92	0.81

$$F - measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (6)$$

Here, TP denotes as *true positive*, FP denotes as *false positive* and FN denotes as *false negative*.

3) Experimental Setup: We use our corpus consists of SO posts that are labeled into two classes (issue and non-issue) to perform the evaluation. We apply 10 fold stratified cross-validation to measure the performance of each compared technique. We split the dataset into 10 different folds of equal sizes. We use the 9 folds (90% of data) to train the technique and the remaining fold is used to test the performance of the technique. We repeat the process 10 times by rotating the training and test folds. The MALLET [29] tool is used to train the CRF and we reuse the implementation of the logistic regression available in the Weka [36].

4) Evaluation Results: Table VI summarizes the results of our evaluation for both issue and non-issue classes. The results clearly suggest that *CAPS* outperforms the other three baseline approaches. For the Android API, reputation-based technique performs the worst. While the precision and recall values are 0.63 and 0.43 for the issue class, both values are dropped for the non-issue class (0.27 and 0.25 respectively). The CRF-based technique improves the performance on the issue class but it does not work well on the non-issue class. Since the number of non-issue posts are expected to be much higher than the issue posts, the low recall value for the non-issue class makes the technique ineffective. Text classification technique also does not perform well comparing other three baseline approaches. While the precision and recall values are 0.54 and 0.56 for the issue class, for the non-issue class the values are 0.49 and 0.51 respectively. *CAPS* achieves the best precision and recall values for both classes. While the precision and recall values are 0.95 and 0.71 for the issue class, the technique achieves 0.76 and 0.95 for non-issue class. We also observe similar results for the Neo4j and Jenkins API. The reputation-based technique performs the worst again. The text classification technique and the CRF-based technique rank the second and the third positions respectively. *CAPS* performs the best among all four techniques.

B. Comparison with the Work of Wang et al. [9]

A related work to our study is that of Wang et al. [9]. Given a collection of SO posts, their technique recommends a ranked list of API issue-related posts. Since the implementation is not publicly available, we re-implement the technique. The technique first detects experts and retains only those posts that are asked by expert users. It then detects dominant SO discussion topics and selects only those posts for recommendation that are related to those topics. Next, the technique filters late answered posts using a statistical quality control technique, called *control charts*. The remaining posts are sorted based on a set of metrics derived from SO data.

Their work does not focus on classifying issue-related posts and does not utilizes textual features of SO posts. Thus, it is difficult to compare *CAPS* with their work. However, we follow the following approach for the purpose of comparison. We select an equal number of issue and non-issue-related posts (1000 in total) of Android API from our corpus for training. The remaining 1500 issue posts of Android API are used for testing. We would like to find how many of these issue-related posts are detected by the technique of Wang et al.. We fed all the Android related posts except those we use for training *CAPS* as input to the technique. After filtering, the technique selects 91,234 posts out of the 994,237 Android posts. This selected set of posts only contain 324 issue-related posts out of the 1,500 we selected for testing. Thus, the technique achieves 21.6% accuracy. However, *CAPS* correctly classifies 1050 issue-related posts out of the 1500 and achieves 70% accuracy.

VII. Discussion

This section discusses a set of questions related to our study.

A. Testing Generalizability of the CAPS Results

SO posts that discuss issues of different APIs may use different words and jargon. This is because these APIs are quite different from each other. We are interested to know

TABLE VII: Evaluation results of *CAPS* in classifying unseen APIs

Unseen API	Class	Precision	Recall	F-measure
Android	Issue	0.83	0.74	0.78
	Non-issue	0.77	0.85	0.80
Neo4j	Issue	0.92	0.86	0.89
	Non-issue	0.84	0.91	0.87
Jenkins	Issue	0.90	0.68	0.77
	Non-issue	0.69	0.90	0.78

TABLE VIII: Impact of different sets of features on the performance of *CAPS*

No	Different Sources of Information	Issue			Non-issue		
		Precision	Recall	F-measure	Precision	Recall	F-measure
1.	Title Features	0.61	0.56	0.58	0.59	0.63	0.60
2.	Body Features	0.63	0.58	0.60	0.61	0.66	0.64
3.	CRF Features	0.93	0.57	0.70	0.70	0.95	0.80
4.	Answer Features	0.91	0.31	0.46	0.59	0.95	0.73
5.	Experience Features	0.61	0.39	0.47	0.57	0.72	0.63
6.	Title + Body Features	0.66	0.61	0.63	0.64	0.68	0.66
7.	Title + Body + CRF Features	0.94	0.62	0.73	0.71	0.92	0.80
8.	Title + Body + CRF + Answer Features	0.97	0.68	0.79	0.75	0.96	0.84
9.	Title + Body + CRF + Answer Features + Experience Features	0.95	0.71	0.81	0.76	0.95	0.84

