Empirical Study of Fault Introduction Focusing on the Similarity among Local Variable Names

Hirohisa Aman
Center for Information Technology
Ehime University
Matsuyama, Ehime, Japan
aman@ehime-u.ac.jp

Sousuke Amasaki
Faculty of Computer Sc. & Systems Eng.
Okayama Prefectural University
Soja, Okayama, Japan

Tomoyuki Yokogawa
Faculty of Computer Sc. & Systems Eng.
Okayama Prefectural University
Soja, Okayama, Japan

Minoru Kawahara
Center for Information Technology
Ehime University
Matsuyama, Ehime, Japan

***Abstract*—Well-chosen variable names play significant roles in program comprehension and high-quality software development and maintenance. However, even though all variables have easy-to-understand names, attention needs to be paid to the similarity among those names as well because a highly similar pair of variables may decrease the readability of code and might cause a fault. In order to analyze the relationship of variable similarity with the risk of introducing faults, this paper collects variable data from ten Java open source development projects and conducts an empirical study on the following three research questions: (1) Does the distribution of similarity differ among software development projects?, (2) What is the appropriate threshold of similarity?, and (3) How can the threshold of similarity contribute to the fault-prone method prediction? Then, the empirical results show the following findings: (1) The distribution of similarity is nearly identical regardless of the project; (2) Programmers should avoid giving similar names to different variables to prevent a fault introduction, and the threshold of Levenshtein similarity is 0.35; (3) By classifying Java methods with the above threshold, the risk of overlooking fault-introducing events in the fault-prone prediction model is effectively reduced; The recall of the prediction model improves by 12.8% on average.**

I. Introduction

Code changes form essential parts of software development and maintenance, i.e., drive a software evolution. On the other hand, a code change also has a risk of introducing a new fault [1]. Because the code change is a human intellectual activity, the risk of a fault introduction would become higher when the program is more complex and harder to understand. Hence, the readability of code plays a significant role in the successful development and maintenance [2]. By making the code readable, the programmer can review the code more clearly and deeply, and may quickly detect a potential fault if it exists. Moreover, readable code is easy-to-understand for many other programmers as well, and they can review and maintain the source code smoothly. One of the most critical matters for producing readable code is the proper naming of the variables [3]. Indeed, there is a report saying that 24% of code review feedbacks were related to the variable naming [4]. Well-chosen names of variables can be useful clues to the understanding of what the program does [5] and can lead to the advantages of a low maintenance cost [6]. On the other hand, we can readily decrease the readability and understandability of programs by selecting meaningless names for the variables.

There have been various studies on a better naming of variables in the past. For example, Lawrie et al. [7] and Scanniello et al. [8] reported empirical results that fully spelled English words or their abbreviated forms are better for naming variables in terms of program comprehension and effective fault detection. Binkley et al. [9] showed that the camel case-style naming is useful for enhancing the understandability.

Although the naming of variables has been studied, we should consider not only the naming of an individual variable but also the relationship among the names of variables. If two variables have highly similar names to each other, they may be easily confused. For example, suppose a method (or a function) has some local variables including lineIndex and lineIndent. Although each of these two names looks easy-to-understand, a mix of them may be confusable [10]. When a programmer uses a sophisticated editor which can automatically complement or suggest the name of a variable, the programmer might accidentally select a similar but wrong variable [11]. When we use two or more variables, we should give more distinguishable names to the variables while keeping the meaningfulness of the names.

In this paper, we report our quantitative study on the risk of introducing faults by focusing on the similarity among the names of the local variables. The key contributions of this paper are as follows:

- We present a quantitative guideline of the local variable name to prevent the fault introduction: When a programmer declares two or more local variables in a Java method, the names of them should be dissimilar to each other. The threshold of Levenshtein similarity between variable names is 0.35.

- We empirically show that the classification of Java methods by the above threshold can be helpful in the fault introduction prediction; Especially, it reduces the risk of overlooking fault introductions effectively: The classification based on the similarity of local variable names improves the recall, the precision, and the F value of the random forest-based fault introduction prediction model by 12.8%, 2.45%, and 4.2% on average, respectively.

The remainder of this paper is organized as follows: In Section II, we describe the related work regarding variable names and present our research motivation and research questions (RQs). Then, in Section III, we report the empirical work that we conducted on the RQs. Finally, we give our conclusion and future work in Section IV.

II. Related Work and Motivation

In this section, we briefly explain the related work focusing on the names of local variables. Then, we describe our research motivation and put our research questions (RQs) to clarify our goal in this paper.

A. Naming Variables

There have been several empirical studies regarding better naming of variables in the past.

Lawrie et al. [7] conducted an empirical study focusing on the comprehensibility of variables from the perspective of the naming form. They prepared three variations of a variable name as (a) fully spelled English word, (b) an abbreviated form of the word, and (c) a single character then got 128 programmers to compare the ease of comprehension of them. The empirical results showed that the trend of comprehensibility is "(a) $\geq$ (b) $>$ (c)," but there is no statistically significant difference between (a) and (b). That is, a fully spelled English word or a well-chosen abbreviated work would be a better name for a variable than a single character name. Scanniello et al. [8] also performed an empirical study regarding the relationships of the variable naming with the program comprehension and the fault detection/fix. The empirical results involving 100 programmers showed a trend similar to the one reported by Lawrie et al. [7].

