Do Bugs Lead to Unnaturalness of Source Code?

Yanjie Jiang
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
yanjiejiang@bit.edu.cn

Hui Liu*
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
liuhui08@bit.edu.cn

Yuxia Zhang*
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
yuxiazh@bit.edu.cn

Weixing Ji
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
jwx@bit.edu.cn

Hao Zhong
Shanghai Jiao Tong University
Shanghai, China
zhonghao@sjtu.edu.cn

Lu Zhang
Key Laboratory of High Confidence Software Technologies, Peking University
Beijing, China
zhanglu@sei.pku.edu.cn

ABSTRACT

Texts in natural languages are highly repetitive and predictable because of the naturalness of natural languages. Recent research validated that source code in programming languages is also repetitive and predictable, and naturalness is an inherent property of source code. It was also reported that buggy code is significantly less natural than bug-free one, and bug fixing substantially improves the naturalness of the involved source code. In this paper, we revisit the naturalness of buggy code and investigate the effect of bug-fixing on the naturalness of source code. Different from the existing investigation, we leverage two large-scale and high-quality bug repositories where bug-irrelevant changes in bug-fixing commits have been explicitly excluded. Our evaluation results confirm that buggy lines are often less natural than bug-free ones. However, fixing bugs could not significantly improve the naturalness of involved code lines. Fixed lines on average are as unnatural as buggy ones. Consequently, bugs are not the root cause of the unnaturalness of source code, and it could be inaccurate to identify buggy code lines solely by the naturalness of source code. Our evaluation results suggest that the naturalness-based buggy line detection results in extremely low precision (less than one percentage).

CCS CONCEPTS

• **Software and its engineering → Software post-development issues**.

KEYWORDS

Code Entropy, Bugs, Source Code, Bug Fixing, Naturalness

*corresponding authors

ACM Reference Format:
Yanjie Jiang, Hui Liu, Yuxia Zhang, Weixing Ji, Hao Zhong, and Lu Zhang. 2022. Do Bugs Lead to Unnaturalness of Source Code?. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549149

1 INTRODUCTION

Natural languages are often highly repetitive and predictable [21]. Consequently, a few approaches have been proposed to model natural languages [10, 20, 37]. N-gram [14] is one of the most well-known statistic models for natural languages. It is based on the assumption that the next token is determined by the preceding n tokens. It learns the probability for the token t to appear after $t_1 t_2 \ldots t_n$ from a large corpus. The trained n-gram model could be used to predict the next token in typewriting and to facilitate other natural language processing tasks, e.g, text classification [32, 40, 45, 46] and text compression [15, 35, 39].

Programming languages, as a special kind of human-oriented languages, are often repetitive and predictable as well [24]. Hindle et al. modeled Java programs and C programs with a widely used language model, i.e., n-gram. Their evaluation results suggest that source code is far more repetitive and more predictable than natural language (English). The n-gram based cross-entropy on source code is significantly lower than that of English documents, which suggests that n-gram model works better on source code than on English documents. Consequently, Hindle et al. [24] leveraged n-gram models to predict the next token in source code. Their evaluation results suggest that n-gram based code complete is significantly more accurate than the widely used built-in code complete in Eclipse [6]. Besides, natural language models have been successfully exploited to migrate source code from one programming language to another [36, 37, 43], to learn coding styles from source code [10, 11, 13], and to guide code generation [16].

Inspired by Hindle et al. [24], Ray et al. [42] investigated whether the n-gram based cross-entropy (called *code entropy* for short) could be leveraged to identify buggy code lines. The key hypothesis is that buggy source code is often less natural and thus has higher entropies than bug-free code (and the fixed version of the buggy code as well). To validate this hypothesis, they identified bug-fixing commits in version control systems, identified bug-free code lines, buggy code lines (i.e., code lines modified or removed by the bug-fixing commits), and the fixed lines (i.e., lines modified or inserted by the bug-fixing commits) associated with the bug-fixing commits. They computed the entropies of the code lines automatically. By comparing the entropies of buggy lines and bug-free lines, they found that buggy lines often have higher entropies. They also compared buggy lines against their fixed version. The comparison suggests that bug-fixing commits increase the naturalness of code lines. The two findings together suggest that bugs lead to unnaturalness of source code and fixing bugs makes code natural again. Based on these findings, they proposed an approach to identify buggy code by the entropy of source code. Their evaluation results suggest that such an entropy-based approach is comparable to the state-of-the-art static bug finders.

In this paper, we revisit the naturalness of buggy code and investigate the impact of bug-fixing operations on the naturalness of source code. Different from the existing investigation [42], we exploit two large-scale and high-quality bug repositories, Defects4J [28] and GrowingBugs [7]. Defects4J is a highly popular repository whose patches have been manually validated and bug-irrelevant changes in bug-fixing commits have been manually excluded. Notably, on average 37% of the changes in bug-fixing commits are bug-irrelevant [26]. GrowingBugs is similar to Defects4J. The major difference is that GrowingBugs employs an automated approach, called *BugBuilder* [26], to exclude bug-irrelevant changes automatically. Based on the concise patches in both Defects4J and GrowingBugs, we can accurately identify bug-free code lines, buggy lines, and fixed lines automatically, which in turn facilitates the analysis on the naturalness of buggy lines and the impact of bug-fixing operations.

By analyzing the 809 bug-fixing commits in Defects4J and 487 bug-fixing commits in GrowingBugs, we confirm that buggy lines are often less natural than bug-free ones. However, fixing the bugs could not significantly improve the naturalness of involved code lines: The average entropy is even increased by bug fixing operations. The evaluation results may suggest that although buggy lines are less natural, bugs in such code lines are not the root causes of their low naturalness. Consequently, it could be inaccurate to identify buggy code lines by the naturalness of source code: it results in extremely low precision (less than one percentage). To facilitate replication and further analysis on code naturalness, we make the replication package for our empirical study (including the datasets, scripts, and tools involved in the study) publicly available on GitHub [9].

The rest of this paper is structured as follows. Section 2 introduces related work on the naturalness. Section 3 presents the setup of our empirical study, and Sections 4-6 present the results of our empirical study. Section 7 discusses related issues and limitations whereas Section 8 makes conclusions.

2 RELATED WORK

2.1 Naturalness of Source Code

Hindle and his colleagues [23, 24], under the support of NSF grants led by Devanbu [18], are the pioneers in the field of code naturalness. They applied a language model (i.e., n-gram) to Java and C source code, and their evaluation results suggest that this language model works well on source code. To quantify how well n-gram works on source code, they leveraged *cross-entropy* of source code. Given a document (or a piece of source code) $s = a_1 \dots a_n$ of length n and a language model $\mathcal{M}$, the probability of $s = a_1 \dots a_n$ computed by the model $\mathcal{M}$ is noted as $p_{\mathcal{M}}(s)$. The cross-entropy (called *entropy* for short in the rest of this paper) of the document s is computed as follows:

$$\begin{aligned} H_{\mathcal{M}}(s) &= -\frac{1}{n}\log p_{\mathcal{M}}(s) \\ &= -\frac{1}{n}\log p_{\mathcal{M}}(a_1 \dots a_n) \\ &= -\frac{1}{n}\sum_{i=1}^{n}\log p_{\mathcal{M}}(a_i|a_1 \dots a_{i-1}) \end{aligned} \tag{1}$$

Tu et al. [44] further improved the n-gram model on source code by exploiting the localness of source code. Source code is often more repetitive within a small module, e.g., a method, a class, or a package. For example, different methods within the same class often share many common tokens whereas source code in different projects shares few common tokens besides keywords of programming languages. Consequently, while predicting the next token for source code, we should exploit the localness of source code, i.e., the contexts of the source code. To this end, Tu et al. [44] proposed the cache model as follows:

$$\begin{aligned} p(a_i|a_1 \dots a_{i-1}, cache) = \lambda \cdot p_{ngram}(a_i|a_1 \dots a_{i-1}) \\ + (1-\lambda) \cdot p_{cache}(a_i|a_1 \dots a_{i-1}) \end{aligned} \tag{2}$$

where *cache* is the list of n-grams in the cache (e.g., enclosing files), p_{ngram} is the prediction by a traditional n-gram model trained with all source code, and p_{cache} is the prediction by n-gram models trained with *cache* only. λ assigns different weights to the global model p_{ngram} and local model p_{cache}. Their evaluations suggest that such a hybrid model outperforms the traditional n-gram model.

