Characterizing the Natural Language Descriptions in Software Logging Statements

Pinjia He, Zhuangbin Chen, Shilin He, Michael R. Lyu
pinjiahe@gmail.com,{zbchen,slhe,lyu}@cse.cuhk.edu.hk
The Chinese University of Hong Kong, Hong Kong, China
Shenzhen Research Institute, The Chinese University of Hong Kong, Shenzhen, China

ABSTRACT

Logging is a common programming practice of great importance in modern software development, because software logs have been widely used in various software maintenance tasks. To provide high-quality logs, developers need to design the description text in logging statements carefully. Inappropriate descriptions will slow down or even mislead the maintenance process, such as postmortem analysis. However, there is currently a lack of rigorous guide and specifications on developer logging behaviors, which makes the construction of description text in logging statements a challenging problem. To fill this significant gap, in this paper, we systematically study what developers log, with focus on the usage of natural language descriptions in logging statements. We obtain 6 valuable findings by conducting source code analysis on 10 Java projects and 7 C# projects, which contain 28,532,975 LOC and 115,159 logging statements in total. Furthermore, our study demonstrates the potential of automated description text generation for logging statements by obtaining up to 49.04 BLEU-4 score and 62.1 ROUGE-L score using a simple information retrieval method. To facilitate future research in this field, the datasets have been publicly released.

CCS CONCEPTS

• **Software and its engineering** → **Software development methods**; • **Computing methodologies** → *Natural language processing*;

KEYWORDS

Logging, natural language processing, empirical study

ACM Reference Format:
Pinjia He, Zhuangbin Chen, Shilin He, Michael R. Lyu. 2018. Characterizing the Natural Language Descriptions in Software Logging Statements. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, Montpellier, France, 12 pages. https://doi.org/10.1145/3238147.3238193

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238193

1 INTRODUCTION

Logging is a common programming practice of practical importance for modern software developers. Developers mainly conduct logging by writing logging statements (e.g., *printf(), logging.info()*) and inserting them into the code snippets. As in-house debugging tools (e.g., debugger), all too often, are inapplicable in production settings [62], software logs have become the principal source of information when diagnosing a problem. Specifically, software logs have been used in various reliability enhancement tasks, including anomaly detection [18, 57], fault diagnosis [55, 63], program verification [17, 51], performance monitoring [27, 43], etc. The performance of these tasks highly depends on the quality of the collected logs. Thus, for modern software development and maintenance, appropriate logging statements are of great importance.

Typically, a logging statement contains description text and variables. Real-world examples of logging statements can be found in Fig. 2. Description text describes the specific system operation in runtime, which is the main focus of this paper, while variables record necessary system status (e.g., IP address). Elaborate description text can accelerate reliability enhancement tasks by providing better understanding of the system runtime information. On the contrary, immature description text (e.g., outdated text) slows down the log analysis process or may even mislead the developers [13].

However, logging is not easy because of the following reasons. First, there is currently a lack of rigorous specification on logging practice. For example, Fu et al. [19] find 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. Moreover, with the popularity of source code sharing platforms (e.g., Github), modern software may consist of components written by multiple developers all over the world. This further increases the difficulty for a developer to align with the project logging style. This problem is compounded by the fact that logging statements update frequently in modern software development (e.g., hundreds of new logging statements every month [56]). Although there are some existing logging frameworks (e.g., Microsoft's ULS [6] and Apache's log4net [1]), developers still need to make their own logging decisions [62]. Thus, where to log and what to log have become crucial but challenging problems.

Recently, Cinque et al. [15] summarize empirical rules regarding the placement of logging statements. Zhu et al. [62] propose LogAdvisor, which recommends whether developers should write a logging statement in a code snippet or not. They focus on the "where to log" problem, while do not consider "what to log". Yuan et al. [60] develop LogEnhancer, a tool that can enhance logging statements by adding informative variables. However, their method

focuses on existing statements and does not consider the description text.

To fill in this significant gap of "what to log", we conduct the first empirical study on the context of logging statements, with focus on their natural language descriptions. Specifically, we collect 10 Apache Java projects and 7 C# projects, which contain 28,532,975 LOC and 115,159 logging statements in total. We first study the purpose of logging by manually inspecting 383 logging statements and their surrounding code snippets. Then, we study the repetitive usage of certain n-grams (i.e., a sequence of n tokens), which we call n-gram patterns, in the natural language descriptions. For simplicity, in the following, we use *logging descriptions* to represent the natural language description text in logging statements. In particular, could we generate the logging descriptions based on historical logs using n-gram language model [5]? Are logging descriptions locally repetitive (e.g., in a source file)? Moreover, based on the manual inspection experience and quantitative evaluation, we further study the possibility of automated logging description generation. In particular, is it potentially feasible to implement a logging description suggestion tool to assist developers in determining what to log? By answering the above questions through systematic analysis, this investigation helps to characterize the current usage of logging descriptions and serves as the first step towards automated logging description generation.

The results of our study show that there are generally three categories of logging descriptions, including description for program operation (37.34%), description for error condition (39.16%), and description for high-level code semantics (23.5%) (*Finding* 1). Besides, compared with common English, the repetitiveness in logging descriptions can be better captured by statistical language models. (*Finding* 2). However, the n-gram patterns in different projects vary a lot (*Finding* 3), which is caused by the localness [53] of logging descriptions. In particular, the n-gram patterns in logging descriptions are endemic to one source file or frequently used in a few source files (*Findings* 4~5). In addition, we evaluate the potential feasibility to automatically generate logging descriptions based on historical logs. The high BLEU score [45] and ROUGE score [39] imply that automated logging description generation is feasible and deserves more future exploration (*Finding* 6).

In summary, our paper makes the following contributions:

- This paper conducts the first empirical study on the usage of natural language in logging practice by an evaluation of 10 Java projects and 7 C# projects.
- It summarizes three categories of logging descriptions in logging statements, including description for program operation, description for error condition, and description for high-level code semantics, covering all the scenarios observed in our study.
- We demonstrate the repetitiveness in logging descriptions globally (i.e., in a project) and locally (i.e., in a source file), and further present the potential feasibility of automated logging description generation.
- The datasets studied have been publicly released [3], allowing easy use by practitioners and researchers for future study.

The remainder of this paper is organized as follows. Section 2 explains the methodology in our study. Section 3 summarizes logging descriptions into different categories. Section 4 characterizes the natural language descriptions by quantitative analysis. Based on the evaluation results, the potential feasibility of automated logging description generation is assessed in Section 5. We discuss potential directions to improve logging in Section 6. Section 7 introduces related work. Finally, we conclude this paper in Section 8.

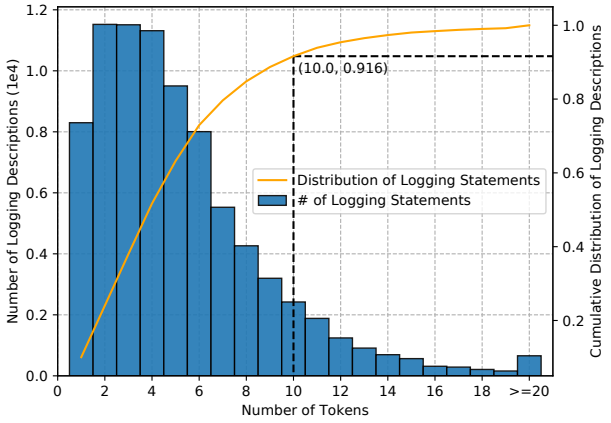


Figure 1: Distributions of the length of logging descriptions.