When a variable has a more complex role, programmers tend to use a longer and more descriptive name. The major ways of such naming are the concatenations of two or more words (or abbreviated words) by the camel case (for example, indexOfArray) or the snake case (for example, index_of_array) [9].

Although the above previous studies are remarkable empirical work to discuss the appropriate naming of variables, their main focus is on the naming of an *individual* variable. In other words, the *relationship* between names has not been well-discussed. When we see two or more variables and the names of them are similar to each other, we might get confused by the similar names even if each variable has an easy-to-understand name.

B. Similar Names of Variables

As mentioned above, when there are two or more local variables and the names are highly similar to each other, the code comprehensibility and the readability may be low even if each of the names is easy-to-understand. Binkley et al. [11] concerned the risk of selecting the wrong variable when the name is long. Indeed, many programmers may use an advanced code editor like Atom[1] or an integrated development environment like Eclipse[2], which can automatically complete the name of the variable or line up available variables [12]. Then, a programmer might overlook a wrong completion or choose the wrong candidate (see Fig. 1).

Tashima et al. [10] analyzed the issue of similar variable names from the perspective of the fault-proneness. In work by Tashima et al., they measured the similarity between two names by the Levenshtein distance [13]. We briefly describe their approach below. The Levenshtein distance between two names is defined to be the minimum number of operations required to change one name to the other name. The available operations are the following three ones:

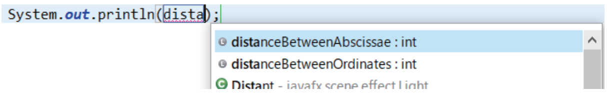
- to add one character,
- to delete one character, and
- to replace one character by another character.

For example, we can produce lineIndent from lineIndex through the following two operations: 1) replace x (in lineIndex) by n, and 2) add t to the end of it. That is, the Levenshtein distance between the above two names is two. The smaller the Levenshtein distance, the higher the similarity between the names. However, the length of the name has also an impact on the similarity assessment. For example, although the Levenshtein distance between file and pipe is also two, the similarity of (file, pipe) does not look at as the same as the similarity of (lineIndex, lineIndent). Hence, Tashima et al. used the following normalized Levenshtein distance (*NLD*) between two names, s_1 and s_2, in their work:

$$NLD(s_1, s_2) = \frac{LD(s_1, s_2)}{\max\{\lambda(s_1), \lambda(s_2)\}}, \tag{1}$$

where $LD(s_1, s_2)$ is the Levenshtein distance between s_1 and s_2, and $\lambda(s_i)$ is the length (character count) of s_i (for $i = 1, 2$).

Tashima et al. conducted an empirical study using Java programs and showed that more fault fixes tend to occur in Java methods which have a pair of local variables with highly similar names [10]. Although their study is a useful previous work which drew attention to the similarity of the variable names, there are the following two problems to be solved.



```
System.out.println(dista);
```

distanceBetweenAbscissae : int
distanceBetweenOrdinates : int
Distant - javafx.scene.effect.Light

Fig. 1. Example of code completion which may cause a wrong selection.

[1] https://atom.io/

[2] https://www.eclipse.org/

1) *The fault introduction events were missed*: The previous work used a snapshot when a fault was fixed. It is better to focus on the fault introduction events rather than the fault fix events for discussing the relationship between the similarity of variable names and the fault-proneness.
2) *The threshold of the similarity was not discussed*: The previous work did not discuss the threshold of the similarity. To construct the guideline about the similarity, we need to find an appropriate threshold value of similarity.

These two problems are our research motivations in this paper.

C. Research Questions

To address the two problems mentioned in Section II-B, we consider the following three research questions (RQs). In this study, we use the following Levenshtein similarity (LS),

$$LS(s_1, s_2) = 1 - NLD(s_1, s_2) \ , \tag{2}$$

as our similarity measure because *NLD* is an inverse measure of similarity.

RQ1: *Does the distribution of similarity differ among software development projects?*

Because different projects may have different developers, the trends of naming variables might also differ among projects. To discuss an appropriate threshold value of the similarity between variable names, we first need to check whether the difference of projects affects the distribution of similarity or not. If there is a significant variation in the distributions, we have to study the appropriate threshold value for each project separately. Otherwise, we can discuss the standard threshold value, which can be commonly used for all projects in this study.

RQ2: *What is the appropriate threshold of similarity?*

If there are two or more local variables whose names are highly similar to each other in a Java method, they are confusable and may adversely affect the code quality of the method. Although Tashima et al. [10] empirically showed that the presence of such a confusing pair of local variables in a Java method is related to the fault-proneness of the method, they did not discuss the appropriate threshold value of the similarity between variable names. Thus, we explore the appropriate threshold in this study and present the result as a guideline about the variable naming.

RQ3: *How can the threshold of similarity contribute to the fault-prone method prediction?*

Once we obtain an appropriate threshold value of the similarity between variable names through the empirical study regarding RQ2, then we can classify Java methods by the threshold. Thus, we examine the usefulness of the classification by the threshold of similarity. More specifically, we build fault-prone method prediction models with and without using the above threshold and compare the prediction performance values between them to evaluate the usefulness of the classification.

We conduct an empirical study on the above three RQs in the following section.

III. Empirical Study

In this section, we report the empirical study that we conducted to address the three RQs mentioned above. First, we describe our aim and data source in Section III-A and explain the procedure of our data collection and analysis in Section III-B. Then, we show our results in Section III-C and give our discussions about the results in Section III-D. Finally, we describe our threats to validity in Section III-E.