Musfiqur et al. [41] revisited the naturalness of source code by investigating how language specific syntax tokens (e.g., semi-colons and brackets in Java) influence the naturalness of source code. Their evaluation results suggest that syntax tokens are popular, accounting for 59% of Java tokens. Removing such tokens could significantly reduce the naturalness of the source code. For example, removing Java-specific tokens increases the entropy of Java programs by more than 90% (evaluated with 3-gram).

We conclude that source code is natural and cross-entropy is widely employed to measure the naturalness of source code.

2.2 Naturalness of Abnormal Source Code

If normal source code is natural, it is likely that abnormal source code, e.g., buggy code and smelly code, could be unnatural. To validate this hypothesis, Ray et al.[42] investigated whether buggy code is less natural than bug-free code, and whether such unnaturalness could be leveraged to predict defects in source code. They selected 10 open-source Java projects from GitHub and Apache Software Foundation, and identified bug-fixing commits by checking keywords (e.g., 'error', 'fix', and 'bug') in commit messages. From the resulting bug-fixing commits, they automatically identified *bug-free lines*, *buggy lines*, and *fixed lines*:

Bug-free lines: Code lines in the faulty programs that are not influenced by the associated bug-fixing operations.

Buggy lines: Code lines in the faulty programs that have been deleted or changed by the bug-fixing operations.

Fixed lines: Code lines in the fixed version that have been added or changed by the bug-fixing operations.

Ray et al. [42] computed code entropies by the cache-based language model (n-gram) proposed by Tu et al. [44] (we call it *cached model* for short in this paper). Notably, they did not compute the entropy of a code line directly by Equation 1 or its variant (i.e.,the cached model). Instead, they computed the entropy for each token on the code line with the cached model, and assigned the average entropy of the tokens on a line as the final entropy of the code line. Because the cached model does not consider types of source code, more repetitive statements like for-loop statements and catch clauses often have lower entropies than other less repetitive statements, e.g., method declarations. To balance the entropies of different statements, they adjusted the entropies of code lines as follows:

$$H'_{\mathcal{M}}(l, type) = \frac{H_{\mathcal{M}}(l) - \mu_{type}}{SD_{type}} \quad (3)$$

where $H_{\mathcal{M}}(l)$ is the code entropy of line l computed with the cached n-gram model [24, 44]. $type$ is the statement type of code line l, μ_{type} denotes mean entropy of the lines of the given type, and SD_{type} denotes standard deviation. The type of a code line is the type of the lowest AST node encompassing the full line [42]. $H'_{\mathcal{M}}(l, type)$ is the type-based code entropy of line l, measuring how natural the line is with regard to other lines of the same type. Empirical study conducted by Ray et al. [42] suggests that buggy lines are significantly less natural than bug-free ones, and bug fixing can significantly improve the naturalness of source code. They also proposed an entropy-based approach to detect buggy lines that has been proved comparable to the state-of-the-art static bug finders.

Inspired by the findings reported by Ray et al. [42] (i.e., buggy code lines are less natural than bug-free ones), Matthieu et al. [27] investigated how program mutation influences the naturalness of source code, and how the naturalness of the mutants may help mutant selection. They leveraged Major [29] to create mutants automatically, and their evaluation results suggest that "*mutants sometimes make the code more natural and sometimes less natural*" [27]. From the generated mutants, they tried to select the most useful mutants according to 1) the naturalness of the mutants, 2) the naturalness of the code where mutation operations have been conducted, and 3) the decrease of naturalness caused by the mutation operations. However, their evaluation results suggest that naturalness does not help in mutant selection: Such naturalness-based approach fails to outperform random selection in selecting fault-revealing mutants.

Following the findings reported by Ray et al. [42], Bin et al. [34] investigated whether naturalness of source code could be exploited for code smell detection, i.e., identifying source code where software refactorings are needed. Smelly code refers to badly designed but functionally working code snippets that should be restructured (refactored). Similar to buggy code, such smelly code could be less natural than well-designed source code. To this end, they collected a corpus of refactorings and analyzed how such refactorings changed the naturalness of smelly code. Their evaluation results suggest, however, smelly code is not necessarily less natural than its revised (improved) version. On 44.5% of the cases, smelly code is even more natural than its improved version.

Table 1: Selected Bug Repositories

	Defects4J	GrowingBugs
# Bugs	809	487
# Projects	16	135
# Buggy-Free Lines	81,907,946	38,704,122
# Buggy Code Lines	2,563	893
# Fixed Code Lines	6,642	2,332
# Test Cases	2,473,481	828,746

We conclude that buggy code was found less natural [42], and this finding has stimulated a new line of research investigating how unnaturalness is associated with abnormal source code (e.g., buggy code, smelly code, and mutants) and how such unnaturalness could be exploited to identify anomalies. However, such investigations often led to unexpected negative results [27, 34], which prompts us to revisit the naturalness of buggy code in this paper.

3 SETUP

3.1 Research Questions

To investigate the naturalness of buggy code, our empirical study investigates the following research questions:

- **RQ1**: Are buggy code lines less natural than bug-free ones?
- **RQ2**: Is it accurate to identify buggy lines by code entropy?
- **RQ3**: Does bug fixing improve the naturalness of source code? If yes, is the fixed code as natural as bug-free code?

RQ1 investigates the naturalness of buggy code. More specifically, it validates whether buggy code lines are often less natural than bug-free ones. Although this issue has been investigated by Ray et al. [42], revisiting this issue with different datasets is still valuable, especially when our datasets are of higher quality. **RQ2** investigates the usefulness of code entropy. More specifically, it validates whether code entropy could be exploited for automated identification of buggy code. If buggy code is significantly less natural than others, it is likely that we can identify buggy code by the naturalness (entropy) of source code. **RQ3** investigates the impact of bug-fixing on the naturalness of source code. More specifically, it validates whether fixed lines are more natural than buggy ones and whether the fixed lines are as natural as bug-free ones. Answering this question may reveal whether the unnaturalness of buggy code is caused by bugs.

