6-2014

Towards more accurate content categorization of API discussions

Bo Zhou

Xin Xia

David LO
Singapore Management University, davidlo@smu.edu.sg

Cong Tian

Xinyu Wang

Citation

Zhou, Bo; Xia, Xin; LO, David; Tian, Cong; and Wang, Xinyu. Towards more accurate content categorization of API discussions. (2014). *22nd International Conference on Program Comprehension (ICPC 2014): Proceedings: June 2-3, 2014, Hyderabad, India*. 95-105.
Available at: https://ink.library.smu.edu.sg/sis_research/2420

Towards More Accurate Content Categorization of API Discussions

Bo Zhou[1], Xin Xia[1,*], David Lo[2], Cong Tian[3], and Xinyu Wang[1†]
[1]College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[2]School of Information Systems, Singapore Management University, Singapore
[3]ICTT and ISN Lab, Xidian University, Xi'an, China
{bzhou, xxkidd}@zju.edu.cn,davidlo@smu.edu.sg, ctian@mail.xidian.edu.cn, wangxinyu@zju.edu.cn

ABSTRACT

Nowadays, software developers often discuss the usage of various APIs in online forums. Automatically assigning pre-defined semantic categorizes to API discussions in these forums could help manage the data in online forums, and assist developers to search for useful information. We refer to this process as *content categorization of API discussions*. To solve this problem, Hou and Mo proposed the usage of naive Bayes multinomial, which is an effective classification algorithm.

In this paper, we propose a Cache-bAsed compoSitE algorithm, short formed as *CASE*, to automatically categorize API discussions. Considering that the content of an API discussion contains both textual description and source code, *CASE* has 3 components that analyze an API discussion in 3 different ways: text, code, and original. In the text component, *CASE* only considers the textual description; in the code component, *CASE* only considers the source code; in the original component, *CASE* considers the original content of an API discussion which might include textual description and source code. Next, for each component, since different terms (i.e., words) have different affinities to different categories, *CASE* caches a subset of terms which have the highest affinity scores to each category, and builds a classifier based on the cached terms. Finally, *CASE* combines all the 3 classifiers to achieve a better accuracy score. We evaluate the performance of *CASE* on 3 datasets which contain a total of 1,035 API discussions. The experiment results show that *CASE* achieves accuracy scores of 0.69, 0.77, and 0.96 for the 3 datasets respectively, which outperforms the state-of-the-art method proposed by Hou and Mo by 11%, 10%, and 2%, respectively.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

*The work was done while the author was visiting Singapore Management University.
†Corresponding author.

ICPC'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06...$15.00
http://dx.doi.org/10.1145/2597008.2597142

General Terms

Algorithms and Experimentation

Keywords

API Discussion, Text Categorization, Composite Method, Cache-Based Method

1. INTRODUCTION

Learning to use software frameworks and their corresponding APIs (Application Programming Interfaces) can be a hard job for software developers, which would impede their productivity [7, 18, 20, 24]. Nowadays, developers commonly use online forums to discuss the usage of APIs, ask API usage questions, and seek help. For example, Figure 1 shows an API discussion in a Java Swing forum.[1] A developer asked a question about the bad display of icons when using JLabel and JList. Another developer later discovered the root cause of the problem and advised to wrap the invocation of method `ensureIndexIsVisible` inside method `invokeLater` of `SwingUtilities` when the auto scrolling feature is selected.

Over the years, these online forums store massive amount of valuable API usage knowledge, and most of the time, developers just need to search over the forums to find their contents of interest. To better manage the organization of contents in the forums, and reduce the time that developers need to spend to search for their contents of interest, we need an automated way to index these forum data according to their semantic similarity [14]. Hou and Mo proposed the problem of content categorization of API discussions, which is the task of automatically assigning pre-defined semantic categorizes to API discussions in software forums [14].

Various text categorization and machine learning algorithms [10, 27] could be used to solve the content categorization problem, e.g., naive Bayes [10], kNN [10], SVM [10], etc. Hou and Mo investigated the performance of naive Bayes multinomial (NBM) [19], and they concluded that NBM achieves a remarkable high accuracy [14]. However, NBM is a general algorithm which focuses on the general text categorization problem and API discussions are inherently different from general text. For example, in the API discussions, normally developers would attach source code (see Figure 1). The terms in the source code and the textual description are often different, and thus we need to treat them differently.[2]

In this paper, we propose a Cache-bAsed compoSitE algorithm, short formed as *CASE*, to improve the accuracy of the content

[1]https://community.oracle.com/thread/2594342
[2]For more details, please refer to Section 2.

categorization task. Considering that the content of API discussions includes both textual description and source code, we first extract and separate the textual description from the source code in API discussions. Then, CASE analyzes API discussions using 3 components: text, code, and original. In the text component, it categorizes the API discussions by only using textual description; in the code component, it categorizes the API discussions by only using source code; in the original component, it categorizes the API discussion by using both textual description and source code. Next, for each component (i.e., text, code, and original), since different terms have different affinity scores to different categories, we cache a subset of terms which have the highest affinity scores to each category, and we build a classifier based on the cached terms. Finally, we combine all the 3 classifiers to achieve better accuracy scores.

To evaluate the performance of *CASE*, we reuse the 3 datasets provide by Hou and Mo [14].[3] The 3 datasets are taken from Java Swing forums and we refer to them as Data-1.0, Data-2.0, and Data-3.0 respectively[4]. Data-1.0 and Data-2.0 are more challenging datasets as they include more semantic categories and there are less API discussions per category. The total number of API discussions across the three datasets is 1,035. The experiment results show that *CASE* achieves accuracy scores of 0.69, 0.77, and 0.96 for each of the 3 datasets respectively, which outperform the state-of-the-art method proposed by Hou and Mo by 11%, 10%, and 2%, respectively.

The main contributions of this paper are:

1. Considering the special structure of API discussions, we propose a composite algorithm which combines 3 components which separately analyze the text, code, and overall content of API discussions to achieve a better performance.

2. We also propose a cache-based algorithm which caches the terms with high affinity scores for each API category, and build classifiers by only using these cached terms.

3. We evaluate the performance of our algorithm using 3 publicly available datasets which were also used in the previous study by Hou and Mo [14]. We show *CASE* outperforms the the method proposed by Hou and Mo by a substantial margin – especially for Data-1.0 and Data-2.0.

