**An in-depth understanding of inconsistent labels in multi-version-project defect data sets**

Shiran Liu1,2 Zhaoqiang Guo1,2 Yanhui Li1,2\* Chuanqi Wang1,2

Lin Chen1,2 Zhongbin Sun3 Yuming Zhou1,2\* Baowen Xu1,2

1State Key Laboratory for Novel Software Technology, Nanjing University

2Department of Computer Science and Technology, Nanjing University

3School of Computer Science & Technology, China University of Mining and Technology

\*Corresponding author

# **Appendix A. The running time of our TSILI algorithm**

The method we adopted to generate the source code databases required by the TSILI algorithm is to download the codes corresponding to each version from the official website of each target project, and then use the Understand[[1]](#footnote-1) tool to parse the code to generate source code databases (.udb file). We write a Python[[2]](#footnote-2) script to implement the TSILI algorithm. In the second stage of TSILI, the source code of each module is parsed and filtered (b9 in Fig. 17) based on the Python API (application programming interface) provided by the Understand tool.

In order to observe the time required for the TSILI algorithm to detect inconsistent labels on projects of different orders of magnitude, we selected Log4j, Hive, and Eclipse projects from multi-version-project defect data sets Metrics-Repo-2010 [1], JIRA-RA-2019 [2], and ECLIPSE-2007 [3], respectively, and then ran TSILI and recorded the time spent. Table 1 lists the details of these three projects and the single thread running time of TSILI. The 2nd column lists the versions included in each project. The 3rd column lists the order of magnitude of the project size, where *n*, *totalIns*, and *sumSLOC* represent the number of versions, the total number of instances of all versions, and the total number of code lines of all versions, respectively. The 4th column reports the running time of TSILI. These three projects (Log4j, Hive, and Eclipse) were selected because they represented orders of magnitude of the size of the projects in their respective data sets (the ECLIPSE-2007 data set only has the Eclipse project). In addition, the size of the total number of instances of these three projects varies in turn by one order of magnitude, which is conducive to observe the running time of TSILI under different orders of magnitude data sets.

Table 1 shows that the running time of TSILI is positively correlated with the size of the data set. For the project with hundreds of instances (i.e., Log4j project), the running time is at the second level. For the projects with ten thousands of instances (i.e., Eclipse project), the running time is at the hour level. Because TSILI is an offline algorithm, the hour-level (even minute-level or second-level) running time is acceptable in practice.

Table 1. Time consuming of the TSILI algorithm

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dataset | Project (versions) | n, totalIns, sumSLOC | Running Time | Experimental environment |
| Metrics-Repo-2010 | Log4j (1.0, 1.1, 1.2) | n=3, totalIns=411, sumSLOC=74857 | ≈ 25 seconds | Inter(R) Core(TM) i7-7700 CPU @ 3.6GHz and 16G RAM |
| JIRA-RA-2019 | Hive (0.9.0, 0.10.0, 0.12.0) | n=3, totalIns=5285, sumSLOC=974774 | ≈ 11 minutes |
| ECLIPSE-2007 | Eclipse (2.0, 2.1, 3.0) | n=3, totalIns=25203, sumSLOC=3089619 | ≈ 3 hours |

# **Appendix B. Metrics in the MA-SZZ-2020 data set**

Table 2 describes the size, complexity, coupling, and inheritance metrics in the MA-SZZ-2020 data set we collected. In Table 2, column “Type” represents the type to which each metric belongs to, column “Name” gives the acronym of each metric, column “Deﬁnition” provides an informal description of the corresponding metric, and column “Tool for measuring metrics” gives the source of the tool that we measure metrics from. Note that inheritance metrics are indeed a form of coupling metrics. In practice, however, many researchers distinguish inheritance metrics from coupling metrics. Our study follows a metric classiﬁcation framework similar to that in Briand et al. [4].

Table 2. List of metrics in the MA-SZZ-2020 data set

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Name | Deﬁnition | Tool for measuring metrics |
| Size Metrics | SLOC (loc in data set) | the non-commentary source lines of code in a class | We used the Perl script developed in previous studies [5, 6] to collect metrics based on the udb database, where the udb database is generated by the commercial software Understand. |
| NMIMP | the number of methods implemented in a class |
| NumPara | sum of the number of parameters of the methods implemented in a class |
| NM | the number of methods in a class, both inherited and non-inherited |
| NAIMP | the number of attributes in a class excluding inherited ones |
| NA | the number of attributes in a class including both inherited and non-inherited |
| Stms | the number of declaration and executable statements in the methods of a class |
| Nmpub | number of public methods implemented in a class |
| NMNpub | number of non-public methods implemented in a class |
| NIM | Number of Instance Methods |
| NCM | Number of Class Methods |
| NLM | Number of Local Methods |
| AvgSLOC | Average Source Lines of Code |
| Complexity Metrics | CDE | Class Definition Entropy |
| CIE | Class Implementation Entropy |
| WMC | Weighted Method Per Class |
| SDMC | Standard Deviation Method Complexity |
| AvgWMC | Average Weight Method Complexity |
| CCMax | Maximum cyclomatic complexity of a single method of a class |
| NTM | Number of Trivial Methods |
| Coupling Metrics | CBO | Coupling Between Object |
| DAC | Data Abstraction Coupling: Type is the number of attributes of other classes. |
| DACquote | Data Abstraction Coupling: Type is the number of other classes. |
| ICP | Information-flow-based Coupling |
| IHICP | Information-flow-based inheritance Coupling |
| NIHICP | Information-flow-based non-inheritance Coupling |
| Inheritance Metrics | NOC | Number Of Child Classes |
| NOP | Number Of Parent Classes |
| DIT | Depth of Inheritance Tree |
| AID | Average Inheritance Depth of a class |
| CLD | Class-to-Leaf Depth |
| NOD | Number Of Descendants |
| NOA | Number Of Ancestors |
| NMO | Number of Methods Overridden |
| NMI | Number of Methods Inherited |
| NMA | Number Of Methods Added |
| SIX | Specialization IndeX = NMO \* DIT / (NMO + NMA + NMI) |
| PII | Pure Inheritance Index. |
| SPA | static polymorphism in ancestors |
| SPD | static polymorphism in descendants |
| DPA | dynamic polymorphism in ancestors |
| DPD | dynamic polymorphism in descendants |
| SP | static polymorphism in inheritance relations |
| DP | dynamic polymorphism in inheritance relations |

