2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Learning to Log: Helping Developers Make Informed Logging Decisions

Jieming Zhu[†], Pinjia He[†], Qiang Fu[§], Hongyu Zhang[‡], Michael R. Lyu[†], Dongmei Zhang[‡]
[†]Shenzhen Research Institute, The Chinese University of Hong Kong, Shenzhen, China
[†]Ministry of Education Key Laboratory of High Confidence Software Technologies (CUHK Sub-Lab)
[†]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong
[§]Microsoft, Washington DC, USA [‡]Microsoft Research, Beijing, China
[†]{*jmzhu, pjhe, lyu*}*@cse.cuhk.edu.hk* [§]*qifu@microsoft.com* [‡]{*honzhang, dongmeiz*}*@microsoft.com*

***Abstract*—Logging is a common programming practice of practical importance to collect system runtime information for postmortem analysis. Strategic logging placement is desired to cover necessary runtime information without incurring unintended consequences (*e.g.*, performance overhead, trivial logs). However, in current practice, there is a lack of rigorous specifications for developers to govern their logging behaviours. Logging has become an important yet tough decision which mostly depends on the domain knowledge of developers. To reduce the effort on making logging decisions, in this paper, we propose a "learning to log" framework, which aims to provide informative guidance on logging during development. As a proof of concept, we provide the design and implementation of a logging suggestion tool, *LogAdvisor*, which automatically learns the common logging practices on *where to log* from existing logging instances and further leverages them for actionable suggestions to developers. Specifically, we identify the important factors for determining *where to log* and extract them as structural features, textual features, and syntactic features. Then, by applying machine learning techniques (*e.g.*, feature selection and classifier learning) and noise handling techniques, we achieve high accuracy of logging suggestions. We evaluate *LogAdvisor* on two industrial software systems from Microsoft and two open-source software systems from GitHub (totally 19.1M LOC and 100.6K logging statements). The encouraging experimental results, as well as a user study, demonstrate the feasibility and effectiveness of our logging suggestion tool. We believe our work can serve as an important first step towards the goal of "learning to log".**

I. Introduction

Logging is a common programming practice in software development, typically issued by inserting logging statements (*e.g.*, *printf*(), *Console.Writeline*()) in source code. As in-house debugging tools (*e.g.*, debugger), all too often, are inapplicable in production settings, logging has become a principal way to record the key runtime information (*e.g.*, states, events) of software systems into logs for postmortem analysis. To facilitate such log analysis, the underlying logging that directly determines the quality of collected logs is a matter of vital importance.

Due to the criticality of logging, it would be bad to log too little, which may miss the runtime information necessary for postmortem analysis. For example, systems may fail in the field without any evidence from logs, thus significantly increasing the difficulty in failure diagnosis [43]. However, it is also not the case that the more logging, the better. As the practical experiences reported in [4], [13], logging too much can yield many problems too. First, logging means more code, which takes time to write and maintain. Furthermore, logging consumes additional system resources (*e.g.*, CPU and I/O) and can have noticeable performance impact on system operation, for example, when writing thousands of lines to a log file per second [4]. Most importantly, excessive logging can produce numerous trivial and useless logs that eventually mask the truly important information, thus making it difficult to locate the real issue [13]. As a result, strategic logging placement is desired to record runtime information of interest yet not causing unintended consequences.

To achieve so, developers need to make informed logging decisions. However, in our previous developer survey [26], we found that even in a leading software company like Microsoft, it is difficult to find rigorous (i.e., thorough and complete) specifications for developers to guide their logging behaviors. Although we found a number of online blog posts (*e.g.*, [1], [2], [3], [4], [9], [11]) sharing best logging practices of developers with deep domain expertise, they are usually high-level and application-specific guidelines. Even with logging frameworks (*e.g.*, Microsoft's ULS [12] and Apache's log4net) provided, developers still need to make their own decisions on where to log and what to log, which in most cases depend on their own domain knowledge. Therefore, logging has become an important yet tough decision during development, especially for new developers without much domain expertise.

Current research has seldom focused on studying how to help developers make such logging decisions. To bridge this gap, in this paper, we propose a "learning to log" framework, which aims to automatically learn the common logging "rules" (*e.g.*, where to log, what to log) from existing logging instances, and further leverage them to provide informative guidance for new development. Motivated by our observations (detailed in Section II-B), we extract a set of contextual features from the source code to construct a learning model for predicting where to log. Our logging suggestion tool built on this model, named *LogAdvisor*, can thus provide actionable suggestions for developers and reduce their effort on logging. As an initial step towards "learning to log", this paper focuses on studying where to log (or more specifically, whether to log a focused code snippet), while leaving other aspects (such as what to log) of this research for future work.

We have conducted both within-project evaluation and

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.60

TABLE I
SUMMARY OF THE STUDIED SOFTWARE SYSTEMS (SOME ENTRIES ARE ANONYMIZED FOR CONFIDENTIALITY)

Software Systems	Start Time	Description	Version	LOC	#Logging		#Commits		
					Logging Statements	LOC of Logging	Total	#Commits with Logging	#Patches with Logging
System-A	–	Online service	–	2.5M	23,624	77,945	–	–	–
System-B	–	Online service	–	12.7M	69,057	240,395	–	–	–
SharpDevelop	2001	.NET platform IDE	5.0.2	1.4M	2,896	9,261	13,886	4,593 (33.1%)	724 (15.8%)
MonoDevelop	2003	Cross-platform IDE	4.3.3	2.5M	4,996	13,043	29,357	9,437 (32.1%)	1,157 (12.3%)
Total				19.1M	100.6K	327.6K	43.2K	14.0K (32.4%)	1.9K (13.6%)

cross-project evaluation on *LogAdvisor* using two industrial software systems from Microsoft and two open-source software systems from GitHub. Additionally, a user study is performed to evaluate whether the suggestions provided by *LogAdvisor* can help developers in practice. The comprehensive evaluation results have demonstrated the feasibility and effectiveness of our logging suggestion tool. For ease of reproducing and applying our approach to future research, we release our source code and detailed study materials (*e.g.*, data, questionnaire) on our project page[1].

The rest of this paper is organized as follows. Section II introduces our studied software systems and the motivation of this work. Section III provides the overview and the detailed techniques of learning to log. Section IV reports the evaluation results, and Section V presents our user study. We discuss the limitations in Section VI and the related work in Section VII. Finally, we conclude this paper in Section VIII.

II. OBSERVATIONS AND MOTIVATION

In this section, we first introduce the subject software systems under study. Then we provide some key observations on logging practices and present the motivation of our study.

