5-2014

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

Follow this and additional works at: http://ink.library.smu.edu.sg/sis_research

Part of the Computer Sciences Commons

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). Research Collection School Of Information Systems. Available at: http://ink.library.smu.edu.sg/sis_research/2420

This Conference Paper is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.
Towards More Accurate Content Categorization of API Discussions

Bo Zhou, Xin Xia, David Lo, Cong Tian, and Xinyu Wang

1College of Computer Science and Technology, Zhejiang University, Hangzhou, China
2School of Information Systems, Singapore Management University, Singapore
3IGIT and ISN Lab, Xi’an 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 categories 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 naïve 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

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. 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 categories 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., naïve Bayes [10], kNN [10], SVM [10], etc. Hou and Mo investigated the performance of naïve 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. In this paper, we propose a Cache-based compositE algorithm, short formed as CASE, to improve the accuracy of the content.

1https://community.oracle.com/thread/2594342
2For 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]. 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. 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 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.

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Text</th>
<th>Code</th>
<th>Original</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-1.0</td>
<td>0.5333</td>
<td>0.3778</td>
<td>0.6222</td>
</tr>
<tr>
<td>Data-2.0</td>
<td>0.6772</td>
<td>0.6519</td>
<td>0.6962</td>
</tr>
<tr>
<td>Data-3.0</td>
<td>0.9411</td>
<td>0.9291</td>
<td>0.9315</td>
</tr>
</tbody>
</table>

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
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.

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).

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).

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

For more detail of APIComposer, please refer to Section 5.
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 Swing GUI 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 \( j \)-th API discussion as a vector of weights denoted by \( API_j = \{ w_{i,j} \} \), where \( w_{i,j} \) represents the number of times the term \( t_j \) appears in the \( j \)-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:

\[
Aff(c, t_j) = \frac{\sum_{i \in (c \cap \{ w_i \})} w_{i,j}}{\sum_{i} w_{i,j}}
\]  

(1)

Table 2 presents an example of dataset with 4 terms and 2 categories (A and B). The Data in the Cells are the Weights.

<table>
<thead>
<tr>
<th>Discuss. ID</th>
<th>Term 1</th>
<th>Term 2</th>
<th>Term 3</th>
<th>Term 4</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5</td>
<td>0</td>
<td>0.25</td>
<td>0.25</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0.5</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>0.4</td>
<td>0.1</td>
<td>0.4</td>
<td>0.1</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>0.3</td>
<td>0.2</td>
<td>0.4</td>
<td>B</td>
</tr>
</tbody>
</table>

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

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

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

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 com-
bine 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 $C_{\text{t}}$, $C_{\text{a}}$, and $C_{\text{o}}$, respectively. Given an unknown API discussion, $C_{\text{t}}$, $C_{\text{a}}$, and $C_{\text{o}}$, 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 $\text{TEXT}$, and suppose there are $L$ categories. We build a classifier $C_{\text{t}}$, trained on $\text{TEXT}$. For a new API discussion $\text{api}$, for each category $l \in L$, we use $C_{\text{t}}$ to get the likelihood that $\text{api}$ will belong to the category $l$. We refer to these likelihood scores as text scores, and denote each of them as $\text{Text}(\text{api}, l)$, for each $l \in L$.

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

**Definition 4.** (Original Scores.) Consider a training API discussion collection API, and its corresponding original component $\text{ORIG}$, and suppose there are $L$ categories. We build a classifier $C_{\text{o}}$, trained on $\text{ORIG}$. For a new API discussion $\text{api}$, for each category $l \in L$, we use $C_{\text{o}}$, to get the likelihood that $\text{api}$ will belong to the category $l$. We refer to these likelihood scores as original scores, and denote each of them as $\text{Orig}(\text{api}, l)$, 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 $\text{api}$. 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 $\text{BR}$ and $L$ categories, and the corresponding classifiers for text, code, and original components ($C_{\text{t}}$, $C_{\text{a}}$, and $C_{\text{o}}$), respectively. For a new API discussion $\text{api}$, for each category $l \in L$, we compute its corresponding text, code, and original scores, and then its APIComposer scores, denoted as $\text{Comp}(\text{api}, l)$, which are linear combinations of 3 scores, defined as follows:

$$\text{Comp}(\text{api}, l) = \alpha \times \text{Text}(\text{api}, l) + \beta \times \text{Code}(\text{api}, l) + \gamma \times \text{Orig}(\text{api}, l)$$

(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 $\text{api}$, after we compute the APIComposer scores for each category $l \in L$, the final category for $\text{api}$ would be the category which has the highest APIComposer scores, i.e.,

$$\text{Category}(\text{api}) = \arg\max_{l \in L} \text{Comp}(\text{api}, l)$$

(3)

To automatically produce good $\alpha$, $\beta$, and $\gamma$ values for APIComposer, we propose a greedy algorithm. Algorithm 3 presents the detailed steps to estimate good $\alpha$, $\beta$, and $\gamma$ values. We initialize $\alpha$, $\beta$, and $\gamma$ values to 0 at Line 9. Then, we build the classifiers (i.e., $C_{\text{t}}$, $C_{\text{a}}$, and $C_{\text{o}}$) 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 $\alpha$, $\beta$, and $\gamma$ values (Lines 13 to 15). We increase $\alpha$, $\beta$, and $\gamma$ values from 0 to 1, in 0.1 increments. We use a rather coarse granularity step (i.e., 0.1) to tune $\alpha$, $\beta$, and $\gamma$ values to reduce the computational cost in the tuning process. For each configuration of $\alpha$, $\beta$, and $\gamma$ values, we build a composite model and compute the resultant accuracy using API discussions in API (Lines 16 to 23). Finally, Algorithm 3 returns $\alpha$, $\beta$, and $\gamma$ 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 used 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 Weka7 [9].