The remainder of the paper is organized as follows. We describe the motivation of this work in Section 2. We outline our overall framework for content categorization of API discussions in Section 3. We elaborate the cached-based algorithm to cache the terms with high affinity scores to each category in Section 4. We present how we combine the 3 components of *CASE* in Section 5. We report the experiment results in Section 6. We describe related work in Section 7. We present the threats to validity in Section 8. We conclude and mention future work in Section 9.

2. MOTIVATION

In this section, we first describe an example to help readers better understand the motivation for content categorization of API discussions and our cache-based algorithm in Section 2.1. Next, we present the motivation of building a composite model in Section 2.2.

[3]http://www.clarkson.edu/ dhou/projects/swingForum2012.tar.gz

[4]These datasets are referred to as V1.0, V2.0, and V3.0 in [14].

Table 1: Accuracy Scores for Data-1.0, Data-2.0, and Data-3.0 Using Naive Bayes Multinomial.

Dataset	Text	Code	Original
Data-1.0	0.5333	0.3778	**0.6222**
Data-2.0	0.6772	0.6519	**0.6962**
Data-3.0	**0.9411**	0.9291	0.9315

2.1 A Motivating Example

A typical life cycle of an API discussion is as follows: 1. A user meets an API usage problem, and he posts the problem in a forum, and also attaches the source code related to this problem to his post. 2. Other developers who have the necessary expertise and are willing to help, reply to the post. 3. The user judges whether the problem is solved, and gives a reward (e.g., forum score) to the developer who solves the problem, and closes the discussion.

Figure 1 shows a sample API discussion. In the example, a user met a problem with the display of icons when he used JLabel and JList, and attached his source code. Sometimes later, another developer solved the problem by recommending a modification of the user's code.

Observations and Implications. From the above example, we have the following observations:

1. After we read the content of the API discussion, we find that it is an "icon display" problem. Thus, we can assign the label "icon display" to it. The next time another user meets an "icon display" problem, the user could perform a search under the category "icon display". By doing this, the user can potentially find API discussions that contain relevant contents to help him resolve his problem.

2. Some terms in the API discussion appear more number of times than the others. For example, term "size" appears 4 times in the textual description and source code, "icon" appears 3 times, and "display" and "height" appear 2 times. These terms are related to the "icon display" category, and their term frequencies help us to differentiate them from many other terms that are less related to the category.

The above observations tell us that automated categorization of API discussions could help to improve search efficiency, and make the content of discussions more explicit and informative [14]. Some terms which appear many times in an API discussions could help to identify its proper category. Thus, in this paper, we propose a cache-based algorithm, which caches the terms which have high term frequencies for each API discussion category, and we use these terms as the input features to build classifiers.

2.2 Why Composite Model?

As described in the previous section, we can categorize an API discussion in 3 ways: 1. use only the textual description in the API discussion (text); 2. use only the source code in the API discussion (code); 3. use both the textual description and source code in the API discussion (original). In this section, we investigate which one is the best to categorize API discussions using naive Bayes multinomial which was used by Hou and Mo [14].

Table 1 presents the experiment results for Data-1.0, Data-2.0, and Data-3.0 using naive Bayes multinomial. We notice for Data-1.0 and Data-2.0, categorizing API discussions using both textual description and source code achieves the best accuracy scores. However, for Data-3.0, categorizing API discussions using only

Displaying JList holding JLabel with icons of different height screwed up

I started to create a unix 'make' wrapper, which filters for the binaries being created and display their names and potentially some icon for those in a JList.
But sometimes the display of the icons is screwed up. It's cut to the size of a pure text (JLabel) line in the JList.

Below you can find test program.

You need to put two different sized picture files binary1.jpg and binary2.jpg in the run directory of the java build (i use eclipse)
and create some test input file like this:

```
> cat input
binary1
binary2
```

Set TEST_DIR environment variable to the build run directory.
Finally start the java class like this:

```
> bash -c 'while read line; do echo $line; sleep 1; done' < input| (cd $TEST_DIR; java -classpath $TEST_DIR r
```

Then you should see the issue.
(Don't mind about the JAVA code in general - i know, that there are quite some other issues to fix :-))

1. Re: Displaying JList holding JLabel with icons of different height screwed up

I found out, that the issue is caused by trying to use auto-scrolling via line #100:

```
list.ensureIndexIsVisible(list.getModel().getSize() - 1);
```

this helped to solve it:
java - ensureIndexIsVisible(int) is not working - Stack Overflow

The result is:

```
if (autoScroll.isSelected()) {
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        list.ensureIndexIsVisible(list.getModel().getSize() - 1);
  }
  });
}
```

Figure 1: An API Discussion in Java Swing Forum.

textual description achieves the best accuracy score. Thus, for different datasets, the best performing approach could be different.

Due to this reason, if we only use text or code or original, then the categorization performance would be poorer on some datasets. To address this problem, in this work, we propose an algorithm that combines text, code, and original to achieve a better performance.

3. OVERALL FRAMEWORK

Figure 2 shows the overall framework of *CASE*. The whole framework includes two phases: model building phase and prediction phase. In the model building phase, our goal is to build a model from the historical API discussions which have known categories. In the prediction phase, this model would be used to predict the category of new API discussions.

Our framework first extracts and separates the textual content from the source code in API discussions. It then represents each API discussion as three documents containing only textual content (text), only code (code), and both textual content and code (original) – Steps 1, 2, and 3. Then, *CASE* parses the content of each document into tokens, removes tokens corresponding to stop words (e.g., I, you, the, and, etc.), stems the remaining tokens (i.e., reduce the tokens to their root forms, e.g., "reading" and "reads" are reduced to "read"), and represents each document as a "bag of words" [2]. Each processed token (aka. term) becomes a feature. Features are various quantifiable characteristics of API discussions that could potentially distinguish different categories of API discussions. After that, in each component, we use our term cache algorithm to select the features (i.e., terms) which have the highest affinity scores to each category – Steps 4, 5, and 6.[5]

Next, each of the three sets of processed documents (text, code and original), is inputted to the respective component of CASE. Each of this component constructs a classifier based on the cached terms (which are treated as features) – Steps 7, 8, and 9. A classifier is a machine learning model which assigns labels (in our case: categories of API discussions) to a data point (in our case: a piece of API discussion) based on its cached terms. By default, we use naive Bayes multinomial as the underlying classifier following the previous study by Hou and Mo [14]. We then blend or combine the 3 classifiers (i.e., text classifier, code classifier, and original classifier) together to construct an *APIComposer* classifier (Step 10).[6]

In the prediction phase, the *APIComposer* classifier is then used to predict the categories of new API discussions. For each API discussion, we first extract different parts of the API discussion to form the 3 documents (code, text, and original) as we do in the model building phase, and investigate the occurrences of the cached terms to form the text, code, and original features – Steps 11, 12, and 13. Next, we input these three sets of features to the *APIComposer* classifier which would input the respective feature set to each of the 3 classifiers built in the model building phase – Step 14. This step would eventually output a prediction result which is the predicted category for a new API discussion (Step 15).

