SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

January 2017, Vol. 60 012107:1–012107:16
doi: 10.1007/s11432-014-0372-y

Mining authorship characteristics in bug repositories

He JIANG[1,2,3]*, Jingxuan ZHANG[1,2], Hongjing MA[1,2],
Najam NAZAR[1,2] & Zhilei REN[1,2]

[1]*School of Software, Dalian University of Technology, Dalian* 116621*, China;*
[2]*Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, Dalian* 116621*, China;*
[3]*State Key Laboratory of Software Engineering, Wuhan University, Wuhan* 430072*, China*

Received July 8, 2015; accepted August 27, 2015; published online November 23, 2016

Abstract Bug reports are widely employed to facilitate software tasks in software maintenance. Since bug reports are contributed by people, the authorship characteristics of contributors may heavily impact the performance of resolving software tasks. Poorly written bug reports may delay developers when fixing bugs. However, no in-depth investigation has been conducted over the authorship characteristics. In this study, we first leverage byte-level N-grams to model the authorship characteristics and employ Normalized Simplified Profile Intersection (NSPI) to identify the similarity of the authorship characteristics. Then, we investigate a series of properties related to contributors' authorship characteristics, including the evolvement over time and the variation among distinct products in open source projects. Moreover, we show how to leverage the authorship characteristics to facilitate a well-known task in software maintenance, namely Bug Report Summarization (BRS). Experiments on open source projects validate that incorporating the authorship characteristics can effectively improve a state-of-the-art method in BRS. Our findings suggest that contributors should retain stable authorship characteristics and the authorship characteristics can assist in resolving software tasks.

Keywords software maintenance, bug repositories, authorship characteristics, bug report summarization

Citation Jiang H, Zhang J X, Ma H J, et al. Mining authorship characteristics in bug repositories. Sci China Inf Sci, 2017, 60(1): 012107, doi: 10.1007/s11432-014-0372-y

1 Introduction

Recent years have witnessed the growing interests from the society of software engineering in Mining Bug Repositories (MBR) to improve software quality. As stated in [1], more than 45% efforts have been spent over software maintenance to fix bugs. To store and manage increasing number of bugs, many software projects employ bug repositories (also known as bug tracking systems) in software maintenance. Up to May, 2014, a well-known bug repository, namely Bugzilla, has been publicly used by 148 companies, organizations, and projects, including Mozilla, Eclipse, Gnome, and GCC. In addition, there may also exist at least 10 times as many private Bugzilla installations (see http://www.bugzilla.org/installation-list/). Facing vast bug reports in bug repositories, many researchers investigate how to assist software tasks by MBR. Some typical tasks include bug triage [2,3], fault prediction [4,5], bug location [6,7], and

* Corresponding author (email: jianghe@dlut.edu.cn)

BRS [8–10]. To resolve software tasks, researchers usually leverage the contents of bug reports to extract valuable knowledge. For example, a triager needs to understand a bug report well to find a potential developer for fixing this bug [2]. Meanwhile, when a developer fixes a bug, he/she often needs to trace historical bug reports to locate the root cause of this bug [7].

Since bug reports are initialized and updated by people (in this study, we refer to these people as contributors), the authorship characteristics (e.g., the richness of vocabulary, the layout, the length of text) of contributors may heavily impact the performance of resolving software tasks. Intuitively, a messy bug report attracts less attention (comments) than a well-structured one, and might be delayed to fix. As to our statistical results, from Jan. 1, 2003 to Dec. 31, 2012, each fixed bug in Mozilla has 10.8 comments on average while an unfixed bug has only 5.9 comments. In [11], Zimmermann et al. find that "developers are slowed down by poorly written bug reports". However, as to our knowledge, no systematic investigation has been conducted on the authorship characteristics of contributors.

In this study, we show our attempts towards mining the authorship characteristics in bug repositories. First, we model the authorship characteristics with an effective approach, namely byte-level N-grams, which is well-used in authorship attribution [12] and software forensics [13]. Meanwhile, we employ Normalized Simplified Profile Intersection (NSPI) to identify the similarity between the authorship characteristics. Based on these approaches, we address a series of Research Questions (RQs) with experiments conducted over two open source projects, namely Eclipse and Mozilla. We find that, in terms of NSPI, a fraction of contributors retain relative stable authorship characteristics. Their bug reports are more likely to be fixed than other contributors whose authorship characteristics vary sharply over time. In addition, the authorship characteristics of active contributors slightly change among distinct products in open source projects. Furthermore, we employ the authorship characteristics to facilitate a typical task in software maintenance, namely BRS. A new framework named Authorship Characteristics based Summarization (ACS) is proposed and evaluated over two corpora of annotated bug reports. Experiments show that incorporating the authorship characteristics can effectively improve the resulting summaries.

The main contributions of this paper are as follows:

• We model the authorship characteristics of contributors in bug repositories based on the method of byte-level N-grams and evaluate the similarity between the authorship characteristics with NSPI. As to our knowledge, this is the first work to model the authorship characteristics of contributors in bug repositories.

• We find that a fraction of contributors' authorship characteristics in open source bug repositories are relatively stable along time and their bugs are more likely to be fixed. In addition, the authorship characteristics of active contributors vary slightly among distinct products.

• We leverage the authorship characteristics to assist the software tasks in bug repositories. Taking BRS as an example, we design a new framework named ACS.

• We create a new corpus of 96 annotated bug reports from 4 open source projects, namely Eclipse, Mozilla, KDE, and Gnome. The new corpus is publicly available for BRS.

2 Background & motivation

2.1 Reports in bug repositories

Nowadays, bug repositories are widely used in software projects to store and manage bug reports. Along with the development of software projects, these bug repositories contain tremendous bug reports. For example, up to May 1, 2014, Bugzilla has managed over 434000 and 1004000 bug reports for Eclipse and Mozilla, respectively. When a bug is found, a bug report is initialized by a contributor to describe its details, e.g., its related *product*, *component*, *severity*, and *version*, and the description to reproduce this bug. In bug repositories, a bug could be a defect, a new feature, an update to documentation, or a refactoring (see [14] for more details). After a bug report is initialized, other contributors could update this bug report with comments. In this study, when someone initializes or updates a bug report, we state that he/she contributes to this bug.

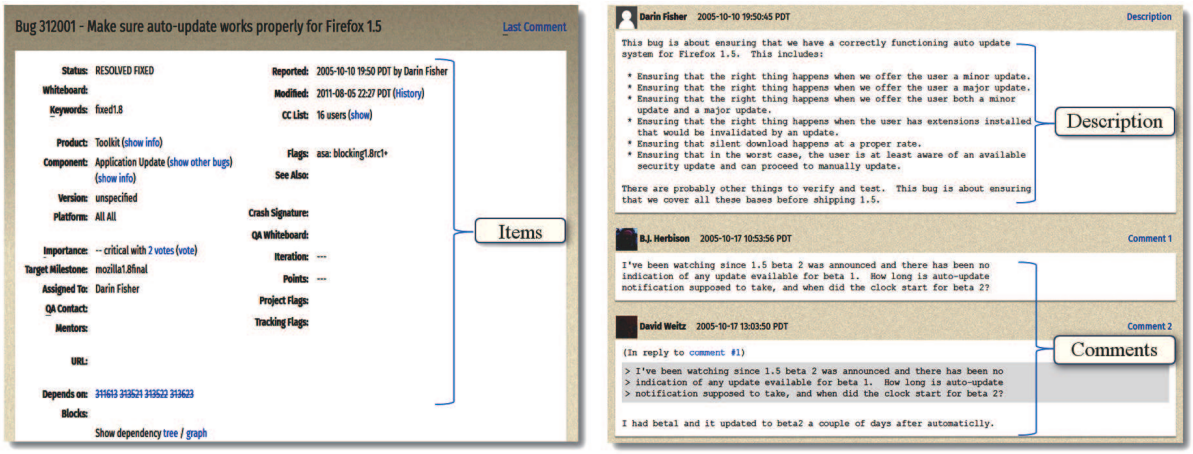

Figure 1 (Color online) Illustration of a bug report of Mozilla. (a) Items of bug 312001 of Mozilla; (b) description and comments of bug 312001 of Mozilla.