A. Aim and Data Source

This study aims to tackle the above three RQs through an empirical data analysis. To this end, we collect fine-grained method-level data of code changes from open source development projects and analyze the risk of introducing faults in terms of the similarity among the names of local variables.

We used ten open source development projects as our data source[3] (see Table I). The main reasons why we use these projects are as follows:

1) The code repository is Git;
2) The primary development language is Java;
3) The issue (bug) tracking system is Apache JIRA;
4) The developers specify the corresponding issue IDs in the commit messages when they commit fixed source files into the code repository.

The reasons 1) and 2) aim to perform a lightweight method data collection. In this study, we collect the change history of methods (functions) from a project. To carry out our data collection effectively, we utilize a Git-based fine-grained code repository, Historage [14], [15] which manages the source code at the method level rather than the file level; Git repositories are convertible to Historage repositories. Because the supported language of Historage repository is Java, we focus only on Java projects in this study.

The reasons 2), 3), and 4) are requirements for collecting fault introduction data. To detect fault introduction events, we use the well-known SZZ algorithm [16]–[18] and one of its implementations, SZZ Unleashed [19]. The SZZ algorithm

TABLE I
Studied open source software development projects

Name	Size (KLOC)	#Contributors	SHA-1
Beam	447	406	`c961139`
Flink	709	501	`4973d8a`
Groovy	187	272	`325653a`
HBase	754	217	`36f0929`
Hive	1,386	183	`4c87512`
Kafka	244	527	`fd79dd0`
RocketMQ	86	139	`971fa8e`
Storm	281	294	`0ea3c89`
Zeppelin	119	279	`4219d55`
Zookeeper	119	77	`7256d01`

[3] `https://{beam,flink,hbase,hive,kafka,rocketmq, storm,zeppelin,zookeeper}.apache.org/, http://groovy-lang.org/`

links a reported issue (bug) with the corresponding issue-fix commits. The SZZ Unleashed is designed to collect issue data from Apache JIRA and to make the above links to commits on Git. Because the issue-commit linking is based on whether or not the issue ID appears in the commit message, the above reason 4) is a requirement in this study.

We performed a project search on GitHub with the search keyword "org:apache," and selected the most popular[4] ten projects (see Table I) satisfying the above requirements.

B. Procedure

We collect data of Java methods and their local variables from each of the studied projects and analyze the collected data in the following procedure.

1) *Fine-grained repository construction*:
Because local variables belong to a Java method, we need to collect data of local variables from each method. Moreover, to capture the fault introduction events in a method, we have to examine the code change history at the method level rather than the file level. Since the Git repositories of the studied projects maintain the code change history at file-level, we convert the repositories into finer-grained method-level repositories—the Historage repositories [14], [15]—by using the conversion tool, Kenja[5] [20].

2) *Fault introduction commit detection*:
According to the SZZ algorithm [16]–[18], we detect the commits in which the code changes introduced faults; We use the SZZ Unleashed [19], an implementation of the SZZ algorithm. In the above detection, we exclude the following kinds of methods because they are not related to any fault introduction: Java methods for testings, demos, and document generations.

3) *Data collection of the fault-proneness of methods and the names of local variables*:
If a method experiences a fault introduction, we define the method to be *faulty*. For a faulty method, we focus on the commit in which the fault introduction occurred first and extract the corresponding revision of the method. Then, for each pair of local variables (including formal parameters) in the selected revision, we compute the similarity between the names of variables. To link a computed similarity with the method in our analysis, we adopt the highest similarity in a method as the representative value of the method.
When a method is not faulty, we single out the latest revision of the method as a sample for our data analysis and compute the similarity of local variables as well.
We henceforth denote the highest similarity between local variable names in method m by $\mathit{HLS}(m)$.

4) *Data analysis for answering RQ1 (Does the distribution of similarity differ among software development projects?)*:
We compare the distributions of computed $\mathit{HLS}(\cdot)$ across the studied projects by using the summary statistics—the minimum, the first quartile (25 percentile), the median, the mean, the third quartile (75 percentile), and the maximum—and the box plots. Moreover, we randomly select 140 samples[6] (methods) from each project, i.e., $1400 (= 140 \times 10)$ samples in total, and perform the Kruskal-Wallis test [21] at significance level 0.05 with the following hypotheses.
   - *Null hypothesis*: There is no difference in the median similarity across the studied projects.
   - *Alternative hypothesis*: At least one project's median similarity is different from the others.

5) *Data analysis for answering RQ2 (What is the appropriate threshold of similarity?)*:
Let M be the set of all Java methods to be analyzed. We consider the following two subsets of M, which are divided by a threshold τ:
   - $M_L(\tau) = \{\ m \in M \mid \mathit{HLS}(m) \leq \tau\ \}$
   - $M_H(\tau) = \{\ m \in M \mid \mathit{HLS}(m) > \tau\ \}$

   Now we explore the appropriate threshold of similarity (τ^*) as follows.
First, for a τ, we compute the following faulty method rates ($\mathit{FMR}_L(\tau)$ and $\mathit{FMR}_H(\tau)$) in $M_L(\tau)$ and $M_H(\tau)$, respectively:
$$\mathit{FMR}_x(\tau) = \frac{\Big|\{m \in M_x(\tau) \mid m \text{ is faulty}\}\Big|}{|M_x(\tau)|} , \quad (3)$$
where $x \in \{L, H\}$.
Then, we evaluate the effect of the classification by τ, using the following odds ratio $\mathit{OR}(\tau)$:
$$\mathit{OR}(\tau) = \frac{\mathit{FMR}_H(\tau) / \{1 - \mathit{FMR}_H(\tau)\}}{\mathit{FMR}_L(\tau) / \{1 - \mathit{FMR}_L(\tau)\}} . \quad (4)$$
The higher $\mathit{OR}(\tau)$ means that the classification by τ is more effective in terms of the fault-prone method detection. We adopt the threshold τ such that $\mathit{OR}(\tau)$ has the highest value, as the appropriate threshold τ^*.