3.2 Bug Repositories

Our empirical study is conducted on two large-scale and high-quality bug repositories. An overview of the selected bug repositories is presented on Table 1. The first bug repository is Defects4J [28]. This repository contains 835 bugs collected from 17 real-world applications. Defects4J identifies bug-fixing commits in version control systems by comparing commit messages against bug report IDs in bug tracking systems. Identifying bug-fixing commits via bug

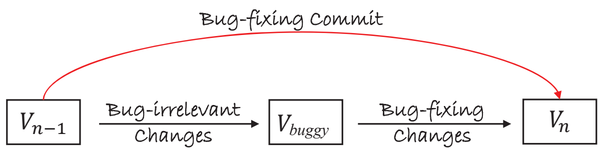

Figure 1: Bug-fixing Changes in Bug-Fixing Commits

tracking systems could be much more reliable than key-word based approaches (like what Hindle et al. [24] do) because the tracking systems have been manually validated. Notably, some bug-irrelevant commits may contain keywords "fix" or "bug", which may result in false positives for keyword-based approaches. For example, the commit [5] from Commons-IO is not a bug-fixing commit although its message ("*Fix whitespace.*") does contain keyword "Fix". The commit did nothing but changed a comment from "* *Handles an Exception .*" to "* *Handles an Exception.*" by removing the last white space before the period.

For each of the bug-fixing commits, Defects4J automatically retrieves the buggy version (noted as V_{n-1}) and the fixed version (noted as V_n) of the associated software application from version control systems. Defects4J requests experts to manually identify bug-irrelevant changes within the bug-fixing commit. As a result of the manual exclusion, Defects4J generates a new version of the application (called V_{buggy} by Defects4J): Applying all bug-irrelevant changes in the commit to V_{n-1} should result in V_{buggy} whereas applying all bug-fixing changes in the commit to V_{buggy} should result in V_n. The relations among V_{n-1}, V_{buggy}, and V_n are visualized in Figure 1. Notably, only 809 out of the 835 bugs in Defects4J are finally employed for the empirical study. All bugs from project Chart [3] (26 bugs in total) are excluded because the commit IDs associated with these bugs are invalid. We fail to retrieve the commits specified by such commit IDs.

To improve the generalizability of the evaluation results, we employ another bug repository (called GrowingBugs [7]) besides Defects4J for the evaluation. The latest version of GrowingBugs (when we accessed it for empirical study) contained 487 real-world bugs collected from real-world open-source applications. It is highly similar to Defects4J in that bug-irrelevant changes in bug-fixing commits have been excluded from the patches in both GrowingBugs and Defects4J. The relations among V_{n-1}, V_{buggy}, and V_n as visualized in Figure 1 holds as well on GrowingBugs. The only difference between GrowingBugs and Defects4J is that the latter excludes bug-irrelevant changes from bug-fixing commits manually, whereas the former does it automatically by BugBuilder [25, 26].

The two bug repositories are selected for evaluation because both of them are of high quality. First, they identify bug-fixing commits via bug tracking systems and version control systems, which is much more reliable than keyword-based alternatives. Second, they explicitly exclude bug-irrelevant changes from patches. As a result, we can automatically and accurately identify buggy code (and their fixed version) according to the patches provided by the bug repositories. The accuracy in the identification of the buggy lines is critical for the evaluation, which may significantly influence the conclusions drawn by the evaluation. Third, the bug repositories explicitly exclude changes on test cases from patches.

4 BUGGY LINES ARE LESS NATURAL

In this section, we investigate whether buggy lines are less natural than bug-free ones.

4.1 Process

According to Figure 1, *bug-free lines*, *buggy lines*, and *fixed lines* in involved bug repositories are identified as follows:

Bug-Free Lines: Code lines in V_{buggy} that are not influenced by the associated bug-fixing operations.

Buggy Lines: Code lines in V_{buggy} that have been deleted or changed by the bug-fixing operations.

Fixed Lines: Code lines in V_n that have been added or changed by the bug-fixing operations.

Notably, bug-fixing operations are the code editions turning V_{buggy} into V_n. They are a subset of the whole bug-fixing commit that may contain some bug-irrelevant changes as well. Notably, we do not exploit any information in V_{n-1} while investigating RQ1.

We compute the entropy for each of the bug-free lines and buggy lines in the same way as Ray et al. [42] did. To make it self-contained, we specify the computation as follows. First, to compute the entropies of code lines in file *FA* of project *PA*, we train an n-gram with all files from project *PA* except for *FA*. Notably, projects in the selected bug repositories are large, containing millions of lines of source code. Consequently, the n-gram could be well-trained even with source code from a single project. The resulting n-gram is then employed to compute entropies for each token in *FA* with the document *FA* as a cache [44]. The raw entropy of a code line is the average entropies of the tokens on the same line. Finally, the raw entropy of a code line is normalized according to Equation 3. To be consistent with the empirical study conducted by Ray et al. [42], we not only follow their way in computing entropies, but also employ the same tool/implementation of the cache-based n-gram model. All of the data, scripts, and tools involved in the study are publicly available on GitHub [9].

Notably, the numbers of bug-free lines and buggy lines are large, which makes it difficult to interpret the results of significance tests because of the well-known *too-large sample size problem* [31]: Too large sample size may amplify the statistical differences. Reducing the size by sampling may not help because it would result in unstable p-values depending on the size of the samples. Consequently, in this paper, we do not conduct significance tests if bug-free lines are involved because the number of bug-free lines is too large (more than 30 millions). In contrast, we visualize the difference with beanplots and assess the difference with statistical values (e.g., averages and medians).

4.2 Results and Analysis

The results are presented on Table 2 and Figure 2 (Bean Plot). From Table 2, we observe that on average buggy lines are less natural, i.e., with higher entropy than bug-free ones. On Defects4J, the average entropy of buggy lines is 0.18, substantially higher than that (0.0011) of bug-free lines. In addition, the median (0.07) is also greater than that (-0.2273) of bug-free lines. On GrowingBugs, the average line entropy of buggy lines is 0.24, substantially higher than that (0.0017) of bug-free lines, too. The median (0.19) is also greater than that (-0.15) of bug-free lines. Notably, the average entropy of buggy

Table 2: Code Entropy of Buggy and Buggy-Free Lines

	Defects4J		GrowingBugs	
	Buggy Lines	Bug-Free Lines	Buggy Lines	Bug-Free Lines
Number of Lines	2,563	81,907,946	893	38,704,122
Average Entropy	0.18	0.0011	0.24	0.0017
Maximal Entropy	4.94	9.9957	5.82	9.9976
Minimal Entropy	-2.42	-3.994	-2.04	-9.9287
The First Quartile (Q1)	-0.62	-0.75	-0.52	-0.82
The Second Quartile (Median)	0.07	-0.2273	0.19	-0.15
The Third Quartile (Q3)	0.87	0.6025	0.89	0.74

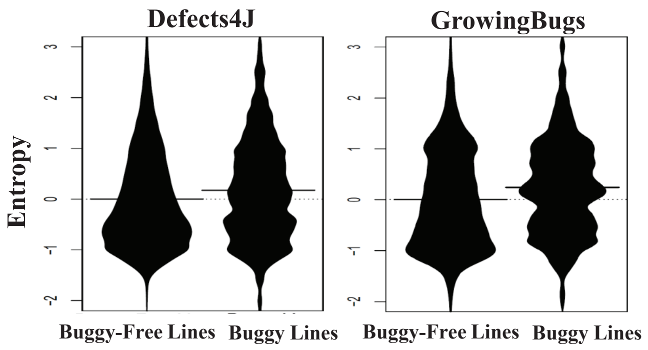

Figure 2: Distribution of Line Entropy (Bean Plot)

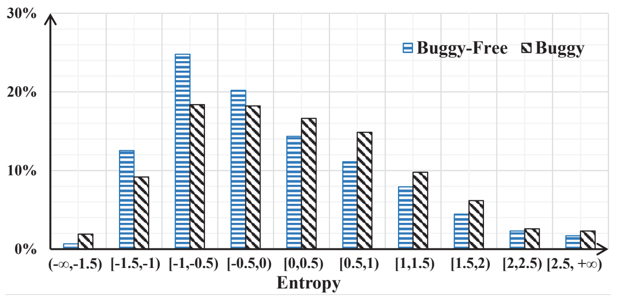

Figure 3: Probability Histogram of Line Entropy (Defects4J)