Figure 1 exhibits a bug report of Mozilla, namely Bug 312001. As shown in Figure 1, a contributor named Darin Fisher initialized a bug report about ensuring a correctly functioning auto update system for Firefox 1.5 on Oct.10, 2005. Darin Fisher presented the detailed description of this bug that "This bug is about ensuring $\cdots$. This bug is about ensuring that we cover all these bases before shipping 1.5". In addition, Darin Fisher also specified some related items of this bug, including *product*, *component*, and *target milestone*. Along the progress of fixing this bug, the item status may change accordingly. After Darin Fisher submitted Bug 312001, totally 21 comments have been conducted over Bug 312001 by 9 contributors. In Figure 1, we only display a part of the comments. Interestingly, Darin Fisher also contributed 6 comments.

2.2 Motivation

Since developers in actual software development process need to understand the contents of bug reports to resolve software tasks, the authorship characteristics of contributors may greatly impact the performance of resolving software tasks. A poorly written bug report is hard to comprehend and process [11]. As shown in Table 2 of [11], 34% and 26% of developers complain that bug reports suffer from unstructured text and too long text by contributors, respectively. Developers are more likely to pay attention to well-written bug reports than poorly-written ones. In some popular question and answer websites, some people often complain that poorly-written bug reports are hard to fix and verify. For example, in StackExchange, Dzieciou complains that "Fixes hard to verify because of poor quality bug reports" (see http://sqa.stackexchange.com/questions/6841/fixes-hard-to-verify-because-of-poor-quality-bug-reports?rq=1). However, as to our knowledge, no in-depth investigation has been performed on the authorship characteristics of contributors.

In contrast, some researchers in software maintenance investigate code ownership, a conception similar to the authorship characteristics, and achieve great success. Code ownership is the process of finding the proper developers who take responsibility of some software components. Rahman and Devanbu [15] exploit a modern version control system to examine the impact of code ownership and developer experience on software quality for open source projects. They find that implicated code is more strongly related to a single developer's contribution. Bird et al. [16] examine the relationship between different ownership measures and software failures in commercial software projects, namely Windows Vista and Windows 7. Bird et al. find that measures of ownership are related to both pre-release faults and post-release failures. Inspired by the research of code ownership, we argue that the authorship characteristics of contributors may influence the process of resolving software task in bug repositories.

Based on the above considerations, we examine the authorship characteristics of contributors in bug repositories. In this study, we employ byte-level N-grams to model the authorship characteristics. With

this approach, we investigate a series of Research Questions (RQs) related to the authorship characteristics.

RQ1: *How contributors' authorship characteristics evolve over time?* In this study, we conduct experiments on two open source projects to investigate whether contributors' authorship characteristics are stable over time or not.

RQ2: *Do contributors' authorship characteristics change among distinct products?* For this question, we examine the change of active contributors' authorship characteristics among several products in open source projects.

RQ3: *How can we employ the authorship characteristics to enhance existing software tasks in bug repositories?* In RQ3, we investigate how to incorporate the authorship characteristics to improve the performance of BRS, a typical task in bug repositories.

3 Authorship characteristics

In this section, we employ byte-level N-grams to model the authorship characteristics in Subsection 3.1. Then, we investigate the answers to RQ1 and RQ2, by conducting some empirical analysis on two open source projects, namely Eclipse and Mozilla. We collect all the bug reports of Eclipse and Mozilla in 10 years, from Jan. 1, 2003 to Dec. 31, 2012. There are 368263 and 588694 bug reports in Eclipse and Mozilla respectively.

3.1 Identifying authorship characteristics

We consider extending the seminal work by Frantzeskou et al. [13] to model the authorship characteristics. In their work for software forensics, Frantzeskou et al. employ byte-level N-grams to define authorship profiles and measure the similarity between two authorship profiles with Simplified Profile Intersection (SPI). In a recent survey, Burrows et al. [17] compare existing techniques for authorship attribution of source code and confirm that the approach of byte-level N-grams [13] is the best among existing methods. In this study, we transfer the seminal work of Frantzeskou et al. [13] to model the authorship characteristics, and extend SPI to NSPI for measuring the similarity between the authorship characteristics of contributors.

Given a contributor in bug repositories, all his/her descriptions and comments are concatenated into a single file. Then, a simplified profile can be defined as the set $\{X_1, X_2, \ldots, X_L\}$ of the L most frequent byte-level N-grams, to identify the authorship characteristics for the contributor, where a byte-level N-gram is a consecutive sequence of N bytes. For example, given a string "debug tool", there are totally 6 byte-level 5-grams, namely "debug", "ebug ", "bug t", "ug to", "g too", and " tool".

Given two contributors with their profiles V_A and V_B, we use $\mathrm{NSPI}(V_A, V_B)$ to denote the similarity between the authorship characteristics of these two contributors.

$$\mathrm{NSPI}(V_A, V_B) = |V_A \cap V_B|/L, \tag{1}$$

where $|\theta|$ represents the size of θ.

In the following part of this section, we set $L = 1000$ and $N = 5$. The values of these parameters are learned from the best results of ACS framework. We use the same values to build the model of the authorship characteristic to analyze the results of each RQ.

3.2 Authorship characteristics over time

To investigate the evolvement of the authorship characteristics of contributors, we first calculate every contributor's authorship characteristics in every year within the interval from Jan. 1, 2003 to Dec. 31, 2012. Since the authorship characteristics (namely simplified profiles) cannot be directly visualized, we use the similarity between the authorship characteristics in years instead to examine the behavior of each contributor. Given a contributor A, his/her authorship characteristics in year 200X can be achieved by

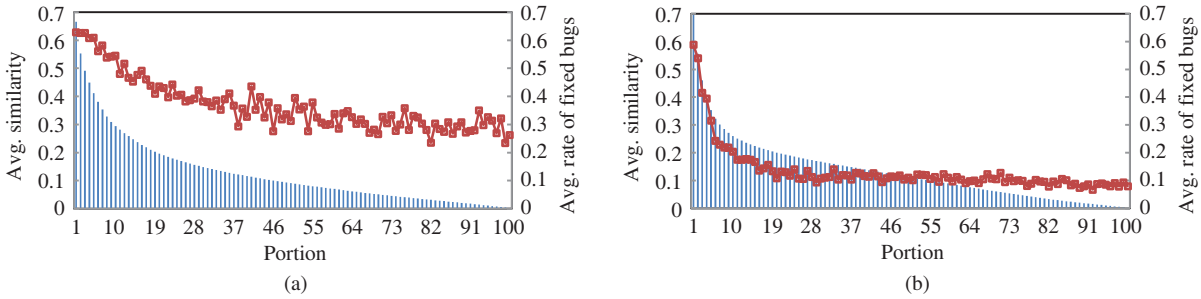

Figure 2 (Color online) Distributions of the authorship characteristics in Eclipse and Mozilla. (a) Distributions of the authorship characteristics of contributors in Eclipse; (b) distributions of the authorship characteristics of contributors in Mozilla.