A. Subject Software Systems

In our study, we investigate four large software systems, including two industrial systems from Microsoft (denoted as System-A and System-B for confidentiality) and two open-source systems from GitHub (SharpDevelop and MonoDevelop). Each of these systems contains millions of lines of code (LOC) written in C# language. Table I provides the summary information of our studied software systems. Both industrial systems are online service systems developed by Microsoft, serving a huge number of users globally. These two systems were also used as subjects in our empirical study on logging practices [26]. To allow for reproducing and applying our approach to future research, we choose another two open-source software systems as subjects. They are two IDE projects: SharpDevelop (supporting .NET platform) and MonoDevelop (supporting cross-platform development). Both of them are selected due to their popularity (well-known C# projects), active updates (10000+ commits) and long history of development (10+ years).

Our targeted systems are supposed to have reasonably good logging implementation, because the produced logs by these systems have mostly met the requirements of usage analysis, troubleshooting, and operating, after undergoing more than 10 years of evolution. This is especially true for the industrial software systems, because each of them is implemented by a group of experienced developers at Microsoft, where the code quality has been strictly controlled. Consequently, the source code of these software systems is well suited for our study on logging practices. All of our code analysis is conducted based on an open-source C# code analysis tool, *Roslyn* [10]. By using Roslyn, we can perform both syntax analysis and semantic analysis on the source code.

B. Observations

1) ***Pervasiveness of logging****:* Logging is pervasively used in software development. As shown in Table I, our studied systems have a total of 100.6K logging statements (containing 327.6K lines of logging code) out of 19.1M LOC. That is, there is a line of logging code in every 58 LOC, as similarly reported in [42], [44]. By drilling down according to the type 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. In addition, by examining the revision histories of the systems, we find that, on average, 32.4% of the commits involve logging modifications, and further, 13.6% of them are modified along with patches[2]. Both its pervasive existence and active modifications reveal that logging plays an indispensable role in software development and maintenance.

2) ***Where to log****:* The logging decisions can resolve to where to log and what to log. *Where to log* determines the locations to place logging statements, while *what to log* denotes the contents recorded by these logging statements. Whereas the goal of "learning to log" is to handle them both, we study where to log in this paper. Our previous empirical study on where developers log [26] has shown that there are some typical categories of logging strategies for recording error sites and execution paths. Error sites indicate some unexpected situations where the system potentially runs into a problem, including exceptions and function-return errors. As two typical ways for error reporting, exception mechanisms are widely used in modern programming languages (*e.g.*, C#) to handle abnormal situations, and function-return errors indicate the situation where an unexpected value (*e.g.*, -1/null/false/empty)

[1] *http://cuhk-cse.github.io/LogAdvisor*

[2] We identify patches by searching commit logs for keywords such as "fix", "bug", "crash" or issue ID like "#42233", the same as in [30].

TABLE II
LOGGING STATISTICS

Software Systems	Exception Snippets			Return-value-check Snippets		
	#Exception Types	#Instances	#Logged Instances	#Call Types	#Instances	#Logged Instances
System-A	188	7,580	3,320 (43.8%)	5,400	43,443	5,127 (13.5%)
System-B	1,657	25,441	5,307 (20.9%)	21,624	131,870	15,081 (11.4%)
SharpDevelop	106	1,346	252 (18.7%)	3,221	17,937	476 (2.7%)
MonoDevelop	220	4,041	771 (19.1%)	5,821	37,360	750 (2.0%)
Total		38.4K	9.7K (25.3%)		230.6K	21.4K (9.3%)

```
/* A code example taken from MonoDevelop (v.4.3.3), at file:
 * main\external\mono-tools\gendarme\console\Settings.cs,
 * line: 116. Some lines are omitted for ease of presentation.
 */
private int LoadRulesFromAssembly (string assembly, ...)
{
    Assembly a = null;
    try {
        AssemblyName aname = AssemblyName.GetAssemblyName(
            Path.GetFullPath (assembly));
        a = Assembly.Load (aname);
    }
    catch (FileNotFoundException) {
        Console.Error.WriteLine ("Could not load rules
            from assembly '{0}'.", assembly);
        return 0;
    }
    ...
}
```

(a) Exception Snippet

```
/* A code example taken from MonoDevelop (v.4.3.3), at file:
 * main\src\core\MonoDevelop.Ide\MonoDevelop.Ide\ImageService.cs,
 * line: 302.
 */
// Converts an image spec into a real stock icon id
string stockid = GetStockIdForImageSpec (name, size);
if (string.IsNullOrEmpty (stockid)) {
    LoggingService.LogWarning ("Can't get stock id for " +  name
        + " : " + Environment.NewLine + Environment.StackTrace);
    return CreateColorBlock ("#FF0000", size);
}
```

(b) Return-value-check Snippet

Structural features:
Exception Type: 0.39 (System.IO.FileNotFoundException)
Containing method: Gendarme.Settings.LoadRulesFromAssembly
Invoked methods: System.IO.Path.GetFullPath,
System.Reflection.AssemblyName.GetAssemblyName,
System.Reflection.Assembly.Load

Textual features:
Exception type/methods: gendarme(1), settings(1), load(2),
rules(1), from(1), assembly(4), system(3), io(1), path(1),
get(2), full, path(1), reflection(2), name(2), file(1), not(1),
found(1), exception(1)
Vaiables: aname(2), assembly(1), a(1) **Comments**: N/A

Syntactic features:

SetLogicFlag: 0	EmptyCatchBlock: 0	
Throw: 0	OtherOperation: 0	
Return: 1	LOC: 3	
RecoverFlag: 0	NumOfMethods: 3	**Label:** Logged

(c) Extracted Contextual Features from Exception Snippet in (a)

Fig. 1. Code Examples and Contextual Features

is returned from a function call. We denote their associated code snippets as exception snippets and return-value-check snippets respectively (as examples shown in Fig. 1(a)(b)). They are the two most common logging strategies [26] and thus become our *focused code snippets*. Although recording information of execution path is crucial for tracking down root causes from the error sites, existing studies (*e.g.*, control-flow instrumentation [23], [35]) have been conducted to achieve this goal, which are orthogonal to our work.

*3) **Why not log everything**:* Log information is immensely useful in maintaining software systems. So the question "why not log everything?" (*e.g.*, StackOverflow questions [5], [7]) does sound reasonable. Yuan et al. also proposed conservative logging (ErrLog) [43], which logs all the generic exceptions (*e.g.*, exceptions and function-return errors) for failure diagnosis. However, as the logging statistics shown in Table II, we observed that, in our studied systems, the majority of exceptions (74.7% on average) and return-value-check snippets (90.7% on average) are actually not logged. To understand this fact, we posted our questions on "why not log all exceptions?" to the mail lists and websites of MonoDevelop [14], SharpDevelp [15] and StackOverflow [7], and received some valuable feedback from the developers. According to their feedback, "logging all exceptions would produce a ton of garbage and make it hard to zoom in on real issues", which conforms with our argument (not logging too much). There are many reasons for not logging an exception. Some exceptions are "expected in normal operation", while some others are satisfactorily handled or "recovered without impacting the user". In a word, not all exceptions are "unexpected" (or errors) [4]. Strategic logging needs to "determine whether or not an exception is worth reporting" [6].