6) *Data analysis for answering RQ3 (How can the threshold of similarity contribute to the fault-prone method prediction?)*:
To examine how the classification by τ^* can contribute to the fault-prone method prediction, we compare the prediction performances of the prediction models with and without using the similarity-based classification. Although various mathematical models have been studied for the fault-prone method prediction in the past, we use the random forest in this study because it has been widely known as one of the most promising models [22], [23].
6a) Examination by random forests without using the similarity-based classification: First, we build a random forest[7] for predicting whether a method is faulty or not,

[4] The popularity is evaluated by the "stars" score on GitHub.

[5] `https://github.com/niyaton/kenja`

[6] We decided the simple size "140" by a simulation using random numbers, where the significance level is 0.05, and the power of a test is 0.8.

[7] We use the randomForest function provided by the randomForest package of R version 3.6.1, with its default settings.

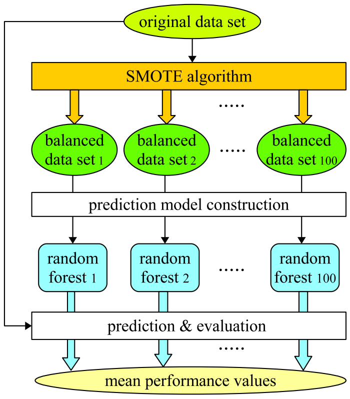


Fig. 2. Summary of the prediction model construction with SMOTE algorithm and the evaluation in step 6a.

by using only the fundamental code metrics: the lines of code (LOC) [24], the cyclomatic complexity (CC) [25], and the number of local variables in a method.

Then, we compute the following performance values: the recall, the precision, and the F value (the harmonic mean of the recall and the precision). These values form the baselines of evaluation in our study.

During the prediction model construction, we have to pay attention to the imbalance between positive samples and negative samples: we have less number of faulty methods than non-faulty ones in our data sets. Such an imbalance of data often leads to a poor prediction model. To overcome this issue of imbalanced data, we use SMOTE algorithm [26], which oversamples the minority by generating similar artificial data and undersamples the majority to balance the data set[8]. By using such a balanced data set, we construct the random forest for predicting faulty methods. Because the behavior of the SMOTE algorithm depends on the random number, we repeat the following two steps 100 times:

6a-1) construct the prediction model, and

6a-2) compute the performance values.

Then, we adopt the mean performance values as the baselines (see Fig. 2).

6b) Examination by random forests with using the similarity-based classification: Next, by τ^* obtained at the step 5, we divide the set of all methods M into $M_L(\tau^*)$ and $M_H(\tau^*)$. Then, we obtain the mean performance values of the prediction models by repeating the following three steps 100 times (see Fig. 3):

6b-1) construct the prediction models for $M_L(\tau^*)$ and $M_H(\tau^*)$, respectively,

6b-2) integrate the prediction results produced by the two models, and

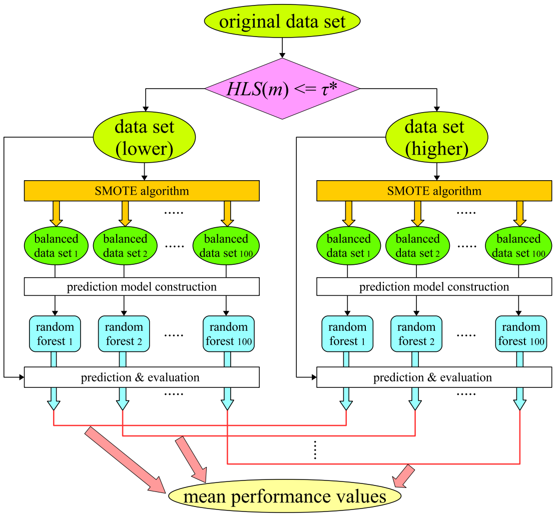


Fig. 3. Summary of the prediction model construction with SMOTE algorithm and the evaluation in step 6b.

6b-3) compute the performance values of the prediction.

6c) Comparison of the prediction performances: Finally, we evaluate the effect of the similarity-based classification by comparing the mean performance values obtained in the step 6a and 6b.

C. Results

By performing the steps 1) – 3) described in Section III-B, we collected $86,104$ methods (including constructors) from the studied ten projects and detected fault introductions by the SZZ algorithm. Notice that the above methods are the ones that have two or more local variables (including formal parameters) because this study focuses on the similarity among the names of local variables. Table II presents the number of methods and that of faulty ones for each project; For example, in Beam project, 63 out of $7,491$ methods are faulty.

We show the results of the steps 4)–6) below, which are corresponding to RQ1, RQ2, and RQ3, respectively. The set of our empirical data is available at http://se.cite.ehime-u.ac.jp/data/QuASoQ2019/.

