Learning to Log: Helping Developers Make Informed Logging Decisions

[Supplementary Report]

Jieming Zhu[†], Pinjia He[†], Qiang Fu[‡], Hongyu Zhang[‡],
Michael R. Lyu[†], Dongmei Zhang[‡]

[†]Department of Computer Science and Engineering
The Chinese University of Hong Kong, Hong Kong
{jmzhu, pjhe, lyu}@cse.cuhk.edu.hk

[‡]Microsoft Research Asia
Beijing, China
{qifu, honzhang, dongmeiz}@microsoft.com

APPENDIX

This report is provided as supplementary material for our ICSE'15 paper [9]. The content comprises three parts: observations, approach, and additional evaluation results. Note that we use **[Section xxx]** along with the section title to point out the place where the corresponding supplementary content is referenced in the main paper.

The paper, supplementary report, source code, and questionnaire for user study are all available at:

http://cuhk-cse.github.io/LogAdvisor

A. OBSERVATIONS

A.1 Logging Statistics [Section II-B1]

Table 1 provides the detailed logging statistics of different types of software entities. We find that about 17.4% of the source files, 14.4% of the classes, 7.7% of the methods, and 25.3% of the catch blocks are logged, respectively.

A.2 Code Examples [Section II-B3]

There are many situations for not logging an exception. Some exceptions have no critical impact on the normal operation of the whole system, some are resolved by recovery actions such as retry or walk-around, and some others are explicitly reported (e.g., by setting flags, re-throwing, or returning special values) to the subsequent or upper-level operations (e.g., caller method) to handle. Figure 1 provides some of the unlogged exception snippets. In Example 1, when an "UnauthorizedAccessException" exception is caught, the program sets the flag "userHasRights" to *false* and then directs the execution to the subsequent logic branch. In Example 2, the caught exception is re-thrown to its caller as a "TestFailedException", then its caller would determine whether to log the exception at a higher level. Example 3 illustrates an example of an exception recovered by retry. When the program fails to create a *Uri* object, it uses the default scheme *http* to create the object again.

A.3 Logging Ratios [Section II-B4]

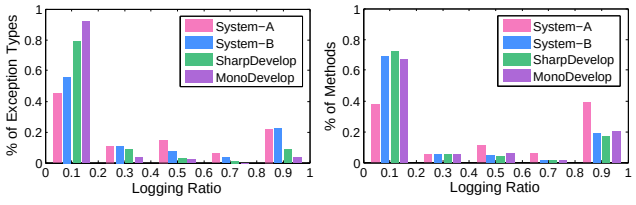
To understand the potential relations between exception types (or methods) and logging decisions of developers. We study the logging ratios of each exception type and each method. The logging ratio, with respect to an exception type (or an invoked method), is measured by the number of logged exceptions divided by the number of all the exception snippets within this exception type (or containing this method). Figure 2 illustrates the bar plot of the distribution of exception types/methods across the corresponding logging ratios. The results show that a significant proportion of exception types (82%) and methods (86%) have either

```
/* Example 1: An exception used to determine logic branch*/
void AccountConfig(MONOAccount user, string propertyName) {
    ...
    bool userHasRights = true;
    try {
        user.DeleteAccountProperty(propertyName);
    }
    catch (UnauthorizedAccessException) {
        userHasRights = false;
    }
    if (userHasRights) {
        ...
    }
}

/* Example 2: An exception re-thrown */
try {
    Type t = Type.GetTypeFromID(guid);
    object instance = Activator.CreateInstance(t);
}
catch (Exception e) {
    throw new TestFailedException("Fail to create Com interface.\t: "
        + Tester.GetExceptionDetails(e));
}

/* Example 3: An exception recovered by retrying */
void DWAppOverride(...) {
   ...
   Uri UriNew = null;
   try {
      UriNew = new Uri(wApp);
   }
   catch (UriObjectFormatException) {
    // Assume http is the scheme and the URL param is the machine name
      if (UriNew == null) {
         try {
            UriNew = new Uri("http://" + wApp);
         }
      }
   }
}
```