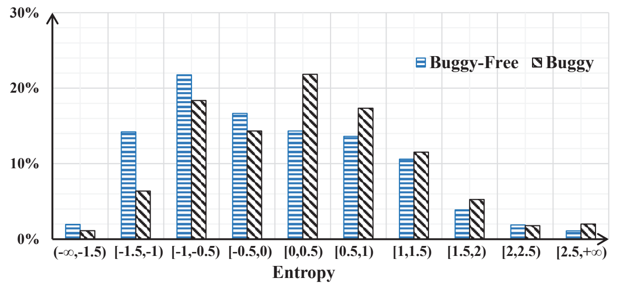

Figure 4: Probability Histogram of Line Entropy (GrowingBugs)

lines in different bug repositories is comparable (0.18 versus 0.24). The same is true for bug-free lines (0.0011 versus 0.0017). It may suggest that the conclusion (i.e., buggy lines are less natural) holds regardless of bug repositories.

From Table 2, we also observe that the number of bug-free lines is substantially larger than that of buggy lines. Defects4J contains more than 81 million bug-free lines whereas less than three thousand lines are buggy. Similarly, GrowingBugs contains more than 38 million bug-free lines whereas the number of buggy lines is less than one thousand.

Besides the comparison on statistic metrics on Table 2, we also employ the bean plot in Figure 2 and probability histograms in Figure 3 and Figure 4 to specify and compare the distribution of the line entropies. From these figures, we make the following observations:

- First, most of the bug-free lines are of low entropies, resulting in a pear-shaped bean in Figure 2 (especially in Defects4J). In contrast, the distribution of buggy lines' entropy is more balanced, suggesting that a substantial part of the buggy lines are of high entropy. It confirms that buggy lines are often less natural.
- Second, the probability histograms in Figure 3 and Figure 4 suggest that buggy lines are more likely to have higher entropy. The horizontal axes of the histograms specify the ranges (also known as bins) of line entropy whereas the vertical axes specify how many percentages of the buggy lines (or bug-free lines) fall into the given bins. From Figure 3 and Figure 4, we can observe that buggy lines have substantially greater chance than bug-free ones to fall into the bins where line entropy is greater than zero. In contrast, bug-free lines have substantially greater chance than buggy lines to fall into bins where line entropy is smaller than zero. However, we also notice from the probability histograms that the difference is smaller than one order of magnitude. The maximal difference appears on the bin [2.5, +∞) (on GrowingBugs) where the probability (2.02%) of buggy lines is 1.83 times of that (1.1%) of bug-free ones.
- Third, not all unnatural lines are buggy. Most unnatural lines are bug-free indeed. In Defects4J, the maximal entropy (9.9957) of bug-free lines is higher than that (4.94) of buggy lines. Similarly, the maximal entropy (9.9976) of bug-free

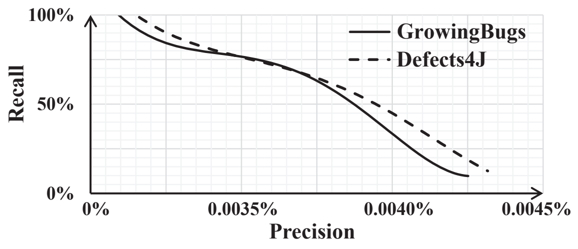

Figure 5: Identifying Buggy Lines by Entropy

lines in GrowingBugs is also higher than that (5.82) of buggy lines.

We conclude based on the preceding analysis that buggy lines on average are less natural than bug-free ones, which is consistent with the findings reported by Ray et al. [42]. However, we also notice that not all unnatural lines are buggy.

5 IDENTIFYING BUGGY LINES BY CODE ENTROPY

In this section, we investigate whether it is accurate to identify buggy code lines by code entropy.

5.1 Process

Since the evaluation results in the preceding section suggest that buggy code lines are often less natural than others, it is likely that we can leverage code entropies (naturalness) to identify buggy code lines, and thus we can inspect such lines first to improve the efficiency in software quality insurance. An intuitive and straightforward approach to identify buggy lines by entropy is to report all code lines whose entropy is greater than a threshold β as buggy lines whereas others are reported as bug-free. We call it entropy-based identification of buggy lines, or *EIBL* for short.

The setting of the threshold β may significantly influence the performance of the approach. More specifically, it may change the precision and recall in opposite directions. Consequently, we are more interested in how the precision and recall change synchronously and whether the approach could result in high precision and high recall at the same time with some optimal setting of the threshold. To this end, we keep changing β automatically, and depict the relationship between the precision and recall of *EIBL*. The relationship may reveal the potential of the approach, and how the performance is influenced by the changing threshold. We also leverage widely-used Receiver Operating Characteristic (ROC) curve [19] to illustrate the potential of *EIBL* because *EIBL* is essentially a binary classifier and ROC curve is frequently employed to illustrate the diagnostic ability of binary classifiers.

5.2 Results and Analysis

The results are presented in Figure 5 and Figure 6. Figure 5 presents the relationship between precision and recall whereas Figure 6 presents the ROC curves on the two bug repositories. The vertical axis in Figure 6 specifies the true positive rate of the classifier, i.e., the percentage of positive items (buggy lines) are correctly predicted as positive (buggy). The horizontal axis specifies the false

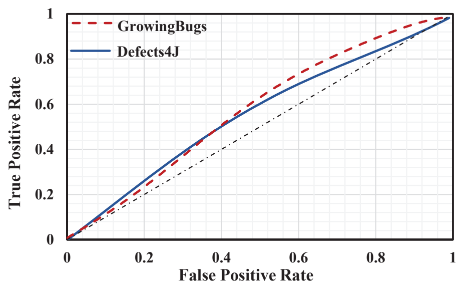

Figure 6: Identifying Buggy Lines by Entropy (ROC Curves)

positive rate, i.e., the percentage of negative items (bug-free lines) are incorrectly predicted as positive (buggy). The area under a ROC curve is called AUC (Area Under the ROC Curve), that is a well-known performance metrics to compare binary classifiers.

From Figure 5, we observe that the precision is extremely low. The precision keeps lower than 0.005% while the recall varies significantly with the changing threshold β. The results suggest that it could result in numerous false positives if we identify buggy lines by code entropy/naturalness only no matter how the threshold β is set. From the figure, we also observe that the performance on Defects4J and GrowingBugs is highly similar.

The ROC curves on Figure 6 confirm the preceding observation that it is inaccurate to identify buggy lines by naturalness alone. The ROC curves are close to the diagonal (dashed black line) that represents the ROC curve of random guess. The AUC is 0.5473 on Defects4J and 0.5661 on GrowingBugs, only slightly greater than AUC of random guess (0.5). All such observations suggest that it is inaccurate to identify buggy lines by naturalness alone, and its performance is only slightly better than random guess.