[5]For more details of our term cache algorithm, please refer to Section 4.

[6]For more detail of APIComposer, please refer to Section 5.

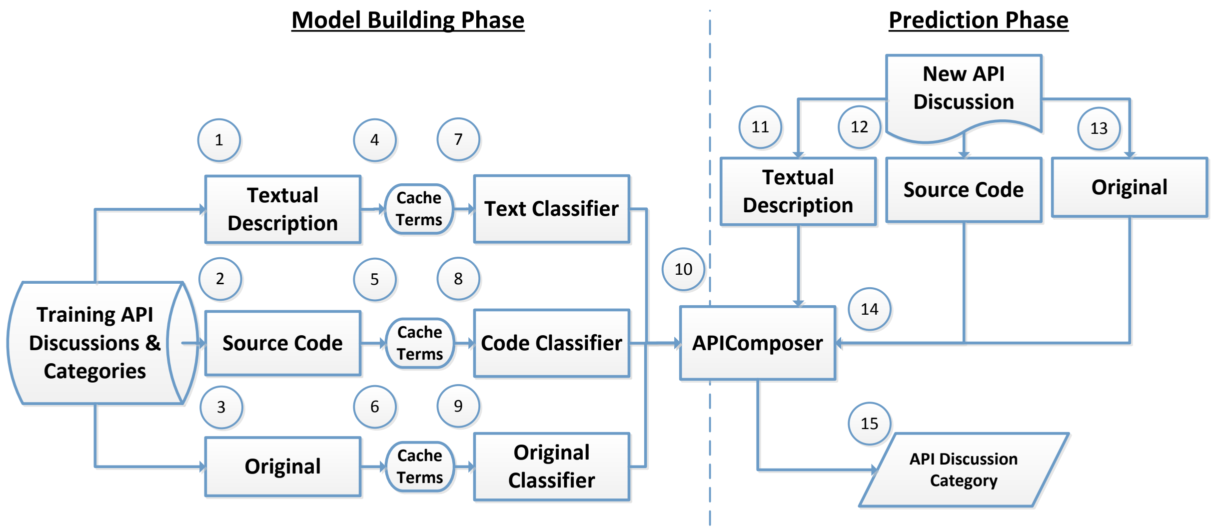

Figure 2: Overall Framework of *CASE*.

4. TERM CACHE ALGORITHM

In *CASE*, all of its 3 components process bags-of-words. Each processed term in the bags-of-words is a feature. Thus, we have a large number of features. In machine learning literature, a feature can be viewed as a dimension, and a data point (i.e., an API discussion) can then be viewed as a point in this high-dimensional space. An overly high number of dimensions can cause the *curse-of-dimensionality* problem [10].

Aside from this, we observe for each category, often some terms appear more often than others, and these terms are important to infer the category of an API discussion. For example, in Data-3.0, terms like "inputborder", "boardertext", "shuffle" appear more often in discussions belonging to category "BoarderandMargin" which corresponds to Swing GUI boarder and margin problem. Also, terms like "drawpanel", "fiddle", "bush" appear more often in discussions belonging to category "drawing" which corresponds to problems in using Swing API to draw customized GUI. To leverage this observation and avoid the *curse-of-dimensionality* problem, we propose our term cache algorithm.

We denote the category of the i^{th} API discussion as c_i, and following vector space modeling [2], we represent the text in the i^{th} API discussion as a vector of weights denoted by $API_i = \langle w_{i,t_1}, w_{i,t_2}, \cdots, w_{i,t_v} \rangle$, where $w_{i,j}$ represents the number of times the term t_j appears in the i^{th} API discussion divided by the total number of terms that appear in the i^{th} API discussion, and v represents the total number of unique terms across the whole API discussion collection. Based on these notations, we define category-term affinity score as follows:

DEFINITION 1. (**Category-term Affinity Score.**) *Consider an API discussion collection A, and a set of categories C. For each category $c \in C$, and term $t_j \in API$, the category-term affinity score of c and t_j, denoted as $Aff(c, t_j)$, is computed as follows:*

$$Aff(c, t_j) = \frac{\sum_{i \in \{i | c_i = c\}} w_{i,j}}{\sum_i w_{i,j}} \quad (1)$$

Table 2 presents an example of dataset with 4 terms and 2 categories (A and B). Considering API discussions 1 and 3 both belong to category A, the affinity score for term 1 and category A is:

Table 2: An Example of Dataset with 4 Terms and 2 Categories (A and B). The Data in the Cells are the Weights.

Discuss. ID	Term 1	Term 2	Term 3	Term 4	Category
1	0.5	0	0.25	0.25	**A**
2	0	0.5	0	0.5	**B**
3	0.4	0.1	0.4	0.1	**A**
4	0.1	0.3	0.2	0.4	**B**

$$Aff(term\ 1, A) = \frac{0.5 + 0.4}{0.5 + 0 + 0.4 + 0.1} = 0.9$$

Similarly, the affinity score for term 2 and category B is:

$$Aff(term\ 2, B) = \frac{0.5 + 0.3}{0 + 0.5 + 0.1 + 0.3} = 0.88$$

For each category, we compute its affinity score to each term, and we rank the terms based on their affinity scores. The higher the affinity score of a term is, the more important the term is to identify the category. Thus, for each category, we cache the terms whose affinity scores appear in the top m% highest affinity scores. Suppose there are l categories, there would be at most $l \times m\% \times v$ terms that are cached. These terms are used as input features to build a classifier. In this paper, by default, we set m% = 5%, i.e., for each category, we cache 5% of the terms.

5. APICOMPOSER: A COMPOSITE ALGORITHM

In *CASE*, we have 3 components: text, code, and original. For each component, we build a classifier based on the cached terms; in total, we have 3 independent classifiers. By default, we use naive Baye multinomial to build the classifiers for the 3 components. Each classifier would output a set of scores for a new API discussion. CASE then composes the three sets of scores together. In this section, first we define the three sets of scores outputted by the three classifiers in Section 5.1. Next, we describe how we combine these scores together to construct the *APIComposer* classifier in Section 5.2.

