Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 14, no. 8, August 2018, pp. 1887-1895
DOI: 10.23940/ijpe.18.08.p27.18871895

Understanding the Similarity of Log Revision Behaviors in Open Source Software

Xu Niu, Shanshan Li*, Zhouyang Jia, Shulin Zhou, Wang Li, and Xiangke Liao

National University of Defense Technology, Changsha, 410073, China

Abstract

As logging code evolves with bug fixes and feature updates, developers may miss some log revisions due to a lack of general specifications and attention from developers. This makes it more troublesome to achieve good logging practices. In this paper, we try to study log revision behaviors from evolutionary history. Motivated by similar edits of clone codes, we assume there also exist similar log revisions that implicated log revision behaviors. Based on this assumption, we study the similarity of log revision behaviors and answer six research questions. Specifically, we find that 54.14% of log revisions belong to groups of similar log revisions and 64.4% of groups contain log revisions that are missed by developers. We stress the importance of branch statements on learning from similar log revisions since 53.51% of sampled similar log revisions are related to the semantics of branch statements.

Keywords: log revision; software evolution; failure diagnose

(Submitted on May 13, 2018; Revised on June 20, 2018; Accepted on July 27, 2018)

1. Introduction

Log statements are widely used to collect the runtime status of software. Log messages play a critical role in postmortem failure diagnosis. Despite the wide applications, it is difficult to achieve good logging practices as software evolves. This is mainly caused by three reasons. First, there are no rigorous specifications and systematic processes to guide the practices of software logging [1-3]. In this case, making log decisions is subjective and arbitrary.

Second, logging code also evolves with feature updates and bug fixes. This makes it more difficult to improve logging practices in evolution. Log revisions displayed in Figure 1 illustrate this problem. Figure 1a shows one log statement that worked well in the original version. For satisfying the new feature of the supporting argument "--help", the variable of this log statement was updated to permit logging into the standard output stream. Thus, evaluation of the same log statement varies in versions of software due to updated features. (Here "versions" means the internal version number, not the release version. This may be incremented many times in one day.) In Figure 1b, there is a patch with an inserted branch structure and corresponding log statement to check the return value of find_next_block(). In this case, developers did not notice the necessity to log until the software had encountered bugs during processing of the sparse array. In other words, real-world bugs in software are so unpredictable that it is troublesome for developers to provide sufficient log messages without triggering bugs. In both cases, log revisions are closely related to feature updates or bug fixes, which emerge over software evolution and are difficult to be predicted by existing tools [4-5].

Third, log revisions may be missed by developers (simplified as "missed log revisions"). For example, the log revision in Figure 2(b) was similar to the one in Figure 2(a) since the static text of both log statements were updated from "malloc" to "calloc". Despite their similarity, these two log revisions were actually performed in different commits, and the one in Figure 2(b) was missed by developers and committed several days later.

* Corresponding author.
E-mail address: shanshanli@nudt.edu.cn

In order to improve logging practices as software evolves and avoid missing log revisions, we plan to learn log revision behaviors from evolution history. Motivated by similar edits of clone codes [6-7], we assume there also exist similar log revisions that implicated log revision behaviors. Based on this assumption, we aim to study the similarity of log revision behaviors.

Commit id: e1dd3bfa4a9fb789b03c1857786d58c45b0b2901 in master of Make
File: main.c
Commit Message: (decode_switches): For –help, print usage to stdout.

```
 if (!env && (bad || print_usage_flag))
 {
+  FILE *usageto;
-  fprintf (stderr, "Usage: %s [options] [target] ...\n", program);
+  usageto = bad ? stderr : stdout;
+  fprintf (usageto, "Usage: %s [options] [target] ...\n", program);
```

(a) Update log statement with feature updates

Commit id: 72290b64a6433a3c88ecb7b3044c3047beffeadc in master of Tar
File: src/extract.c
Commit Message: (extract_archive): Fix sparse array bug: we did not find end of array correctly. Don't assume find_next_block yields nonnull.

```
  data_block = find_next_block ();
+ if (! data_block)
+ {
+   ERROR ((0, 0, _("Unexpected EOF on archive file")));
+   return;}
```