The major reason for the inaccuracy of the entropy-based buggy line identification is that the total number of bug-free lines is significantly larger than that of buggy lines. The precision of the approach could be formalized as follows:

$$precision = \frac{TP}{TP + FP} \tag{4}$$

$$TP = N_2 \times p_2 \tag{5}$$

$$FP = N_1 \times p_1 \tag{6}$$

where N_1 and N_2 are the total numbers of bug-free lines and buggy lines, respectively. p_1 and p_2 specify the percentages of the bug-free lines and buggy lines have high entropy (higher than threshold β), respectively. From Figure 3 and Figure 4, we know that p_2 is significantly greater than p_1 (by less than one order of magnitude) because buggy lines on average have higher entropy. However, N_1 is significantly larger than N_2 (by around four orders of magnitude). As a result, FP (=$N_1 \times p_1$) is significantly larger than TP (=$N_2 \times p_2$), which results in extremely small precision (=$TP/(TP + FP)$) that is even lower than one percentage.

For example, in Defects4J, there are more than 81 million bug-free lines whereas there are only 2,563 buggy lines. The former is larger than the latter by four orders of magnitude. In Defects4J, up to 32,395,683 bug-free lines have higher entropy than the median entropy (0.07) of buggy lines. In contrast, only half (1,282) of the bug-free lines have such great entropy. As a result, if we set

Table 3: Number of Lines with High Entropy

Minimal Entropy	Defects4J		GrowingBugs	
	#Buggy Lines	#Bug-Free Lines	#Buggy Lines	#Bug-Free Lines
$E > 5.82$	0	57,353	0	11,629
$E > 2.5$	59	1,392,820	18	427,322
$E > 2$	125	3,290,250	34	1,155,588
$E > 1$	534	13,413,775	184	6,764,236
$E > 0$	1,341	34,267,764	532	17,430,392

threshold $\beta = 0.07$, we have $TP = 1,282$, $FP = 32,395,683$, and $precision = TP/(TP + FP) = 1,282/(1,28 + 32,395,683) = 0.004\%$ Similarly, in GrowingBugs, up to 15,343,063 bug-free lines have higher entropy than the median entropy (0.19) of buggy lines. In contrast, only half (447) of the bug-free lines have such great entropy. As a result, if threshold $\beta = 0.19$, the precision is as low as 0.003%=447/(447+15,343,063). Table 3 presents a detailed comparison between the numbers of high-entropy buggy lines and bug-free lines. The first column specifies the minimal entropy that a code line should have to be counted as a *high-entropy line*. The other columns specify how many buggy or bug-free lines could be counted as *high-entropy line* with the given threshold specified by the first column. From this table, we observe that the number of high-entropy bug-free lines is always substantially larger than that of high-entropy buggy lines regardless of the setting of the minimal entropy. It may suggest that the entropy-based identification of buggy lines would result in low precision regardless of the setting of the threshold β.

We conclude based on the preceding analysis that it could be inaccurate to identify buggy lines by entropy alone, although buggy lines on average have higher entropy.

6 BUG FIXING'S EFFECT ON CODE ENTROPY

In this section, we investigate whether bug fixing can significantly improve the naturalness of buggy code.

6.1 Process

If bugs are the root cause of high entropy of buggy lines, it is likely that the entropy of such lines could be significantly reduced by bug fixing, and it is likely that the fixed lines could be as natural as other bug-free lines because the fixed lines are bug-free as well. To validate these hypotheses, we identify buggy code lines and their corresponding fixed lines, and compare their entropies to reveal how bug fixing influence the code entropies.

To further investigate the impact of bug fixing operations on the naturalness of involved source code, we divide the bug fixing operations and investigate how each category of the bug fixing operations influence the naturalness of source code. Bug-fixing is further divided into deletion (of buggy lines), insertion (of fixed lines), and modification (of buggy lines). Accordingly, buggy lines could be further divided into *deleted buggy lines* and *modified buggy lines* whereas fixed lines are further divided into *inserted fixed lines* and *updated fixed lines*:

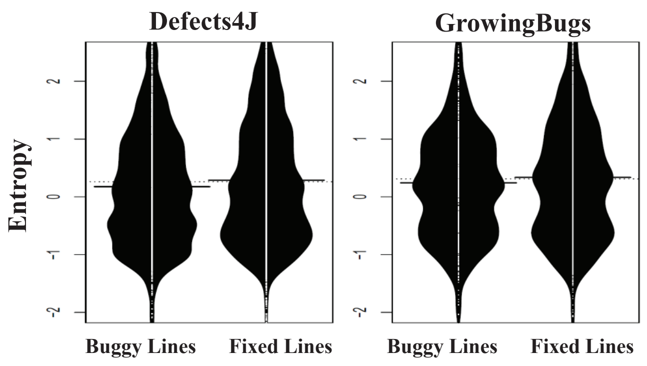

Figure 7: Entropy of Buggy and Fixed Lines (Bean Plot)

Deleted buggy lines: Code lines in V_{buggy} that are deleted by the associated bug-fixing operations.

Modified buggy lines: Code lines in V_{buggy} that are modified by the bug-fixing operations.

Inserted fixed lines: Code lines in V_n that are inserted by the bug-fixing operations.

Updated fixed lines: Code lines in V_n that are modified by the bug-fixing operations. They are the updated version of modified buggy lines in V_{buggy}.

For the example patch [2] on Listing 1, Line 8 is a *deleted buggy line* (red lines beginning with "-"), Line 9 and Line 10 are *inserted fixed line* (green lines beginning with "+"), Line 6 is a *modified buggy line*, and Line 7 is an *updated fixed line*. Notably, we distinguish modified buggy lines via standard `git` common: `git diff -word-diff=plain`. This common explicitly specifies which lines have been modified and which lines have been removed (or replaced with brand-new lines).

```
1   /* This patch has been applied to project Gson to fix Bug-17*/
2
3   final class DefaultDateTypeAdapter extends TypeAdapter<Date
         > {
4     ......
5     public Date read(JsonReader in) throws IOException{
6   -   if ( in.peek() != JsonToken.STRING) {
7   +   if ( in.peek() == JsonToken.NULL) {
8   -     throw new JsonParseException("The date should be a string value");
9   +     in.nextNull();
10  +     return null;
11    }
12    ......
```

Listing 1: Code Lines Influenced by Bug Fixing

6.2 Fixed Lines Remain Unnatural

We first take all bug fixing operations as a whole, and compare the entropy of buggy lines against the entropy of fixed lines. The results are presented on Table 4. We also visualize the results with bean plots in Figure 7. From Table 4 and Figure 7, we make the following observations.

First, bug fixing surprisingly fails to improve the naturalness of buggy code, and the average entropy of fixed lines is even higher than that of buggy lines. On Defects4J, the average entropy of fixed lines is 0.29, substantially higher than that (0.18) of buggy lines. Similarly, on the other bug repository GrowingBugs, the average entropy of fixed lines is 0.34, also substantially higher than that

Table 4: Code Entropy of Buggy and Fixed Lines

	Defects4J		GrowingBugs	
	Buggy Lines	Fixed Lines	Buggy Lines	Fixed Lines
Number of Lines	2,563	6,642	893	2,332
Average Entropy	0.18	0.29	0.24	0.34
Maximal Entropy	4.94	7.05	5.82	7.08
Minimal Entropy	-2.42	-2.46	-2.04	-2.39
The First Quartile (Q1)	-0.62	-0.60	-0.52	-0.54
The Second Quartile (Median)	0.07	0.13	0.19	0.26
The Third Quartile (Q3)	0.87	1.05	0.89	1.05

(0.24) of buggy lines. The median is also increased substantially from 0.07 to 0.13 (on Defects4J) and 0.19 to 0.26 (on GrowingBugs) by the bug-fixing operations, respectively. Following Ray et al. [42], we also conduct Wilcoxon non-parametric tests and compute the Cohen's D effect size to validate the difference. On Defects4J, the results of the test (p-value=0.000566 $\ll$ 0.05) suggest that fixed lines are significantly less natural than buggy ones, although Cohen's D size (0.0359) suggests that the difference is small. On the other bug repository GrowingBugs, however, the results of the test (p-value =0.103 $>$ 0.05, Cohen's Size = 0.0287) suggest that the difference is not significant. Anyway, we do not find any evidence on any bug repository that bug fixing can significantly improve the naturalness of buggy code. The results may suggest that software bugs within source code are not the root cause of the unnaturalness of buggy code lines: Otherwise, the naturalness should have increased after bugs are removed.