*4) **Logging decision and the context**:* To understand this tradeoff in practice, we attempt to study how developers make decisions on whether to log a focused code snippet. Fig. 1(a) presents a real-world example of an exception snippet (*i.e.*, try-catch block). The operations enclosed in the try block attempt to load the rules from the input string, "*assembly*". If this assembly file cannot be found, an exception with type of "FileNotFoundException" will occur and then be caught by the catch block. Here, the exception has been logged with an error message by "*Console.Error.WriteLine*()". Intuitively, from this example, we can see that the logging decision is highly dependent on the context of this code snippet, including the *exception type* (*e.g.*, FileNotFoundException), the invoked *methods* (*e.g.*, GetFullPath, GetAssemblyName, Load) in a try block, etc. The contextual information is crucial because each exception type generally denotes one specific type of exceptional conditions while the invoked methods indicate the functionality of operations. Driven by this intuition, we measure the logging ratio of each exception type and each method. Specifically, the logging ratio, with an exception type (or an invoked method) is measured by the number of logged exceptions divided by the number of all the exception snippets with this exception type (or containing this method). The results show that a significant portion of exception types (82%) and methods (86%) have either high ($> 80\%$) or low ($< 20\%$) logging ratios, which suggests their high correlations (*i.e.*,

either positive or negative correlations) with logging decisions of developers.

C. Motivation

With the ever growing scale and complexity of software systems, it is common that each developer is only responsible for a part of a system (*e.g.*, one or several components). Logging under this situation is notoriously challenging, because developers may not have full knowledge of the whole system. For example, in our user study (Section V), 68% of the participants have logging difficulties. However, there is a lack of rigorous specifications or tool support for developers to aid their logging decisions. Without a well-structured logging strategy, it is difficult for developers to know how to make informed logging decisions, and thus, quite often, the decisions are made based on their own domain knowledge (*e.g.*, understanding of system behaviours, logging experience). Such domain knowledge is seldom documented and it is also hard to do so, since the logging behaviours of developers may vary widely, not only from project to project, but also from developer to developer. Indeed, the pervasively-existing logging instances together can provide strong indication of the developers' domain knowledge embedded with their logging decisions. Thus, we intend to explore whether the logging decisions of developers, such as where to log, can be learnt automatically from these existing logging instances. If so, the constructed model can represent the common knowledge of logging and be further built into tool support to provide valuable suggestions (*e.g.*, whether to log an exception snippet) for developers. Such a tool can improve the logging quality as well as reduce the effort of developers. Following this motivation, we propose "learning to log".

III. Learning to Log

In this section, we present the overview as well as the detailed techniques of "learning to log".

A. Overview

Our goal, referred to as "learning to log", is to automatically learn the common logging practice as a machine learning model, and then leverage the model to guide developers to make logging decisions during new development. We further implement the proposed "learning to log" approach as a tool, *LogAdvisor*. Fig. 2 presents the overview of "learning to log", which can be described as the following steps:

1) ***Instances collection****:* As the first step, we need to extract data instances (focused code snippets) from our target projects. There are two types of frequently-logged code snippets: exception snippets and return-value-check snippets. As shown in Fig. 1(a) and Fig. 1(b), exception logging records the exception context (*e.g.*, exception message) after an exception is captured in the catch block, while return-value-check logging is used to log the situation where an unexpected value (*e.g.*, -1/null/false/empty) is returned from a function call. By employing Roslyn, we extract all these focused code snippets, and use them as training data to learn the logging practices of developers.

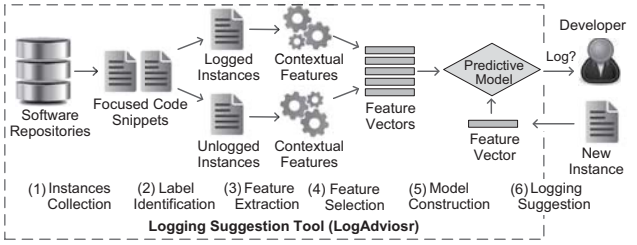


Fig. 2. The Overview of Learning to Log

2) ***Label identification****:* As a key step of preparing training data, each data instance (a code snippet) is labelled "logged" if it contains a logging statement; or "unlogged", otherwise. A logging statement denotes a statement that has an invocation to a logging method (*e.g.*, *Console.Writeline*()). We identify logging methods by searching some keywords in all method names, such as *log/logging*, *trace*, *write/writeline*, etc. The logging statement identification and labelling procedures are automatically performed based on Roslyn.

3) ***Feature extraction****:* In our study, we need to extract useful features (e.g., exception type) from the collected code snippets for making logging decisions, which is one of the most important steps to determine the performance of the prediction model. The details on feature extraction are described in Section III-B.

4) ***Feature selection****:* When there are too many features, some of them are likely redundant or irrelevant since they provide little useful information or even act as noises to degrade the prediction performance. Feature selection [28] is a key technique to remove such redundant or irrelevant features to enhance the prediction performance as well as shorten the training time.

5) ***Model training****:* Through feature extraction and selection, we can generate a corpus of feature vectors, where each denotes a vector of feature values from a data instance. With these feature vectors and their corresponding labels, we can apply a set of machine learning models (*e.g.*, Decision Tree [41]) to learn the common logging practice. In our study, we learn the decision on whether to log a focused code snippet as a classification model.

6) ***Logging suggestion****:* Through the above learning process, we can obtain a predictive model to perform accurate logging predictions. This predictive model can be trained offline and further be built into a logging support tool (namely *LogAdvisor*) to provide online logging suggestions for developers. For example, when a developer composes a new piece of code containing a *try-catch* block, *LogAdvisor* can detect and extract its feature vector in a transparent way. Then LogAdvisor can predict on whether to log, and provide a logging suggestion for the developer through IDE (*e.g.*, like the warning message). By using *LogAdvisor*, developers can make informed logging decisions.

The above learning workflow is generic and works similarly to many other machine learning applications in software engineering (*e.g.*, defect prediction [30], [34], [48]).