(b) Update log statement along with bug fixes

Figure 1. Log revision examples in software evolution

Commit id: 4743f198cfd38e2776bf1a8af8b31c37c4ea537e in master of Collectd
File: src/collectd-tg.c.c
Commit Message: collectd-tg: malloc + memset − > calloc.

```
- vl = malloc (sizeof (*vl));
+ vl = calloc (1, sizeof (*vl));
  if (vl == NULL){
-  fprintf (stderr, "malloc failed.\n");
+  fprintf (stderr, "calloc failed.\n");
   return (NULL);}
```

(a) Update of static text in a manner similar to Figure 2(b)

Commit id: 7383e7cf33edb331e0ec64ba2c1f67b297856dc8 in master of Collectd
File: src/liboconfig/oconfig.c
Commit Message: liboconfig: malloc + memset − > calloc

```
- ci_copy = malloc (sizeof (*ci_copy));
+ ci_copy = calloc (1, sizeof (*ci_copy));
  if (ci_copy == NULL){
-  fprintf (stderr, "malloc failed.\n");
+  fprintf (stderr, "calloc failed.\n");
   return (NULL);}
```

(b) Update of static text in a manner similar to Figure 2(a)

Figure 2. Similar log revisions in software evolution

Many researchers have already studied features of logging practices during software evolution. Yuan et al. [3] and Chen et al. [8] quantitatively studied the general features of log revisions in open-source software. Chen et al. [9] manually analyzed historical log revisions to summarize six anti-patterns of logging code. Li et al. [10] summarized four categories of reasons for log revisions to help decide whether to perform log revision in certain commits. The aforementioned works have not systematically characterized the similarity of log revision behaviors in software evolution.

To fill this gap, in this paper, we perform an empirical study to systematically characterize the similarity of log revision behaviors. Specifically, we study the similar log revisions in six pieces of large, widely used open-source software. We group automatically retrieved log revisions to generate similar log revisions. Through detailed statistical analysis and careful manual analysis, we successfully answer the following six research questions:

RQ1: How pervasive are similar log revisions? We find that an average of 54.14% of log revisions belong to groups of similar log revisions (i.e., simplified as "revision group"). This result validates the feasibility of learning from similar log revisions.

RQ2: How many similar log revisions are not committed simultaneously? On average, 64.4% of revision groups consist of log revisions that are similar but not committed simultaneously. Thus, learning from similar log revisions may help to predict these missed log revisions.

RQ3: How many similar log revisions have influences on software behaviors? On average, 91.0% of revision groups are involved with non-trivial modifications [11], which have influences on software behaviors. In this case, learning from similar log revisions helps to predict non-trivial log revisions.

RQ4: How many categories of log modifications are covered by similar log revisions? We find that categories covered by similar log revisions are so diverse (see section 3.4) that learning from log revisions may predict various sorts of log modifications.

RQ5: What is the size of similar log revision groups? Our study shows that the group size of 60% to 95% of revision groups is smaller than (may equal to) 6. Thus, we suggest that the algorithm used to learn from similar log revisions should be able to mine behaviors from limited samples.

RQ6: What is the reason for similar log revisions? It turns out that an average of 53.51% of similar log revisions are caused by "check similar return value" or "check similar variables" (see section 3.6). Consequently, we stress the importance of branch statements on learning from similar log revisions.

The rest of the paper is organized as follows. Section 2 introduces the six subject systems and how we collect research data. Section 3 discusses the procedure of answering the six research questions. Section 4 describes the limitations of this paper, and section 5 presents the related works. At last, we conclude our work in section 6.

2. Methodology

This section introduces the subject systems and the process of collecting research data.

2.1. Subject Systems

This empirical study is conducted on six open-source projects in C/C++ languages. They are Squid [12], Git [13], Make [14], Collectd [15], Tar [16], and Wget [17]. Each of these projects has a long development history of more than 11 years. This improves the reliability and validity of this research study.

Table 1 lists detailed information of these subject systems. Among these indicators, the line of code (LOC, i.e., the second column) is evaluated by SLOCCount [18] which can eliminate comments and empty lines. The number of commits (the fourth column) summarizes all commits during studied history (the third column). As shown in the table, we collect more than 48K commits which were performed in at least ten years. Given one commit, we judge whether it contains log revisions (see section 2.2 for details of how to recognize log revisions) to decide whether it is a log commit. The fifth column displays the number of log commits in six projects. In total, we have retrieved more than 5K log commits from the six projects. Besides, we evaluate the ratio of log commits to commits to measure the pervasiveness of log revisions in one project and show it in the last column. On average, 11.67% of commits involve log revisions. This result is similar to the findings of a previous study [8]. In addition, the diversity of the aforementioned indicators among subject systems increases the universality of our research.