TABLE II
NUMBER OF ANALYZED METHODS

Project	#Methods	#Faulty methods (faulty rate)
Beam	7,491	63 (0.8%)
Flink	12,854	437 (3.4%)
Groovy	9,926	448 (4.5%)
HBase	13,170	468 (3.6%)
Hive	28,313	523 (1.9%)
Kafka	1,089	30 (2.8%)
RocketMQ	1,998	21 (1.1%)
Storm	6,875	68 (1.0%)
Zeppelin	2,688	249 (9.3%)
Zookeeper	1,700	88 (5.2%)
Total	86,104	2,395 (2.8%)

[8] We use the SMOTE function provided by the DMwR package of R version 3.6.1; To balance the ratio of faulty methods and non-faulty ones as "fifty-fifty" in our data set, we increase the samples of faulty methods by 10-fold through the oversampling and randomly choose (undersample) the same number of non-faulty methods.

TABLE III
SUMMARY STATISTICS OF THE SIMILARITY AMONG THE NAMES OF LOCAL VARIABLES IN THE STUDIED PROJECTS

Project	Min.	25%	50%	Mean	75%	Max.
Beam	0	0.143	0.250	0.332	0.500	0.958
Flink	0	0.143	0.286	0.346	0.556	0.967
Groovy	0	0.167	0.286	0.354	0.556	0.960
HBase	0	0.143	0.267	0.336	0.500	0.955
Hive	0	0.167	0.333	0.384	0.600	0.962
Kafka	0	0.125	0.250	0.320	0.500	0.947
RocketMQ	0	0.143	0.289	0.342	0.539	0.938
Storm	0	0.143	0.267	0.340	0.500	0.962
Zeppelin	0	0.165	0.282	0.351	0.500	0.955
Zookeeper	0	0.123	0.250	0.324	0.500	0.952
All	0	0.150	0.286	0.356	0.556	0.967

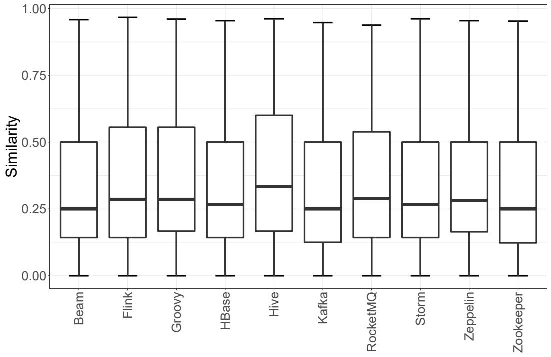


Fig. 4. Boxplot of the similarity among the names of local variables.

Results regarding RQ1 (Does the distribution of similarity differ among software development projects?)

Table III presents the summary statistics of the similarity by the studied projects, and Fig. 4 shows the box plots of them. From the table and figure, the distributions of similarity in projects seem to be close to each other.

Moreover, the p-value of Kruskal-Wallis test was $0.3387 (> 0.05)$, so we cannot reject the null hypothesis "There is no difference in the median similarity across the studied projects." That is, there does not seem to be a significant difference in the central tendency of similarity distribution among projects.

From the above results, our answer to RQ1 (Does the distribution of similarity differ among software development projects?) is: *The distribution of similarity is nearly identical regardless of the project.*

Results regarding RQ2 (What is the appropriate threshold of similarity?)

In the data analysis regarding RQ1, we have seen that the distribution of similarity is nearly identical regardless of the project. Hence, we seek an appropriate threshold of similarity between the names of local variables without distinction of the project.

From Table III and Fig. 4, the range of "relatively high similarity" seems to be between 0.3 and 0.6. Thus, by changing the threshold τ from 0.3 to 0.6 at intervals of 0.01, we divided the set of all methods M into $M_L(\tau)$ and $M_H(\tau)$, obtained the faulty method rates $FMR_L(\tau)$ and $FMR_H(\tau)$, and computed the odds ratio of them, $OR(\tau)$, as the measure of effect. We show the change of the odds ratio over τ in Fig. 5.

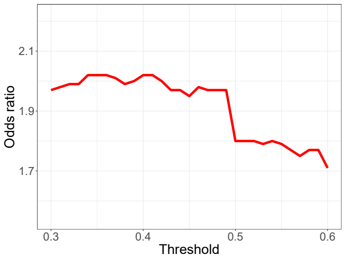


Fig. 5. Changes of the odds ratio over τ.

In the figure, the odds ratio becomes the highest around $\tau = 0.35$–0.4, and it drops around $\tau = 0.5$. Table IV presents a part of the computed odds ratios (OR), the faulty method rates (FMRs), and the ratio between FMRs; In the table, we highlight the highest odds ratio and the highest ratio between FMRs by boldface together with the "$*$" mark.

In Table IV, six thresholds ($\tau = 0.34, 0.35, 0.36, 0.37, 0.40,$ and 0.41) showed the highest odds ratio, i.e., the highest effect of classification by that threshold for detecting fault-prone methods. Because the ratio between faulty method rates also gets the highest value at $\tau = 0.35$ within these six thresholds, we consider it is the appropriate threshold: $\tau^* = 0.35$.

From the above results, our answer to RQ2 (What is the appropriate threshold of similarity?) is: *The appropriate threshold of similarity is around* 0.35.

Results regarding RQ3 (How can the threshold of similarity contribute to the fault-prone method prediction?)