2 STUDY METHODOLOGY

In this paper, we study the logging statements of 10 Java projects and 7 C# projects, which contain 28,532,975 LOC and 115,159 logging statements in total. In particular, the Java projects are collected from Apache Project List [2] with more than 15 committers, while the C# projects are widely-used software with more than 1,000 stars on Github.

Table 1 presents the details of these open-source projects. These projects are of great variety, ranging from distributed system, database, enterprise service bus, SDK, to IDE. *Description* presents a brief introduction of the project. *Version* shows the date of the last commit to the master branch in the repository when we downloaded the project source code. *LOC* indicates the corresponding project's line of code. *# of Logging Statements* is the number of logging statements in the project. *# of Logging Descriptions* is the number of logging statements that have natural language descriptions. *Tokens in Logs* counts the total tokens extracted from the logging descriptions; while *Distinct Tokens* counts the distinct tokens.

We can observe that a majority of the logging statements contain logging descriptions. In particular, 81.9% of the logging statements in ActiveMQ have logging descriptions; while the average percentage of all projects is 69.8%. Thus, we can find that logging descriptions are important and widely adopted by developers.

The lengths of the logging descriptions in all of the projects are illustrated in Fig. 1. In this paper, by length, we mean the number of tokens in a logging description. The lengths of more than 90% of the logging descriptions are in range [1, 10]. Different from other text in software engineering, such as code or documentation, the length of logging description is shorter. Thus, we expect different characteristics of the natural language used in logging descriptions.

These projects are supposed to have good logging practice, especially appropriate logging descriptions, for the following two

Table 1: Java and C# project details.

Java Project	Description	Version	LOC	# of Logging Statements	# of Logging Descriptions	Tokens in Logs	
						Total	Distinct
ActiveMQ	Message Broker	20180212	689.9K	6.8K	5.6K (81.9%)	23,659	2,200
Ambari	Hadoop Monitor	20180220	798K	5K	3.7K (75.2%)	22,590	2,104
Brooklyn	Distributed System Manager	20170901	483.5K	4.5K	3.6K (79.7%)	24,255	2,458
Camel	Integration Framework	20180126	1.8M	11.7K	8.8K (75.5%)	45,288	3,521
CloudStack	Cloud Computing Software	20180131	838.8K	12K	9.8K (81.6%)	67,753	3,579
Hadoop	Distributed Computing Platform	20171214	1.9M	13.9K	11.0K (79.4%)	59,825	4,669
HBase	Distributed Database	20170804	957K	8.3K	6.6K (79.8%)	36,782	2,969
Hive	Data Warehouse	20170812	1.5M	7.3K	5.9K (81.2%)	32,756	3,151
Ignite	Distributed Database	20171031	1.6M	4.6K	3.5K (77.3%)	18,476	1,737
Synapse	Enterprise Service Bus	20171204	586.3K	8.1K	5.1K (64.0%)	31,489	1,436
C# Project	**Description**	**Version**	**LOC**	**# of Logging Statements**	**# of Logging Descriptions**	**Tokens in Logs**	
						Total	Distinct
Azure SDK	Azure Tools for Visual Studio	20170301	2.1M	786	529 (67.3%)	1,804	474
CoreRT	.NET Core Runtime	20180208	537.6K	677	413 (61.0%)	1,364	419
CoreFX	.NET Core Foundational Lib.	20180208	3.6M	7.5K	4.7K (63.2%)	21,219	2,009
Mono	.NET Framework	20180116	7.5M	14.1K	7.9K (56.4%)	26,457	3,681
Monodevelop	Cross Platform IDE	20180103	2.2M	6.6K	3.0K (46.3)	13,540	2,067
Orleans	Distributed Virtual Actor Model	20180228	242.6K	761	484 (66.2%)	2,342	529
Sharpdevelop	Cross Platform IDE	20171221	701.5K	2.1K	1.1K (51.8%)	4,168	1,190

reasons. First, these projects are maintained by developers from large organizations or companies, which provide important software solutions powering modern IT industry. For example, Hadoop is used by a large amount of companies to process large-scale data. Second, these projects have been serving users globally for a long time, and thus their logging statements have fulfilled the use of daily development and maintenance.

We characterize the logging descriptions by both manual inspection and these evaluation metrics. To study the purpose of logging descriptions, we first randomly sample a subset of logging descriptions and their corresponding code snippets. After manually exploring these samples in detail, we categorize logging descriptions into 3 groups. Then, to further study the characteristics of them, we evaluate the global and local repetitiveness in logging descriptions by the evaluation metrics proposed in previous empirical studies on source code [28, 53]. In all the experiments using evaluation metrics, we regard the logging descriptions as plain text and study them from the natural language perspective.

3 CATEGORIES OF LOGGING DESCRIPTIONS

To characterize the usage of natural language in logging descriptions, the first step is to understand the purpose of these descriptions. Thus, in this section, we manually inspect the logging descriptions and summarize them into different categories.

Since the total number of logging descriptions is large, it is prohibitive to manually inspect all the logging descriptions and analyze the corresponding code snippets. Thus, we randomly sample a subset of logging descriptions together with the corresponding code snippets from all the projects in Table 1. Similar to existing studies [20, 21], we calculate the number of samples by standard techniques [35]. Specifically, the number of samples is determined by a desired margin of error, confidence level, and the data size. There are totally 82,476 logging descriptions in our study. Thus, we determine the sample size as 383 after setting ±5% margin of error and 95% confidence.

Table 2: Categories from 383 sampled logging descriptions.

Categories		Samples	% of Samples	
Program Opeartion	Completed	59/383	15.40%	37.34%
	Current	18/383	4.70%	
	Next	66/383	17.23%	
Error Message	Exception	96/383	25.07%	39.16%
	Value-Check	54/383	14.10%	
Semantic Description	Variable	40/383	10.44%	23.50%
	Function	15/383	3.92%	
	Branch	35/383	9.14%	

To figure out the purpose of logging descriptions, we assign each logging statement and its surrounding code snippet to three researchers with 6 years' programming experience in average. Each researcher labels the category of the logging description after manual analysis. Then we compare the labels returned by different researchers. If all the labels for a logging description are the same, we regard it as the final label. Otherwise, the researchers re-visit the case together and produce the final label after discussion. Table 2 shows the details of the categories and the number of samples in each category. In particular, we summarize 3 main categories, under which there are totally 8 mutually exclusive subcategories.

```
/*  Example 1: Completed Operation */
final lbmonitor monitorObj = lbmonitor.get(_netscalerService,
nsMonitorName);
monitorObj.set_respcode(null);
lbmonitor.delete(_netscalerService, monitorObj);
s_logger.info("Successfully deleted monitor : " + nsMonitorName);

/*  Example 2: Current Operation */
while (nm.getServiceState() != STATE.STOPPED && waitCount++ != 20) {
      LOG.info("Waiting for NM to stop..");
      Thread.sleep(1000);
}

/*  Example 3: Next Operation */
LOG.info("Close consumer on A");
clientA.close();

/*  Example 4: Exception */
try {
      count = c.readAndProcess();
    } catch (InterruptedException ieo) {
       LOG.info(Thread.currentThread().getName() + ": readAndProcess
caught InterruptedException", ieo);
       throw ieo;
}

/*  Example 5: Value-Check */
if (jobId == null) {
   s_logger.error("Unable to get a jobId");
   return null;
}

/*  Example 6: Variable Description */
String messageID = (String) element.get("JMSMessageID");
LOG.debug("MessageID: {}", messageID);

/*  Example 7: Function Description */
public void forget(Xid xid) throws XAException {
        LOG.debug("Forget: {}", xid);
        ...
}

/*  Example 8: Branch Description */
if (blobItem instanceof CloudBlockBlobWrapper || blobItem instanceof
CloudPageBlobWrapper) {
   LOG.debug("Found blob as a directory-using this file under it to
infer its properties {}", blobItem.getUri());
...
}
```

Figure 2: Real-world examples of logging statements.