calculating the simplified profile $V_{A/200X}$ over the concatenation of all the descriptions and comments by A in this year. Then, we measure the similarity between A's authorship characteristics in two consecutive years with the approach of NSPI. For example, we can represent the similarity (denoted by $\text{NSPI}_{A/2004}$) between A's authorship characteristics in 2003 and 2004 with the following formula.

$$\text{NSPI}_{A/2004} = \text{NSPI}(V_{A/2004}, V_{A/2003}). \tag{2}$$

By this way, we can obtain a series of similarities over 10 years for contributor A, namely $\text{NSPI}_{A/2004}$, $\text{NSPI}_{A/2005}, \ldots, \text{NSPI}_{A/2012}$. For contributor A, we use NSPI_A to represent the average value of his/her similarities over 10 years.

$$\text{NSPI}_A = \frac{1}{9} \sum_{X=2004}^{2012} \text{NSPI}_{A/X}. \tag{3}$$

If a contributor does not have consecutive 10-year's reported bug reports, we calculate the similarity since the start year (the year he/she begins to submit bug reports). If there exists one year which the contributor does not submit bug reports, we keep the similarity the same as before. In such a way, we can calculate the average value of every contributor's similarities. For both Eclipse and Mozilla, we sort all the contributors in each project by their average value of similarities in descending order. We partition all the contributors in each project into 100 equally sized portions. For the 100 equally sized portions mentioned above, we compute the average rate of fixed bugs associated with the contributors in each portion [18].

Figure 2 presents the average similarity and the average rate of fixed bugs against the portion of contributors. The left vertical axis and the bars show the average similarity, while the right vertical axis and the curves indicate the average rate of fixed bugs. As shown in Figure 2, both projects exhibit a long tail phenomenon. For the average similarity of authorship characteristic, it suggests that only a fraction of contributors show relatively stable authorship characteristics, while most contributors' authorship characteristics evolve significantly over time. For the average rate of fixed bugs, in either Eclipse or Mozilla, it follows a similar tendency as the contributors' similarities of authorship characteristics.

We perform correlation analysis between the average similarity and the average rate of fixed bugs. We calculate the Pearson Correlation Coefficient between them. The result lies in -1 to 1. The bigger result implies more likely correlated between the two variables. The Pearson Correlation Coefficient of Eclipse is 0.95, while Mozilla is 0.88. Correlation is significant at the 0.01 level (2-tailed). We can find that the average similarity shows a strong correlation with the average rate of fixed bugs.

This finding suggests that the stability of authorship characteristics could facilitate bug fixing. The reasons for this finding can be two-fold. First, when a contributor's authorship characteristics keep stable, other contributors focusing on the same products can easily capture the ideas in his/her initialized bug reports. Hence, they can update related bug reports to provide supplemental information. Second, when developers in this project are familiar with the authorship characteristics of a contributor, they spend less time on understanding his/her initialized bug reports.

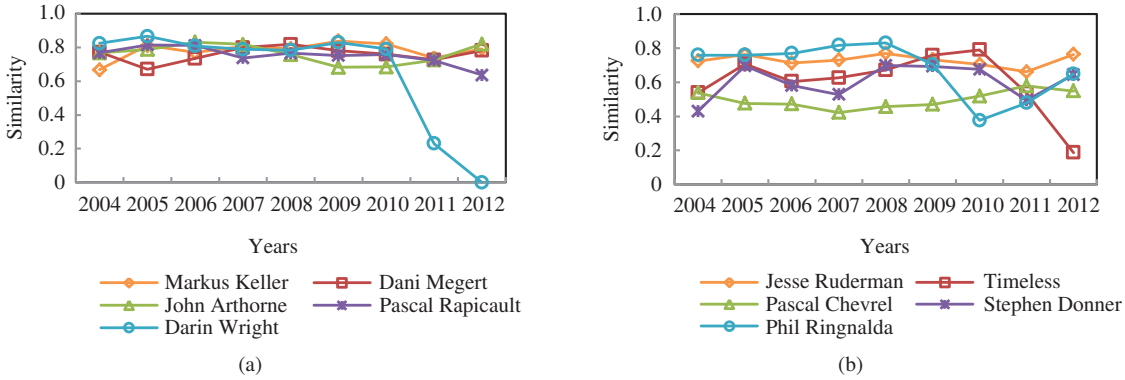

Figure 3 (Color online) Evolvements of authorship characteristics of 5 active contributors in Eclipse and Mozilla. (a) Evolvement in Eclipse; (b) evolvement in Mozilla.

Furthermore, we examine the evolvement of the authorship characteristics of some active contributors. We rank all the contributors in each project by the number of their contributed descriptions and comments in descending order. Out of all the contributors, we select the top 5 contributors and examine their evolvement from Jan. 1, 2003 to Dec. 31, 2012. In Figure 3, we plot each active contributor's similarities over years. In this figure, the similarity of an active contributor A on 2004 represents the similarity of A's authorship characteristics between 2003 and 2004, namely $\text{NSPI}_{A/2004}$. As to Figure 3, most curves of 5 active contributors in either Eclipse or Mozilla are relatively stable. For example, the similarities of John Arthorne from 2004 to 2012 slightly vary in the interval of 0.7 and 0.8. In contrast, there also exist some exceptions in both Eclipse and Mozilla. The similarities of Darin Wright are always close to 0.8 before 2010, however the curve sharply declines to 0.2 in 2011 and down to 0 in 2012. The reason for the sharp decline of Darin Wright's curve lies in that Darin Wright only contributes 2 bug reports in 2011 and no bug report in 2012. For Phil Ringnalda and timeless in Mozilla, the sharp declines of their curves also suffer from similar reasons. To give more insights into the results, we continue to conduct quantitative analysis. We calculate the average variance of the top 5 contributors. The average variance of top 5 contributors is 0.019 for Eclipse and 0.013 for Mozilla. Considering that the average similarity of all the contributors shows a long tail phenomenon, the average variance is relatively small. We can find that top ranked contributors are more likely to keep stable than other contributors in terms of authorship characteristics.

Answer to RQ1: The authorship characteristics of a fraction of contributors (e.g., active contributors) in open source projects are relatively stable. Meanwhile, most contributors' authorship characteristics evolve greatly over time. Interestingly, the stability of the authorship characteristics facilitates bug fixing.

3.3 Authorship characteristics among products

A software project usually consists of many products. In this subsection, we examine the authorship characteristics of active contributors in distinct products. In the experiments, we choose the same active contributors in Subsection 3.2 and investigate their behaviors in 5 typical products to which they contribute a great number of bug reports.

Tables 1 and 2 summarize the number of descriptions and comments submitted by these active contributors for products. In these tables, we use terms "des." and "com." to represent descriptions and comments. As to Tables 1 and 2, every active contributor provides far more comments than descriptions. In addition, an active contributor does not devote efforts evenly to all the products. For example, Markus Keller contributes 2123 descriptions and 4656 comments for the product of *platform*. However, only 77 descriptions and 191 comments are contributed by Markus Keller for the product of *community*.