# **Appendix C. Reasoning process of AP and RR formulas of a random model**

Let *perf*(NC) be the performance of NC and *perf*(CC) be the performance of CC. It is more important for practitioners to evaluate *perfGain*, the relative performance of a model with respect to *random* [7]. In our context, *perfGain*(*NC*) = *perf*(*NC*) − *perf*(*random*) and *perfGain*(*CC*) = *perf*(*CC*) − *perf*(*random*). Therefore, in this study, we also employ the absolute value of *pgr* (performance gain ratio) to evaluate the influence of inconsistent labels in a data set on prediction performance:

Assume that a test set *T* consists of *N* instances, in which *n*1 are defective. For a random model *random*, we calculate AP and RR as:

First, the reasoning process of AP formula is as follows. We consider the calculation of AP as the sum of contribution of each position *i* (1≤ *i*≤ *N*) with defective modules. For each *i*, assuming that there are defective modules at the current position *i*, the remaining *n*1-1 defective modules need to be placed on both sides of position *i*, that is, one part of the defective modules should be placed on 1 to *i*-1, and the other part should be placed on *i*+1 to *N*. At this time, (1) the probability that there is a defective module at the current position *i* is ; (2) the probability that the defective module at the current position *i* is the *k*th defective module (i.e., *k*-1 (*k* ≤ *i*) defective modules are placed on 1 to *i*-1, and *n*1-*k* defective modules are placed on *i*+1 to *N*) is ; (3) the contribution value of position *i* with defective modules is *k*/*i*. Therefore, the total AP contribution value of upstream defective module on current position *i* is calculated as:

Further, we can derive the formula of AP as follows:

Second, the reasoning process of RR formula is as follows. RR can be expressed as:

The first summation term indicates that if the first defective module appears in the first position, then the RR value is 1, and the probability of occurrence of this event is . Because the total possibility is (regarding the defective modules as repeated events), the possibility of the current situation is to select *n*1-1 positions from the subsequent *N*-1 positions to place the defective modules, that is, the total possibility is . By analogy, the first defective module can appear at most in the *N*-*n*1+1 position. After sorting out the above formulas, the results are as follows:

The advantage of using PGR indicator is that the influence of the difficulty of the problem itself is considered. If the performance of a model *m* is significantly increased compared with the random model, it should be considered that model *m* has a good effect. By eliminating the impact of the difficulty of the problem itself, it will be more fair and meaningful to observe or compare the change magnitude of evaluation indicators on different data points (pairs of training set and test set built with different versions of different projects).

**References**

1. M. Jureczko, D. Spinellis. Using object-oriented design metrics to predict software defects. RELCOMEX 2010: 69-81.
2. S. Yatish, J. Jiarpakdee, P. Thongtanunam, C. Tantithamthavorn. Mining software defects: should we consider affected releases? ICSE 2019: 654-665.
3. T. Zimmermann, R. Premraj, A. Zeller. Predicting defects for Eclipse. PROMISE 2007, article no 9: 1-7.
4. L.C. Briand, J. Wust, J.W. Daly, D.V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. Journal of Systems and Software, [51(3)](https://dblp.uni-trier.de/db/journals/jss/jss51.html#BriandWDP00), 2000: 245-273.
5. Y. Zhou, B. Xu, H. Leung, L. Chen. An in-depth study of the potentially confounding effect of class size in fault prediction. ACM Transactions on Software Engineering and Methodology, 23(1), 2014: 1-51.
6. Y. Zhou, H.K.N Leung, B. Xu. Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. IEEE Transactions on Software Engineering, 35(5), 2009: 607-623.
7. S. Lai. Word and document embeddings based on neural network approaches. arXiv: 1611.05962, 2016. http://pdfs.semanticscholar.org/3e23/659a2de955727d548c9f26a044a59c3eb1b0.pdf (in Chinese).

1. https://scitools.com [↑](#footnote-ref-1)
2. https://www.python.org [↑](#footnote-ref-2)