5.1 Component Scores

As illustrated in Figure 2, our proposed framework has 3 different components which correspond to classifiers built based on the cached terms. Let us refer to them as Cla_t, Cla_c, and Cla_o, respectively. Given an unknown API discussion, Cla_t, Cla_c, and Cla_o output the following text score, code score, and original score, respectively:

DEFINITION 2. *(**Text Scores.**) Consider a training API discussion collection API, and its corresponding text component $TEXT$, and suppose there are L categories. We build a classifier Cla_t trained on $TEXT$. For a new API discussion api, for each category $l \in L$, we use Cla_t to get the likelihood that api will belong to the category l. We refer to these likelihood scores as **text scores**, and denote each of them as $Text(api, l)$, for each $l \in L$.*

DEFINITION 3. *(**Code Scores.**) Consider a training API discussion collection API, and its corresponding code component $CODE$, and suppose there are L categories. We build a classifier Cla_c trained on $CODE$. For a new API discussion api, for each category $l \in L$, we use Cla_c to get the likelihood that api will belong to the category l. We refer to these likelihood scores as **code scores**, and denote each of them as $Code(api, l)$, for each $l \in L$.*

DEFINITION 4. *(**Original Scores.**) Consider a training API discussion collection API, and its corresponding original component $ORIG$, and suppose there are L categories. We build a classifier Cla_o trained on $ORIG$. For a new API discussion api, for each category $l \in L$, we use Cla_o to get the likelihood that api will belong to the category l. We refer to these likelihood scores as **original scores**, and denote each of them as $Orig(api, l)$, for each $l \in L$.*

5.2 APIComposer

As shown in Section 5.1, we can get text scores, code scores, and original scores for each new API discussion api. In this section, we propose *APIComposer*, a composite method which uses all of these 3 scores. A linear combination of text scores, code scores, and original scores is used to compute the final *APIComposer* scores.

DEFINITION 5. *(**APIComposer Scores.**) Consider a training API discussion collection BR and L categories, and the corresponding classifiers for text, code, and original components (Cla_t, Cla_c, and Cla_o), respectively. For a new API discussion api, for each category $l \in L$, we compute its corresponding text, code, and original scores, and then its APIComposer scores, denoted as $Comp(api,l)$, which are linear combinations of 3 scores, defined as follows:*

$$Comp(api, l) = \alpha \times Text(api, l) + \beta \times Code(api, l) + \gamma \times Orig(api, l) \quad (2)$$

In the above equation, $\alpha \in [0, 1]$, $\beta \in [0, 1]$, and $\gamma \in [0, 1]$.

Since there are a total of L categories, for a new API discussion api, after we compute the *APIComposer* scores for each category $l \in L$, the final category for api would be the category which has the highest *APIComposer* scores, i.e.,

$$Category(api) = argmax_{l \in L}\ Comp(api, l) \quad (3)$$

Table 3: Statistics of Collected Datasets.

Data	# Doc.	# Categ.	Componet.	# Terms
Data-1.0	45	10	Text	488
			Code	266
			Original	659
Data-2.0	158	17	Text	1,085
			Code	716
			Original	1,430
Data-3.0	832	8	Text	2,384
			Code	831
			Original	3,481

To automatically produce good α, β, and γ values for *APIComposer*, we propose a greedy algorithm. Algorithm 3 presents the detailed steps to estimate good α, β, and γ values. We initialize α, β, and γ values to 0 at Line 9. Then, we build the classifiers (i.e., Cla_t, Cla_c, and Cla_o) for text, code, and original component using API, and compute their corresponding text, code, and original component scores of API discussions in API at Lines 10, 11 and 12, respectively. Next, we incrementally increase α, β, and γ values (Lines 13 to 15). We increase α, β, and γ values from 0 to 1, in 0.1 increments. We use a rather coarse granularity step (i.e., 0.1) to tune α, β, and γ values to reduce the computational cost in the tuning process. For each configuration of α, β, and γ values, we build a composite model and compute the resultant accuracy using API discussions in API (Lines 16 to 23). Finally, Algorithm 3 returns α, β, and γ values resulting in the best accuracy using the training data (Line 24).

6. EXPERIMENTS AND RESULTS

In this section, we evaluate the effectiveness of *CASE*. The experimental environment is an Intel(R) Core(TM) i5 3.20 GHz CPU, 4GB RAM desktop running Windows 7 (32-bit). We first present our experiment setup, and 5 research questions in Sections 6.1 and 6.2, respectively. We then present our experiment results that answer the 5 research questions (Sections 6.3, 6.4, 6.5, 6.6, and 6.7).

6.1 Experiment Setup

We evaluate *CASE* on the 3 datasets provided by Hou and Mo [14] containing a total of 1,035 API discussions. Table 3 presents the statistics of the collected datasets. The columns correspond to the number of API discussion documents (# Doc.), the number of categories (# Categ.), and the number of unique terms in each component (# Terms). Notice that our datasets are slightly different from the original datasets since we remove one API discussion in Data-1.0 and another API discussion in Data-3.0, since these 2 API discussions do not contain text or code.

We use WVTool [37] to extract terms from these 3 datasets. WVTool is a Java library for statistical language modeling, which is used to create word vector representations of text documents. We use WVTool to tokenize the textual description of API discussion, remove stop words, and do stemming. We remove the terms which appear less than 2 times, since these terms can not be used to identify the categories of API discussions. We implement *CASE* on top of Weka[7] [9].

Stratified ten-fold cross validation [10] is used to evaluate the performance of *CASE*. We randomly divide the dataset into 10 fold-

[7] http://www.cs.waikato.ac.nz/ml/weka/

Estimatevalue(API, $TEXT$, $CODE$, $ORIG$)
Input:
API: Training API Discussion Collection
$TEXT$: Text Component of API
$CODE$: Code Component of API
$ORIG$: Original Component of API
Output: α, β, and γ
Method:
α=$0, \beta = 0, and \gamma = 0$
Build Cla_t from $TEXT$, and compute text scores for each API discussion in API;
Build Cla_c from $CODE$, and compute code scores for each API discussion in API;
Build Cla_o from $ORIG$, and compute original scores for each API discussion in API;
for all α from 0 to 1, every time increase α by 0.1 **do**
  for all β from 0 to 1,every time increase β by 0.1 **do**
    for all γ from 0 to 1,every time increase γ by 0.1 **do**
      for all API Discussion api in API **do**
        Compute *APIComposer* score according to Definition 5;
        Predict the category of api by using Equation 3;
      end for
      Evaluate the performance by computing accuracy;
    end for
  end for
end for
Return α, β, and γ which give the best accuracy