Figure 1: Code Examples of NOT Logging from [3]



(a) Exception Types (b) Methods

Figure 2: Distribution of Exception Types/Methods

Table 1: Logging Statistics

Software Systems	Source Files		Classes		Methods		Catch Blocks	
	Total	#Logged	Total	#Logged	Total	#Logged	Total	#Logged
System-A	4,706	2,027 (43.1%)	10,349	2,788 (26.9%)	57,578	9,128 (15.9%)	7,580	3,320 (43.8%)
System-B	50,036	9,380 (18.7%)	62,954	10,098 (16.0%)	324,167	30,988 (9.6%)	25,441	5,307 (20.9%)
SharpDevelop	8,853	666 (7.5%)	10,869	704 (6.5%)	69,108	1,618 (2.3%)	1,346	252 (18.7%)
MonoDevelop	11,567	999 (8.6%)	18,724	1,177 (6.3%)	121,982	2,390 (2.0%)	4,041	771 (19.1%)
Total	75.2K	13.1K (17.4%)	102.9K	14.8K (14.4%)	572.8K	44.1K (7.7%)	38.4K	9.7K (25.3%)

Algorithm 1: Methods extraction algorithm

```
/* All the data structures are used as with Roslyn API and
   system API in C#.                                          */
Input: An abstract syntax tree: SyntaxTree syntaxTree, a
       syntax node of the focused snippet (exception snippet or
       return-value-check snippet): SyntaxNode
       focusedSnippet, the maximal levels to trace back: int
       maxLevel
Output: A list of methods: methodList
1  List<String> methodList ← null;
2  Identify the containing method of focusedSnippet;
3  Get fullMethodName of the containing method, and add it into
   methodList;          /* fullMethodName is comprised of its
   namespace, class name, and method name */
4  Traverse syntaxTree to get allMethodDeclarations;
5  Queue<Tuple<SyntaxNode, int> > methodQueue;
6  Add (focusedSnippet, 0) into methodQueue;      /* trace back
   level=0 */
7  while methodQueue is not empty do
8      Take an element (methodNode, level) out of methodQueue;
9      List<SyntaxNode> invocationList ← null;
10     Add all the methods invoked by methodNode to
       invocationList; /* Skip the methods enclosed in any other
       focused snippet */
11     foreach invokedMethod in invocationList do
12         Get fullMethodName of invokedMethod;
13         if methodList does not contain fullMethodName then
           /* Visit the method that has not been recorded in
           methodList */
14             Add fullMethodName into methodList;
15             if level > maxLevel then          /* Guarantee the
               maximal trace back level to maxLevel */
16                 continue;
17             if fullMethodName.StartWith("System") then
               /* Skip system methods */
18                 continue;
19             if allMethodDeclarations contains
               fullMethodName then /* Add invokedMethod into
               the queue if it is user-defined */
20                 Add (invokedMethod, level + 1) into
                   methodQueue;
21 return methodList;
```

high (> 80%) or low (< 20%) logging ratios, which suggests their high correlations (*i.e.*, either positive or negative correlations) with logging decisions of developers.

B. APPROACH

B.1 Methods Extraction [Section III-B1]

There are two types of focused snippets in our study: exception snippet and return-value-check snippet. Exception snippet corresponds to a *try-catch* block, while return-value-check snippet denotes the *if* block that checks the return value of a function call for contingency (e.g., by using special values such as *false/empty/-1/null*). Algorithm 1 provides the description of our methods extraction process in detail. Specifically, line 1∼3 describe that the containing method of $focusedSnippet$ is identified and its method name