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

<table>
<thead>
<tr>
<th>Table 3: Statistics of Collected Datasets.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
</tr>
<tr>
<td>Data-1.0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Data-2.0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Data-3.0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

7http://www.cs.waikato.ac.nz/ml/weka/
1: Estimatevalue(API, TEXT, CODE, ORIG)
2: Input:
3: API: Training API Discussion Collection
4: TEXT: Text Component of API
5: CODE: Code Component of API
6: ORIG: Original Component of API
7: Output: α, β, and γ
8: Method:
9: α=0, β=0, and γ=0
10: Build Clao from TEXT, and compute text scores for each API discussion in API;
11: Build Clac from CODE, and compute code scores for each API discussion in API;
12: Build Clao from ORIG, and compute original scores for each API discussion in API;
13: for all α from 0 to 1, every time increase α by 0.1 do
14: for all β from 0 to 1, every time increase β by 0.1 do
15: for all γ from 0 to 1, every time increase γ by 0.1 do
16: for all API Discussion api in API do
17: Compute APIComposer score according to Definition 5;
18: Predict the category of api by using Equation 3;
19: end for
20: Evaluate the performance by computing accuracy;
21: end for
22: end for
23: end for
24: 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 used 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 CASEBasic. 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 TextC, CodeC, and OrigC, respectively. We compare the average accuracy of CASE with those of CASEBasic, TextC, CodeC, and OrigC 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].

100
Table 4: Experiment Results for CASE Compared with Naive Bayes Multinomial (NBM).

<table>
<thead>
<tr>
<th>Datasets</th>
<th>CASE</th>
<th>Data Type</th>
<th>NBM</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-1.0</td>
<td>0.6889</td>
<td>Text</td>
<td>0.5333</td>
<td>29.17%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code</td>
<td>0.3778</td>
<td>82.35%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Original</td>
<td>0.6222</td>
<td>10.71%</td>
</tr>
<tr>
<td>Data-2.0</td>
<td>0.7658</td>
<td>Text</td>
<td>0.6772</td>
<td>17.48%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code</td>
<td>0.6519</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Original</td>
<td>0.6962</td>
<td>0.7658</td>
</tr>
<tr>
<td>Data-3.0</td>
<td>0.9615</td>
<td>Text</td>
<td>0.9411</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code</td>
<td>0.9291</td>
<td>3.49%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Original</td>
<td>0.9315</td>
<td>2.17%</td>
</tr>
</tbody>
</table>

Table 5: Experiment Results for CASE Compared with CASE\textsuperscript{BASIC}.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>CASE</th>
<th>CASE\textsuperscript{BASIC}</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-1.0</td>
<td>0.6889</td>
<td>0.6000</td>
<td>14.81%</td>
</tr>
<tr>
<td>Data-2.0</td>
<td>0.7658</td>
<td>0.6962</td>
<td>10.00%</td>
</tr>
<tr>
<td>Data-3.0</td>
<td>0.9615</td>
<td>0.9507</td>
<td>1.14%</td>
</tr>
</tbody>
</table>

6.3 RQ1: Performance of CASE

Table 4 compares the accuracy of CASE and that of naïve 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\textsuperscript{a} for Data-1.0, and Data-2.0, and text\textsuperscript{b} 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\textsuperscript{BASIC}. The accuracy of CASE\textsuperscript{BASIC} varies from 0.6 – 0.9507. We notice the improvement of CASE over CASE\textsuperscript{BASIC} is substantial, CASE outperforms CASE\textsuperscript{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\textsuperscript{C}, Code\textsuperscript{C}, and Orig\textsuperscript{C}. We notice that the improvement of CASE over Text\textsuperscript{C}, Code\textsuperscript{C}, and Orig\textsuperscript{C} are substantial. CASE outperforms Text\textsuperscript{C} by 19.23%, 10%, and 3.47% for Data-1.0, Data-2.0, and Data-3.0, respectively; CASE outperforms Code\textsuperscript{C} by 93.75%, 11.01%, and 22.25% for Data-1.0, Data-2.0, and Data-3.0, respectively; CASE outperforms Orig\textsuperscript{C} by 14.81%, 2.54%, and 8.07% for Data-1.0, Data-2.0, and Data-3.0, respectively.

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.

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
several studies that categorize various software artifacts 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
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 naïve 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 an 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 MUD-ABlue 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 show that CASE achieves accuracy scores of 0.69, 0.77, and 0.96 for each of the 3 datasets respectively, which outper-
forms 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


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