Figure 3: Estimation of Good α, β, and γ Values in *APIComposer*

s. Of these 10 folds, 9 folds are use to train a classifier, while the last one fold (i.e, test fold) is used to evaluate the performance. In the test fold, for an API discussion, if the category we predict is the same as its actual category, we consider it as a *prediction hit*. We iterate the whole process 10 times, and record the average performance across the 10 iterations. The distribution of labels in the training and test folds are the same as the original dataset to simulate the actual usage of *CASE*. Stratified cross validation is a standard evaluation setting, which is widely used in software engineering studies, c.f., [22,28,34,35,38,40]. To evaluate the performance of *CASE*, for each fold, we compute accuracy which is defined as the ratio between the total number of *prediction hit* and the total number of API discussions in our test fold. We report the average accuracy across the 10 iterations.

6.2 Research Questions

We are interested to answer the following research questions:

RQ1 *How effective is CASE? How much improvement could our proposed approach gain over the baseline method by Hou and Mo?*

Hou and Mo propose the usage of naive Bayes multinomial (NBM) to solve the content categorization of API discussions problem [14]. In this research question, we investigate the extent our approach (*CASE*) outperforms this state-of-the-art approach. To answer this research question, we compare the average accuracy of *CASE* with that of NBM for each of the 3 datasets.

RQ2 *Can the term cache algorithm and APIComposer improve the performance of CASE?*

CASE first applies our term cache algorithm to cache the terms for each category, and then apply *APIComposer* to combine 3 classifiers. In this research question, we investigate the performance of *CASE* without the term cache algorithm, and without APIComposer.

To answer this research question, we first remove the term cache algorithm from *CASE*, and we directly combine these 3 components, we refer to this algorithm as $CASE^{Basic}$. Next, we remove the *APIComposer* from *CASE*, i.e., we do not consider the combination of 3 components, we refer to the text, code, and original components with the term cache algorithm as $Text^C$, $Code^C$, and $Orig^C$, respectively. We compare the average accuracy of CASE with those of $CASE^{Basic}$, $Text^C$, $Code^C$, and $Orig^C$ for each of the 3 datasets, respectively.

RQ3 *Do different numbers of cached terms affect the performance of CASE?*

By default, we cache 5% of the terms. We investigate whether different percentages of cached terms would affect the performance of *CASE*. To answer this research question, we vary the percentages of terms cached from 1% to 20%.

RQ4 *What are the best features (i.e., terms) for discriminating different categories?*

Aside from producing a model that can identify different categories of API discussions, we are also interested in finding discriminative features that could help in distinguishing different categories of API discussions. In this research questions, we would like to identify these features (i.e., terms) that we extract from the textual description of API discussions. To answer this research question, we compute the term affinity scores for all the terms and categories we considered.

RQ5: *How much time does it take for CASE to run?*

The efficiency of *CASE* would affect its usability. In this question, we investigate whether the runtime of *CASE* is reasonable. To answer this research question, we report the model building and prediction time of *CASE* and compare them with those of naive Bayes multinomial (NBM) used by Hou and Mo [14].

Table 4: Experiment Results for *CASE* Compared with Naive Bayes Multinomial (NBM).

Datasets	CASE	Data Type	NBM	Improvement
Data-1.0	0.6889	Text	0.5333	**29.17%**
		Code	0.3778	**82.35%**
		Original	0.6222	**10.71%**
Data-2.0	0.7658	Text	0.6772	**13.08%**
		Code	0.6519	**17.48%**
		Original	0.6962	**10%**
Data-3.0	0.9615	Text	0.9411	**2.17%**
		Code	0.9291	**3.49%**
		Original	0.9315	**3.26%**

Table 5: Experiment Results for *CASE* Compared with *CASE*^{BASIC}.

Datasets	CASE	$CASE^{BASIC}$	Improvement
Data-1.0	0.6889	0.6000	**14.81%**
Data-2.0	0.7658	0.6962	**10.00%**
Data-3.0	0.9615	0.9507	**1.14%**

6.3 RQ1: Performance of CASE

Table 4 compares the accuracy of *CASE* and that of naive Bayes multinomial (NBM). The accuracy of *CASE* varies from 0.6889 - 0.9615. We notice the improvement of *CASE* over NBM is substantial. To compare with the best performance of NBM (we choose original[8] for Data-1.0, and Data-2.0, and text[9] for Data-3.0), *CASE* outperforms NBM by 10.71%, 10%, and 2.17% for Data-1.0, Data-2.0, and Data-3.0, respectively.

Notice for Data-3.0, the improvement of *CASE* over NBM is not as high as those for the other 2 datasets; this is because the accuracy for the baseline algorithm is already around 94%, and *CASE* improves it from 94% to 96%. Considering error rate [10], our improvement for Data-3.0 is substantial. For Data-3.0, the error rate for NBM is $(1 - 0.9411) = 0.0589$, while the error rate for *CASE* is $(1 - 0.9615) = 0.0385$. *CASE* improves the error rate of NBM by 39.21%. Thus, the improvement that *CASE* achieves over NBM for all datasets is substantial.

6.4 RQ2: Performance of Term Cache and API-Composer Algorithms

Table 5 compares the accuracy of *CASE* and *CASE*^{BASIC}. The accuracy of *CASE*^{BASIC} varies from 0.6 – 0.9507. We notice the improvement of *CASE* over $CASEBASIC$ is substantial, *CASE* outperforms *CASE*^{BASIC} by 14.81%, 10%, and 1.14% for Data-1.0, Data-2.0, and Data-3.0, respectively.

Table 6 compares the accuracy of *CASE* with $Text^C$, $Code^C$, and $Orig^C$. We notice that the improvement of *CASE* over $Text^C$, $Code^C$, and $Orig^C$ are substantial. *CASE* outperforms $Text^C$ by 19.23%, 10%, and 3.47% for Data-1.0, Data-2.0, and Data-3.0, respectively; *CASE* outperforms $Code^C$ by 93.75%, 11.01%, and 22.25% for Data-1.0, Data-2.0, and Data-3.0, respectively; *CASE* outperforms $Orig^C$ by 14.81%, 2.54%, and 8.07% for Data-1.0, Data-2.0, and Data-3.0, respectively.