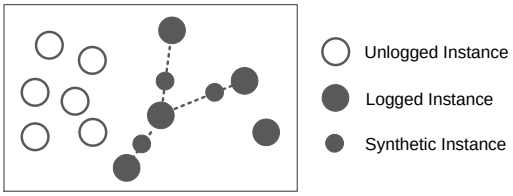


Figure 3: Illustration of Imbalance Handling

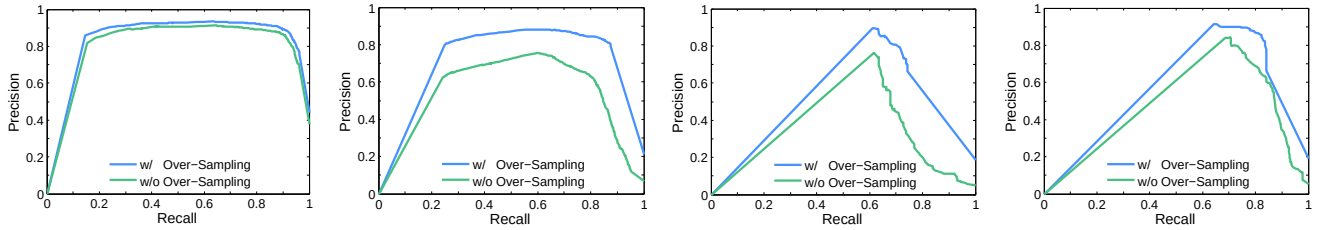
is added to *methodList*. In line 4, all of the syntax nodes with method declaration type are obtained and stored into a list, $allMethodDeclarations$. We put the syntax node $focusedSnippet$ (with its level number) into a queue in line 5∼6. Then line 7∼20 describe a while loop to extract all of the invoked methods. In line 8∼10, a *methodNode* is taken out from the queue, and then extract the directly-invoked methods. Especially, in line 10, any other focused snippet in *methodNode* will be skipped during extraction, and therefore, the methods enclosed in other focused snippets will be ignored for the input $focusedSnippet$. In line 11∼20, we examine each of the invoked methods, and check whether a method has already been recorded or it is a system method or external API method (*i.e.*, not contained in $allMethodDeclarations$). If yes, we will skip to next iteration; otherwise, we put the syntax node of its caller method into the queue for the subsequent extraction. In particular, we set a parameter *maxLevel*, which can control the maximal levels we trace back to extract the invoked methods. For example, in Figure 3 of our main paper, the levels of Method1, Method3, Method6 are 0, 1, and 2, respectively. In our experiment, *maxLevel* is set to 5 by default, which has shown good results in performance evaluation. At last, a list of methods (*methodList*) related to the context of $focusedSnippet$ is returned in line 21.

B.2 Textual Features Extraction [Section III-B2]

We extract the textual features using the bag-of-words model, through a set of widely-used text processing operations: **1) Tokenization**. In our case, we exploit the common use of camel case and special characters (*e.g.*, "_") in naming to split all the text into terms. This tokenization approach is simple yet effective, which works well in our studied projects as well as some other related work [1, 7]. Then we convert all the uppercase characters into lowercase ones. **2) Stemming**. This step is to identify the ground form of each term, where the affixes and other lexical components are removed. For example, "methods" will be stemmed into "method". **3) Stop words removal**. To get rid of some useless and noisy terms, we use a list of stop words to filter them, where terms like "the", "that" are discarded. Besides, we remove all the terms with a length smaller than 3, such as "i", "on", "is", etc. **4) Term weighting**. We use the stan-

Table 2: Prediction Accuracy Results (*w.r.t.* Precision, Recall, and F-Score)