Table V shows the mean performance values of the random forest models constructed in the steps 6a and 6b and the improvements of them by using the similarity-based classi-

TABLE IV
PART OF COMPUTATIONAL RESULTS: THE ODDS RATIO, FAULTY METHOD RATES, AND THE RATIO BETWEEN FAULTY METHOD RATES

threshold	odds ratio	faulty method rates (%)		ratio
τ	$OR(\tau)$	$FMR_L(\tau)$	$FMR_H(\tau)$	$\frac{FMR_H(\tau)}{FMR_L(\tau)}$
0.30	1.97	1.91	3.70	1.932
0.31	1.98	1.91	3.71	1.944
0.32	1.99	1.91	3.73	1.954
0.33	1.99	1.91	3.73	1.954
0.34	**2.02***	1.97	3.90	1.983
$\boxed{\mathbf{0.35}}$	**2.02***	1.97	3.91	**1.984***
0.36	**2.02***	1.97	3.92	1.983
0.37	2.01	1.99	3.92	1.971
0.38	1.99	2.01	3.93	1.953
0.39	2.00	2.01	3.95	1.960
0.40	**2.02***	2.04	4.02	1.975
0.41	**2.02***	2.04	4.02	1.977
0.42	2.00	2.05	4.02	1.960
0.43	1.97	2.09	4.03	1.928
⋮	⋮	⋮	⋮	⋮
0.60	1.71	2.43	4.09	1.682

TABLE V
COMPARISON OF MEAN PERFORMANCE VALUES

	Baseline random forest (RF)			RF + similarity classification			Improvement (%)		
Project	Recall	Precision	F value	Recall	Precision	F value	Recall	Precision	F value
Beam	0.4814	0.0528	0.0950	0.6208	0.0515	0.0951	$+28.95\%$	-2.36%	$+0.11\%$
Flink	0.3845	0.2104	0.2707	0.4569	0.1961	0.2732	$+18.83\%$	-6.80%	$+0.90\%$
Groovy	0.4876	0.2520	0.3314	0.5400	0.2660	0.3557	$+10.73\%$	$+5.54\%$	$+7.31\%$
HBase	0.4187	0.2863	0.3394	0.4856	0.2988	0.3694	$+15.97\%$	$+4.37\%$	$+8.80\%$
Hive	0.4052	0.1565	0.2255	0.4667	0.1522	0.2293	$+15.17\%$	-2.72%	$+1.66\%$
Kafka	0.7507	0.2415	0.3643	0.7777	0.2155	0.3363	$+3.60\%$	-10.77%	-7.69%
RocketMQ	0.9833	0.1364	0.2390	0.9838	0.1327	0.2333	$+0.05\%$	-2.67%	-2.36%
Storm	0.4910	0.0870	0.1477	0.6375	0.0809	0.1433	$+29.83\%$	-7.13%	-2.96%
Zeppelin	0.4522	0.4491	0.4499	0.5183	0.5161	0.5165	$+14.60\%$	$+14.93\%$	$+14.81\%$
Zookeeper	0.5717	0.4362	0.4937	0.6341	0.4548	0.5287	$+10.91\%$	$+4.27\%$	$+7.10\%$
All	0.5426	0.2308	0.2957	0.6121	0.2365	0.3081	$+12.80\%$	$+2.45\%$	$+4.20\%$

fication; In the table, "Baseline random forest (RF)" column and "RF + similarity classification" column correspond to the mean performance values of the random forests constructed in the step 6a (Fig. 2) and 6b (Fig. 3), respectively, and "Improvement" column presents the improvement rates of the latter value from the former value:

$$\frac{\text{latter value} - \text{former value}}{\text{former value}} \times 100\ (\%). \qquad (5)$$

For example of the recall in Beam project, the former value is 0.4814 and the latter one is 0.6208, so the improvement rate is computed as: $(0.6208 - 0.4814)/0.4814 \times 100 \simeq\ +28.95\ (\%)$.

In the table, the recall value, the precision value, and the F value improves by 12.8%, 2.45%, and 4.2% on average of all projects. Moreover, each of all projects shows an improvement in recall value as well. On the other hand, the precision values decrease in six out of ten projects. Because of the reduction of precision values in those projects, the F values also get lower in three out of ten projects.

As a result, the classification using the above threshold ($\tau^* = 0.35$) always works for improving the recall of the fault-prone method prediction. That is, the similarity-based classification contributes to the reduction of risk of overlooking fault-introducing events. Even though some projects showed decreases in the precision and the F values, these measures also got improved on average.

From the above results, our answer to RQ3 (How can the threshold of similarity contribute to the fault-prone method prediction?) are: *That threshold contributes to the reduction of risk of overlooking faults in the fault-prone method prediction. Moreover, although the precision value might be decrease in some cases, the performance of prediction tends to get improved—the recall:* $+12.8\%$*; the precision:* $+2.45\%$*; the F value:* $+4.2\%$.

D. Discussions

To answer RQ1, from each of Java methods in the studied projects, we collected the highest Levenshtein similarity (HLS) among the names of local variables in the method and compared the distribution of HLS values across the projects. As a result, the distributions seem to be close to each other. Moreover, there is no significant difference in the median among the projects. Because the highest similarity tends to be less than 0.5 in many Java methods, we can say that programmers would avoid having a pair of highly similar (confusing) variable names within the same method regardless of the project. To see this trend intuitively, we show samples of variable pairs appearing in the studied projects in Table VI. If the similarity between the two names is greater than 0.5, these names tend to get harder to discriminate each other because the half or more parts of them are duplicated. Especially, pairs of variable names whose similarity is around 0.8–0.9 may lead to an erroneous selection as they are a pair of a word and its plural form or a pair of almost the same names in which the only difference between them is the used number (for example, 1 or 2). It is better to avoid using such highly similar pairs for preventing human errors during the programming activity, and we should support it by providing a quantitative guideline and an automated checking tool.