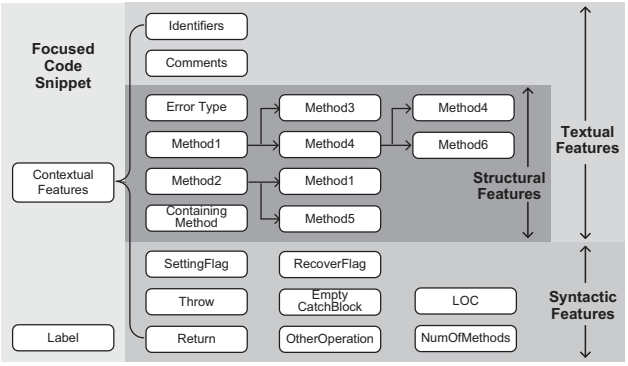


Fig. 3. Framework of Contextual Feature Extraction

B. Contextual Feature Extraction

Feature extraction lies in the core of "learning to log", because the quality of extracted features directly determines the performance of the model. The context information (*e.g.*, the functionality of operations, the impact of exceptions) of logging points are crucial for developers to make logging decisions. However, it is challenging to effectively extract such context information, because the target code snippet is usually short and linguistically sparse compared to natural language text. To address this issue, we propose a novel feature extraction framework, as illustrated in Fig. 3, which involves three types of features: structural features, textual features, and syntactic features.

1) ***Structural features****:* Source code has a well-defined structure. It is desired to leverage the structure information of source code to help extract context information. To achieve this goal, we extract two types of structural features: error type and associated methods.

Error Type: The error type, such as exception type or call type, can largely reveal the context of our focused code snippets, which is highly correlated with logging decisions of developers (as indicated in Section II-B4). For an *exception snippet*, the exception type generally denotes one specific type of exceptional conditions with informative semantic meanings, *e.g.*, "*FileNotFoundException*" in Fig. 1(a). For a *return-value-check snippet*, the call type is denoted as the prototype of the checked function, e.g., *string GetStockIdForImageSpec*(*string, int*) in Fig. 1(b), which indicates one specific type of potential function-return errors. Therefore, we extract error type as a key feature.

Each instance has a single error type, but there exist a wide variety of error types among the training data. We avoid directly using each error type as a feature dimension, which can lead to highly sparse and ineffective feature vectors. Instead, we construct only one feature dimension as the logging ratio of each error type, that is, the ratio of logged instances against all the instances within that error type. Fig. 1(c) presents an illustration of the contextual features extracted from the code example in Fig. 1(a). In this example, the "*FileNotFoundException*" type has a logging ratio of 39% regarding training instances in MonoDevelop, so we take the feature value of error type as 0.39.

Methods: The associated methods of a focused code snippet also provide indicative information to help understand the functionality of the operations. For example, we can figure out the intention of developers (*i.e.*, to load an assembly file) in the example of Fig. 1(a) according to the method names, including *LoadRulesFromAssembly*, *GetFullPath*, *GetAssemblyName*, and *Load*. Therefore, we extract these methods as important contextual features.

Specifically, there are two types of methods: the containing method and the invoked methods. The former is the method that contains the focused code snippet (*e.g.*, *LoadRulesFromAssembly* in Fig. 1(a)), while the latter includes all the methods that are invoked by the snippet. The operations can be seen as a sequence of API method invocations. Thus, instead of using only the methods within the code snippet, we also track the callee methods. Fig. 3 provides a prototype of our approach, where the arrows represent the invocation relationships between methods. For example, Method1 and Method2 are invoked by the focused code snippet, where Method1 invokes Method3 and Method4, and Method4 further invokes itself and Method6. The extraction of methods continues tracking down until the invoked method is a system API or external library API method (*e.g.*, *System.IO.Path.GetFullPath*) or until a certain number of levels has been attained. The extraction process is implemented as a breadth-first search (BFS) variant, where all the recorded (visited) methods will be skipped. In particular, all the logging methods are excluded in this process. Due to space limits, the details of the method extraction algorithm is provided in our supplementary report [8].

After extracting the list of associated methods, we obtain the full qualified name (*e.g.*, System.IO.Path.GetFullPath) of each method as a feature dimension, which contains namespace, class name, and its (short) method name. Fig. 1(c) provides an example for these features.

2) ***Textual features****:* Source code is also text. Using code as flat text has been widely employed in the field of mining software repositories, and its effectiveness are demonstrated and reported in tasks such as API mining [47], code example retrieval [17], etc. Driven by these encouraging results, we also employ the similar approach to extract textual features from source code text.

More specifically, we extract all the texts in the focused code snippet excluding method names, such as variables and types. Then we combine them with the extracted list of structural features (*i.e.*, error type and methods) as the full text. In contrast to extracting all the text directly, our approach not only excludes the text of logging methods, but also includes the names of the callee methods, the containing method, as well as their namespaces and classes. With such text, we can extract the textual features using the bag-of-words model through a set of widely-used text processing operations, including tokenization, stemming, stop words removal, and TF-IDF term weighting [41]. Since the use of these techniques in code processing has been carefully reported in [17], [18], [47], we omit the details here and refer the interested readers to our supplementary report [8]. In our study, these processing

steps are performed using Weka [29].

*3) **Syntactic features**:* As indicated in Section II-B3, there are many situations of not logging, even for typical error sites such as exceptions and function-return errors. Some potential errors 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.

To capture these contextual factors, we also extract some key syntactic features from each focused code snippet: 1) *SettingFlag*. We identify whether there is an assignment statement with an assigned value like -1/null/false/empty. 2) *Throw*. We identify whether there is a *throw* statement. 3) *Return*. We identify whether any special value (*e.g.*, -1/null/false/empty) is returned. 4) *RecoverFlag*. We check whether there is a new *try* statement inside. 5) *OtherOperation*. We check whether there is any other operations included except the above five ones. 6) *EmptyBlock*. We find that the developers sometimes catch and then do nothing. We thus identify whether the catch block is empty. Note that all these identification processes have excluded logging statements at the first place, and all these features have *Boolean* values. In addition, we employ the feature *LOC* to measure the lines of code in the code snippet, and the feature *NumOfMethods* to measure the number of the extracted methods. An example is shown in Fig. 1(c).

C. Feature selection

The above feature extraction process, however, can generate tens of thousands of features, due to the large vocabulary of methods and (textual) terms extracted from the data instances. These features further lead to high-dimensional (*e.g.*, 72K features in System-B) yet highly-sparse feature vectors, because most of the features are actually infrequent across all data instances. Furthermore, some of these features (*e.g.*, textual features parsed from some specific variable names) may be irrelevant and have negative impact on the performance of the predictive model.

In such a setting, we make use of a two-step feature selection process to remove irrelevant features and reduce the dimensionality of feature vectors. First, we institute a threshold that constraints the minimum frequency of a feature that occurs across all data instances. We set the threshold to 5 in our experiments and thus eliminate a significant number (*e.g.*, 68% in System-B) of infrequent features. Second, we employ a well-known approach, *information gain* [16], to perform further feature selection. Information gain is widely-used and effective in text categorization [16]. We carefully set the minimum information gain to filter out many irrelevant features and reduce the feature dimensionality to around 1000.