Given an active contributor A, we first calculate his/her simplified profile V_A over the concatenation of all the descriptions and comments contributed by A within the whole project. To examine A's behaviors among products, we measure the similarity between A's authorship characteristics in a product X and

Table 1 The number of descriptions and comments by 5 active contributors in Eclipse

Contributor	*platform*		*jdt*		*pde*		*equinox*		*community*	
	des.	com.	des.	com.	des.	com.	des.	com.	des.	com.
Markus Keller	2123	4656	2162	8293	325	460	78	160	77	191
Dani Megert	2231	19416	1578	19326	442	1489	96	433	78	446
John Arthorne	1211	12217	299	735	189	492	555	4745	110	750
Pascal Rapicault	571	2234	114	131	546	2425	1101	6381	101	157
Darin Wright	1016	12319	745	11942	498	3447	23	122	7	31

Table 2 The number of descriptions and comments by 5 active contributors in Mozilla

Contributor	*core*		*firefox*		*seamonkey*		*mozilla.org*		*toolkit*	
	des.	com.	des.	com.	des.	com.	des.	com.	des.	com.
Jesse Ruderman	4586	12319	292	5276	13	215	79	192	185	1448
timeless	2088	9089	82	2191	159	916	107	337	97	631
Pascal Chevrel	51	181	49	167	30	82	369	573	7	24
Stephen Donner	142	1683	49	1359	50	1115	200	1419	51	1375
Phil Ringnalda	798	12025	265	7770	6	134	231	892	188	3764

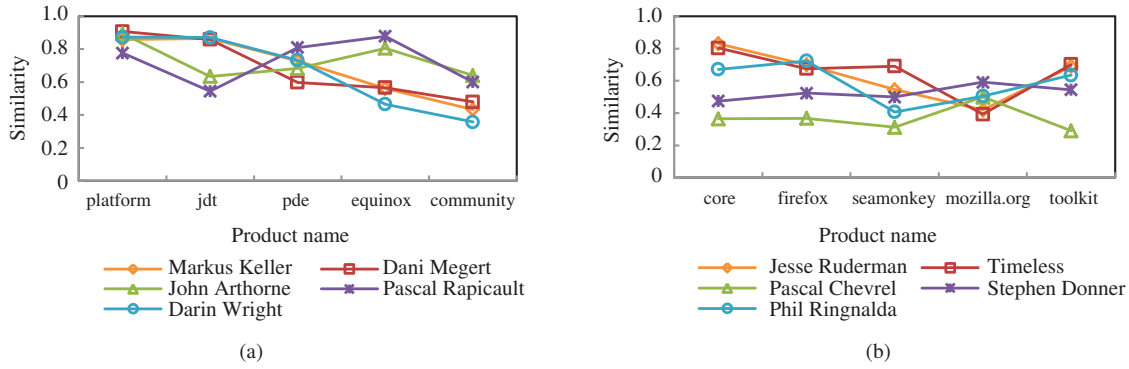

Figure 4 (Color online) Authorship characteristics of 5 active contributors in products of Eclipse and Mozilla. (a) Results in products of Eclipse; (b) results in products of Mozilla.

that in the whole project with formula (4).

$$\mathrm{NSPI}_{A/X} = \mathrm{NSPI}(V_{A/X}, V_A). \tag{4}$$

In this formula, $V_{A/X}$ is A's simplified profile over the concatenation of all the descriptions and comments contributed by A within product X.

For both Eclipse and Mozilla, we present in Figure 4 the changes of 5 active contributors' authorship characteristics over 5 products. We can observe that all the active contributors show relatively high similarities of authorship characteristics among distinct products, when compared with the distributions of authorship characteristics shown in Figure 2. The average variance among distinct products of the 5 active contributors is 0.026 for Eclipse and 0.012 for Mozilla. The similarities of each contributor's authorship characteristics slightly change among distinct products. Ideally, we expect that a contributor's authorship characteristics stemming from a large number of descriptions and comments should be perfectly coincident with that from a segment of descriptions and comments. However, we can find that the number of descriptions and comments may impact on the similarity of authorship characteristics. The reason might lie in that we adopt byte-level N-grams approach, in which a larger number of descriptions and comments may better capture one's authorship characteristics, while a small amount of descriptions and comments tend to introduce more bias in terms of NSPI.

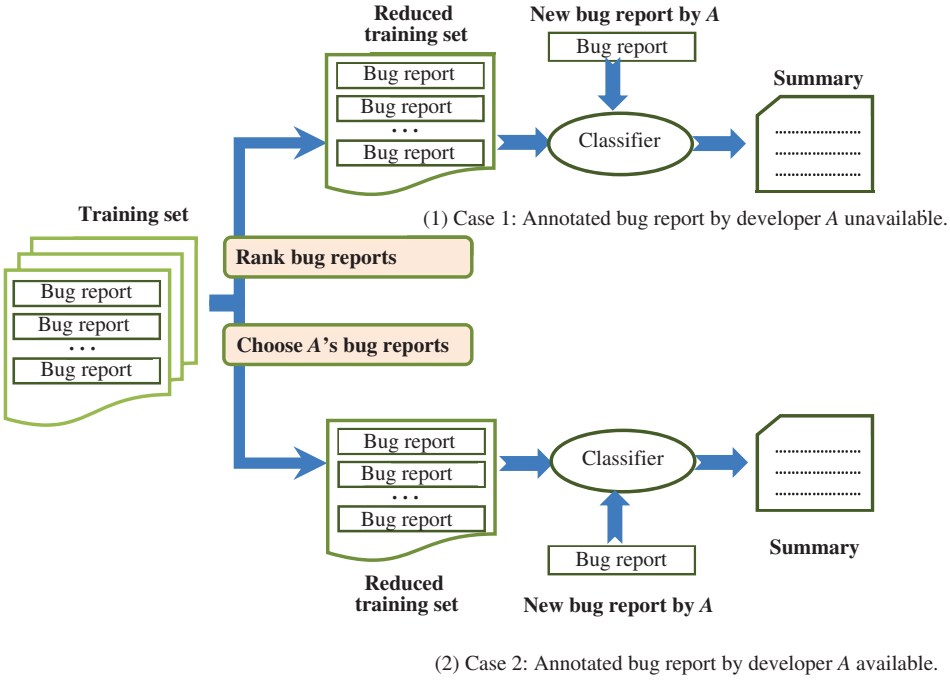

Figure 5 (Color online) Roadmap of Authorship Characteristics based Summarization (ACS).

Answer to RQ2: The authorship characteristics of contributors slightly change over distinct products. The volume of a contributor's descriptions and comments might impact his/her stability of authorship characteristics.

4 ACS framework & experiments

In this section, we investigate how to employ the authorship characteristics to assist a typical software task in bug repositories, namely BRS. First, we illustrate our new framework for BRS, namely Authorship Characteristics based Summarization (ACS) in Subsection 4.1. Then, we present the experiments and results of ACS over two corpora in Subsection 4.2 and 4.3, respectively.

The goal of BRS is to extract a short summary from the description and the comments of a given bug report. As one typical software task in bug repositories, BRS can be resolved in either an unsupervised way [8, 10] or a supervised way [9]. The key difference between a supervised way and an unsupervised way lies in whether a training set of annotated bug reports is given or not. In addition to bug reports, other documents can also be summarized [19]. In this study, we focus on the supervised BRS, in which a training set of annotated bug reports is provided.

4.1 ACS framework

In this section, we incorporate the authorship characteristics into the task of BRS and present the framework of ACS. The intuition of ACS is that, given a new bug report initialized by contributor A, a classifier trained over annotated bug reports by A is highly likely to perform better than a classifier trained over annotated bug reports by other contributors. Figure 5 illustrates the roadmap of ACS.