[8]We extract both textual descriptions and code from bug reports.
[9]We only take textual descriptions from bug reports.

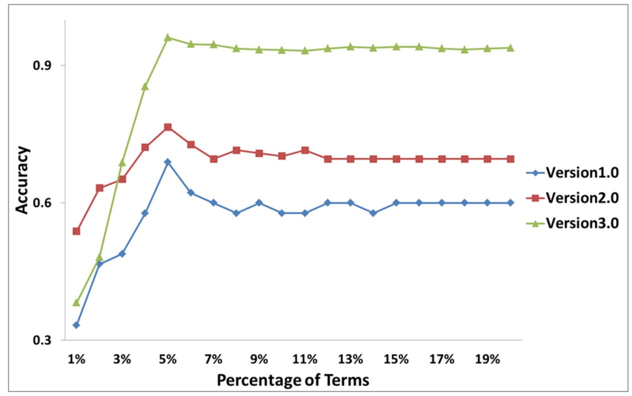

Figure 4: Experiment Results of *CASE* with Textual Terms from 1% to 20%.

6.5 RQ3: Effect of Varying the Number of Terms

We vary the percentage of cached terms from 1% to 20% for Data-1.0, Data-2.0, and Data-3.0, respectively. Figure 4 presents the experiment results of *CASE* with different percentages of cached terms. We notice that for very small percentages of terms, such as 1% to 4%, the accuracy is relatively low. For example, in Data-3.0, the accuracy for 1%, 2%, 3%, and 4% are 0.3822, 0.4820, 0.6887, and 0.8546, respectively. Then the accuracy achieves a peak value at 5% percent, i.e., 0.6889, 0.7658, and 0.9615, for Data-1.0, Data-2.0, and Data-3.0, respectively. When the percentage of cached terms increases from 6% to 20%, the accuracy of *CASE* is stable. For example, for Data-3.0, the accuracy for 6%, 10%, 15%, and 20% are 0.9471, 0.9338, 0.9411, 0.9387, respectively.

6.6 RQ4: Important Terms for API Discussion Categorization

From the API discussions, we extract thousands of features (i.e., terms). For this RQ, we also report discriminative features from the thousands of features. We extract the top-10 terms based on their category-term affinity score. Tables 7, 8, and 9 present the top-10 terms (considering both text and code) per category for Data-1.0, Data-2.0, and Data-3.0, respectively.

Some terms are good indicators to identify the category of an API discussion. For example, for the category "borderandMargin", terms "inputborder", "compoundborder", "titledborder", "borders" are all good indicators; for the category "textIconPosition", terms "basicbutton", "windowbutton", "settextposit", "alight", "settrenderer" are all good indicators.

6.7 RQ5: Time Efficiency of CASE

Table 10 presents the average model building time and prediction time it takes for the 2 algorithms, i.e., CASE and NBM. We notice that the model building time and prediction time of CASE are longer than those of NBM. However, they are still reasonable. On average, we need 4.33 seconds to train a model, and 0.44 seconds to predict the categories of API discussions in a test set. Note that the model building phase can be done offline (e.g., overnight).

7. RELATED WORK

In this section, we first introduce Hou and Mo's work which is most related to ours in Section 7.1. Next, we briefly review studies on software information sites in Section 7.2. Finally, we describe

Table 6: Experiment Results for *CASE* Compared with $Text^C$, $Code^C$, and $Orig^C$, Respectively.

Datasets	*CASE*	$Text^C$	Improvement	$Code^C$	Improvement	$Orig^C$	Improvement
Data-1.0	0.6889	0.5778	**19.23%**	0.3556	**93.75%**	0.6000	**14.81%**
Data-2.0	0.7658	0.6962	**10.00%**	0.6899	**11.01%**	0.7468	**2.54%**
Data-3.0	0.9615	0.9303	**3.47%**	0.7476	**22.25%**	0.9014	**8.04%**

Table 7: Top-10 Terms (Considering Both Text and Code) Per Category (Excluding the Category Others) for Data-1.0.

action	border	dispose	drawing	focus	icon	layout	rendererEditor	title
absolut	surround	parameter	trans	foc	setcontentpan	detail	jtree	setundecor
tooltip	createborder	forgot	layer	keyboard	sourc	repl	child	minim
myact	javadoc	dispos	chessboard	getbackground	geticon	layoutmanager	nod	maxim
print	occup	proces	width	addfocuslistener	docum	constraint	treecellrenderer	titl
separ	model	window	height	focusgain	entr	poster	treecelledit	decor
command	dear	stupid	math	focusev	horizon	shift	renderer	meta
attribut	consider	clos	replac	focuslost	icon	gridbag	jcheckbox	advis
alignm	setborder	perform	graph	affect	cast	gridwidth	listen	hid
setaction	lower	textfield	drawstr	square	getwidth	nest	accord	vis
const	border	expect	getcompon	plat	background	cover	figur	system

Table 8: Top-10 Terms (Considering Both Text and Code) Per Category for Data-2.0.

action	borderAndMar.	defaultButton	dispose	drawing	dynamicHie.	focus	layout	loadingIcons
myact	createtitleborder	initialvalu	memor	painticon	validatetre	keyfocus	constraint	seticonim
attribut	getborder	messagetyp	garb	getconwidth	stack	veto	poster	director
indentif	abstractborder	optiontyp	report	dash	subcomponent	cares	weightx	jpeg
lot	matteborder	defaultbutton	twic	geticonheight	revel	focusgain	gridx	ioexcept
moment	compoundborder	keylistener	lock	draw	revalis	marc	smaller	shuffl
creatborder	emptyborder	getkeycod	prim	rot	lightweight	testfoc	bigger	geticon
correspons	painborder	setdefaultbutton	rock	drawstr	progres	grabfoc	detail	path
iter	margin	fun	getjbutton	bufferedim	root	clearfocus	port	bles
clean	occuo	keybind	jtextfield	bunch	neces	focuslistenr	crus	environment
statement	pixel	bind	shuttl	scal	invalis	focusev	gridwidth	recogn
mouseMotion.	**OOP**	**rendererEditor**	**social**	**textIconPos.**	**threading**	**titleBar**	**titleBarFont**	
detect	illegalaugument	customrenderer	netbean	dict	timer	captur	metatitlepan	
visiblerect	getlogger	selectionback	additem	basicbutton	isdisplay	maxim	titlefont	
awteventlistener	mymainfram	setrenderer	combobox	windowsbutton	blink	minim	myfont	
head	logger	defaultrenderer	shout	triv	fetch	renam	dialogu	
mouselistener	login	listcellrenderer	flag	imageur	getlayout	observ	tall	
requirement	invocationtarget	cellhasfoc	alloc	alttext	getloc	char	getlayeredpan	
setcur	jam	editor	explan	verticalalignm	record	repla	getcomponent	
convers	dead	jtree	bug	jradiobutton	upload	opinion	font	
mousemov	jerom	treecelledit	opens	swingconst	rect	decor	ital	
mousedrag	guid	listen	extr	align	swingworker	entr	setfont	