In addition, we provide a selected example from the studied project for each subcategory in Figure 2. The details of these categories are introduced as follows. In this study, we start with 7 subcategories based on previous experience and preliminary inspection on the logging statements. After exploring all the 383 samples, we find a new subcategory (i.e., branch description) and finally summarize with 8 subcategories in this paper.

Category 1: Program Operation. Logging descriptions in this category summarize the detailed actions or intentions of the surrounding program. Based on the position of the described program statements, this category can be further divided into three subcategories, including *completed operation*, *current operation*, and *next operation*. In particular, a *completed operation* logging description concludes the behavior of program statements preceding a logging statement. This kind of logging statements are often placed at the end of a function/block scope. Example 1 shows a completed operation that deletes a monitor successfully. In *current operation*, developers log the current status of a program to trace the progress of an action. Typical usage of such description is in a *while* loop or a *for* loop, as illustrated in Example 2. Compared with aforementioned two types, *next operation* is more widely utilized to forecast the following behaviors of a program, which often indicates the start of some operations. In Example 3, developers log the next operation of closing the consumer on client A. In our study, more than 37% of samples belong to the *program operation* category.

Category 2: Error Message. It is a common practice for developers to log the error message for maintenance. In this category, logging descriptions mainly present the occurrence or the behind-reasons of an error/exception. There are two types of logging description in this category: *exception* and *value-check*. *Exception* uses the *try-catch* block and the logging description is often written in the *catch* clause. Some error-related keywords (e.g., failed, error, exception) are frequently employed and often indicate a failed execution in the try block. Example 4 is a representative example. *Value-check* logging descriptions explain errors without explicitly employing the *try-catch* block. Instead, an *if-statement* is usually applied to check the value of a variable in current program or the return value of a specific function. For instance, Example 5 shows the value checking of a variable against *null*. Other values such as *false*, *empty* are also widely checked and the corresponding error messages are then logged. Most samples (around 40%) in our study are categorized into the *error message* category.

Category 3: Semantic Description. In this category, there are three subcategories of logging descriptions according to the object they describe, i.e., *variable description*, *function description* and *branch description*. In particular, *variable description* records the value of a pivotal variable during execution. As shown in Example 6, the detailed value of message ID is logged. *Function description* is widely utilized to depict the functionality and usage of a function and its arguments. Generally, *function description* is placed at the beginning of a function body, as illustrated in Example 7. *Branch description* describes the semantic meaning of a branch/path. Different from program operation and error message, *branch description* is mainly used in *if-else* blocks to indicate the execution path in software runtime. Example 8 presents a logging description that captures the semantic meaning of this branch. 23.50% of our studied samples are in the *semantic description* category.

Finding 1: There are three main categories of logging descriptions, i.e., description for program operation, description for error condition, and description for high-level code semantics.

4 LANGUAGE PATTERNS OF LOGGING DESCRIPTIONS

In this section, we try to answer the following research questions:

- RQ1: Is there any repetitiveness in logging descriptions?
- RQ2: Can the repetitiveness be captured by cross-project n-gram models?
- RQ3: Are there any n-grams in logging descriptions appearing in only one source file?
- RQ4: Are n-grams in logging descriptions locally specific to a few source files?

By answering these research questions with quantitative analysis of the open-source projects, we aim to facilitate better understanding of the current logging description usage by developers. In addition, the answers to these questions demonstrate the potential of automated logging description generation.

4.1 RQ1: Is There Any Repetitiveness in Logging Descriptions?

The repetitiveness of n-grams in common English has already been successfully used in various tasks, such as speech recognition, machine translation, and code completion. Thus, it would be of great interest to explore whether logging descriptions are also repetitive and predictable for potential applications (e.g., automated description generation). To answer this question, in this section, we try to use n-gram language models to predict the next token in a logging description. Intuitively, if there is observable repetitiveness in logging descriptions, the models should have decent prediction performance. In the following, we first introduce the n-gram language model and the evaluation metric. After that, we analyze the experimental results and summarize with a finding.

N-gram language model. Language models are statistical models that predict the probability of sequences of words. N-gram models assume a *Markov property*, i.e., the probability of a token only depends on the preceding $n-1$ tokens. For example, for 4-gram models, the probability p of a token is calculated based on the frequency counting of the previous 3 tokens, as the following:

$$p(a_4|a_1a_2a_3) = \frac{count(a_1a_2a_3a_4)}{count(a_1a_2a_3*)}. \quad (1)$$

If there is observable repetitiveness in logging descriptions, an n-gram model $\mathcal{M}$ should be able to learn the probability distributions of n-grams from a corpus. In practice, the n-gram model often encounters some unseen n-grams during prediction. This makes the probability $p_{\mathcal{M}}(a_i|a_1...a_{i-1}) = 0$. Smoothing is a technique that can handle such cases, and assign reasonable probability to the unseen n-grams. In this paper, we use Modified Kneser-Ney Smoothing [34], which is a standard smoothing technique and can give good results for software corpora [28].

Evaluation metric: cross-entropy. We use 10-fold cross validation, where 90% of the logging descriptions are the training data, and the remaining 10% are the testing data. To evaluate the performance of the n-gram models, we use *cross-entropy*, which is defined as follows:

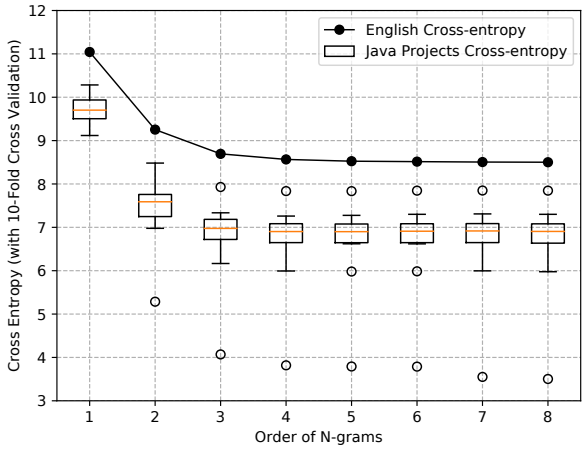
$$H_{\mathcal{M}}(s) = -\frac{1}{n}\sum_{i=1}^{n}\log p_{\mathcal{M}}(a_i|a_1...a_{i-1}) \quad (2)$$

$H_{\mathcal{M}}$ is the cross-entropy of the n-gram model $\mathcal{M}$; s is a logging description of length n and $s = a_1...a_n$; and $p_{\mathcal{M}}$ is the probability that the next token is a_i given the preceding token sequence $a_1...a_{i-1}$. A good model has low cross-entropy for logging descriptions. For example, if a model can correctly predict all the tokens in a logging description, the probability $p_{\mathcal{M}}(a_i|a_1...a_{i-1})$ will be 1 for all tokens a_i, and hence the cross-entropy will be 0. We calculate the cross-entropy for projects in Table 1. Specifically, for each project, we calculate the cross-entropy for every logging description using Equ. 2 and use their average as the cross-entropy for the project.