D. Noise Handling

Another challenge lies in the data noises. In the framework of "learning to log", we implicitly assume good logging quality in the training data, which therefore facilitates the automatic learning of good logging "rules" for new development. However, there is no guarantee about the quality of logging in reality, due to the lack of "ground truth" on what is optimal logging. Considering the active maintenance and the long history of evolution of our studied software systems, it is still reasonable to assume that "most" of the data instances are enclosed with good logging decisions, while only a small portion of them may reveal incorrect decisions, which we refer to as *data noises*. For example, some instances that deserve logging are actually not logged, while some others without the need of logging are logged. These data noises thus have flipped logging labels.

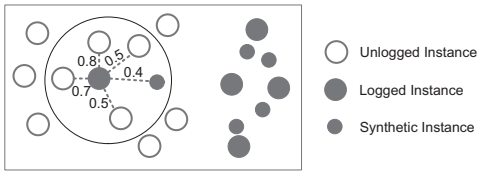


Fig. 4. Illustration of Noise Handling

We attempt to detect and eliminate such data noises, and help the model learn the common knowledge of logging more effectively. In many real-world applications, perfect data labels are impossible (or difficult) to obtain [24]. Kim et al. have proposed a simple and effective noise detection approach (namely CLNI) for defect prediction [30]. We adapt this approach to deal with our specific case, and find that it works well (as demonstrated in Section IV-D).

Traditionally, CLNI identifies the k-nearest neighbours for each instance and examines the labels of its neighbours. If a certain number of neighbours have an opposite label, the examined instance will be flagged as a noise. However, we observe a high imbalance ratio, for example up to $48.8:1$ in MonoDevelop, between unlogged (majority) instances and logged (minority) instances. Therefore, the majority instances tend to dominate the neighbourhood of an examined instance, which makes the identification of k-nearest neighbours in CLNI biased to the majority class. To handle this issue, we apply a state-of-the-art imbalance handling approach, SMOTE [21]. SMOTE balances the data instances by creating synthetic logged instances as shown in Fig. 4. Consequently, both classes have an equal number of data instances, which eliminates the inherent bias to the majority class when we identify the k-nearest neighbours of an instance. Next, we quantify each examined instance i with a noise degree value: $\varphi_i = \sum_{j \in S_i} w_{ij}$, where S_i denotes the set of neighbours with opposite label with i, and w_{ij} is the weight to characterize the different impacts of different neighbours in S_i. In contrast to CLNI that uses $w_{ij} = 1$, we take w_{ij} as the cosine similarity between features of i and j. This is based on the intuition that instances with higher similarity between each other are more likely to share the same label. Therefore, the greater the value φ_i is, the higher probability the examined instance i is a noise. For example in Fig. 4, $\varphi_i = 2.5$. We flag the instances with top ranked φ_i values and remove them as noises, while leveraging the remaining data for model training.

TABLE III
BALANCED ACCURACY OF DIFFERENT APPROACHES

Approaches	Exception Snippets				Return-value-check Snippets			
	System-A	System-B	SharpDev	MonoDev	System-A	System-B	SharpDev	MonoDev
Random	0.499	0.500	0.496	0.503	0.500	0.494	0.505	0.503
ErrLog	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500
Error Type	0.719	0.637	0.724	0.797	0.743	0.748	0.829	0.813
Methods	0.672	0.690	0.603	0.678	0.689	0.699	0.772	0.769
Textual Features	0.768	0.712	0.719	0.797	0.814	0.768	0.781	0.808
Syntactic Features	0.884	0.858	0.779	0.829	0.762	0.764	0.794	0.758
LogAdvisor	**0.934**	**0.927**	**0.846**	**0.932**	**0.903**	**0.927**	**0.865**	**0.918**

TABLE IV
BALANCED ACCURACY OF DIFFERENT LEARNING MODELS

Models	Exception Snippets				Return-value-check Snippets			
	System-A	System-B	SharpDev	MonoDev	System-A	System-B	SharpDev	MonoDev
Naive Bayes	0.701	0.623	0.686	0.714	0.746	0.766	0.788	0.762
Bayes Net	0.729	0.751	0.688	0.862	0.802	0.814	0.845	0.859
Logistic Regression	0.881	0.834	0.772	0.858	0.806	0.834	0.856	0.848
SVM	0.898	0.886	0.878	0.903	0.815	0.885	0.873	0.877
Decision Tree	0.934	0.927	0.846	0.932	0.903	0.927	0.865	0.918

IV. EVALUATION

In this section, we conduct comprehensive experiments to evaluate the effectiveness of *LogAdvisor*. In particular, we intend to answer the following research questions.

RQ1: What is the accuracy of *LogAdvisor*?
RQ2: What is the effect of different learning models?
RQ3: What is the effect of noise handling?
RQ4: How does *LogAdvisor* perform in the cross-project learning scenario?

A. Experimental Setup

After obtaining the feature vectors and their corresponding logging labels, we employ Weka [29] to perform model training and evaluation. Due to the imbalanced nature of our data, we apply the Weka implementation of SMOTE [21] to balance the training data for model construction. By default, we use decision tree (J48) as the learning model, because of its good performance (Section IV-C) and ease of interpretation. Except for the cross-project evaluation (Section IV-E), all of the experiments are evaluated on all of the extracted data instances by using the 10-fold cross evaluation mechanism [41].

As recommended in other related work [22], [46], we evaluate *LogAdvisor* using *balanced accuracy* (BA) [19], which is the average of the proportion of logged instances and the proportion of unlogged instances that are correctly classified. BA is calculated as follows:

$$BA = \frac{1}{2} \times \frac{TP}{TP+FN} + \frac{1}{2} \times \frac{TN}{TN+FP} \quad , \qquad (1)$$

where TP, FP, TN, and FN denote true positives, false positives, true negatives, and false negatives, respectively. BA weights the performance on each of the two classes equally, thus avoiding inflated performance evaluation on imbalanced data. For example, with an imbalance ratio of $48.8 : 1$ in MonoDevelop, a trivial classifier that always predict "not logging (unlogged)" can achieve 98% accuracy, but would result in a low balanced accuracy of 49%. For reference purpose, the results on other metrics such as precision, recall and F-score are provided in our supplementary report [8].