Table 9: Top-10 Terms (Considering Both Text and Code) Per Category for Data-3.0.

borderAndMargin	dispose	drawing	dynamicHierarchy	focus	layout	*textIconPosition*	titleBar
accrod	filechooser	transluc	prgsitem	keyboardfocus	layout	basicbutton	setundecord
inputborder	cancelselect	phot	typeitem	veto	diction	windowsbutton	iconif
inparam	opens	getdevic	typemenu	firstfield	proport	layoutlabel	trayicon
bordertext	freed	logger	simulation	secondfield	closest	settextposit	spacebar
view	profiler	getlogger	unformat	testfoc	former	justf	pit
compoundborder	demostr	newx	framepanel	focusmanager	nextint	alight	removetitlebar
irrespect	childfram	movement	verticalpan	focustpolic	quadr	settrenderer	nullif
titledborder	methot	seticon	jscrollbar	getfocus	ipad	textposit	setdecor
borders	closewindow	drawgraph	reload	clearfocus	setconstraint	gettrenderer	myrootpaneu
inputparameter	wish	drawpanel	disappear	getfocusowner	btnpanel	cellrenderer	reimpl

several studies that categorize various software artifacts in in Section 7.3.

7.1 Content Categorization

To our best knowledge, Hou and Mo's work is the most related to ours [14]. Hou and Mo proposed the problem of content categorization of API discussion, and they solved the problem by leveraging naive Bayes multinomial. They collected 3 API discussion datasets from Swing forums, and the experiment results showed naive Bayes multinomial achieved a reasonable performance. Our work extends theirs; we propose a more accurate algorithm for the

Table 10: Average Model Building Time and Prediction Time (Seconds) for CASE and NBM.

Datasets	Model Building Time (s)		Prediction Time (s)	
	CASE	NBM	*CASE*	NBM
Data-1.0	0.265	0.026	0.004	0.002
Data-2.0	0.673	0.034	0.003	0.007
Data-3.0	12.062	0.081	1.296	0.007

same problem. We first create 3 documents per API discussion: text, code, and original. And based on these three, we cache terms for each category. Finally, we combine 3 classifiers built using the cached terms from these 3 sets of documents. The experiment results show that our algorithm achieves a substantial improvement over naive Bayes multinomial.

7.2 Software Information Sites

Software information sites refer to the online media (e.g., social coding sites, online forums, Q&A sites) which help software engineers improve their performance in the whole lifecycle of software development, maintenance and test processes [40]. There have been a number of studies on software information sites and social media for software engineering [5, 8, 12, 15, 23, 30, 31, 40]. Storey et al. [30] and Begel et al. [5] write two position papers to describe the future of research in social media for software engineering. They propose a set of research questions at community, project, and individual development level. Hong et al. study the developer social networks in open source projects, and compare them with the general social networks such as Facebook, twitter [12]. Gottipati et al. develop a semantic search engine to automatically infer tags for posts in software engineer forums and recover relevant answers according to user queries [8]. Surian et al. mine the collaboration patterns from a large-scale developer social network extracted from SourceForge.Net, and recommend developer based on these mined patterns [31]. Prasetyo et al. propose an automated technique to categorize software related microblogs into different labels [23]. Barua et al. use LDA to automatically infer the main topics in StackOverflow [3]. Jiang et al. study the project dissemination phenomenon in GitHub, and they conclude that social relationships are not reciprocal, and social links play a important role for project dissemination [15]. Xia et al. propose a composite method which combines 3 components (i.e., multi-label ranking component, similarity based ranking component, and tag-term based ranking component) to recommend tags in software information sites [40].

Hou and Li study the API usage obstacles on 172 API discussions in Swing forums, and they analyze the root cause of these obstacles [13]. Rupakheti and Hou perform an empirical study on API usage problem in Swing forum, and build a critic to advise the usage of an API [25]. Zhang and Hou apply natural language processing and sentiment analysis techniques to extract problematic API features from forum discussions [41].

Our study is orthogonal to the above studies; after a developer posts an API discussion in an online forum, our tool automatically predicts its category.

7.3 Software Categorization

There have been a number of studies that categorize various software artifacts [1,4,11,16,17,21,22,29,33,35,36,38]. Baruchelli and Giancarlo propose a fuzzy set based approach to classify software components [4]. Kawaguchi et al. propose a tool named MUDABlue which not only automatically categorizes software systems, but also extract categories from the software systems collection automatically [16]. Sandhu et al. use different pure and hybrid approaches such as Probabilistic Latent Semantic Analysis (PLSA) approach, LSA, Singular Value Decomposition (SVD) technique, and LSA Semi-Discrete Matrix Decomposition (SDD) to classify software components [26]. Antoniol et al. apply text mining techniques to distinguish bug reports from enhancements on Mozilla, Eclipse, and JBoss [1]. Kim et al. use machine learning techniques to classify if a change set is clean or buggy [17]. Hindle et al. build an automated classifier to assign a change request to one of the 5 categories: corrective, adaptive, perfective, feature addition, and non-functional improvement, using machine learning techniques [11]. Tian et al. propose a semi-supervised learning algorithm to identify Linux bug fixing patches based on the changes and commit messages recorded in code repositories [35]. Menzies and Marcus propose a machine learning algorithm to predict the severity of a bug report [22]. Mcmillan et al. use Application Programming Interface (API) calls from third-party libraries as attributes to automatically classify software applications [21]. Thung et al. collect various features from bug and code repositories to predict the type of a defect [33]. Tian et al. propose DRONE which predicts the priority of a bug report by leveraging a logistic regression algorithm [36]. Somasundaram and Murphy propose the usage of LDA to predict the correct component of a bug report [29]. Xia et al. use genetic algorithm to combine different multi-label learning algorithms to categorize failure reports [38].

Our study is orthogonal to the above studies; we categorize a different kind of software artifacts namely API discussions.

8. THREATS TO VALIDITY

In this section, we highlight threats to internal validity, external validity, and construct validity.