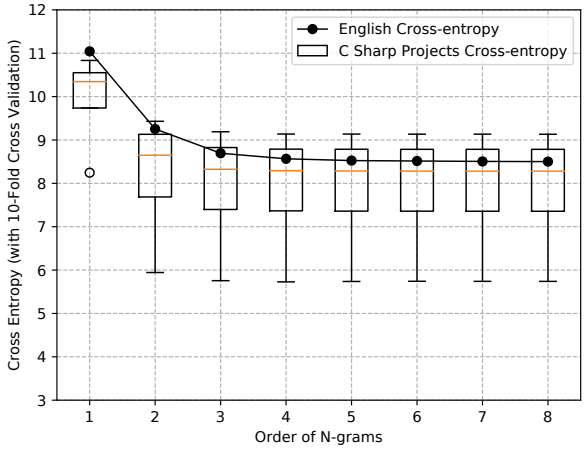
Results. The cross-entropy results of Java and C# projects are demonstrated by the boxplots in Fig. 3. In this experiment, we evaluate the projects with more than 1,000 logging descriptions. To figure out the potential differences between logging descriptions and common English, we also calculate the cross-entropy of two common English corpora, which is demonstrated by the single lines in Fig. 3. The details of these two corpora are illustrated in Table 3. By doing so, we can have an intuitive understanding of the repetitiveness of natural language in logging descriptions and in common English. We analyze each project separately (boxplots), while we analyze the two English corpora as a whole (a line).

Table 3: English corpora.

English Corpus	Version	Lines	Tokens	
			Total	Distinct
Brown	20171201	56,832	1,023,161	53,090
Gutenberg	20171201	98,326	2,136,001	53,253



(a) Java projects



(b) C# projects

Figure 3: Cross-entropy of natural language descriptions in logging statements.

We can observe that both the single lines and boxplots have similar trends. The single lines start at 11 for 1-gram models and trail down to about 8.5 for 8-gram models. This means that for both logging descriptions and common English, using more preceding tokens (i.e., larger n) can lead to more accurate results. Besides, according to the figures, cross-entropy saturates around 3- or 4-grams.

Thus, 3- or 4-gram models are the best choice for the investigated projects considering the trade-off between cross-entropy and model complexity.

In addition, compared with common English, the cross-entropy of the logging descriptions is generally smaller, which means the tokens in logging descriptions are easier to predict. This phenomenon is more obvious for logging descriptions in Java projects. We think this is mainly because all the Java projects are collected from Apache Project List, which share similar logging styles. The observable repetitiveness in logging descriptions is encouraging, and it is promising to utilize the repetitiveness for automated logging in the future.

Finding 2: Compared with common English, the repetitiveness of logging descriptions can be better captured by statistical language models.

4.2 RQ2: Can the Repetitiveness be Captured by Cross-project N-gram Models?

In this section, we further study the repetitive n-gram patterns in logging descriptions. Specifically, does repetitiveness exist across different projects, or does it lie in individual projects locally? To answer this question, we conduct the cross-project experiments. For each Java project, we train a 3-gram model based on 90% of the project logging descriptions. Then, we use the model to predict tokens in (1) the remaining 10% of the logging descriptions and (2) the logging descriptions of all other 9 Java projects. We chose 3-gram models because they do not require too much memory while achieving decent performance. We mainly focus on Java projects, because they are all Apache Java projects which may share more cross-project similarities than C# projects.

The results are shown in Fig. 4. The x-axis lists all the projects that are used to train the n-gram models. The single line illustrates the cross-entropy of the models on the remaining 10% logging descriptions (in-project), while the boxplot shows the cross-entropy of the models on all other projects (cross-project). We can observe that the cross-project cross-entropy is clearly larger than the in-project cross-entropy. This indicates that the repetitive n-gram patterns in different projects are quite different, and thus the n-gram patterns can hardly be captured by cross-project models. The in-project cross-entropy of project Synapse is very low because it contains many identical logging descriptions.

Finding 3: N-gram models trained on other projects cannot capture repetitive n-gram patterns well, indicating that the n-gram patterns in different projects vary a lot.

4.3 RQ3: Are There Any N-grams in Logging Descriptions Appearing in Only One Source File?

Results in the previous section indicate that n-gram patterns tend to appear locally inside the project. To further study the repetitiveness in local context, in the following, we explore whether some n-grams in logging descriptions can be found in only one source file. These n-grams are called endemic n-grams. Table 4 demonstrates the percentage of endemic n-grams in the studied projects. For example, if the logging descriptions of a project have 100 2-grams and 20 of them can only be found in one source file, the percentage of endemic 2-grams is 20%. We can observe that 28.86% 2-grams in Java logging descriptions and 47.91% 2-grams in C# logging descriptions are endemic. The percentage rapidly increases for settings with longer n-grams, because it is more difficult to spot identical longer n-grams in different source files. In addition, among the endemic n-grams, 16.86% endemic 3-grams and 17.37% endemic 4-grams are found more than once in Java projects. For C# projects, the percentages are even larger. These endemic, but locally repeating n-grams further demonstrate the local repetitiveness in source file level.

Besides logging descriptions, we also calculate the percentage of n-grams from all code statements that only appears in one source file. As illustrated in Table 4, the percentages of the endemic n-grams in logging descriptions are generally larger than that in source code, which indicates that logging descriptions are more locally endemic than source code. As reported by a previous paper [53], localness cannot be found in common English, so we regard it as a special feature of the natural language in logging statements. We think this feature can be utilized to improve the automated logging method. For example, we can improve the performance of language models with a cache mechanism similar to that proposed in [53].

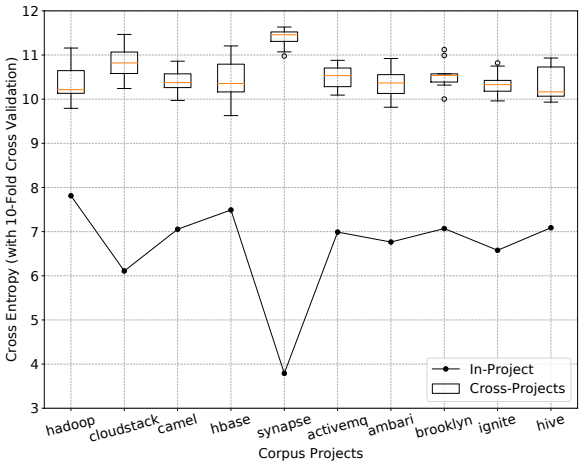


Figure 4: Cross-project cross-entropy versus in-project cross-entropy of the 10 Java projects.

Table 4: Percentage of the endemic n-grams.

Lang.	Freq.	1-gram	2-gram	3-gram	4-gram
Java Log	>=1	2.83%	28.86%	57.62%	68.89%
	>=2	1.32%	10.01%	16.86%	17.37%
Java Code	>=1	2.84%	15.82%	31.54%	40.61%
	>=2	2.54%	7.25%	12.05%	14.03%
C# Log	>=1	8.85%	47.91%	67.23%	74.81%
	>=2	4.87%	21.02%	25.28%	25.28%
C# Code	>=1	2.71%	15.48%	31.52%	42.24%
	>=2	2.43%	8.17%	13.87%	16.97%

Finding 4: Logging descriptions are locally endemic. A number of N-grams are repetitively used in the logging descriptions in only one source file.