Table 1. Subject systems

Software	LOC	Studied history	#Commits	#Log commits	#Log commits /#commits
Git	429166	2007-05-12 to 2018-04-03	25463	2889	11.35%
Squid	214211	1996-03-27 to 2018-04-03	9377	1698	18.11%
Make	35199	1988-04-23 to 2018-04-03	2492	278	11.16%
Collectd	97475	2005-12-17 to 2018-04-08	4698	221	4.70%
Tar	77310	1990-11-01 to 2018-04-03	2895	169	5.84%
Wget	84678	1999-12-01 to 2018-04-03	3837	435	11.34%
Total	938039		**48762**	**5690**	**11.67%**

2.2. Data Collection

In order to study the similarity of log revision behaviors in evolution, we should primarily generate similar log revisions in each software as input for the data research. This procedure can be divided into four steps.

First, for each software, we crawl all available historical commits of the master branch from its version control website (Commits of Squid, Git, and Collect are crawled from Github [19], while commits of Make, Tar, and Wget are collected from the GNU official site [20]). For each commit, we record its commit messages, commit time, and patches. Second, given one patch, its containing hunks (a hunk is the basic unit in a patch which begins with range information and is immediately followed by line additions, line deletions, and any number of contextual lines) are roughly filtered using regex to select hunks that contain log statements. The regex pattern used here depends on log functions that are recognized by traditional methods [4-5]. Third, all selected hunks are then passed to GumTree [21-22], which generates the syntactical edit scripts. These edit scripts help us record log revisions that syntactically edit log statements rather than empty lines or comments. Lastly, we generate similar log revisions by grouping these log revisions according to their edit scripts.

The above processes produce groups of similar log revisions, which serve as the input of this empirical study.

3. Similarity of Log Revision Behaviors

In this section, we study the similarity of log revision behaviors from three aspects by answering six research questions. First, we validate the feasibility of learning from similar log revisions by evaluating the proportion of similar log revisions in evolution (see section 3.1). Second, we quantitatively illustrate the necessity of learning from similar log revisions (see sections 3.2, 3.3, and 3.4). Particularly, we quantitatively evaluate how many similar log revisions are not committed simultaneously and how many have influences on software behaviors. Furthermore, we summarize the categories of log modifications that are covered by revision groups. Third, we aim to provide some suggestions on how to learn from similar log revisions automatically (see sections 3.5 and 3.6). To do this, we study the size of revision groups (simplified as "group size") and the reason for similar log revisions.

3.1. How Pervasive are Similar Log Revisions?

Motivation: The pervasiveness of similar log revisions in evolution history seriously affects the feasibility of learning from similar log revisions. In order to understand how pervasive similar log revisions are, this section intends to quantitatively characterize the proportion of similar log revisions.

Approach: This indicator is evaluated with the ratio of similar log revisions to all log revisions. Its detailed formula is shown as follows.

$$Proportion\ of\ similar\ log\ revisions = \frac{Number\ of\ similar\ log\ revisions}{Number\ of\ all\ log\ revisions} \quad (1)$$

Result: As shown in Table 2, the proportion of similar log revisions ranges from 39.11% to 64.29% in subject systems. Despite the diversity, on average, 54.14% of log revisions belong to similar log revisions. This result validates the feasibility of learning from similar log revisions. Besides, around 54.14% of log revisions may be predicted if we could make full use of similar log revisions.

Table 2. Proportion of similar log revisions and the average group size

Software	#Log revisions	#Similar log revisions	Proportion of similar log revisions	#Group	Average group size
Git	10339	5026	48.61%	1255	4.00
Squid	13112	7425	56.63%	2046	3.63
Make	2603	1327	50.98%	367	3.62
Collectd	1624	1044	64.29%	381	2.74
Tar	813	318	39.11%	100	3.18
Wget	2973	1894	63.71%	456	4.15
Total	31464	17034	**54.14%**	4605	**3.70**

Finding 1: On average, the proportion of similar log revisions to all log revisions in evolution reaches 54.14%.