B. Results of RQ1: Prediction Accuracy

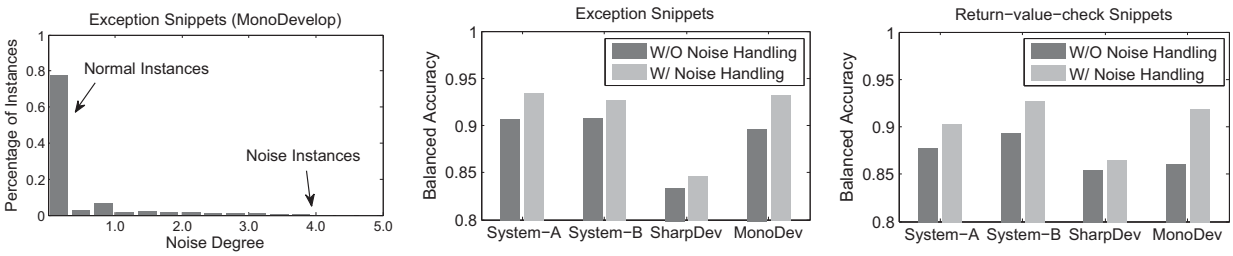
We compare *LogAdvisor* with two baseline approaches: random and ErrLog [43]. By random, we mimic the situation where a developer has no knowledge about logging and perform the logging decision with a random probability of 0.5. ErrLog is proposed in [43] that makes conservative logging (*i.e.*, log all the generic exceptions such as exceptions and function-return errors) for failure diagnosis. The results are provided in Table III.

As we can observe, both random and ErrLog have balanced accuracy of approximately 50%. Random logging has equal accuracy of 50% on either class. ErrLog logs every instance, achieving 100% accuracy on logged class, and 0% on unlogged class. Overall, the balanced accuracy of *LogAdvisor* is high, ranging from 84.6% to 93.4%, indicating high similarity to the logging decisions manually made by developers. Thus, *LogAdvisor* can learn a good representation of the common logging knowledge, and serve as a good baseline for guiding developers' logging behaviors towards better logging practice.

We also evaluate the effect of different contextual features (error type, methods, textual features, and syntactic features) on the prediction accuracy, as presented in Table III. We can see that every type of contextual feature is useful, which leads to much higher balanced accuracy than random and ErrLog. *LogAdvisor*, by combining all these useful features, makes further improvement and achieves the highest balanced accuracy. These results also reveal that the contextual features extracted from the focused code snippets provide good indication of logging practices of developers.

C. Results of RQ2: The Effect of Different Learning Models

By default, we use decision tree (J48) to train our model, due to its simplicity as well as its effectiveness shown in our previous study [26]. We also examine the impact of



(a) Instance Distribution over Noise Degree (b) The Effect on Exception Snippets (c) The Effect on Return-value-check Snippets

Fig. 5. Noise Handling Evaluation Results

different learning models on the prediction accuracy. We have tried a number of popular learning models, including Naive Bayes, Bayes Net, Logistic Regression, Supprot Vector Machine (SVM), and Decision Tree, by using their Weka implementations. The evaluation results in Table IV show that all the learning models lead to overall good prediction accuracy. In particular, Bayes-based learning models are based on probability theory. Unlike natural language text, the features extracted from source code are short and linguistically sparse, so Bayes-based learning models work slightly worse in our settings. Logistic Regression is a linear classifier, thus it may not fit well with our data. Decision Tree achieves the best overall accuracy, because this algorithm can solve non-linear classification problem. Furthermore, this algorithm can implicitly perform feature selection, which removes the redundant or irrelevant features and runs much faster than SVM for our data.

D. Results of RQ3: The Effect of Noise Handling

To evaluate the effect of noise handling approach, we first study the instance distribution across the noise degree (φ_i) values, and then compare the prediction results with noise handling and those without noise handling. For ease of presentation, we only plot the instance distribution regarding exception snippets of MonoDevelop in Fig. 5(a), while the results of other systems are also similar. In particular, we set the number of nearest neighbours, k, to 5. So φ_i has a value range of $0 \sim 5$. It shows that the majority (about 88%) of instances have a noise degree value close to 0, indicating that each examined instance has the same logging label with almost all of its nearest neighbours. Only a small proportion of instances are likely noise data (*e.g.*, those with noise degree $\varphi_i > 3$). To some extent, this reveals the quality of data. In our study, we tune the threshold and flag about 5% of instances with top ranked φ_i values as noises, which are removed them in the training phase. As the evaluation results shown in Fig. 5(b)(c), the noise handling approach makes further improvement on the prediction accuracy. It indicates that properly removing potential noise data can make our model learn the common logging knowledge more effectively.

E. Results of RQ4: Cross-Project Evaluation

In within-project learning, *LogAdvisor* leverages the existing logging instances within the same project as training data to construct the predictive model. The above experiments provide promising results on the prediction accuracy of within-project evaluation, strongly indicating that *LogAdvisor* likely work well in the scenario of developing some new components in the same project. However, many real-world projects are small or new, which have limited training data for model construction. In such cases, it is valuable to explore whether cross-project learning can help.

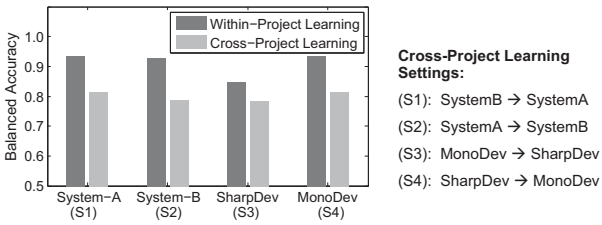


Fig. 6. Cross-Project Evaluation Results

In cross-project learning, we enrich the training data by incorporating the data instances extracted from a similar project (*source project*), and then apply the trained model to the *target project* for logging prediction. However, in contrast to within-project learning, cross-project learning is significantly more challenging [48], such as handling project-specific features. To address these challenges, we extract the common features that are shared between projects. We find that many system APIs and error types are actually common among different projects. We further leverage these common features to evaluate the performance of cross-project learning between different pairs of our studied systems (one source project for training and one target project for testing).

Due to space limits, we only provide four pairs of cross-project evaluation results in Fig. 6 (and others in our supplementary report [8]), with comparison to their corresponding within-project evaluations. The settings of these cross-project evaluations are shown as well. For example, by using System-A as the target project, and System-B as the source project, we can get a balanced accuracy of 81.5%, compared with 93.4% in within-project learning. This result, indeed, indicates that the performance of cross-project learning is largely degraded compared with within-project setting. The reason is that different projects may follow different logging practices, and some project-specific knowledge (*e.g.*, domain exceptions and methods) are challenging to adapt to other projects. However, these results can serve as a baseline for further improvement by exploring other sophisticated techniques, such as transfer learning across projects [34].

V. User Study

To further measure the effectiveness of *LogAdvisor*, we conduct a controlled user study among engineers from Microsoft and a local IT company in China. We invited 37 participants in total, including 23 staff developers and 14 interns, who have an average of 4.9 years of programming experience. In addition, 22 (59%) of them use logging frequently while 12 (32%) of them use logging occasionally. The user study is conducted through an online questionnaire, which consists of 11 questions: 5 questions for the background of participants and their understanding on logging practices, 4 questions for case studies on logging, and 2 questions for assessment of our logging suggestion results. For reproducibility, a copy of the questionnaire is provided on our project page [8].