4.4 RQ4: Are N-grams in Logging Descriptions Locally Specific to a Few Source Files?

We have studied the endemic n-grams that only appear in one source file. In this section, we further explore whether the non-endemic n-grams in logging descriptions also favor a specific locality. By definition, each non-endemic n-gram can be found in a set of files F, and thus, there is a discrete probability distribution p for F. For example, if an n-gram is uniformly distributed, then each file in F contains the same number of this n-grams. We hypothesize that, if the non-endemic n-grams favor specific locality, the distribution p will be skewed. For example, an n-gram, which is found in 100 source files, appears 20 times in one source file and once in the remaining 99 source files. Inspired by [53], we use locality entropy $H_{\mathcal{L}}$, which is defined as follows, to measure the skewness of the distribution of an n-gram σ in F.

$$H_{\mathcal{L}}(\sigma) = -\sum_{f \in F} p(f_\sigma) \log_2 p(f_\sigma), \tag{3}$$

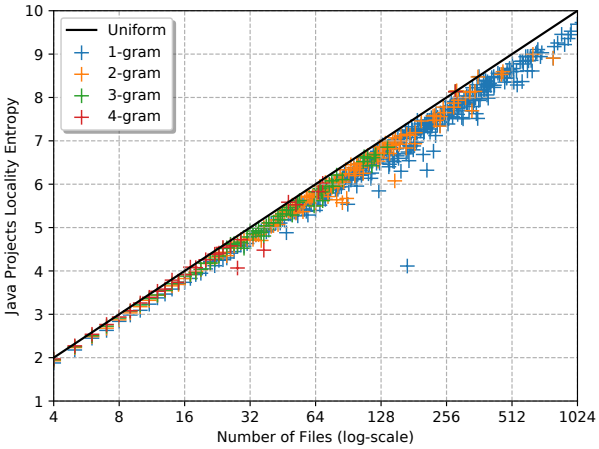
where $p(f_\sigma)$ is defined as follows:

$$p(f_\sigma) = \frac{\text{count(n-gram) } \sigma \text{ in } f}{\text{count(n-gram) } \sigma \text{ in } project} \tag{4}$$

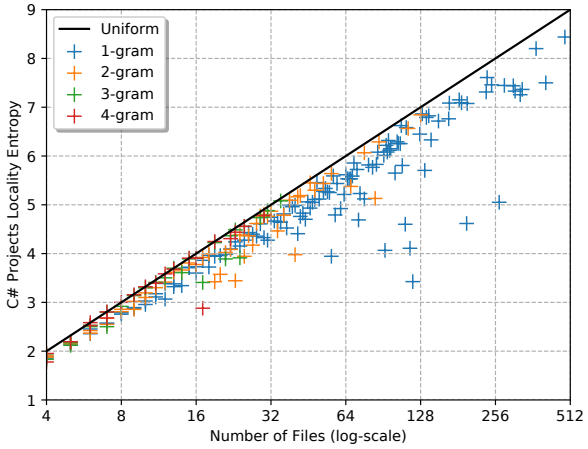
Note f is a source file that contains the non-endemic n-gram σ; *project* is the collection of all the source files in a project; and *count* calculates the number of n-grams. Intuitively, the more skewed the distribution, the lower the entropy. For example, if an n-gram is found only in the logging descriptions of one source file, the entropy will be the lowest (i.e., 0). On the contrary, if an n-gram is uniformly distributed in the logging descriptions of source files F, the entropy will be the highest. Thus, the lower the entropy, the more the logging descriptions are locally specific.

The entropy of both Java and C# projects is shown in Fig. 5. The x-axis is the number of files that contain the non-endemic n-gram. We can observe that almost all non-endemic n-grams have lower entropy than uniform distribution, which demonstrates their locally specific property. Besides, the n-gram with varying orders, which are marked by different colors, share similar trends. We can also observe that some 1-grams have much lower entropy than the others. This is because some common tokens are intensively used in the logging descriptions of a file. For example, in project Hive, token "writing" is found in the logging descriptions in 169 source files (i.e., $|F| = 169$), which is non-endemic. While token "writing" only appears once in most of these source files, it appears 366 times in one source file, because logging description "Exception *writing* to internal frame buffer" is repetitively used for 366 times locally in that file.

Finding 5: Logging descriptions are locally specific. The non-endemic n-grams are repetitively used in a few source files.



(a) Java projects



(b) C# projects

Figure 5: Entropy of the file distributions for non-endemic n-grams. "Uniform" denotes that the n-grams are distributed uniformly in the files.

5 AUTOMATED LOGGING DESCRIPTION GENERATION

Based on the experimental results in Section 3 and 4, it is valuable to explore whether it is possible to automatically generate the logging description for a logging statement. If shown possible, such an automated tool will be of great help for developers, because it can greatly accelerate the development process and potentially improve the quality of their logging descriptions. In this section, we propose a simple but effective automated logging description generation method, in order to demonstrate the potential feasibility of logging automation.

5.1 Methodology

Our method is based on an assumption: similar code snippets tend to contain similar logging descriptions. This assumption is triggered by the results and findings in previous sections. In particular, as explained in Section 3, all categories of logging descriptions are

used to describe certain code statements, thus the logging descriptions should be closely related to the corresponding code snippets. Additionally, in Section 4, the locality experiments demonstrate that similar n-gram patterns are repetitively used in a local context (source file-level), where code providing similar functionalities gathers. Based on this assumption, we propose a simple information retrieval-based method. In particular, to generate the logging description for a logging statement, we extract its corresponding code snippet, and search for the most similar code snippet in the corpus (i.e., existing code). Then, we use the logging description in the searched code snippet as the description for current logging statement. The similarity is measured by Levenshtein distance [4], which regards a code snippet as a string and calculate the distance using character-based edit distance. For example, the Levenshtein distance between "public boolean createFile();" and "public boolean newFile();" is 5, because we need 5 substitutions (i.e., change "cr" to "n" and "ate" to "w") to make them identical.

In this experiment, for each project, we use 10-fold cross validation. Specifically, we first extract the $< code, log >$ pairs from the training data. Considering the code snippet, we study two code ranges: *Pre-10* and *Sur-20*, where *Pre-10* indicates the 10 lines of code preceding the logging statement and *Sur-20* indicates the 10 lines of code preceding and succeeding the logging statement. We generate the logging description for each code snippet in the testing data by searching the most similar code snippet in the training data.

5.2 Evaluation Metrics

5.2.1 BLEU. To measure the accuracy of automated logging description generation, we use BLEU [45], a popular evaluation metric used in text summarization and machine translation tasks [9, 41, 49, 50]. We use BLEU because it can measure the similarity between the candidate and the reference. In our experiments, the logging description generated by our method is regarded as the candidate, while the original logging description written by the developer is regarded as the reference. Specifically, BLEU is calculated as follows:

$$BLEU = BP \cdot \exp(\sum_{n=1}^{N} w_n \log p_n), \tag{5}$$

where BP is a brevity penalty that penalizes overly short candidates; N is the maximum number of grams used in the experiments; p_n is the modified n-gram precision; and w_n is the weight of each p_n. BLEU-1 means the BLEU score considering only the 1-grams in the calculation, where $w_1 = 1$ and $w_2 = w_3 = w_4 = 0$. Specifically, BP is calculated as follows:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{6}$$

where r is the length of the reference; c is the length of the candidate. The modified n-gram precision is defined as follows:

$$p_n = \frac{\text{\#n-grams appear in the reference}}{\text{\#n-grams in the candidate}} \tag{7}$$

From the definition of BLEU, we know that the higher the BLEU, the better the logging statement generation performance. The range of BLEU is $[0, 1]$, which is often presented as a percentage value (i.e., $[0, 100]$). Thus, if none of the n-grams in the candidate appear in the reference, BLEU score is 0. In the contrary, if the candidate is exactly the same as the reference, BLEU score is 100.