3.2. How Many Similar Log Revisions are not Committed Simultaneously?

Motivation: Revision groups whose members are not committed simultaneously are considered missed revision groups. Thus, we answer this research question by evaluating the proportion of missed revision groups. This indicator is positively related to the necessity of learning from historically similar log revisions. For example, it is meaningless if there are no missed revision groups.

Approach: This paper evaluates the proportion of missed revision groups with the ratio of missed revision groups to all revision groups. Its formula is listed as follows.

As for implementation, we automatically recognize missed revision groups by comparing the commit time of log revisions. A revision group is treated as a missed revision group if its members are not committed at the same time.

Results: As shown in Figure 3, on average, 64.4% of revision groups belong to missed revision groups whose members are not committed simultaneously. This points out the importance of learning from similar log revisions, which may predict these missed log revisions.

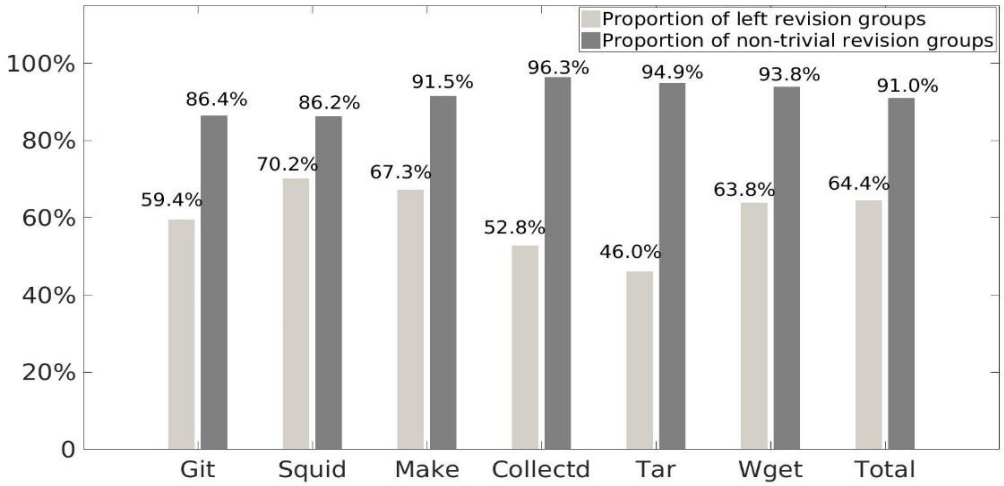

Figure 3. Proportion of missed revision groups and non-trivial revision groups

$$Proportion\ of\ missed\ revision\ groups = \frac{Number\ of\ missed\ revision\ groups}{Number\ of\ all\ revsion\ groups} \quad (2)$$

Finding 2: On average, 64.4% of revision groups consist of log revisions that are similar but not committed simultaneously.

3.3. How Many Similar Log Revisions Have Influences on Software Behaviors?

Motivation: Non-trivial revision groups are revision groups whose members are all non-trivial modifications [11, 23] on log statements that have influences on software behaviors. In this section, we evaluate how many similar log revisions have influences on software behaviors with the proportion of non-trivial revision groups. This indicator is positively related to the potential usefulness of log revision behaviors learned from similar log revisions. Specifically, it is critical to predict non-trivial log revisions.

Approach: We sample 1084 revision groups (8732 pieces of log revisions) and identify whether they are non-trivial revision groups by manually checking their commit messages and changed codes. For example, rename-inducing modifications, local variable extractions, and whitespace updates are not considered non-trivial modifications because they do not influence software behaviors. Then, we calculate the proportion of non-trivial revision groups with the ratio of non-trivial revision groups to all revision groups (see the following formula).

$$\text{Proportion of non} - \text{trivial revision groups} = \frac{\text{Number of non} - \text{trivial revision groups}}{\text{Number of all revision groups}} \quad (3)$$

Results: As displayed in Figure 3, on average, there are 91.0% of revision groups that belong to non-trivial revision groups that have impacts on software behaviors. Thus, learning from similar log revisions can predict non-trivial log revisions.