TABLE VI
SAMPLE PAIRS OF VARIABLE NAMES

Similarity	Pairs of variable names
0.1	(from, components) (name, aggregator)
0.2	(annotation, node) (value, input)
0.3	(method, parameters) (filePath, fieldTypes)
0.35	(descriptorProperties, propertiesMap) (walDirPath, walDirForServerNames)
0.4	(that, other) (metaMethod, metaClass)
0.5	(residualElements, residualSource) (callable, callback)
0.6	(transform, translator) (data, datum)
0.7	(annotation, annotated) (newEntries, newEntry)
0.8	(file, files) (serverName, serverNode)
0.9	(sideInput, sideInputs) (offset1Adj, offset2Adj)

Next, to answer RQ2, we experimented with the similarity-based classification of Java methods and evaluated the effect of classification by the odds ratio in terms of the faulty method rate, while changing threshold τ. As a result, the odds ratio showed the highest value around 0.35–0.4, and we found that the appropriate threshold is 0.35. As we have seen in Table VI, pairs of variable names with the similarity are around 0.35–0.4 may have one or more common words in the compound names. These pairs would not be especially confusing. However, if they become a more similar pair, i.e., the similarity gets larger than 0.5, the risk of a human error such as a mix-up of the variable would also increase. Hence, we consider $\tau = 0.35$ would be a reasonable threshold to warn regarding the confusing pair of variable names.

Finally, to answer RQ3, we prepared two different random forest-based prediction models with and without using the similarity-based classification, and we evaluated how the prediction model with using the similarity (Fig. 3) outperforms the baseline model (Fig. 2) which did not use the similarity. As a result, the mean prediction performance values showed the following improvements: the recall improved by $+12.8\%$, the precision did by $+2.45\%$, and the F value did by $+4.2\%$. Notably, the recall got improved in all of the ten projects. Thus, we consider that the similarity-based classification can work for preventing the overlooking of faults more effectively. Although projects Kafka and RocketMQ showed relatively small improvements ($+3.6\%$ and $+0.05\%$) in Table V, the following two things would be significant reasons: their recall values were already at relatively high levels in the baseline model, and these projects have the least faulty methods in the data set (Table II). On the other hand, the precision values decreased from the baseline in some projects. In general, if we predict more objects as *positive* (i.e., *faulty*) to prevent overlooking the true-positive objects, the precision value tends to decrease. In our data set, because of a small number of faulty methods (Table II), we cannot avoid the depression of the precision when we increase the number of *positive* prediction cases. Nonetheless, the top four projects in terms of the faulty rate (Groovy, HBase, Zeppelin, and Zookeeper) showed improvements in both the recall and the prediction. Therefore, the similarity-based classification tends to help for improving the performance of the prediction model.

In the experiment for answering RQ3, we used only the optimal threshold (τ^*) obtained in the previous experiment regarding RQ2 because we aim to examine the effect of τ^* on the fault-prone method prediction model. Although other threshold values might also have similar effects, a detailed further investigation using various threshold values is our future work.

E. Threats to Validity

We discuss the threats to validity that can affect our results.

Conclusion Validity: The naming of variables may depend on the developers' experience and preference, so the heterogeneity of the project might have an impact on our empirical results. Because there was no significant difference in the median of similarity among the studied projects, we performed our study using a single threshold common to all projects. However, it may be better to collect more data from more projects and to analyze the effect of the threshold by the project domain. That is one of our significant future work.

Internal Validity: We quantitatively analyzed the relationship between the fault-introducing risk in a Java method and the similarity among the names of local variables in the method. However, the observed fault-introducing events may not always be related to the local variables of interest. In other words, there might be a fault at the part which is independent of the variable, and this is a threat to the internal validity. We need a finer-grained code analysis to link a fault-introducing event with the local variables in the future.

Construct Validity: Although we measured the similarity between variable names by using the Levenshtein distance, it is not the only way of evaluating the similarity. There are other edit distance metrics such as the longest common subsequence distance [27] and the Hamming distance [28], and the use of a different metric may lead to a different evaluation of similarity. Moreover, we can consider not only edit distance but also *semantic* distance by using the Doc2Vec [29] method. A further analysis using other similarity metrics is our significant future work.

To evaluate the effect of the similarity-based classification, we constructed the random forest together with the SMOTE algorithm. The selection of parameters in the model construction may affect the results: We specified the parameters of oversampling and undersampling in the SMOTE function so that the ratio between faulty methods and non-faulty methods becomes fifty-fifty, and used the default parameters in the randomForest function. Although a different parameter setting may make a different result, we did not do any special tuning to avoid yet another threat to validity, i.e., the issue of the appropriate setting selection.

External Validity: The threat to external validity is that our data set consists of ten Java open source software projects. Although we collected data from various projects, our results might not represent the results on all data of all software products. Moreover, the difference in the programming language may affect the trend of variable naming. A further data collection and analysis would be needed to mitigate this threat.