whether it is possible to train *CAPS* using SO posts discussing issues of one API and then classify issue posts of another API. To do that we use a three-fold cross-validation where the folds are not randomly created. Instead, we create one fold for each API in our corpus. We use any two folds to train the classifier and then test using the third fold. Table VII reports the results of our experiment. We use the term *unseen API* to refer to the API under testing. The results indicate that the performance of the *CAPS* drops, which is not surprising. However, the results are not affected much which is an indication that *CAPS* can be used to classify issue-related posts of unseen APIs.

B. Effects of Different Sets of Features

The proposed classification model considers five different sets of carefully selected features. This section investigates how different feature sets impact the performance of *CAPS*. Towards this goal we run experiments on Android. Table VIII shows the overall performance of *CAPS* when we use different sets of features. All other settings of our technique remain unchanged. To simply our discussion, we consider the precision and recall of the issue class in this discussion. From the table we can see that the precision and recall values of *CAPS* reaches to 0.61 and 0.56 for the issue class when we only use the title feature set. After adding the body feature set with title, the performance improves again. We observe a significant increase in the precision value when we use the CRF feature set. The precision reaches from 0.66 to 0.94. We observe small increase in the recall value. We also observe significant improvement on precision and recall values for the non-issue class. After adding the answer feature set, we find a small increase in precision, but the recall value increases to 0.68. This is mostly contributed by the fact that sometimes answerer provides issue links or use specific terms while discussing API issues which can be a potential source of information for classifying issue-related posts. Adding the experience feature set does not improve the precision and recall values of the issue class. It only slightly increase the recall value for the non-issue class. Since, the number of non-issue posts are significantly greater than the issue-related posts, we recommend to use the experience feature set.

C. Runtime Performance

To measure the runtime performance of *CAPS*, we calculate the time required to train our model and classify the posts. The majority of the time involves in annotating the sentences for training the CRF model. However, this is a one time operation only. For 5000 SO posts, *CAPS* takes around 7s to train the CRF model and generate the features. It takes around 1.2s on average to train the discriminative classification model for classifying the issue posts. For testing each of the instance post, it only takes 1ms on average.

VIII. Threats to Validity

This section summarizes the threats to the validity of our study.

First, we re-implemented the technique developed by Wang et al. [9] since the data and the technique were not publicly available at the time of writing the paper. Although we cannot guarantee that our implementation of the technique does not contain any errors, we spent a considerable time in replicating and testing the technique to ensure its correctness.

Second, our data set consists of posts concerning API issues of three different APIs written in the Java language. One can argue that the results obtained in our study may not hold for other APIs or for different languages. However, we would like to point to the fact that our selection was based on our familiarity with these APIs. To avoid bias, we considered posts from three different APIs. The features we used to develop our technique is also not specific to any particular language.

IX. Conclusions and Future Works

Stack Overflow posts concerning API issues become a valuable source of information to API designers. Towards the goal of classifying API issue-related posts, we develop a supervised learning approach using a CRF that can classify issue-related sentences. We combine the features collected from the output of CRF to that of the SO posts and user reputation. This leads to the development of a issue classifier, called *CAPS*. We evaluate *CAPS* using SO posts from three different API types. Results from the study reveals that *CAPS* achieves high precision and recall values for both classes. We also compare *CAPS* with three other baseline approaches and the technique outperforms all of them. Our approach also enables highlighting problematic issue sentences which can allow developers to (i) filter irrelevant sentences and focus on the API issue-related information, (ii) understand issues more quickly and (iii) be more responsive to issues submitted by users. As a future work, we plan to implement the technique as a Chrome plugin that enables a wide range of API designers to use the technique.

REFERENCES

[1] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE Softw.*, vol. 26, no. 6, pp. 27–34, 2009.

[2] M. F. Zibran, F. Z. Eishita, and C. K. Roy, "Useful, but usable? factors affecting the usability of apis," in *Proc. WCRE*, 2011, pp. 151–155.

[3] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," in *Proc. ICPC*, 2014, pp. 83–94.

[4] K. Aggarwal, F. Timbers, T. Rutgers, A. Hindle, E. Stroulia, and R. Greiner, "Detecting duplicate bug reports with software engineering domain knowledge," *Journal of Software: Evolution and Process*, vol. 29, no. 3, 2017.