Finding 3: On average, 91.0% of revision groups are involved with non-trivial modifications that have influences on software behaviors.

3.4. How Many Categories of Log Modifications are Covered by Similar Log Revisions?

Motivation: As proven by previous studies [3, 8, 10], log revisions in evolution may cover many categories of log modifications (e.g., log insertion, log deletion). In this subsection, we study the categories of log modifications covered by similar log revisions which imply categories of log modifications that may be predicted with similar log revisions.

Approach: According to previous works [3, 8], we divide log revisions into five categories: log insertion, log deletion, update of log function, update of log variables, and update of static text. To identify the category for log revisions, we

design and implement a simple classifier. It utilizes syntactical edit scripts mentioned in section 2.2 to decide what components (e.g. log variables, static text) of the log statements have been modified.

Results: With the help of this classifier, we characterize the distribution of revision groups among the five categories and display it in Table 3. This result meets with the distribution of all log revisions among categories of log modifications [3, 8] and implies the diversity of categories covered by log revisions. In other words, learning from similar log revisions can help predict various categories of log modifications.

Table 3. Categories of log modifications covered by similar log revisions (Proportion is the ratio of one value to the sum of all values in the same row)

Software	Total	Log insertion	Log deletion	Update of log function	Update of variables	Update of static text
Git	7215	2051(28.43%)	621(8.61%)	1130(15.66%)	2600(36.04%)	813(11.27%)
Squid	9177	3470(37.81%)	1611(17.55%)	934(10.18%)	1913(20.85%)	1249(13.61%)
Make	1653	631(38.17%)	186(11.25%)	187(11.31%)	532(32.18%)	117(7.08%)
Collectd	1089	505(46.37%)	474(43.53%)	11(1.01%)	64(5.88%)	35(3.21%)
Tar	426	149(34.98%)	51(11.97%)	26(6.10%)	126(29.58%)	74(17.37%)
Wget	2489	926(37.20%)	333(13.38%)	120(4.82%)	763(30.65%)	347(13.94%)
Total	22049	7732(35.07%)	3276(14.86%)	2408(10.92%)	5998(27.20%)	2635(11.95%)

Finding 4: Generally speaking, categories covered by similar log revisions share similar distributions with those covered by all log revisions.

3.5. What is the Size of Similar Log Revision Groups?

Motivation: The size of one revision group measures the number of log revisions within it. This indicator affects which type of algorithms should be used to learn from similar log revisions. For example, we should not choose deep learning models if most revision groups consist of few members.

Approach: In detail, we take two indicators to study the feature of group size. First, the average group size, which evaluates the ratio of the number of similar log revisions to the number of revision groups, is used to express the average case of group size. Second, we use the cumulative density diagram shown in Figure 4 to depict the distribution of group size.

Results: As listed in Table 2, the average group size is 3.7. In other words, on average there are only 3.7 samples in one revision group. Meanwhile, Figure 4 points out that the group size of 60% to 95% of revision groups ranges from 2 to 6. Considering the rather small group size, the algorithm used to learn from similar log revisions should be able to mine behaviors from limited samples (e.g., program synthesis algorithms [24]).

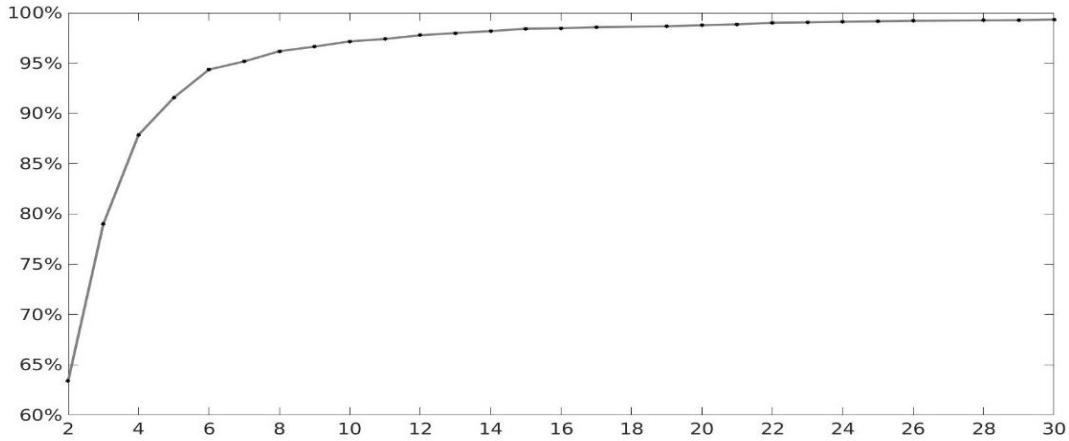

Figure 4. Cumulative density of group size among similar log revisions (horizontal axis is group size while vertical axis is cumulative density)