More detailed, ACS works as follows. Given a new bug initialized by contributor A and a training set consisting of annotated bug reports, the process of ACS can be divided into two cases regarding whether there exist annotated bug reports by the same contributor A. Noticed that in bug repositories, e.g., Bugzilla, the contributor initializing a bug report is a predefined item of this bug report, we can check whether an annotated bug report is initialized by A in $\Theta(1)$ running time.

Case 1: Annotated bug reports by contributor A unavailable. In this case, no annotated bug reports

by A can be directly used. For every contributor initializing an annotated bug report, ACS calculates the similarity between his/her authorship characteristics and A's authorship characteristics. Then, ACS ranks all the annotated bug reports by their similarities in descending order. In this way, the writing style of a top annotated bug report is more close to that of the new bug report by developer A, than that of a bottom one. Finally, ACS chooses the top M annotated bug reports to train the classifier and feeds the new bug report by developer A to achieve the resulting summary.

Case 2: Annotated bug reports by contributor A available. For this case, ACS simply chooses all the annotated bug reports by contributor A as the new training set to train a classifier. Then, ACS feeds the new bug report by A to the classifier to achieve the resulting summary.

In the remaining part of this subsection, we briefly discuss some features of ACS. First, any existing supervised technique for BRS can be embedded to ACS after the training set is reduced. Second, in contrast to existing supervised techniques (e.g., [9]), ACS performs a preprocessing step to reduce the training set. One might argue that the preprocessing might consume some additional running time. For a traditional supervised technique (e.g., [9]), multiple new bug reports could be predicted after a classifier is trained over a large set of annotated bug reports. In contrast, ACS needs to achieve a specific training set for every contributor who provides new bug reports. However, it is still rational to use ACS though suffering from the overhead time. The key goal of BRS is to save a developer's time for tracing historical bug reports when he/she fixes a bug. A developer can save his/her time by manually checking the summaries instead of the raw bug reports. Since ACS can be automatically run on computers, the overhead time of the preprocessing step in ACS is acceptable.

4.2 Evaluation metrics

We evaluate our new framework ACS in terms of 4 metrics, namely *Precision*, *Recall*, *F-score*, and *Pyramid*, where the first 3 metrics are widely used in information retrieval and the last one is developed in text summarization [20, 21].

Precision measures the fraction of selected sentences that belong to the Gold Standard Summary (GSS) which is annotated by people. The high value of *Precision* means more GSS sentences are selected in top ranked sentences, and a perfect 1.0 value reflects that all the top ranked sentences are GSS sentences. *Precision* can be considered as a measure of exactness. *Precision* can be computed as follows.

$$Precision\ (\%) = \frac{\#\text{sentences selected from GSS}}{\#\text{sentences in summary}} \times 100\%. \tag{5}$$

Recall measures the fraction of the sentences in GSS that are selected in the resulting summary. A perfect 1.0 value reflects that all the GSS sentences are selected. *Recall* can be considered as a measure of completeness which can be calculated as follows.

$$Recall\ (\%) = \frac{\#\text{sentences selected from GSS}}{\#\text{sentences in GSS}} \times 100\%. \tag{6}$$

In information retrieval, there usually exists a trade-off between *Precision* and *Recall*. It's likely to increase one at the expense of the other one. *F-score* is the harmonic mean of *Precision* and *Recall* as follows.

$$F\text{-}score\ (\%) = 2 \times \frac{Precison \times Recall}{Precision + Recall} \times 100\%. \tag{7}$$

To date, the corpora of BRS are annotated by people. For each sentence in a bug report, 3 annotators vote whether it belongs to GSS or not. A bug report's GSS consists of all the sentences gaining at least 2 votes. *Pyramid* is used to assess the quality of the resulting summary when there are multiple annotations available [20, 21]. Given a summary, *Pyramid* can be calculated as follows.

$$Pyramid\ (\%) = \frac{\#\text{votes of sentences in summary}}{\text{Max. num of votes for summary length}} \times 100\%. \tag{8}$$

The computing of *Pyramid* can be demonstrated with a simple example. Given a bug report and its GSS which consists of 6 sentences with 2 votes and 8 sentences with 3 votes, the maximum number of

votes for a summary with 10 sentences is $(8\times3)+(2\times2)=28$. Then, given a summary with 10 sentences gaining 21 votes, its *Pyramid* equals to $21/28\times100\%=75\%$.

4.3 Experiments on BRC corpus

In this section, we evaluate our new framework ACS over existing corpus which meets the scenario of Case 1 and compare it against the state-of-the-art classifier.

4.3.1 *BRC corpus*

In this subsection, we briefly introduce some basic information of BRC corpus, a publicly available corpus for BRS. BRC corpus is firstly provided in the seminal work of Rastkar et al. [9]. BRC corpus consists of 36 annotated bug reports from 4 open source projects, namely Eclipse, Mozilla, KDE, and Gnome. The 36 bug reports (9 bug reports from every project) are annotated by graduate students from the Department of Computer Science at the University of British Columbia. Totally, there are 2361 sentences in these 36 bug reports. Each sentence is voted by 3 annotators to determine whether it belongs to GSS. When a sentence gains at least 2 votes, it is included in GSS.

4.3.2 *Experimental results*

In this subsection, we run ACS over BRC data set and compare its performance against the state-of-the-art supervised classifier BRC [9]. BRC is a conversation-based classifier trained over a series of annotated bug reports. Similar as [9], we use a leave-one-out strategy, namely, each bug report is selected out as the new bug report for summarization and the remaining 35 annotated bug reports are used as its training set. We employ BRC as the underlying classifier in ACS. Given a new bug report R, ACS ranks all the remaining 35 annotated bug reports and chooses some top ranked annotated bug reports as R's reduced training set. In this study, we choose the top 15 bug reports as the reduced training set. Then, these 15 annotated bug reports are fed into BRC as the training set to train a classifier for summarizing R.

Since ACS needs to rank all the annotated bug reports to reduce the training set, we first specify the parameters used in ACS for calculating the similarities between authorship characteristics. There are two parameters, namely the scale L of simplified profile and the value of N. In Figure 6, we evaluate the performance of ACS with a series of combinations of two parameters, i.e., $L=600$, 1000, 1400, $N=3$, 4, 5, 6, 7, 8. In each subfigure of Figure 6, we set the horizontal axis as the value of N. The vertical axes in Figure 6 are *Precision*, *Recall*, *F-score*, and *Pyramid*, respectively. For comparison, in each subfigure of Figure 6, we also present the result of BRC as a baseline. In Figure 6, we use ACS(600), ACS(1000), ACS(1400) to denote the curves of ACS with $L=600$, 1000, 1400.

As shown in Figure 6(a), the value of *Precision* of ACS is impacted to a certain extent by the combinations of parameters. For three combinations (namely, $L=1000$ and $N=5$, $L=1400$ and $N=5$, $L=600$ and $N=8$), ACS performs better than BRC in terms of *Precision*. For other combinations of parameters, ACS is not so promising. When checking Figure 6 (b)–(d), we can find that ACS can undoubtedly outperform BRC in terms of *Recall*, *F-score*, and *Pyramid*, for nearly all the combinations of parameters. Based on the above observation, in this study, we set $L=1000$ and $N=5$ for ACS.