Threats to internal validity relate to errors in our experiments. We have double checked our experiments and the datasets collected from the 4 projects, still there could be errors that we did not notice. We use the same datasets as those used by Hou and Mo [14].

Threats to external validity relate to the generalizability of our results. We have evaluated our approach using 1,035 API discussions from Swing forum, and investigate 25 different categories. In the future, we plan to reduce this threat further by analyzing more API discussions from more software forums.

Threats to construct validity refer to the suitability of our evaluation measures. We use the average accuracy scores as our evaluation measure which is also used by past studies to evaluate the effectiveness of a prediction technique in various software engineering studies [11, 14, 17, 28]. Thus, we believe there is little threat to construct validity.

9. CONCLUSION AND FUTURE WORK

In this paper, we propose a more accurate algorithm named *CASE* for content categorization of API discussions. *CASE* has 3 components which analyze an API discussion considering different kinds of data: text, code, and both text and code (original). For each of the component, *CASE* first caches terms that have the highest affinity scores to each category, and then builds a classifier using the cached terms. Finally, *CASE* combines these 3 classifiers to achieve a better performance. The experiment results on 3 API discussion datasets shows that *CASE* achieves accuracy scores of 0.69, 0.77, and 0.96 for each of the 3 datasets respectively, which outperforms the accuracy scores of the method used by Hou and Mo by 11%, 10%, and 2%, respectively.

In the future, we plan to evaluate CASE using more API discussions from various online forums, and evaluate CASE using different underlying classifiers (such as SVM and decision tree [10]), analyze the misclassified cases to understand why our approach fails to correctly classify a number of API discussions, and develop a more accurate algorithm (such as ensemble learning algorithms [6]). We also plan to evaluate our algorithms using the longitudinal data setup as described in [32, 39].

ACKNOWLEDGMENTS

This research is sponsored in part by NSFC Program (No.61103032) and National Key Technology R&D Program of the Ministry of Science and Technology of China (2014BAH24F02). We thank Daqing Hou and Lingfeng Mo for making the 3 datasets available for public use.

10. REFERENCES

[1] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, page 23. ACM, 2008.

[2] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[3] A. Barua, S. W. Thomas, and A. E. Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, pages 1–36, 2012.

[4] F. Baruchelli and G. Succi. A fuzzy approach to faceted classification and retrieval of reusable software components. *ACM SIGAPP Applied Computing Review*, 5(1):15–20, 1997.

[5] A. Begel, R. DeLine, and T. Zimmermann. Social media for software engineering. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 33–38. ACM, 2010.

[6] T. G. Dietterich. Ensemble methods in machine learning. In *Multiple classifier systems*, pages 1–15. Springer, 2000.

[7] G. Fischer. Cognitive view of reuse and redesign. *IEEE Software*, 4(4):60–72, 1987.

[8] S. Gottipati, D. Lo, and J. Jiang. Finding relevant answers in software forums. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 323–332. IEEE Computer Society, 2011.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[10] J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques*. Morgan kaufmann, 2006.

[11] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. Automatic classication of large changes into maintenance categories. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 30–39. IEEE, 2009.

[12] Q. Hong, S. Kim, S. Cheung, and C. Bird. Understanding a developer social network and its evolution. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 323–332. IEEE, 2011.

[13] D. Hou and L. Li. Obstacles in using frameworks and apis: An exploratory study of programmers' newsgroup discussions. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 91–100. IEEE, 2011.

[14] D. Hou and L. Mo. Content categorization of api discussions. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 60–69. IEEE, 2013.

[15] J. Jiang, L. Zhang, and L. Li. Understanding project dissemination on a social coding site. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 132–141. IEEE, 2013.

[16] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Mudablue: An automatic categorization system for open source repositories. *Journal of Systems and Software*, 79(7):939–953, 2006.

[17] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181–196, 2008.

[18] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 199–206. IEEE, 2004.

[19] A. McCallum, K. Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Citeseer, 1998.

[20] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi. Building more usable apis. *Software, IEEE*, 15(3):78–86, 1998.

[21] C. McMillan, M. Linares-Vásquez, D. Poshyvanyk, and M. Grechanik. Categorizing software applications for maintenance. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 343–352. IEEE, 2011.

[22] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 346–355. IEEE, 2008.

[23] P. K. Prasetyo, D. Lo, P. Achananuparp, Y. Tian, and E.-P. Lim. Automatic classification of software related microblogs. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 596–599. IEEE, 2012.

[24] M. P. Robillard and R. Deline. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[25] C. R. Rupakheti and D. Hou. Evaluating forum discussions to inform the design of an api critic. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 53–62. IEEE, 2012.

[26] P. S. Sandhu, J. Singh, and H. Singh. Approaches for categorization of reusable software components. *Journal of Computer Science*, 3(5), 2007.

[27] F. Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[28] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 249–258. IEEE, 2010.

[29] K. Somasundaram and G. C. Murphy. Automatic categorization of bug reports using latent dirichlet allocation.

In *Proceedings of the 5th India Software Engineering Conference*, pages 125–130. ACM, 2012.

[30] M. Storey, C. Treude, A. van Deursen, and L. Cheng. The impact of social media on software engineering practices and tools. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 359–364. ACM, 2010.

[31] D. Surian, D. Lo, and E.-P. Lim. Mining collaboration patterns from a large developer network. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 269–273. IEEE, 2010.

[32] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 365–375. ACM, 2011.

[33] F. Thung, D. Lo, and L. Jiang. Automatic defect categorization. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 205–214. IEEE, 2012.

[34] F. Thung, D. Lo, and J. L. Lawall. Automated library recommendation. In *WCRE*, pages 182–191, 2013.

[35] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 386–396. IEEE, 2012.

[36] Y. Tian, D. Lo, and C. Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 200–209. IEEE, 2013.

[37] D. Tutorial. The word vector tool.

[38] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang. Towards more accurate multi-label software behavior learning. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 134–143. IEEE, 2014.

[39] X. Xia, D. Lo, X. Wang, and B. Zhou. Accurate developer recommendation for bug resolution. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 72–81. IEEE, 2013.

[40] X. Xia, D. Lo, X. Wang, and B. Zhou. Tag recommendation in software information sites. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 287–296. IEEE Press, 2013.

[41] Y. Zhang and D. Hou. Extracting problematic api features from forum discussions. In *Program Comprehension (ICPC), 2013 IEEE 21th International Conference on*, pages 142–151, 2013.