5.2.2 ROUGE. BLEU measures how many n-grams in the generated logging statement appear in the reference, which enjoys similar sense as "precision". Compared with BLEU, ROUGE [39] is like "recall", which measures how many n-grams in the reference appear in the generated logging statement.

Specifically, ROUGE is defined as follows:

$$ROUGE - N = \frac{\sum_{S \in Ref} \sum_{gram_n \in S} count_{match}(gram_n)}{\sum_{S \in Ref} \sum_{gram_n \in S} count(gram_n)}, \tag{8}$$

where n represents the length of the n-gram, $gram_n$; S is a reference; Ref is the set of all references; $count_{match}(gram_n)$ is the maximum number of n-grams co-occurring in the candidate and the reference; and $count(gram_n)$ is the number of n-grams in the reference. In our experiments, we calculate ROUGE-1 to ROUGE-3, and ROUGE-L. ROUGE-L does not require a predefined n-gram length. Instead, it measures the longest matching sequence of words using LCS (Longest Common Subsequence).

Similar to BLEU, the range of ROUGE is $[0, 1]$, which is often presented as a percentage value (i.e., $[0, 100]$). If none of the n-grams in the reference appear in the candidate, ROUGE score will be 0. In the contrary, if all n-grams in the reference appear in the candidate, ROUGE score is 100.

5.3 Results

The BLEU scores and the ROUGE scores are shown in Table 5. We run the experiments on 5 Java projects and 3 projects, which are selected based on the number of logging descriptions. We can observe that the BLEU-1 scores on all the evaluated projects are larger than 35, and the BLEU-1 score on CoreFx is 68.76, which means that 68.76% of the tokens in the generated logging descriptions can be found in the ground truth. The BLEU scores gradually decrease as the n-grams become longer. For example, the BLEU-1 score on Hadoop Pre-10 is 36.59, while the corresponding BLEU-4 score is 16.96. This is reasonable because BLEU-4 score considers the match of consecutive 4 tokens. Besides, the BLEU scores and ROUGE scores for Java projects (Hadoop, Cloudstack, Camel, Hbase, and Hive) and C# (Mono, CoreFx, Monodevelop) are similar, which show that the effectiveness of our approach is robust against different programming languages.

In addition, previously we expect to obtain larger BLEU scores and ROUGE scores with Sur-20 than with Pre-10, since we consider more code statements in the information retrieval process. We are surprised to observe that the BLEU scores and ROUGE scores with Pre-10 are better in most cases. After manual inspection of the corresponding code snippets, we think it is caused by two main reasons. First, considering the succeeding 10 lines of code may bring in some noises, which mislead the method. Recall the manual categorization in Table 2, we can observe that only 17.23% of the logging descriptions are used to explain the succeeding program operation. Second, in this section, we propose a simple information retrieval method based on Levenshtein distance, which gives equal weights to the edit distances of all the tokens. However, in practice,

Table 5: Log generation results.

Dataset	Code Scope	BLEU					ROUGE			
		1	2	3	4	avg	L	1	2	3
Hadoop	Pre-10	36.59	25.57	20.98	16.96	24.02	36.24	36.88	24.26	19.99
	Sur-20	35.30	23.06	18.02	13.93	21.26	35.31	36.14	22.40	17.52
Cloudstack	Pre-10	47.60	36.35	31.17	27.57	34.92	46.05	47.11	34.64	28.97
	Sur-20	45.33	33.41	28.09	24.29	31.88	43.94	45.19	32.07	26.43
Camel	Pre-10	51.98	41.98	36.41	30.74	39.53	49.62	50.23	38.57	33.53
	Sur-20	50.45	39.25	33.21	27.51	36.67	48.46	49.37	36.46	30.79
Hbase	Pre-10	37.69	27.05	22.40	18.28	25.42	37.71	38.47	26.76	22.76
	Sur-20	37.36	25.88	20.93	16.69	24.11	37.55	38.37	25.29	20.69
Hive	Pre-10	40.78	31.26	26.97	23.04	29.83	40.08	40.58	29.83	25.41
	Sur-20	41.42	31.07	26.28	22.37	29.49	40.91	41.55	29.93	25.12
Mono	Pre-10	40.54	31.99	26.27	18.65	28.23	35.95	36.10	27.28	24.69
	Sur-20	38.08	28.54	22.60	15.49	24.83	34.15	34.32	24.63	21.66
CoreFx	Pre-10	68.76	60.69	55.26	49.04	57.99	62.10	62.30	53.12	49.17
	Sur-20	65.28	55.87	50.05	44.00	53.23	57.74	57.96	46.15	41.86
Monodevelop	Pre-10	40.74	31.40	25.68	20.07	28.48	37.86	38.14	29.88	26.95
	Sur-20	41.43	31.75	26.05	21.12	29.14	36.57	36.92	28.25	25.05

some tokens or statements are more important in this context. Thus, the performance of our model is affected.

This paper presents the first step towards automated logging description generation, and thus there is no existing baseline method to compare with. However, in software engineering, some similar tasks also generate natural language text using corresponding code snippets, such as code summarization, which aims to generate a line of text to summarize a code snippet. The BLEU-4 scores reported in the state-of-the-art code summarization papers [31, 40] range from 6.4% to 34.3%. Meanwhile, the BLEU-4 scores of our simple information retrieval-based method range from 16.96% to 49.04% with the "Pre-10" setting, which is encouraging. Note that we do not intend to directly compare the performance of methods for different tasks. However, we want to provide an intuitive understanding of the BLEU scores and ROUGE scores we achieve. Furthermore, we vision plenty of space for improvement by adopting more mature models or specialized feature engineering. However, this is not the focus of this paper, so we leave it as our future work.

Finding 6: A simple information retrieval-based method, which generates logging descriptions by finding similar code snippets, can achieve decent performance in terms of BLEU score and ROUGE score.

6 FUTURE DIRECTIONS

This paper aims to study the usage of natural language in logging practice and further trigger follow-up research work in this field. In this section, we present some potential directions based on our study of software logs.

Improved Information Retrieval Models. To this end, we use a simple character similarity-based method to find suitable natural language descriptions for logging statements, which demonstrates decent performance. However, it has obvious limitations. For example, as a reviewer mentioned, a method being called at different program locations can have different logging descriptions, which cannot be addressed by this model. There are several avenues for extension. First, as explained in [28], very large bodies of code can be readily parsed, typed, scoped, and even subject to simple semantic analysis. Thus, all these data could be used to develop a more sophisticated approach to search similar code snippets. Besides, code clone detection [16, 33, 54] is a classical topic that has been widely studied in software engineering area. Based on the results of our automated logging generation experiments, we believe it is promising to adapt code clone detection methods to further improve the generation performance.

Logging Statement Generation from Code. Although the potential of information retrieval-based logging description generation has been validated in the experiments, the model has some limitations. In particular, it assumes that the current code snippet can be described by an existing logging description. However, new projects often contain only a few logging statements, which may make the proposed model ineffective. This is also a typical problem known as "cold start" in the field of information retrieval. Thus, it will be of great help if we can generate the logging descriptions based on the corresponding code snippets. We vision it is feasible because logging descriptions are mainly used to explain the surrounding codes. To achieve this goal, existing work on code summarization [31, 40], code comment generation [52], and commit message generation [32] are good starting points.