In Table 3, we present the numerical results of ACS (with $L=1000$ and $N=5$) and BRC. Since ACS uses less annotated bug reports (namely 15 bug reports) than BRC as the training set for a new bug report, one may argue that the success of ACS derives from the small training set size. Hence, we also implement a random algorithm (denoted as Random) to check the effect of the training set size. Similar to ACS and BRC, Random conducts a leave-one-out strategy. For every bug report R, Random arbitrarily chooses 15 bug reports from the remaining 35 annotated bug reports as the training set. Then, Random feeds these 15 chosen bug reports into BRC to train a classifier for producing the summary for R.

The best results among algorithms are presented in bold font. As shown in Table 3, the difference of results achieved by BRC and Random is trivial. For *Precision*, *Recall*, and *F-score*, BRC slightly outperforms Random. For *Pyramid*, Random is a bit better than BRC. Out of 3 algorithms, ACS always

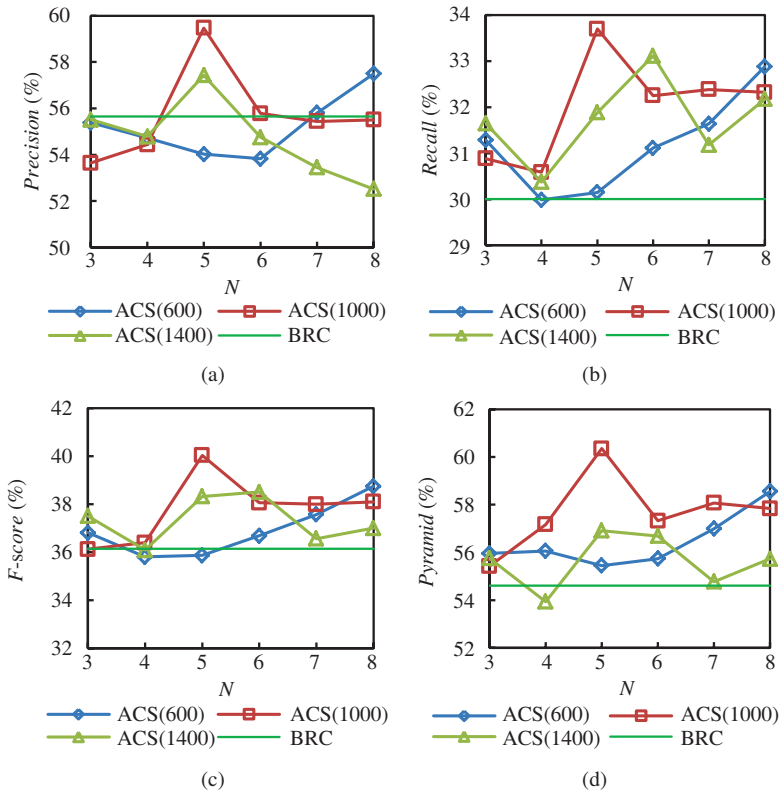

Figure 6 (Color online) Performance of algorithms with parameters. (a) *Precision*; (b) *Recall*; (c) *F-score*; (d) *Pyramid*.

Table 3 Performance comparison of ACS, BRC, and Random on BRC corpus

Algorithm	BRC	Random	ACS
Precision (%)	55.66	54.04	**59.47**
Recall (%)	30.00	29.66	**33.70**
F-score (%)	36.15	35.17	**40.04**
Pyramid (%)	54.61	55.75	**60.36**

performs best. ACS improves BRC in terms of 4 metrics by 3.7% to 5.75%. The comparison implies that irrelevant annotated bug reports might hamper the task of BRS.

4.4 Experiments on new corpus

To further evaluate our new framework, namely Case 2 of ACS, we conduct experiments on a series of new annotated bug reports.

4.4.1 *New corpus preparation*

In this subsection, we illustrate how to prepare a new corpus for further evaluation of ACS. It will be ideal if we would have a corpus of annotated bug reports in which each contributor initializes multiple bug reports. Unfortunately, such a corpus is not available to date. Following [9], we employ 7 graduate students from School of Software, Dalian University of Technology to annotate 96 bug reports from 4 open source projects, namely Eclipse, Mozilla, KDE, and Gnome. For each project, out of 24 bug reports, every 6 bug reports are initialized by the same contributor. In such a way, there are 4 contributors initializing bug reports in a project. In addition to these contributors initializing bug reports, some other contributors who only comment on these bug reports are also involved in the corpus.

In Table 4, we list some statistical information of these bug reports. The second row of Table 4 presents the number of bug reports chosen in each project, namely 24 bug reports for each project. Since

Table 4 Statistical information of new corpus

Project	Eclipse	Mozilla	KDE	Gnome
#bug reports	24	24	24	24
#comments per bug report	10.83	15.42	6.67	5.08
#sentences per bug report	43.67	61.71	30.58	16.88
#words per bug report	329.67	511	216.83	106.04
#contributor per bug report	3.75	5.21	3.71	2.92
#sentences per Manual Summary	5.29	4.92	2.63	3.36
#words per Manual Summary	106.07	75.50	39.61	41.01
#linked sentences per bug report	15.01	13.68	6.78	6.83
#sentences of GSS per bug report	12.88	11.08	6.21	6.63

a bug report is composed of one description and some comments. We also give the average number of comments of each bug report in Table 4. For these chosen bug reports in Eclipse, a bug report contains 10.83 comments on average. The fourth row presents the average number of sentences in each bug report. For example, in Mozilla, a bug report consists of 61.71 sentences. In addition, we also present the average number of words in a bug report. In the sixth row of Table 4, the average number of contributors involved in each bug is presented. For Eclipse and Mozilla, 3.75 and 5.21 contributors are involved in each bug report on average, respectively.

Similar as [9], each bug report in this new corpus is annotated by 3 annotators. We employ a BC3 web-based annotation software (http://www.cs.ubc.ca/nest/lci/bc3/framework.html) to assist the annotation process. Given a bug report, an annotator is requested to manually summarize it in his/her own language with up to 250 words and achieves his/her Manual Summary (MS). Then, the annotator links each sentence in MS to one or more sentences from the original bug report. For each bug report, the set of sentences gaining at least 2 votes forms its GSS. In Table 4, we present some statistical information of MS and GSS. The seventh and eighth rows of Table 4 list the average number of sentences and the average number of words in each MS. In the ninth row of Table 4, the average number of linked sentences from each bug report is given. Finally, we present the average number of sentences in GSS for each bug report. Our new corpus of annotated bug reports is publicly available on http://oscar-lab.org/paper/ac/corpus.htm.

4.4.2 *Experimental results*

In this subsection, we present the experimental results of ACS on the new corpus of annotated bug reports and compare the performance of ACS with existing methods.

The new corpus of annotated bug reports can be divided into 4 data sets (each for one project). Each data set consists of 24 annotated bug reports involving 4 contributors who initialize bug reports. As stated in Subsection 4.4.1, there may exist some other contributors submitting only comments for the bug reports. To evaluate the performance of ACS, we conduct a leave-one-out procedure over each data set. More specifically, for each data set, one bug report is summarized with ACS which is trained on the remainder of the annotated bug reports. In such a way, every bug report in a data set is summarized and the results of all the bug reports are averaged and reported.

In this experiment, we use BRC as the embedded classifier in ACS. Given a bug report R initialized by contributor A in the new corpus, contributor A also initializes 5 other bug reports in the remainder of annotated bug reports. For clarity, we let T denote these 5 bug reports. Therefore, for the new corpus, ACS follows its Case 2 to summarize bug reports. More specifically, ACS is trained on T, the 5 bug reports initialized by A, to summarize bug report R.