Finding 5: On average, the size of revision groups is 3.7 and 60% to 95% of revision groups have group sizes which are smaller than (may equal to) 6.

3.6. What is the Reason for Similar Log Revisions?

Motivation: Given one revision group, the reason for similar log revisions determines why developers perform similar edits on these log statements. Thus, it is the prerequisite for learning from similar log revisions. For example, the pair of similar log revisions shown in Figure 2 was modified similarly because their branch statements both checked the return value of malloc() (i.e. "check similar return value").

Approach: To understand why developers perform similar log revisions, we sample 1084 revision groups (8732 pieces of log revisions) and manually analyze the reason for similar log revisions in reference to the commit messages and changed codes. As shown in Table 4, we have summarized ten categories of reasons for similar log revisions.

Table 4. Detailed explanation of reasons for similar log revisions

Reasons	Explanation
Block change	Similar log revisions are associated with massive edits of unfunctional code (e.g. adding or deleting entire functions or entire files)
Invoke same log function	Modified log statements have invoked the same log function which is updated
Invoke same function	Log variables of modified log statements have invoked the same function which is updated
Check similar return value	Modified log statements are under control of similar branch statements which check the return value of similar functions (e.g. functions that share similar names)
Check similar variables	Modified log statements are under control of similar branch statements which check similar variables (e.g. variables that have similar variable types)
Inner similar functions	Modified log statements are inner similar functions (e.g. functions that share similar names)
Use same text	Static text of modified log statements shares common content
Use same text format	Static text of modified log statements shares common text format
Use same constant	Modified log statements involve with the same constant
Use similar variables	Modified log statements have used similar variables (e.g. variables that have similar variable types)

Results: Table 5 lists the distribution of sampled revision groups among the ten reasons. Among these reasons, "check similar return value" and "check similar variables" counter the highest proportion (i.e. 27.31% and 26.20%). "Check similar return value" indicates the log revisions are similar because the branch statements of modified log statements check the return value of similar functions, while "check similar variables" means these branch statements check similar variables (e.g. variables that share the same type). According to their definitions, these two reasons are both related to the branch statement that controls the modified log statement. Consequently, we suggest for works that aim to learn from similar log revisions to take branch statements into consideration.

Finding 6: On average, 53.51% of similar log revisions are caused by "check similar return value" or "check similar variables". This implies the influence of branch statements on similar log revisions.

4. Threats to Validity

In this section, we will discuss threats to the validity of our empirical study.

4.1. Representativeness of Subject Systems

This paper studies the similarity of log revision behaviors by conducting an empirical study on similar log revisions in six subject systems. Consequently, the accuracy of our finding is closely related to the representativeness of the six subject systems. To promise the reliability and universality of our work, these six projects are all popular in respective fields with a long development history.

4.2. Accuracy of Manual Analysis

As we have mentioned, the procedures of deciding whether a log revision is non-trivial or not and why groups of log revisions are similarly modified both require a thorough understanding of commit messages and changed codes. Consequently, we must perform a careful manual analysis to answer the two research questions. In this case, the accuracy of our findings is also affected by the accuracy of the manual analysis.