Approaches	System-A			System-B			SharpDevelop			MonoDevelop		
	Prec.	Recall	F-Score	Prec.	Recall	F-Score	Prec.	Recall	F-Score	Prec.	Recall	F-Score
Random	0.437	0.499	0.466	0.209	0.500	0.295	0.188	0.503	0.274	0.191	0.499	0.276
ErrLog	0.438	1	0.609	0.209	1	0.346	0.187	1	0.315	0.191	1	0.321
Exception Type	0.639	0.784	0.705	0.307	0.796	0.443	0.328	0.853	0.474	0.413	0.903	0.567
Methods	0.501	0.907	0.646	0.604	0.459	0.522	0.519	0.278	0.362	0.529	0.450	0.486
Textual Features	0.738	0.698	0.718	0.656	0.713	0.683	0.588	0.532	0.558	0.606	0.634	0.620
Syntactic Features	0.816	0.933	0.871	0.695	0.783	0.736	0.908	0.587	0.713	0.853	0.671	0.751
LogAdvisor	0.871	0.921	0.895	0.810	0.869	0.838	0.793	0.714	0.752	0.856	0.824	0.839



(a) System-A (w = 1.28) (b) System-B (w = 3.7) (c) SharpDevelop (w = 4.34) (d) MonoDevelop (w = 4.24)

Figure 4: Impact of Data Imbalance Handling on Prediction Accuracy

dard TF-IDF [5] weighting scheme to assign values to each term. Note that the processing steps 2∼4 are performed using Weka [4].

B.3 Imbalance Handling [Section III-C]

One critical challenge faced by our model is the high imbalance of data. For our studied systems, only 25.3% of exception snippets and 9.3% of return-value-check snippets are logged. This reveals an imbalance ratio up to 48.8 : 1 between unlogged (majority) instances and logged (minority) instances. Data imbalance is a common issue in real-world machine learning applications, and as we will show in Appendix C.2, it can heavily influence the prediction performance.

In our study, we employ a state-of-the-art approach, SMOTE [2], to balance the data by creating synthetic instances from the minority class. SMOTE first identifies the k-nearest minority neighbors (measured by the cosine similarity between their feature vectors) for each examined minority instance, and then randomly generates synthetic instances between the instance and its neighbour. As illustrated in Figure 3, three synthetic instances are generated between the examined instance and its 3-nearest neighbors. The value k and the number of synthetic instances to generate are set as user input. In our experiments, we employ the Weka [4] implementation of SMOTE.

C. ADDITIONAL EVALUATION RESULTS

C.1 Prediction Accuracy [Section IV-B]

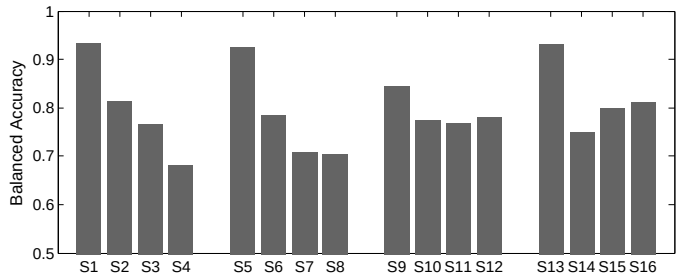
Table 2 provides the prediction accuracy results for exception snippets, regarding precision, recall, and F-score. Under a random setting, the experiments are run for 100 times and the average values are reported. As we can observe, for balanced dataset like System-A (43.8% positives), the F-Score is approximately 0.5. For other datasets whose data are imbalanced, the F-Score is much lower. For ErrLog, because of its conservative logging, it will logs each exception instance, and thus get a recall of 100%. However, its precision is heavily influenced by the data imbalance, leading to low F-Score close to the random approach. Our LogAdvisor, by combining all useful features, achieves a high F-Score with 0.752∼0.895.

C.2 The Effect of Imbalance Handling

The data we collected from our studied software systems are imbalanced between logged instances and unlogged instances. To study the impact of data imbalance on prediction accuracy, we apply the state-of-the-art imbalance handling approach (SMOTE) described in Appendix B.3 to balance the training data and evaluate the performance improvement. Figure 4 provides the precision-recall plots of the prediction results on exception snippets, which present the trade-off between precision and recall. The weights used in our experiments are also shown in the figure. For example, we create sythetic logged instances of System-B by a weight of 3.7. We can see that System-B, SharpDevelop and MonoDevelop have large improvement on prediction accuracy, while the improvement on System-A is small, because the data of System-A is more balanced compared to the others. The results indicate that our imbalance handling approach is helpful at improving the prediction accuracy.