Second, fixed lines and buggy lines share similar entropy distribution (i.e., shapes in the bean plot in Figure 7). Their distribution is significantly different from that of bug-free lines (in Figure 2): Most bug-free lines have low entropy (and thus are presented on the large bottom of the beans) whereas the entropies of buggy lines and fixed lines are distributed much more evenly across the whole beans (especially the long central parts of the beans).

Notably, our finding is inconsistent with what was reported by Ray et al. [42]. According to their findings, bug fixing can significantly improve the naturalness of buggy code. However, in our evaluation, bug-fixing operations do not significantly improve the naturalness of buggy code. A possible reason for the conflicting observations is that the dataset we employ is different from that employed by Ray et al. [42]. Employing different data sets may result in completely different conclusions. Another possible reason is that we distinguish bug-fixing changes from bug-irrelevant changes (including changes on test cases) within bug-fixing commits as shown in Figure 1, and our analysis is based on the pure bug-fixing changes only. In contrast, Ray et al. [42] took all changes within bug-fixing commits as bug-fixing changes. The third possible reason is that the bug-fixing commits involved in our empirical study have been manually confirmed on bug tracking systems whereas Ray et al. [42] retrieved bug-fixing commits automatically by keywords. The latter many contain some false positives as discussed in Section 3.2.

To qualitatively analyze why bug fixing may not improve the naturalness of source code, we manually inspect some typical examples of the bugs in the involved repositories. The example in Listing 2 is a typical example from open-source application *Lang* [8].

```
1  /* From NumverUtils.java in Lang*/:
2
3  public class NumberUtils {
4   ......
5   if(chars[i] == 'l'
6    || chars[i] == 'L'){
7    // not allowing L with an exponent or decimal point
8  -   return foundDigit && ! hasExp
9  +   return foundDigit && !hasExp && !hasDecPoint;
10   }
11 ......
```

Listing 2: Code Entropies Before and After Bug Fixing

```
/* From ZipArchiveInputStream.java in Compress*/:

public ZipArchiveEntry getNextZipEntry() throws IOException
        {
 ......
if (sig.equals(ZipLong.CFH_SIG)||sig.equals(ZipLong.AED_SIG
        )) {
        hitCentralDirectory = true;
        skipRemainderOfArchive();
+       return null;
  }

  if (!sig.equals(ZipLong.LFH_SIG)) {
+   throw new ZipException(String.format("Unexpected record signature:
        0X%X", sig.getValue()));
-   return null;
}
......
```

Listing 3: Bug Fixing Dose Not Improve Code's Naturalness

In this example, the buggy line (Line 8) is modified into the fixed line (Line 9): It turns the logical expression from "*foundDigit && ! hasExp*" into "*foundDigit && !hasExp && !hasDecPoint*". The buggy line is not natural (with a high entropy of 1.825) because the statement is not common, the two expressions (i.e., "*foundDigit*" and "*!hasExp*") have never been concatenated with "&&" before, and none of its 3-grams (sequences of tokens) has appeared in training data (i.e., other parts of the project). However, fixing the bug does not improve the naturalness because it does not remove any of the uncommon token sequences (3-grams). Instead, it appends additional an uncommon expression "*!hasDecPoint*" to the logical expression, resulting in an uncommon 3-gram ("*&&!hasDecPoint*") that further reduces the naturalness of the whole statement. As a result, the entropy (2.204) of the fixed line (Line 9) is substantially higher than that (1.825) of the buggy line (Line 8).

Table 5: Entropy of Lines Influenced by Different Bug-Fixing Operations (Defects4J)

Type of Operations	DELETION	MODIFICATION		INSERTION
Influenced Code Lines	Deleted buggy lines	Modified buggy lines	Updated fixed lines	Inserted fixed lines
Number of Lines	1,546	1,017	1,017	5,625
Average Entropy	0.15	0.23	0.38	0.27
Maximal Entropy	4.05	4.94	4.94	7.05
Minimal Entropy	-2.42	-1.94	-2.07	-2.46
The First Quartile (Q1)	-0.62	-0.58	-0.38	-0.61
The Second Quartile (Median)	0.08	0.13	0.32	0.13
The Third Quartile (Q3)	0.81	0.94	1.07	1.05

Table 6: Entropy of Lines Influenced by Different Bug-Fixing Operations (GrowingBugs)

Type of Operations	DELETION	MODIFICATION		INSERTION
Influenced Code Lines	Deleted buggy lines	Modified buggy lines	Updated fixed lines	Inserted fixed lines
Number of Lines	451	442	442	1890
Average Entropy	0.12	0.36	0.43	0.31
Maximal Entropy	3.73	5.82	5.78	7.08
Minimal Entropy	-2.04	-1.82	-1.54	-2.39
The First Quartile (Q1)	-0.67	-0.34	-0.25	-0.60
The Second Quartile (Median)	0.05	0.36	0.43	0.22
The Third Quartile (Q3)	0.76	0.95	1.09	1.05

Another example is presented in Listing 3. The buggy method `getNextZipEntry` comes from project `Compress` [4] that is associated with a bug-fixing commit in Defects4J. From the code snippet, we observe that Line 13 is buggy whereas Line 8 and Line 12 are fixed lines. We notice that the buggy Line 13 "*return null*" is a common statement, appearing many times within the same application. Consequently, it is natural with a low code entropy of -1.6895. We also notice that the fixed Line 8 is exactly the same as the buggy line whereas the other fixed line (Line 12) looks much more unnatural: Its entropy (0.45) is significantly higher than that (-1.6895) of the buggy line. Line 12 is less natural because it contains uncommon token sequences like "*Unexpected record signature:*" that do not appear in other parts of the project. As a result, replacing the buggy Line 13 with fixed Line 8 and Line 12 dose not improve the naturalness of source code. From this example, we observe that bugs do not necessarily lead to unnaturalness of code: Common and natural statements (like "*return null*") could be buggy (Line 13) or bug-free (Line 8), whereas bug-free lines that are leveraged to replace buggy ones could be unnatural as well. Fixing bugs does not improve code naturalness because the unnaturalness is not caused by bugs, bugs are more likely to appear on unnatural lines.

We conclude based on the preceding analysis that bug fixing as a whole cannot significantly improve the naturalness of source code. Fixed lines on average are not more natural than buggy lines, which may suggest that software bugs are not the root causes of the unnaturalness of buggy code lines. It also explains why it is inaccurate to identify buggy lines by code entropy alone.

6.3 Fine-grained Analysis

To further investigate the impact of different bug-fixing operations (i.e., deletion, insertion, and modification), we compare the entropies of deleted buggy lines, modified buggy lines, inserted fixed lines, and updated fixed lines. More specifically, we investigate the effect of modification by comparing the entropies of modified buggy lines against the entropies of updated fixed lines. We also investigate the effect of replacement (i.e., replacing buggy lines with brand-new lines) by comparing the entropies of deleted buggy lines against the entropies of inserted fixed lines. Evaluation results are presented on Table 5 and Table 6, and corresponding bean plots are presented in Figure 8. From such tables and figure, we make the following observations.