Table 5. Distribution of sampled revision groups among reasons (Proportion is the ratio of value to the sum of all values in the same column)

Reasons	Total	Git	Squid	Make	Collectd	Tar	Wget
Block change	63	4	7	17	14	9	12
	(5.81%)	(1.87%)	(3.45%)	(8.02%)	(8.70%)	(9.09%)	(6.15%)
Invoke same log function	66	21	18	18	2	3	4
	(6.04%)	(9.81%)	(8.87%)	(8.49%)	(1.24%)	(3.03%)	(2.05%)
Invoke same function	61	30	10	3	4	0	14
	(5.63%)	(14.02%)	(4.93%)	(1.42%)	(2.48%)	(0.00%)	(7.18%)
Check similar return value	296	60	48	46	52	45	45
	(27.31%)	**(28.04%)**	**(23.65%)**	**(21.70%)**	**(32.30%)**	**(45.45%)**	**(23.08%)**
Check similar variables	284	28	38	82	47	23	66
	(26.20%)	(13.08%)	**(18.72%)**	**(38.68%)**	**(29.19%)**	**(23.23%)**	**(33.85%)**
Inner similar functions	60	15	18	5	7	2	13
	(5.54%)	(7.01%)	(8.87%)	(2.36%)	(4.35%)	(2.02%)	(6.67%)
Use same text	56	15	26	7	2	3	3
	(5.17%)	(7.01%)	(12.81%)	(3.30%)	(1.24%)	(3.03%)	(1.54%)
Use same text format	40	11	3	5	2	6	13
	(3.69%)	(5.14%)	(1.48%)	(2.36%)	(1.24%)	(6.06%)	(6.67%)
Use same constant	12	1	2	2	3	3	1
	(1.11%)	(0.47%)	(0.99%)	(0.94%)	(1.86%)	(3.03%)	(0.51%)
Use similar variables	146	29	33	27	28	5	24
	(13.47%)	(13.55%)	**(16.26%)**	(12.74%)	**(17.39%)**	(5.05%)	(12.31%)
Total	1,084	214	203	212	161	99	195
Total revisions	8,732	2,721	2,785	1,014	604	316	1,292

5. Related Works

There are two areas of research that are closely related to our work: empirical studies on logging practices and improving logging practices.

5.1. Empirical Studies on Logging Practices

Yuan et al. [3] and Chen et al. [8] quantitatively studied the logging practices in open-source subjects and concluded several impressive findings. Fu et at. [1] studied practices of log placements in industrial software. Chen et al. [9] manually summarized six anti-patterns from historical log revisions. Li et al. [10] studied the reason for log revisions. In this paper, we also perform an empirical study on logging practices, but our focus is on the similarity of log revision behaviors. In this case, our work is supplementary of the above works.

5.2. Improving Logging Practices

Errlog [4], LogAdvisor [5], Log2 [25], and Log20 [26] suggested where to place log statements in one code by understanding log patterns or balancing informativeness and performances. LogEnhancer [27] improved logging practices by detecting uncertainty variables and appending them to log statements. LCAnalyzer [9] detected defective logging codes by comparing them with six anti-patterns. The above works focus on improving logging practices while our work studies the similarity of log revision behaviors in evolution, which may inspire tools for leveraging knowledge in similar log revisions.

6. Conclusions

In this paper, we study the similarity of log revision behaviors to make up for existing researches on logging practices. We conduct an empirical study on six open source projects to quantitatively validate the necessity and feasibility of learning from similar revisions and recommend two suggestions on how to learn from similar revisions automatically.

Specifically, we find that 54.14% of log revisions belong to groups of similar log revisions and 64.4% of groups contain log revisions that are missed by developers. 53.51% of sampled revision groups are modified similarly due to the similar semantics of branch statements. Considering the importance of branch statements, for future works, we plan to leverage the semantics of branch statements to learn log revision behaviors from similar log revisions automatically. For example, it seems meaningful to mine the correlation between the semantics of branch statements and the log revisions.

Acknowledgements

This research is supported by the National Natural Science Foundation of China (Project No. 61690203 and U1711261) and the National Key R&D Program of China (Project No. 2017YFB1001802 and 2017YFB0202201).