C.3 Cross-Project Evaluation

Figure 5 presents the cross-project evaluation results with comparison to the within-project evaluation results. The corresponding experimental settings are shown in the right panel of the figure, where we use one project for training and one project for testing. Especially for within-project evaluation, we use 10-fold cross evaluation within the same project to evaluate the performance. Taking System-A as an example, we obtain a balanced accuracy of 93.4% for within-project evaluation (S1), while achieving 81.5%, 76.7% , and 68.0% balanced accuracy by using System-B (S2), SharpDevelop (S3), and MonoDevelop (S4) as training project respectively. Overall, the results indicate that the performance of cross-project learning is largely degraded compared with within-project learning. The reason is that different projects may follow different logging practices, and some



Cross-Project Learning Settings:

S1: System-A → System-A	S9: SharpDev → SharpDev
S2: System-B → System-A	S10: SystemA → SharpDev
S3: SharpDev → System-A	S11: System-B → SharpDev
S4: MonoDev → System-A	S12: MonoDev → SharpDev
S5: System-B → System-B	S13: MonoDev → MonoDev
S6: System-A → System-B	S14: System-A → MonoDev
S7: SharpDev → System-B	S15: System-B → MonoDev
S8: MonoDev → System-B	S16: SharpDev → MonoDev

Figure 5: Cross-Project Evaluation Results (Training Project → Testing Project)

Table 3: Accuracy on Golden Set

Focused Code Snippets	SharpDevelop	MonoDevelop
Exception Snippets	0.667	0.902
Return-value-check Snippets	0.789	0.751
Overall	0.750	0.854

project-specific knowledge (e.g., domain exceptions and methods) are challenging to adapt to other projects. Similar to our intuition, we find that the cross-project evaluation results among similar projects (*e.g.*, System-A and System-B, SharpDevelop and MonoDevelop) are slightly better than the results among dissimilar projects (*e.g.*, System-A and SharpDevelop, System-B and MonoDevelop). Furthermore, these results can serve as a baseline for further improvement by exploring other sophisticated techniques, such as transfer learning [6].

C.4 Evaluation on Golden Set

Due to the lack of "ground truth" on optimal logging, there is no guarantee about the logging quality of the collected data, even though we employ several mature software systems as our subjects. The logging behaviors of developers can be ad-hoc. In some cases, developers perform logging as afterthoughts, for example, after a failure happens and logs are needed. Logging statements may be added, modified, and even deleted with the evolution of systems, as reported in [8].

To extract the "golden practice" of logging, we examine the revision histories of the two open-source projects, and find out the logging statements that are added or modified along with patches. We use these logging statements as the "golden set" for evaluation, because they are likely good representatives of useful logging instances. Specifically, we collect 75 such logging instances in SharpDevelop and 135 such logging instances in MonoDevelop, including a total of 116 exception snippets and 94 return-value-check snippets. We utilize this golden set as the testing data to evaluate the performance of *LogAdvisor*. Table 3 show the accuracy results, with an overall accuracy ranging from 75%∼85%, which denotes the percentage of logging statements that are covered by *LogAdvisor*.

D. REFERENCES

[1] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proc. of ACM FSE*, pages 157–166, 2010.

[2] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, 2002.

[3] 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 *Proc. of ACM/IEEE ICSE*, 2014.

[4] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[5] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[6] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proc. of ACM/IEEE ICSE*, pages 382–391, 2013.

[7] R. Saha, M. Lease, S. Khurshid, and D. Perry. Improving bug localization using structured information retrieval. In *Proc. of IEEE/ACM ASE*, pages 345–355, 2013.

[8] D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *Proc. of ACM/IEEE ICSE*, pages 102–112, 2012.

[9] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to log: Helping developers make informed logging decisions. In *Proc. of ACM/IEEE ICSE*, 2015.