To validate the effectiveness of ACS, we also run two comparative classifiers, namely BRC and Random.

(1) BRC for comparison. Given a data set, each bug report is summarized by BRC trained on the remaining 23 annotated bug reports. We employ BRC to show the effectiveness of the authorship characteristics in reducing training sets.

Table 5 Performance comparison of ACS, BRC, and Random on new corpus

Project	Eclipse			Mozilla			KDE			Gnome		
Algorithm	BRC	Random	ACS	BRC	Random	ACS	BRC	Random	ACS	BRC	Random	ACS
Precison (%)	63.08	61.28	**67.03**	40.95	37.81	**42.52**	55.50	50.65	**59.30**	67.50	64.18	**73.06**
Recall (%)	32.39	31.29	**37.22**	39.06	35.96	**40.76**	37.93	36.07	**42.82**	30.42	30.39	**35.47**
F-score (%)	41.42	41.14	**46.34**	38.78	35.56	**39.93**	43.55	40.75	**47.77**	41.13	40.75	**46.89**
Pyramid (%)	69.09	67.32	**69.18**	55.73	52.56	**56.13**	66.74	61.55	**68.65**	72.08	68.96	**74.62**

(2) Random for comparison. Since ACS uses 5 annotated bug reports as the training set to summarize a bug report, rather than 23 bug reports in a project. When evaluating the performance of ACS, we use Random to avoid the bias caused by the reduced training set size. Given a bug report, out of the remaining 23 annotated bug reports, Random arbitrarily chooses 5 bug reports to train the BRC classifier and produces the resulting summary.

Table 5 summarizes the experimental results of ACS, BRC, and Random over 4 data sets. For each data set, the best results among three algorithms are given in bold font. As to Table 5, BRC always outperforms Random in all the metrics, including *Precision*, *Recall*, *F-score*, and *Pyramid*. It implies that an arbitrarily chosen subset of annotated reports is not a good choice for training set in BRS. When comparing ACS with BRC, we can find that ACS achieves better performance than BRC. In terms of *Precision*, ACS improves BRC by from 1.57% to 5.56%. For the other three metrics, we can also find similar phenomena. We can observe that the performance difference between ACS and BRC varies from data set to data set. For Eclipse, KDE, and Gnome, ACS improves BRC by over 4% in terms of *F-score*, however for Mozilla, the improvement achieved by ACS is 1.15%. On the other hand, far remarkable improvements can be observed when comparing ACS with Random. For example, the value of *Precision* of ACS for Gnome is 73.06%, while that of Random is only 64.18%. It implies that the strength of ACS derives from the chosen annotated bug reports based on the authorship characteristics, rather than the reduced training set size.

Answer to RQ3: The authorship characteristics can be beneficial for typical software tasks (e.g., BRS) in bug repositories. With the authorship characteristics, it is promising to find more accurate training sets specific for contributors to facilitate software maintenance.

5 Discussion

In this section, we discuss a series of issues related to this study, including the modeling, the authorship characteristics in other projects, and how to facilitate software tasks.

5.1 Modeling of authorship characteristics

It deserves further investigations on finding new effective approaches to model the authorship characteristics. In this study, we employ byte-level N-grams to identify the authorship characteristics in bug repositories. As shown in the literature, the approach of byte-level N-grams is an effective tool in author attribution [12] and software forensics [13]. With byte-level N-grams, we gain a series of findings. Meanwhile, we also encounter some difficulties in this study. First, the simplified profile calculated by the approach of byte-level N-grams cannot be easily visualized. Hence, we have to employ NSPI to measure the similarity between authorship characteristics instead. A better modeling approach may help us to investigate RQ1 and RQ2 more directly. Second, we expect that an ideal modeling approach is insensitive to the length of bug reports submitted by contributors. However, we find that the number of bug reports may impact on the similarity between authorship characteristics, when examining the answer to RQ2 with byte-level N-grams. Although the approach of byte-level N-grams shows its effectiveness in authorship attribution and software forensics, this approach may not be perfect for bug repositories.

Bug reports are usually far shorter than texts in authorship attribution meanwhile they deal with more domain specific topics (e.g., *products*, *components*) than source code in software forensics.

5.2 Authorship characteristics in other projects

We examine the authorship characteristics of open source projects, including Eclipse, Mozilla, KDE, and Gnome. All the above projects employ Bugzilla, a popular bug repository, to store and manage bugs. In Bugzilla, bug reports are submitted by people (contributors). For commercial projects, software companies may use their own bug tracking systems to manipulate bugs. In the commercial bug tracking systems, bug reports may be generated and submitted automatically. Hence, the conclusions achieved in this study may not directly apply to commercial projects.

However, we hope that commercial bug tracking systems may also benefit from this study. For example, by investigating the impact of the authorship characteristics on bug fixing in Bugzilla, developers of commercial bug tracking systems can improve the layouts of those automatically generated bug reports, to help with bug fixing. To validate the above idea, more case studies are needed in commercial bug tracking systems.

5.3 Enhancing software tasks

The behaviors of contributors submitting comments in bug repositories should be further examined in ACS. Given a new bug report initialized by contributor A, in Case 2 of ACS, we reduce the training set by choosing the annotated bug reports which are initialized by A. In Case 2 of ACS, we have risked ignoring the authorship characteristics of contributors submitting comments. For future work, more sophisticated strategies should be taken to consider the contributors submitting comments.

In this study, we have found the effectiveness of the authorship characteristics in improving the task of BRS. To facilitate other bug report oriented software tasks, e.g., bug triage, severity identification, it needs more efforts to seek for strategies to fully leverage the authorship characteristics.

6 Related work

In this section, we briefly review the relate work of this study. In Subsection 6.1, we present the research of MBR. In Subsection 6.2, we present two closely related areas of the authorship characteristics, namely authorship attribution and software forensics.

6.1 MBR

Since many projects employ bug repositories to store and manage bug reports, MBR has been a hot topic in software maintenance. Following the methods of mining software repositories [22], researchers in MBR usually leverage the contents of bug reports to build either a machine learning model or an information retrieval model for resolving software tasks. Some traditional tasks include bug triage [2,3], fault prediction [4,5], bug location [6,7], and BRS [8–10]. Our research in this study differs from traditional tasks in that we focus on the authorship characteristics of contributors, rather than the contents of bug reports.

In recent years, a lot of efforts have been spent on "people" factors in bug repositories [23]. Jeong et al. [24] leverage the tossing relationship between developers to assist bug triage. Xuan et al. [25] model the developer prioritization and assist three predictive tasks in bug repositories, including bug triage, severity identification, and reopened bug prediction. Our work can also be viewed as an investigation into "people" factors in bug repositories. However, the existing researches (e.g., [24, 25]) consider the collaborations among people, we focus on a contributor's personality, namely authorship characteristics.

Some researchers spend a lot of efforts on investigating the bug report quality. Zimmermann et al. [11] conduct a survey on what makes a good bug report and propose a prototype to rate a bug report mainly based on its content. Lotufo and Czarnecki [26] develop a new bug tracking system named BugBot to

facilitate reasoning about bug reports. No in-depth investigation has been conducted on the authorship characteristics of contributors.