References

1. Q. Fu, J. Zhu, W. Hu, J. G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do Developers Log? an Empirical Study on Logging Practices in Industry," in *Proceedings of the 36th International Conference on Software Engineerin*g (*ICSE*), pp. 24-33, Hyderabad, India, May 2014
2. A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry Practices and Event Logging: Assessment of a Critical Software Development Process," in *Proceedings of the 37th IEEE International Conference on Software Engineering* (*ICSE*), pp. 169-178, Florence, Italy, May 2015
3. D. Yuan, S. Park, and Y. Zhou, "Characterizing Logging Practices in Open-source Software," in *Proceedings of the 34th International Conference on Software Engineering* (*ICSE*), pp. 102-112, Zürich, Switzerland, June 2012
4. D. Yuan, S. Park, P. Huang, Y. Liu, and M. Lee, "Be Conservative: Enhancing Failure Diagnosis with Proactive Logging," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (*OSDI*), pp. 293-306, California, USA, October 2012
5. J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to Log: Helping Developers Make Informed Logging Decisions," in *Proceedings of the 37th International Conference on Software Engineering* (*ICSE*), pp. 415-425, Firenze, Italy, May 2015
6. N. Meng, M. Kim, and K. S. McKinley, "LASE: Locating and Applying Systematic Edits by Learning from Examples," in *Proceedings of the 35th International Conference on Software Engineering* (*ICSE*), pp. 502-511, San Francisco, America, May 2013
7. R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning Syntactic Program Transformations from Examples," in *Proceedings of the 39th International Conference on Software Engineering* (*ICSE*), pp. 404-415, Buenos Aires, Argentina, May 2017
8. B. Chen and Z. M. Jiang, "Characterizing Logging Practices in Java-based Open Source Software Projects a Replication Study in Apache Software Foundation," *Empirical Software Engineering*, Vol. 22, No. 1, pp. 330-37, 2017
9. B. Chen and Z. M. Jiang, "Characterizing and Detecting Anti-Patterns in the Logging Code," in *Proceedings of the 39th International Conference on Software Engineering* (*ICSE*), pp. 71-81, Buenos Aires, Argentina, May 2017
10. H. Li, W. Shang, Y. Zou, and A. E. Hassan, "Towards Just-in-time Suggestions for Log Changes," *Empirical Software Engineering*, Vol. 22, No. 4, pp. 1831-1865, 2017
11. D. Kawrykow and M. P. Robillard, "Non-essential Changes in Version Histories," in *Proceedings of the 33th International Conference on Software Engineering* (*ICSE*), pp. 351-360, Hawaii, USA, May 2011
12. T. A. S. Foundation, "Httpd - Apache Hypertext Transfer Protocol Server," (http://httpd.apache.org/docs/2.4/programs/httpd.html)
13. S. F. Conservancy, "Git," (https://git-scm.com/)
14. W. Venema, "The Postfix Home Page," (http://www.postfix.org/)
15. Collectd, "The System Statistics Collection Daemon," (http://collectd.org/)
16. F. S. Foundation, "Tar - Gnu Project - Free Software Foundation," (https://www.gnu.org/software/tar/)
17. F. S. Foundation, "Wget - Gnu Project - Free Software Foundation," (https://www.gnu.org/software/wget/)
18. S. Media, "Sloccount Download - Sourceforge.net," (https://sourceforge. net/projects/sloccount/)
19. Github, "Github," (https://github.com/)
20. Cgit, "Savannah Git Hosting," (http://git.savannah.gnu.org/cgit/)
21. J. R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus, "Fine-grained and Accurate Source Code Differencing," in *Proceedings of the 29th international conference on Automated Software Engineering* (*ASE*), pp. 313-324, Vasteras, Sweden, September 2014
22. Github, "Github - Gumtreediff/gumtree: a Neat Code Differencing Tool," (https://github.com/GumTreeDiff/gumtree)
23. M. Mondai, C. K. Roy, and K. A. Schneider, "Micro-clones in Evolving Software," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering* (*SANER*), pp. 50-60, Campobasso, Italy, March 2018
24. O. Polozov and S. Gulwani, "FlashMeta: a Framework for Inductive Program Synthesis," *ACM SIGPLAN Notices*, Vol. 50, No. 10, pp. 107-126, 2015
25. R. Ding, H. Zhou, J. G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log 2: a Cost-aware Logging Mechanism for Performance Diagnosis," in *Proceedings of the 2015 USENIX Conference on USENIX Annual Technical Conference* (*USENIX ATC*), pp. 139-150, California, America, July 2015
26. X. Zhao, K. Rodrigues, and M. Stumm, "Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles* (SOSP), pp. 565-581, Shanghai, China, October 2017
27. D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving Software Diagnosability via Log Enhancement," *ACM Transactions on Computer Systems*, Vol. 30, No. 1, pp. 1-28, Zürich, Switzerland, June 2012