To perform logging case studies, we randomly select 20 exception snippets and 20 return-value-check snippets from MonoDevelop. Half of them are logged, while the other half are not. We remove the logging statements in code snippets and ask participants to make logging decisions on whether to log. The original logging labels made by code owners are taken as the "ground truth". However, sometimes, it is hard for participants (not code owners themselves) to understand the code logic well by reading only a small code snippet. To mitigate this issue, we group two code snippets with different logging labels (*e.g.*, one logged exception and one unlogged exception) into a pair. Then we ask the participants to choose which one is more likely to be logged from the pair, because it is easier for an participant to make choice through comparison. To evaluate the effectiveness of *LogAdvisor*, two groups of pairs are provided: one group with our logging suggestions, and the other group without logging suggestions. The suggestion results are provided from our trained model, with an accuracy of approximately 80% on these case-study snippets. To make a fair comparison, each participant marks an equal number of pairs in each group, and each pair is marked by at least three participants. In particular, we leverage the online survey system, Qualtrics[3], to build 10 questionnaires, each using 4 different pairs of code snippets. We distribute the survey links evenly to the participants. Furthermore, we record the time they spend on making each logging choice using the timing functionality of Qualtrics.

Results: We evaluate the accuracy that the participants correctly recover the logging decisions of the code owners. For the group without logging suggestions, the accuracy is 60%, while the group with logging suggestions achieves an accuracy of 75%, with a relative improvement of 25%. As for time consumption, the participants took 33% less time on average to make a logging choice with our logging suggestions (28 seconds v.s. 42 seconds). In addition, we query the feedback from the participants by the question "Do you think the suggestion result is useful for your logging choice?", and 70% of participants think it is useful. These results provide a strong evidence in the effectiveness of our logging suggestion.

[3] *http://qtrial.qualtrics.com*

VI. Discussions

Logging quality: The approach of "learning to log" works under the premise that the training data have high logging quality. In such a setting, the constructed model can represent the common (and good) logging knowledge and generalize well to predictions of new instances. However, there is no "ground truth" on what is high-quality (or optimal) logging. In our study, we assume that our studied software systems have reasonably good logging implementations due to their high code quality, active maintenance and long history of evolution. To a certain degree, it has been endorsed by our evaluation results (*e.g.*, high prediction accuracy, positive user feedback). Besides, our noise handling approach can further mitigate the data quality issue by detecting and omitting the noisy logging instances from the training data, thus improving the performance of *LogAdvisor*.

Diversity of subject software systems: Our study was conducted on four software systems written in C#, thus its validity may be threatened by the limited diversity of our studied systems. To mitigate this threat, we choose the subjects including both commercial software systems from a leading software company like Microsoft and popular open-source software systems on GitHub. These systems are actively maintained and have a long history of evolution, which can serve as a representative of real practice. Besides, two of them are online services while the other two are IDEs, thus yielding both similar projects and dissimilar projects for our study. We believe that our approach and the results derived from these systems are easily reproducible and can be generalizable to many other software systems. Future studies on more types of software systems may further reduce this threat.

Where to log v.s. what to log: To achieve good logging quality, developers need to make informed decisions on both where to log and what to log. The ideal of "learning to log" is to help developers resolve both decisions. However, as an initial step towards this goal, we focus primarily on where to log in this paper, because it is the first logging decision to make and sometimes can determine (or narrows down) what to log. For example, when developers decide to log an exception, the contents to be recorded become much more specific, including the exception message, stack trace, etc. Besides, a recent study [45] has built an *LogEnhancer* tool that can enrich the recorded contents by automatically identifying and inserting critical variable values into the existing logging statements. As part of our future work, this tool can be further integrated into our "learning to log" framework to facilitate log automation, where *LogAdvisor* determines where to log and *LogEnhancer* determines what to log.

Potential Improvements: Towards "learning to log", we still have a number of potential directions that deserve further exploration for improvements: *1) Other factors on logging decision*. The logging behaviours of developers can be quite complex and vary among developers. Also, the logging statements can be dynamically updated, such as deletion and modification. Thus, additional consideration of factors such as

code owner, check-in time and execution frequency of code may further enhance the performance of logging prediction. *2) Interdependence of logging statements*. Our approach identifies each logging point sequentially and in isolation. In some cases, logging at one point may impact another. For example, a *try-catch block* may be enclosed in another *catch* block, and the exception may be thrown to the upper one to log. Or sometimes, logging statements at critical points are used together to record the execution path. Further exploration of a joint inference model (*e.g.*, graphical models, Markov chains) may help in this case. *3) Runtime logging*. Current logging statements are mostly statically inserted into the code. There is a new proposal for runtime logging, in which whether to log or not can be determined at runtime. For example, logs may be recorded by adaptive sampling [43] or only be recorded when encountering some problems (*e.g.*, a failed request or a long response) [11]. Although such sophisticated runtime logging mechanism is not supported by our studied systems, it is a promising direction for exploration to balance utility and overhead of logging.

VII. Related Work

Log Analysis: Logs contain a wealth of information that are useful in aiding software system maintenance, and thus become an important data source for postmortem analysis [36]. For instance, logs have been widely analyzed for various tasks, such as anomaly detection [25], [42], problem diagnosis [33], [43], program verification [37], security monitoring [32], usage analysis [31], etc. In addition to the usage of logs, Shang et al. [39] studied how to automatically enrich the produced log messages with development knowledge (*e.g.*, source code, commits, issue reports) and further assist users in log understanding. Instead, our work aims to improve the underlying logging practice, thus can potentially benefit these tasks on log analysis and log understanding.

Logging Practices: Current research has mostly focused on the usage of logs, but little on logging itself. Two empirical studies [26], [44] have recently been conducted to characterize the logging practices. Yuan et al. [44] reported the characteristics of logging modifications by investigating the revision histories of open-source software systems. Our previous work [26] focused on studying where developer log through both code analysis and developer survey at Microsoft, and summarized five typical categories of logging strategies. Additionally, Shang et al. [38] studied the relationship between logging characteristics and the code quality of platform software. All these studies provide comprehensive logging characteristics that shed insights into our design of *LogAdvisor*.

Improving Logging: Towards improving the logging quality, Yuan et al. have recently pioneered two prior studies: LogEnhancer [45] and ErrLog [43]. LogEnhancer [45] aims to enhance the recorded contents in existing logging statements by automatically identifying and inserting critical variable values into them. ErrLog [43] summarizes a set of generic exception patterns (*e.g.*, exceptions, function-return errors) that potentially cause system failures, and then suggests conservative logging to automatically log all of them (*e.g.*, log all exceptions). Their work takes the first step towards automatic logging and provides promising results in reducing diagnosis time of system failures. Our work, instead, makes an initial attempt to help developers make informed logging decisions. Furthermore, we argue that logging too much can cause unintended problems and aim to draw a good balance via "learning to log".