6.2 Authorship attribution & software forensics

Given a text of unknown authorship, the goal of authorship attribution [27] is to assign this text to one candidate author. In this task, the text samples for a set of candidate authors are available in advance. Authorship attribution can be viewed as a multiclass, single-label text categorization problem. In [12], Keselj et al. propose a method based on byte-level N-grams to address authorship attribution. In contrast to existing literatures [28, 29], this method is simple and language-independent.

Software forensics [17] can be viewed as the task of authorship attribution on source code. Frantzeskou et al. [13] extend the work of Keselj et al. [12] to characterize the software authors. As to the comparison by Burrows et al. [17], the approach of byte-level N-grams [13] is best among existing methods (e.g., [30–32]). In this study, we employ a similar method as Frantzeskou et al. [13] to measure the authorship characteristics of contributors.

7 Conclusion

In this study, we investigate the authorship characteristics of contributors in bug repositories and leverage the authorship characteristics to resolve software tasks. By modeling the authorship characteristics with byte-level N-grams, we achieve a series of interesting findings. First, active contributors tend to show stable authorship characteristics. Meanwhile, the stability of the authorship characteristics could facilitate bug fixing. Second, the authorship characteristics of contributors slightly vary over products in a project. Finally, with the help of the authorship characteristics, the training sets of predictive tasks in software maintenance can be effectively reduced while achieving satisfactory performance.

For future work, we plan to investigate the following directions. First, we aim to find better modeling approaches for the authorship characteristics. In this study, we employ byte-level N-grams as the modeling approach, which is effective in authorship attribution and software forensics. However, bug reports are usually short and domain specific, it would be ideal if a specific modeling approach can be proposed. Second, we hope to further improve ACS. In the framework ACS, we use only the description contributor to determine the training set. Since a bug report usually involves multiple contributors, we will seek to leverage the authorship characteristics of all the contributors in a bug report. Third, motivated by the success of ACS, we consider to incorporate the authorship characteristics to facilitate software tasks other than BRS.

Acknowledgements This work was supported by National Basic Research Program of China (973) (Grant No. 2013CB035906), National Natural Science Foundation of China (Grant Nos. 61175062, 61370144), and New Century Excellent Talents in University (Grant No. NCET-13-0073). We greatly thank Rastkar, Murphy, and Murray with University of British Columbia for sharing their BRS corpus.

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Pressman R S, Ince D. Software Engineering: A Practitioner's Approach. New York: McGraw-Hill, 2010
2 Anvik J, Hiew L, Murphy G C. Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering, Shanghai, 2006. 361–370
3 Anvik J, Murphy G C. Reducing the effort of bug report triage: recommenders for development-oriented decisions. ACM Trans Softw Eng Methodol, 2011, 20: 10
4 Bishnu P S, Bhattacherjee V. Software fault prediction using Quad Tree-based K-means clustering algorithm. IEEE Trans Knowl Data Eng, 2012, 24: 1146–1150
5 Shivaji S, Whitehead J, Akella R, et al. Reducing features to improve code change based bug prediction. IEEE Trans Softw Eng, 2012, 22: 1–17

6 Artzi S, Kiezun A, Dolby J, et al. Finding bugs in web applications using dynamic test generation and explicit state model checking. IEEE Softw, 2010, 36: 474–494
7 Zhou J, Zhang H Y, Lo D. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012. 14–24
8 Mani S, Catherine R, Sinha V S, et al. AUSUM: approach for unsupervised bug report summarization. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, New York, 2012. 11–21
9 Rastkar S, Murphy G C, Murray G. Automatic summarization of bug reports. IEEE Trans Softw Eng, 2014, 40: 366–380
10 Lotufo R, Malik Z, Czarnecki K. Modelling the 'hurrie' bug report reading process to summarize bug report. In: Proceedings of the International Conference on Software Maintenance, Trento, 2012. 430–439
11 Zimmermann T, Premraj R, Bettenburg N, et al. What makes a good bug report? IEEE Trans Softw Eng, 2010, 36: 618–643
12 Keselj V, Peng F, Cercone N, et al. N-gram based author profiles for authorship attribution. In: Proceedings of Pacific Association for Computational Linguistics, Harifax, 2003. 255–264
13 Frantzeskou G, Stamatatos E, Gritzalis S, et al. Effective identification of source code authors using byte-level information. In: Proceedings of the 28th International Conference on Software Engineering, Shanghai, 2006. 893–896
14 Herzig K, Just S, Zeller A. It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the 35th International Conference on Software Engineering, San Francisco, 2013. 392–401
15 Rahman F, Devanbu P. Ownership, experience and defects: a fine-grained study of authorship. In: Proceedings of the 33rd International Conference on Software Engineering, New York, 2011. 491–500
16 Bird C, Nagappan N, Murphy B, et al. Don't touch my code!: examining the effects of ownership on software quality. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, New York, 2011. 4–14
17 Burrows S, Uitdenbogerd A L, Turpin A. Comparing techniques for authorship attribution of source code. Softw Pract Exper, 2014, 44: 1–32
18 Zou W Q, Xia X, Zhang W Q, et al. An empirical study of bug fixing rate. In: Proceedings of the 39th Annual International Computers, Software & Applications Conference, Taichung, 2015. 254–263
19 Zhang R, Yu W Z, Sha C F, et al. Product-oriented review summarization and scoring. Front Comput Sci, 2015, 9: 210–223
20 Nenkova A, Passonneau R. Evaluating content selection in summarization: the pyramid method. In: Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, Boston, 2004. 145–152
21 Carenini G, Ng R T, Zhou X. Summarizing emails with conversational cohesion and subjectivity. In: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, New York, 2008. 353–361
22 Xie T, Thummalapenta S, Lo D, et al. Data mining for software engineering. Computer, 2009, 8: 55–62
23 Zhang W Q, Nie L M, Jiang H, et al. Developer social networks in software engineering: construction, analysis, and applications. Sci China Inf Sci, 2014, 57: 121101
24 Jeong G, Kim S, Zimmermann T. Improving bug triage with tossing graphs. In: Proceedings Joint Meeting of 12th European Software Engineering Conference & 17th ACMSIGSOFT Symposium on Foundations of Software Engineering, Amsterdam, 2009. 111–120
25 Xuan J F, Jiang H, Ren Z L, et al. Developer prioritization in bug repositories. In: Proceedings of 34th International Conference on Software Engineering, Zurich, 2012. 25–35
26 Lotufo R, Czarnecki K. Improving Bug Report Comprehension. Technical Report GSDLAB-TR 2012-09-01, University of Waterloo, 2012
27 Stamatatos E. A survey of modern authorship attribution methods. J Amer Soc Inf Sci Technol, 2009, 60: 538–556
28 Stamatatos E, Fakotakis N, Kokkinakis G. Computer-based authorship attribution without lexical measures. Comput Hum, 2001, 35: 193–214
29 Zheng R, Li J X, Chen H C, et al. A framework for authorship identification of online messages: writing style features and classification techniques. J Amer Soc Inf Sci Technol, 2006, 57: 378–393
30 Kothari J, Shevertalov M, Stehle E, et al. A probabilistic approach to source code authorship identification. In: Proceedings of the 4th International Conference on Information Technology, Las Vegas, 2007. 243–248
31 Lange R, Mancoridis S. Using code metric histograms and genetic algorithms to perform author identification for software forensics. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, London, 2007. 2082–2089
32 Shevertalov M, Kothari J, Stehle E, et al. On the use of discretised source code metrics for author identification. In: Proceedings of the 1st International Symposium on Search Based Software Engineering, Windsor, 2009. 69–78