First, modifying buggy lines does not significantly improve the naturalness of source code. On Defects4J, the average entropy (0.38) of the updated fixed lines is even higher than that (0.23) of the modified buggy lines. Wilcoxon non-parametric test (p-value=$0.67 * 10^{-3} \ll 0.05$) suggests that updated fixed lines are significantly less natural than modified buggy lines although Cohen's D size (0.07) suggests that the difference is small. On the other repository GrowingBugs, the average entropy (0.43) of the updated fixed lines is also higher than that (0.36) of the modified buggy lines. However, Wilcoxon non-parametric test (p-value=$0.34 > 0.05$, and size=0.0319) suggests that the difference is not significant. Anyway, on neither of the bug repositories we can find any evidence to support the hypothesis that modifying buggy lines can significantly improve the naturalness of source code.

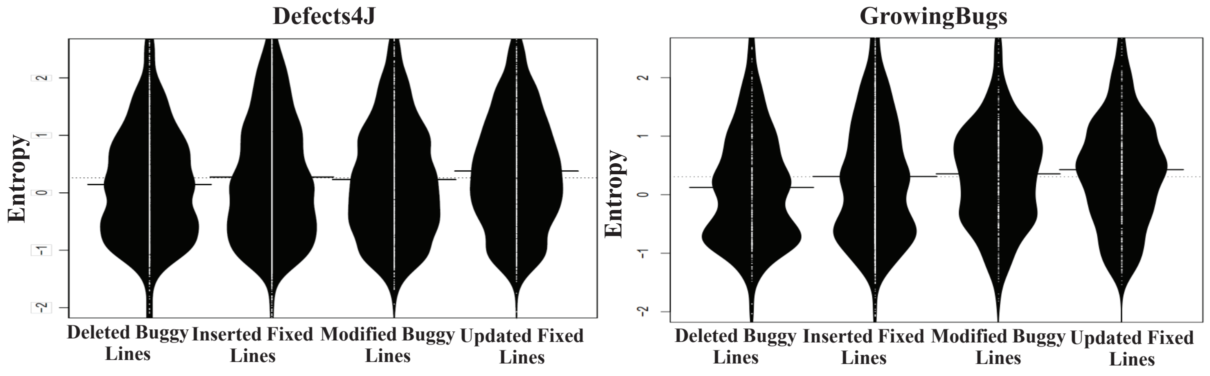

Figure 8: Entropy of Lines Influenced by Bug-Fixing Operations

Second, replacing buggy lines with new lines does not significantly improve the naturalness of source code, either. On Defects4J, the average entropy (0.27) of inserted fixed lines is even higher than that (0.15) of deleted buggy lines. The same is true on GrowingBugs: the average entropy (0.31) of inserted fixed lines is higher than that (0.12) of deleted buggy lines.

Third, not all buggy lines are modified by bug-fixing operations. More than half of the buggy lines are simply removed. On Defects4J, only 40% = 1017 / (1017 + 1546) of buggy lines are modified into fixed lines in the fixed version, and the others (accounting for 60%) are removed completely. Similarly, on GrowingBugs, less than half (49%=442/(451+442)) of the buggy lines are modified into fixed lines in the fixed version, and the others (51%) are removed completely.

Finally, most of the fixed lines are newly inserted. On Defects4J, 85% = 5625/(5625+1017) of the fixed lines come from insertion of brand-new lines, and only 15% are derived from buggy lines by modification. Similarly, on the other repository GrowingBugs, 81%(= 1890/(1890+442)) of fixed lines are newly inserted, and only 19% are derived from buggy lines. The results may suggest that insertion of new lines is much more popular than modification of buggy lines while developers are fixing bugs.

We conclude from the preceding analysis that neither modifying nor replacing buggy lines with brand-new lines can significantly improve the naturalness of source code.

7 DISCUSSIONS

7.1 Threats to Validity

A threat to external validity is that our study was conducted on only two bug repositories. Some special characters of the repositories may have biased our conclusions. Notably, it is challenging to construct large-scale and high-quality bug repositories because it remains challenging to exclude bug-irrelevant changes from bug-fixing commits [26]. To the best of our knowledge, Defects4J and GrowingBugs, are two of the largest general bug repositories where bug-irrelevant changes have been explicitly excluded. Other well-known bug repositories, like iBUGS [17], ManyBugs [33] and ManySStuBs4J [1, 30], are not employed because they do not explicitly exclude bug-irrelevant changes or focus on some special categories of bugs only. Notably, the conclusions drawn in this paper are consistent on the two repositories, which helps to generalize the conclusions. We also notice that our empirical study involves 151 open-source applications in total, compared to 10 applications exploited by comparable previous study [42].

The second threat to external validity is that it investigates Java projects only. Our study is confined to Java because of the following reasons. First, our empirical study was inspired by the high-impact study conducted by Ray et al. [42], and the latter was conducted on Java applications. Consequently, focusing our empirical study on Java applications may facilitate direct comparison between our study and the high-impact existing empirical study [42]. Second, Defects4J and GrowingBugs contain Java projects only. Although bug repositories in other programming languages are available, e.g, SIR in both C and Java, and BugsJS [22] in JavaScript, such bug repositories do not explicitly exclude bug-irrelevant changes as Defects4J and GrowingBugs do. Consequently, such repositories could not yet be exploited by our empirical study. In the future, we should extend the study to validate whether the conclusions drawn on Java hold on other languages.

A threat to internal validity is that we do not explicitly exclude potential confounds, e.g., characters of developers, development guidelines employed by different companies, characters and domains of software applications. All such confounds may influence the naturalness and popularity of code lines. We do not explicitly exclude such confounds because of the following reasons. First, some of them (e.g., development guidelines) are not explicitly specified by the bug repositories. Second, further dividing the dataset according to the confounds (e.g., developers, companies, and application domains) would significantly reduce the size of the dataset, which may result in severe threats to statistical validity.

The first threat to construct validity is that the measurement of source code's naturalness depends on the employed language model, and the quality of the measurement may significantly influence the conclusions drawn in this paper. Our empirical study depends on a single entropy model [42, 44]. Switching to other entropy models may break the conclusions drawn in this paper. The model is selected because of the following reasons. First, this model has been widely used and its implementation is publicly available. Second, this model has been used to investigate the naturalness of buggy code by existing research [42], and thus employing the same model facilitates comparison against related work.

Another threat to construct validity is that the bug-fixing commits and bug-fixing changes in Defects4J/GrowingBugs could be inaccurate. For example, although the patches in Defects4J have been manually validated, it has been reported that some of these patches could be inaccurate [26]. Such inaccuracy could make the patch-based analysis in our study inaccurate. Notably, it is time-consuming and challenging to manually validate all of the patches. We also notice that the bug repositories may not include all bugs in the involved applications. As a result, some source code marked as bug-free according to the bug repositories could be buggy in fact.

A threat to statistical conclusion validity is that the conclusions depend on significance tests applied to datasets of different sizes. Because of the *too-large sample size problem* [31], results of significance tests applied to large datasets could be misleading. However, to the best of our knowledge, there is no explicit and well-accepted guidance on the maximal size of datasets where significance tests could be applied.