Mining Software Repositories: Some technical insights in the design of *LogAdvisor* are also inspired from the existing work on mining software repositories, especially from software defect prediction [30], [48]. The defect prediction methods extract features from the defective and non-defective modules, and then construct a classification model to predict the defect-proneness of a new module. Kim et al. [30] proposed the CLNI method to address the data quality issue (data noise) in defect prediction. Zimmermann et al. [48] evaluated cross-project defect predictions among 12 real-world applications, and highlighted the critical challenges in cross-project learning. Our work applies a similar machine learning approach, and also considers issues such as data quality and cross-project learning.

Exception Handling: Exception handling mechanisms [27] have been widely studied to improve the reliability and maintainability of software systems. Cacho et al. [20] evaluated how changes in exceptional code can impact system robustness. Thummalapenta et al. [40] leveraged association rules to mine some specific exception handling patterns. In our work, we focus on logging in two types of code snippets with regard to exceptions as well as function-return errors.

VIII. Conclusion

Strategic logging is important yet difficult for software development. However, current logging practices are not well documented and cannot provide strong guidance on developers' logging decisions. To fill this gap, we propose a "learning to log" framework, which aims to automatically learn the common logging practices from existing code repositories. As a proof of concept, we implement an automatic logging suggestion tool, *LogAdvisor*, which can help developers make informed logging decisions on where to log and potentially reduce their effort on logging. Evaluation results on four large-scale software systems, as well as a controlled user study, demonstrate the feasibility and effectiveness of *LogAdvisor*. We believe it is an important step towards automatic logging.

Acknowledgment

The work described in this paper was substantially supported by the National Basic Research Program of China (973 Project No. 2011CB302603), the National Natural Science Foundation of China (Project No. 61332010, 61272089), and the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 415113 of the General Research Fund).

REFERENCES

[1] 10 best practices with exceptions. http://www.wikijava.org/wiki/10_best_practices_with_Exceptions.

[2] 7 good rules to log exceptions. http://codemonkeyism.com/7-good-rules-to-log-exceptions.

[3] 7 more good tips on logging. http://codemonkeyism.com/7-more-good-tips-on-logging.

[4] The art of logging. http://www.codeproject.com/Articles/42354/The-Art-of-Logging.

[5] Code to logging ratio? http://stackoverflow.com/questions/153524/code-to-logging-ratio#153547.

[6] Exception logging in javascript. https://developer.mozilla.org/en-US/docs/Exception_logging_in_JavaScript.

[7] Exception logging: Why not log all exceptions? http://stackoverflow.com/questions/25560953/exception-logging-why-not-log-all-exceptions.

[8] Learning to log: Helping developers make informed logging decisions (supplementary report). http://cuhk-cse.github.io/LogAdvisor.

[9] Logging best practices. https://idea.popcount.org/2013-12-31-logging-best-practises.

[10] Microsoft "Roslyn" CTP. http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx.

[11] Optimal logging (Google) testing blog. http://googletesting.blogspot.com/2013/06/optimal-logging.html.

[12] Overview of Unified Logging System (ULS). http://msdn.microsoft.com/en-us/library/office/ff512738(v=office.14).aspx.

[13] The problem with logging. http://blog.codinghorror.com/the-problem-with-logging.

[14] Why not log all exceptions in MonoDevelop? http://lists.ximian.com/pipermail/monodevelop-list/2014-August/016201.html.

[15] Why not log all exceptions in SharpDevelop? https://github.com/icsharpcode/SharpDevelop/issues/554.

[16] C. C. Aggarwal and C. Zhai. A survey of text classification algorithms. pages 163–222, 2012.

[17] 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.

[18] B. Bassett and N. A. Kraft. Structural information based term weighting in text retrieval for feature location. In *Proc. of IEEE ICPC*, pages 133–141, 2013.

[19] K. H. Brodersen, C. S. Ong, K. E. Stephan, and J. M. Buhmann. The balanced accuracy and its posterior distribution. In *Proc. of IEEE ICPR*, pages 3121–3124, 2010.

[20] N. Cacho, T. César, T. Filipe, E. Soares, A. Cassio, R. Souza, I. García, E. A. Barbosa, and A. Garcia. Trading robustness for maintainability: an empirical study of evolving C# programs. In *Proc. of ACM/IEEE ICSE*, pages 584–595, 2014.

[21] 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.

[22] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. of Usenix OSDI*, pages 231–244, 2004.

[23] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *Proc. of ACM EuroSys*, pages 199–214, 2011.

[24] B. Frénay and M. Verleysen. Classification in the presence of label noise: A survey. *IEEE Trans. Neural Netw. Learning Syst.*, 25(5):845–869, 2014.

[25] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proc. of IEEE ICDM*, pages 149–158, 2009.

[26] 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.

[27] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[28] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[29] 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.

[30] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Proc. of ACM/IEEE ICSE*, pages 481–490, 2011.

[31] G. Lee, J. Lin, C. Liu, A. Lorek, and D. V. Ryaboy. The unified logging infrastructure for data analytics at twitter. *VLDB Endowment*, 5(12):1771–1780, 2012.

[32] M. Montanari, J. H. Huh, D. Dagit, R. Bobba, and R. H. Campbell. Evidence of log integrity in policy-based security monitoring. In *Proc. of IEEE DSN Workshops*, pages 1–6, 2012.

[33] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proc. of USENIX NSDI*, pages 26–26, 2012.

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

[35] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *Proc. of IEEE ASE*, pages 378–388, 2013.

[36] A. J. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *Commun. ACM*, 55(2):55–61, 2012.

[37] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proc. of ACM/IEEE ICSE*, pages 402–411, 2013.

[38] W. Shang, M. Nagappan, and A. E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 2013.

[39] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang. Understanding log lines using development knowledge. In *Proc. of IEEE ICSME*, 2014.

[40] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proc. of ACM/IEEE ICSE*, pages 496–506, 2009.

[41] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques (Second Edition)*. Morgan Kaufmann Publishers Inc., 2005.

[42] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. of ACM SOSP*, pages 117–132, 2009.

[43] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *Proc. of USENIX OSDI*, pages 293–306, 2012.

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

[45] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. *ACM Trans. Comput. Syst.*, 30(1):4:1–4:28, 2012.

[46] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *Proc. of IEEE DSN*, pages 644–653, 2005.

[47] W. Zheng, Q. Zhang, and M. Lyu. Cross-library api recommendation using web search engines. In *Proc. of ACM ESEC/FSE*, pages 480–483, 2011.

[48] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proc. of ESEC/SIGSOFT FSE*, 2009.