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

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

[7] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web? (nier track)," in *Proc. ICSE*, 2011, pp. 804–807.

[8] C. Sutton and A. McCallum, "An introduction to conditional random fields," *Found. Trends Mach. Learn.*, vol. 4, no. 4, pp. 267–373, 2012.

[9] W. Wang, H. Malik, and M. W. Godfrey, "Recommending posts concerning api issues in developer q&a sites," in *Proc. MSR*, 2015, pp. 224–234.

[10] A. Baltadzhieva and G. Chrupala, "Predicting the quality of questions on stackoverflow," in *Proc. RANLP*, 2015, pp. 32–40.

[11] B. V. Hanrahan, G. Convertino, and L. Nelson, "Modeling problem difficulty and expertise in stackoverflow," in *Proc. CSCW*, 2012, pp. 91–94.

[12] L. Ponzanelli, A. Mocci, A. Bacchelli, M. Lanza, and D. Fullerton, "Improving low quality stack overflow post detection," in *Proc. ICSME*, 2014, pp. 541–544.

[13] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," *Empirical Softw. Engg.*, vol. 19, no. 3, pp. 619–654, 2014.

[14] B. Bazelli, A. Hindle, and E. Stroulia, "On the personality traits of stackoverflow users," in *Proc. ICSM*, 2013, pp. 460–463.

[15] A. Bacchelli, L. Ponzanelli, and M. Lanza, "Harnessing stack overflow for the ide," in *Proc. RSSE*, 2012, pp. 26–30.

[16] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proc. MSR*, 2014, pp. 102–111.

[17] M. Asaduzzaman, A. S. Mashiyat, C. K. Roy, and K. A. Schneider, "Answering questions about unanswered questions of stack overflow," in *Proc. MSR*, 2013, pp. 97–100.

[18] D. Correa and A. Sureka, "Fit or unfit: Analysis and prediction of 'closed questions' on stack overflow," in *Proc. COSN*, 2013, pp. 201–212.

[19] ——, "Chaff from the wheat: Characterization and modeling of deleted questions on stack overflow," in *Proc. WWW*, 2014, pp. 631–642.

[20] A. Sandor, N. Lagos, N.-P.-A. Vo, and C. Brun, "Identifying user issues and request types in forum question posts based on discourse analysis," in *Proc. WWW*, 2016, pp. 685–691.

[21] M. P. Robillard and R. Deline, "A field study of api learning obstacles," *Empirical Softw. Engg.*, vol. 16, no. 6, pp. 703–732, 2011.

[22] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation?: The case of a smalltalk ecosystem," in *Proc. FSE*, 2012, pp. 56:1–56:11.

[23] W. Wang and M. W. Godfrey, "Detecting api usage obstacles: A study of ios and android developer questions," in *Proc. MSR*, 2013, pp. 61–64.

[24] M. Silfverberg, T. Ruokolainen, K. Lindén, and M. Kurimo, "Part-of-speech tagging using conditional random fields: Exploiting sub-label dependencies for improved accuracy," in *Proc. ACL*, 2014, pp. 259–264.

[25] S. Ding, G. Cong, C.-Y. Lin, and X. Zhu, "Using conditional random fields to extract contexts and answers of questions from online forums." in *Proc. ACL*, 2008, pp. 710–718.

[26] H. Wang, C. Wang, C. Zhai, and J. Han, "Learning online discussion structures by conditional random fields," in *Proc. SIGIR*, 2011, pp. 435–444.

[27] P. Raghavan, R. Catherine, S. Ikbal, N. Kambhatla, and D. Majumdar, "Extracting problem and resolution information from online discussion forums." in *Proc. COMAD*, 2010, p. 77.

[28] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, 2003.

[29] A. K. McCallum, "Mallet: A machine learning for language toolkit," 2002.

[30] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proc. ACL*, 2014, pp. 55–60.

[31] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Proc. NAACL*, 2003, pp. 173–180.

[32] S. Baccianella, A. Esuli, and F. Sebastiani, "Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining," in *Proc. LREC*, 2010, pp. 2200–2204.

[33] K. Bajaj, K. Pattabiraman, and A. Mesbah, "Mining questions asked by web developers," in *Proc. MSR*, 2014, pp. 112–121.

[34] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining api types using text classification," in *Proc. ICSE*, 2015, pp. 869–879.

[35] P. D. Allison, *Logistic Regression Using SAS: Theory and Application, Second Edition*, 2nd ed., 2012.

[36] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, 2009.