Data Augmentation. Compared with most NLP (natural language processing) applications in software engineering, such as code completion [11, 38], the data volume of "what to log" is not large. As illustrated in Table 1, the largest Java project in our experiments (i.e., Hadoop) contains 1.9M LOC but only 13.9K logging statements. The relatively small data volume could stay in the way of the application of deep learning-based algorithms [14, 24], which dominates many difficult research problems in recent years. We think "what to log" is such a difficult problem, because researchers need to understand the semantic meaning of the corresponding code snippets. Thus, effective data augmentation techniques, which aim at the generation of more training data, is in high demand. The basic idea of data augmentation for "what to log" is to generate more $< code, log >$ pairs using existing data. For example,

a simple data augmentation method is to change the identifier names in a code snippet and keep the original log, which leads to a new $< code, log >$ pair. Towards this end, researchers could start with data augmentation methods in image processing field [29, 30], which have been widely studied.

7 RELATED WORK

7.1 Log Analysis

Software logs contain a wealth of runtime information, and thus have been widely used in various software reliability enhancement tasks, including anomaly detection [18, 57], fault diagnosis [55, 63], program verification [17, 51], performance monitoring [27, 43, 61], etc. Most of these tasks use data mining models to extract critical runtime information from a large volume of software logs. To facilitate log analysis, researchers also focus on related log data preprocessing topics, including log collection [12, 17] and log parsing [25, 26, 42]. These existing papers study how to utilize logs printed by existing logging statements in software. Instead, in this paper, we focus on the design of logging statements, and thus can potentially benefit these log analysis tasks.

7.2 Logging Practice

Current research has mostly focused on the usage of logs printed by existing logging statements, but little on logging itself. Recently, some empirical studies [10, 13, 19, 36, 46, 58, 59] have been conducted to characterize logging practice. Specifically, Yuan et al. [58, 59] study the logging practice of open-source software and further propose proactive logging strategy. [13, 36] characterize the logging practice of Java projects. Considering logging practice in industry, Fu et al. [19] conduct an empirical study on the logging practice of software used in Microsoft. Pecchia et al. [46] study the logging practice in a critical software development process. All these studies provide insightful findings on logging practice, which shed lights into our study of the natural language descriptions in software logs.

7.3 Improving Logging

Towards improving logging practice, there are two categories of work: "where to log" [15, 62] and "what to log" [60]. "Where to log" studies focus on strategic logging, which recommends developers the suitable logging places. Specifically, Cinque et al. [15] propose a logging method based on a set of rules about logging placement, which makes the logs able to detect more software failures. Zhu et al. [62] design a tool LogAdvisor that informs developers whether they should place a logging statement in a code snippet or not. Different from these methods, we focus on the contents of the logging statements, which is the goal of "what to log" research work, instead of logging placement. Yuan et al. [60] propose a tool LogEnhancer that can enhance existing logging statements by augmenting important variables. Li et al. [37] design a regression model to recommend the log level in a logging statement. Our paper also targets on improving the "what to log" part of logging practice. Different from [37, 60], we focus on the natural language descriptions in logging statements. Besides, we present a simple but effective description generation tool. Thus, we believe our study can complement existing logging improving work.

7.4 NLP in Software Engineering

Natural language widely exists in software artifacts, such as design documents, user manuals, bug reports, source code comments, and identifier names [23]. In recent years, various techniques have been proposed by researchers to analyze natural language text for the improvement of modern software engineering. Gabel and Su [20] study the syntactic redundancy of source code, which reveals a general lack of uniqueness in software. Hindle et al. [28] study the naturalness of software, showing that repetitive and predictable regularities of source code can be captured by a simple n-gram language model. Tu et al. [53] further explore the localness characteristics of software. These three studies regard source code as natural language text and study the related characteristics of them. Inspired by these studies, in this paper, we study the characteristics of natural language in logging statements. Additionally, NLP methods have been adapted to many software engineering scenarios, including defect prediction [48], code completion [44], program synthesis [47], API recommendation [22], identifier/method name suggestion [7, 8], etc. Different from these papers, we study "what to log" with the focus on the natural language descriptions in logging statements. We believe this study paves the path for the design of NLP techniques for the "what to log" problem.

8 CONCLUSION

To facilitate software development and maintenance, developers are expected to provide informative and appropriate logging descriptions. However, there is currently a lack of investigations and specifications on studying such descriptions in logging practice. To fill this significant gap, this paper presents an empirical study on the logging statements in 10 Java projects and 7 C# projects, with focus on *what to log*. We summarize with 6 valuable findings, ranging from the logging description's categories, the globally and locally repetitive usage of n-gram patterns, to an encouraging indication towards automated logging description generation. In addition, some valuable directions for improving current logging practice are discussed. In summary, this paper systematically characterizes the natural language descriptions used in logging practice, which serves as the first work towards automated logging description generation. With our datasets released, we hope to trigger related research projects and push this field forward.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their feedback which helped improve this paper. The work described in this paper was fully supported by the National Natural Science Foundation of China (Project Nos. 61332010 and 61472338), the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14234416 of the General Research Fund), the National Basic Research Program of China (973 Project No. 2014CB347701), and Microsoft Research Asia (2018 Microsoft Research Asia Collaborative Research Award).

REFERENCES

[1] [n. d.]. Apache log4net. https://logging.apache.org/log4net/
[2] [n. d.]. Apache Project List. https://projects.apache.org/projects.html
[3] [n. d.]. Datasets. https://github.com/logpai/LoggingDescriptions

[4] [n. d.]. Levenshtein Distance. https://en.wikipedia.org/wiki/Levenshtein_distance

[5] [n. d.]. N-gram. https://en.wikipedia.org/wiki/N-gram

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

[7] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. 2014. Learning natural coding conventions. In *FSE'14: Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[8] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. 2015. Suggesting accurate method and class names. In *FSE'15: Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[9] D. Bahdanau, K. Cho, and Y. Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *ICLR'15: Proc. of the International Conference on Learning Representations*.

[10] T. Barik, R. Deline, S. Drucker, and D. Fisher. 2016. The Bones of the System: A Case Study of Logging and Telemetry at Microsoft. In *ICSE'16: Proc. of the 38th International Conference on Software Engineering*.

[11] A. Bhoopchand, T. Rocktaschel, E. Barr, and S. Riedel. 2016. Learning Python Code Suggestion with a Sparse Pointer Network. In *arXiv preprint arXiv:1611.08307*.

[12] N. Busany and S. Maoz. 2016. Behavioral Log Analysis with Statistical Guarantees. In *ICSE'16: Proc. of the 38th International Conference on Software Engineering (Companion Volume)*.

[13] B. Chen and Z. Jiang. 2017. Characterizing logging practices in Java-based open source software projects - a replication study in Apache Software Foundation. *Empirical Software Engineering* 22 (2017), 330–374. Issue 1.

[14] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *EMNLP'14: Proc. of the Empirical Methods in Natural Language Processing*.

[15] M. Cinque, D. Cotroneo, and A. Pecchia. 2013. Event Logs for the Analysis of Software Failures: A Rule-Based Approach. *IEEE Transactions on Software Engineering (TSE)* 39, 6 (2013), 806–821.

[16] Y. Dang, D. Zhang, S. Ge, R. Huang, C. Chu, and T. Xie. 2017. Transferring Code-Clone Detection and Analysis to Practice. In *ICSE'17: Proc. of the 40th International Conference on Software Engineering (SEIP-track)*.

[17] R. Ding, H. Zhou, J.G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie. 2015. Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis. In *ATC'15: Proc. of the USENIX Annual Technical Conference*.