IV. Conclusion

In this paper, we focused on the similarity among the names of local variables in a Java method. Because the presence of a highly similar, i.e., "confusing" pair of local variables may decrease the code readability and cause a human error, we quantitatively analyzed such a risk by using the data of fault-introducing events. Through an empirical study using the data of $86,104$ Java methods collected from ten open source projects, we got the following findings:

- To reduce the risk of introducing faults into a Java method, programmers should avoid giving similar names to different variables. The threshold of Levenshtein similarity is 0.35.
- The classification of the Java method by the above threshold works for reducing the risk of overlooking fault-introducing events in the fault-prone prediction model; The recall of the prediction model improves by 12.8% on average.

When a programmer develops a Java method having two or more local variables, it is better to make the names of local variables dissimilar to each other. If there are highly similar names, they can be confusing and may decrease the readability, and consequently, may raise the risk of introducing faults. The above findings can form a quantitative guideline.

Our future work includes: 1) to further analyze the impacts of the differences in the project domain and the programming language by collecting more project data; 2) to develop the tool or the plugin for alerting the presence of similar variables based on our empirical results; 3) to examine the similarity of variable names by using other metrics including other edit distance metrics and semantic metrics.

Acknowledgment

This work was supported by JSPS KAKENHI #18K11246.

References

[1] C. Jones, *Applied Software Measurement: Global Analysis of Productivity and Quality*, 3rd ed. New York: McGraw-Hill, 2008.

[2] D. Boswell and T. Foucher, *The Art of Readable Code: Simple and Practical Techniques for Writing Better Code*. Sebastopol, CA: Oreilly & Associates, 2011.

[3] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Softw. Quality J.*, vol. 14, no. 3, pp. 261–282, Sept. 2006.

[4] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Softw. Eng.*, Nov. 2014, pp. 281–293.

[5] A. Corazza, S. D. Martino, and V. Maggio, "Linsen: An efficient approach to split identifiers and expand abbreviations," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, Sept. 2012, pp. 233–242.

[6] T. M. Pigoski, *Practical Software Maintenance: Best Practices for Managing Your Software Investment*, 1st ed. N.J.: Wiley, 1996.

[7] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Syst. & Softw. Eng.*, vol. 3, no. 4, pp. 303–318, Dec. 2007.

[8] G. Scanniello, M. Risi, P. Tramontana, and S. Romano, "Fixing faults in c and java source code: Abbreviated vs. full-word identifier names," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 2, pp. 6:1–6:43, Jul. 2017.

[9] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Empir. Softw. Eng.*, vol. 18, no. 2, pp. 219–276, Apr. 2013.

[10] K. Tashima, H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "Fault-prone java method analysis focusing on pair of local variables with confusing names," in *Proc. 44th Euromicro Conf. Softw. Eng. & Advanced App.*, Aug. 2018, pp. 154–158.

[11] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "Identifier length and limited programmer memory," *Sc. Comp. Programming*, vol. 74, no. 7, pp. 430–445, May 2009.

[12] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the elipse ide?" *IEEE Softw.*, vol. 23, no. 4, pp. 76–83, July 2006.

[13] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge Univ. Press, 1997.

[14] H. Hata, O. Mizuno, and T. Kikuno, "Historage: Fine-grained version control system for java," in *Proc. 12th Int. Workshop Principles Softw. Evolution & 7th Annual ERCIM Workshop Softw. Evolution*, Sep. 2011, pp. 96–100.

[15] ——, "Bug prediction based on fine-grained module histories," in *Proc. 34th Int. Conf. Softw. Eng.*, Jun. 2012, pp. 200–210.

[16] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. 2005 Int. Workshop Mining Softw. Repositories*, May 2005, pp. 1–5.

[17] C. Williams and J. Spacco, "SZZ revisited: Verifying when changes induce fixes," in *Proc. Workshop on Defects in Large Softw. Systems*, Jul. 2008, pp. 32–36.

[18] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 641–657, July 2017.

[19] M. Borg, O. C. Svensson, K. Berg, and D. Hansson, "SZZ Unleashed: An open implementation of the SZZ algorithm – featuring example usage in a study of just-in-time bug prediction for the jenkins project," in *arXiv:1903.01742*, Mar. 2019, pp. 1–6.

[20] K. Fujiwara, H. Hata, E. Makihara, Y. Fujihara, N. Nakayama, H. Iida, and K. ichi Matsumoto, "Kataribe: A hosting service of historage repositories," in *Proc. 11th Working Conf. Mining Softw. Repositories*, May 2014, pp. 380–383.

[21] W. W. Daniel, *Applied Nonparametric Statistics*, 2nd ed. Boston, MA: Cengage Learning, 1990.

[22] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE Trans. Pattern Analysis & Machine Intelligence*, vol. 20, no. 8, pp. 832–844, Aug. 1998.

[23] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, July 2008.

[24] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, Aug. 2000.

[25] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[26] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artificial Intelligence Research*, vol. 16, pp. 321–357, Jun. 2002.

[27] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proc. 7th Int. Symp. String Processing & Inf. Retrieval*, Sep. 2000, pp. 39–48.

[28] D. J. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge: Cambridge University Press, 2003.

[29] J. H. Lau and T. Baldwin, "An empirical evaluation of doc2vec with practical insights into document embedding generation," in *Proc. 1st Workshop on Representation Learning for NLP*, Aug. 2016, pp. 78–86.