[18] Q. Fu, J. Lou, Y. Wang, and J. Li. 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *ICDM'09: Proc. of International Conference on Data Mining*.

[19] Q. Fu, J. Zhu, W. Hu, J. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *ICSE'14: Companion Proc. of the 36th International Conference on Software Engineering*. 24–33.

[20] M. Gabel and Z. Su. 2010. A study of the uniqueness of source code. In *FSE'10: Proc. of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[21] Z. Gao, C. Bird, and E. T. Barr. 2017. To Type or Not to Type: Quantifying Detectable Bugs in Javascript. In *ICSE'17: Proc. of the 39th International Conference on Software Engineering*.

[22] X. Gu, H. Zhang, D. Zhang, and S. Kim. 2016. Deep API learning. In *FSE'16: Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[23] S. Haiduc, V. Arnaoudova, A. Marcus, and G. Antoniol. 2016. The Use of Text Retrieval and Natural Language Processing in Software Engineering. In *ICSE'16: Proc. of the 38th International Conference on Software Engineering (Companion Volume)*.

[24] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR'16: Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*.

[25] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. 2016. An Evaluation Study on Log Parsing and Its Use in Log Mining. In *DSN'16: Proc. of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[26] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. 2017. Towards Automated Log Parsing for Large-Scale Log Data Analysis. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2017). https://doi.org/10.1109/TDSC.2017.2762673

[27] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. 2014. Location-based Hierarchical Matrix Factorization for Web Service Recommendation. In *ICWS'14: Proc. of the 21st International Conference on Web Services*.

[28] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. 2012. On the naturalness of software. In *ICSE'12: Proc. of the 34th International Conference on Software Engineering*.

[29] H. Hosseini, B. Xiao, M. Jaiswal, and R. Poovendran. 2017. On the Limitation of Convolutional Neural Networks in Recognizing Negative Images. In *arXiv preprint arXiv:1703.06857*.

[30] H. Inoue. 2018. Data Augmentation By Pairing Samples for Images Classification. In *arXiv preprint arXiv:1801.02929*.

[31] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *ACL'16: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics*.

[32] S. Jiang, A. Armaly, and C. McMillan. 2017. Automatically Generating Commit Messages from Diffs using Neural Machine Translation. In *ASE'17: Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering*.

[33] H. Kim, Y. Jung, S. Kim, and K. Yi. 2011. MeCC: memory comparison-based clone detector. In *ICSE'11: Proc. of the 33rd International Conference on Software Engineering*.

[34] P. Koehn. 2010. *Statistical Machine Translation*. Cambridge University Press.

[35] Robert V Krejcie and Daryle W Morgan. 1970. Determining sample size for research activities. *Educational and psychological measurement* 30, 3 (1970), 607–610.

[36] H. Li, Shang W. Chen, T., and A. E. Hassan. 2017. Studying Software Logging Using Topic Models. *Empirical Software Engineering* (2017).

[37] H. Li, W. Shang, and A. E. Hassan. 2017. Which log level should developers choose for a new logging statement? *Empirical Software Engineering* 22 (2017), 1684–1716. Issue 4.

[38] J. Li, Y. Wang, I. King, and M. R. Lyu. 2017. Code Completion with Neural Attention and Pointer Networks. In *arXiv preprint arXiv:1711.09573*.

[39] C. Y. Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *WAS'04: Proc. of the Workshop on Text Summarization Branches Out*.

[40] P. Loyola, E. Marrese-Taylor, and Y. Matsuo. 2017. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. In *ACL'17: Proc. of the 55th Annual Meeting of the Association for Computational Linguistics*.

[41] M. Luong, H. Pham, and C. D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *EMNLP'15: Proc. of the Empirical Methods in Natural Language Processing*.

[42] A. Makanju, A.N. Zincir-Heywood, and E.E. Milios. 2012. A Lightweight Algorithm for Message Type Extraction in System Application Logs. *TKDE'12: IEEE Transactions on Knowledge and Data Engineering* (2012).

[43] K. Nagaraj, C. Killian, and J. Neville. 2012. structured comparative analysis of systems logs to diagnose performance problems. In *NSDI'12: Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*.

[44] A. T. Nguyen, T. V. Nguyen, H. D. Phan, and T. N. Nguyen. 2018. A Deep Neural Network Language Model with Contexts for Source Code. In *SANER'18: Proc. of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*.

[45] K. Papineni, S. Roukos, T. Ward, and W. Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *ACL'02: Proc. of the 40th Annual Meeting of the Association for Computational Linguistics*.

[46] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo. 2015. Industry practices and event logging: assessment of a critical software development process. In *ICSE'15: Proc. of the 37th International Conference on Software Engineering*. 169–178.

[47] M. Raghothaman, Y. Wei, and Y. Hamadi. 2016. SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In *ICSE'16: Proc. of the 38th International Conference on Software Engineering*.

[48] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. 2016. On the "Naturalness" of Buggy Code. In *ICSE'16: Proc. of the 38th International Conference on Software Engineering*.

[49] A. M. Rush, S. Chopra, and J. Weston. 2015. A Neural Attention Model for Abstractive Sentence Summarization. In *EMNLP'15: Proc. of the Empirical Methods in Natural Language Processing*.

[50] A. See, P. Liu, and C. Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *ACL'17: Proc. of the 55th Annual Meeting of the Association for Computational Linguistics*.

[51] W. Shang, Z. Jiang, H. Hemmati, B. Adams, A.E. Hassan, and P. Martin. 2013. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *ICSE'13: Proc. of the 35th International Conference on Software Engineering*. 402–411.

[52] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *ASE'10: Proc. of the 25th IEEE/ACM International Conference on Automated Software Engineering*.

[53] Z. Tu, Z. Su, and P. Devanbu. 2014. On the localness of software. In *FSE'14: Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[54] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *ASE'16: Proc. of the 31st IEEE/ACM International Conference on Automated Software Engineering*.

[55] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham. 2012. Effective software fault localization using an RBF neural network. *TR'12: IEEE Transactions on Reliability* (2012).

[56] W. Xu. 2010. *System Problem Detection by Mining Console Logs*. Ph.D. Dissertation. University of California, Berkeley.

[57] W. Xu, L. Huang, A. Fox, D. Patterson, and M.I. Jordon. 2009. Detecting Large-Scale System Problems by Mining Console Logs. In *SOSP'09: Proc. of the ACM Symposium on Operating Systems Principles*.

[58] D. Yuan, S. Park, P. Huang, Y. Liu, M. Lee, X. Tang, Y. Zhou, and S Savage. 2012. Be conservative: enhancing failure diagnosis with proactive logging. In *OSDI'12: Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*. 293–306.

[59] D. Yuan, S. Park, and Y. Zhou. 2012. Characterizing logging practices in open-source software. In *ICSE'12: Proc. of the 34th International Conference on Software Engineering*. 102–112.

[60] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. 2011. Improving software diagnosability via log enhancement. In *ASPLOS'11: Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–14.

[61] A. Zhou, S. Wang, Z. Zheng, C. Hsu, M. R. Lyu, and F. Yang. 2016. On cloud service reliability enhancement with optimal resource usage. *IEEE Transactions on Cloud Computing (TCC)* 4 (2016), 452–466.

[62] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In *ICSE'15: Proc. of the 37th International Conference on Software Engineering*.

[63] D. Q. Zou, H. Qin, and H. Jin. 2016. UiLog: Improving Log-Based Fault Diagnosis by Log Analysis. *Journal of Computer Science and Technology* 31, 5 (2016), 1038–1052.