7.2 Limitations

Our empirical study reveals some interesting and potentially valuable findings, e.g., buggy lines are often less natural, but bug fixing may not significantly improve their naturalness. However, the reasons behind such findings remain unclear. The evaluation results reported in this paper validates only the association (rather than causation) between bugs and unnaturalness, i.e., bugs are more likely to appear on unnatural lines. Although causation is a special kind of association, Section 6 suggests that unnaturalness is not caused by bugs (otherwise, unnaturalness should have disappeared with the fixes). We also present some examples in Section 6.2 to explain why fixing bugs may not improve naturalness. Bugs are more likely to appear on unnatural lines (i.e. their association) because both bugs and unnaturalness are associated with (i.e., more likely to happen on) unpopular code lines. In contrast, most of the popular and simple lines (like "{", "}", and import statements) are often bug-free, and such popular lines are inherently natural because code entropy models represent how surprising a code line is. As a result, bugs often appear on unpopular lines (rather than popular ones), and unpopular lines are inherently less natural. And thus, we observe that buggy lines are often less natural although bugs do not cause unnaturalness. We remove popular lines (i.e., excluding all lines whose popularity is greater than one) and then compare the naturalness of the remaining unpopular buggy lines and bug-free lines. Analysis results suggest that unpopular bug-free lines are almost as unnatural as unpopular buggy lines. For Defects4J, the average entropy of unique bug-free lines is up to 0.53, almost equivalent to that (0.60) of unique buggy lines. Similarly, for GrowingBugs, the average entropy of unique bug-free lines in up to 0.50, almost equivalent to that (0.53) of unique buggy lines.

In this paper, all bugs are taken as a whole (without further division) to investigate how bugs and their patches change the naturalness of source code. However, it is likely that different kinds of bugs, e.g., crashes, misusing of APIs, and misspelling of literals, may have different impact on the naturalness. In future, we should classify the bugs, and conduct a fine-grained empirical study to investigate how each category of the bugs (and their associated bug-fixing operations) influence the naturalness of source code.

It remains unclear whether (and how) source code's naturalness could be exploited for automated identification of buggy code. In Section 4 we find a correlation between code's naturalness and its health condition (buggy or bug-free). We also leverage a simple and intuitive approach to identify buggy code by naturalness alone (in Section 5), which is proved unsuccessful. However, we have not yet tried more advanced approaches with various inputs besides naturalness, and the failed approach itself does not necessarily mean that code's naturalness cannot be exploited for automated identification of buggy code.

8 CONCLUSIONS AND FUTURE WORK

Naturalness is a very interesting and useful property of source code. This property has been successfully exploited for code complete [12, 24], learning of code styles [10], and source code migration [37, 38]. Inspired by such work, Ray et al. [42] conducted an empirical study, suggesting that buggy code is often less natural and code naturalness could be leveraged for fault localization. In this paper, we investigate the naturalness of buggy code with two high-quality large-scale bug repositories. The results of our study suggest that buggy code lines on average are less natural than bug-free ones. However, bug fixing often fails to significantly improve the naturalness of buggy code. It may suggest that bugs in source code are not the root cause of buggy code's unnaturalness. Our investigation helps to figure out the *nature* of source code's naturalness, especially the naturalness of buggy code.

In future, it would be interesting to investigate the root causes of buggy code's unnaturalness. Although our empirical study in this paper suggests that bugs are not the root causes of source code's unnaturalness, it remains unclear why buggy code lines are less natural than bug-free ones.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers from ESEC/FSE'2022 for their insightful comments and constructive suggestions.

This work was partially supported by the National Natural Science Foundation of China (62172037 and 62141209), and CCF-Huawei Formal Verification Innovation Research Plan.

REFERENCES

[1] 2020. ManySStuBs4J Dataset. https://doi.org/10.5281/zenodo.3653444.
[2] 2021. Gson-21. https://github.com/rjust/defects4j/blob/master/framework/projects/Gson/patches/17.src.patch.
[3] 2021. JFreeChart. https://sourceforge.net/projects/jfreechart/.
[4] 2022. https://issues.apache.org/jira/browse/COMPRESS-367.
[5] 2022. A commit from Commons-IO. https://github.com/apache/commons-io/commit/ac36a6df.
[6] 2022. Eclipse. https://www.eclipse.org/.
[7] 2022. GrowingBugs. https://github.com/jiangyanjie/GrowingBugs.
[8] 2022. Lang-Bug24. https://github.com/rjust/defects4j/blob/master/framework/projects/Lang/patches/24.src.patch.
[9] 2022. Replication Package. https://github.com/jiangyanjie/RevisitingNaturalness.
[10] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)* (Hong Kong, China). Association for Computing Machinery, New York, NY, USA, 281–293.
[11] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. 281–293.

[12] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[13] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[14] Lalit R Bahl, Frederick Jelinek, and Robert L Mercer. 1983. A Maximum Likelihood Approach to Continuous Speech Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 2 (1983), 179–190.

[15] William B Cavnar, John M Trenkle, et al. 1994. N-gram-based Text Categorization. In *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*. 161–175.

[16] Saikat Chakraborty, Yangruibo Ding Toufique Ahmed, and Baishakhi Ray Premkumar Devanbu. 2022. NATGEN:Generative pre-training by "Naturalizing" source code. In *Proceedings of the 2022 30th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'22)*.

[17] Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of Bug Localization Benchmarks from History. In *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 433–436.

[18] Premkumar T. Devanbu. 2015. New Initiative: The Naturalness of Software. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 543–546. https://doi.org/10.1109/ICSE.2015.190

[19] Tom Fawcett. 2006. An introduction to ROC analysis. *Pattern recognition letters* 27, 8 (2006), 861–874.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[21] Mark Gabel and Zhendong Su. 2010. A Study of the Uniqueness of Source Code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE'10)*. 147–156.

[22] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. BugsJS: A Benchmark of JavaScript Bugs. In *12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 90–101.

[23] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar T. Devanbu. 2016. On the Naturalness of Software. *Commun. ACM* 59, 5 (2016), 122–131. https://doi.org/10.1145/2902362

[24] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (Zurich, Switzerland). IEEE Press, 837–847.

[25] Yanjie Jiang, Hui Liu, Xiaoqing Luo, Zhihao Zhu, Xiaye Chi, Nan Niu, Yuxia Zhang, Yamin Hu, Pan Bian, and Lu Zhang. 2022. BugBuilder: An Automated Approach to Building Bug Repository. *IEEE Transactions on Software Engineering* (2022), 1–22. https://doi.org/10.1109/TSE.2022.3177713

[26] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE'21)*.

[27] Matthieu Jimenez, Thierry Titcheu Chekam, Maxime Cordy, Mike Papadakis, Marinos Kintis, Yves Le Traon, and Mark Harman. 2018. Are Mutants Really Natural?: A Study on How "Naturalness" Helps Mutant Selection. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*, Markku Oivo, Daniel Méndez Fernández, and Audris Mockus (Eds.). ACM, 3:1–3:10. https://doi.org/10.1145/3239235.3240500

[28] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*. 437–440.

[29] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. 2011. MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 612–615. https://doi.org/10.1109/ASE.2011.6100138

[30] Rafael-Michael Karampatsis and Charles Sutton. 2020. How often do single-statement bugs occur? the manysstubs4j dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 573–577.

[31] Peter Kennedy. February 19, 2008. *A Guide to Econometrics. 6th edition 6th Edition*. Wiley-Blackwell.

[32] Kamran Kowsari, Kiana Jafari Meimandi, Mojtaba Heidarysafa, Sanjana Mendu, Laura Barnes, and Donald Brown. 2019. Text Classification Algorithms: A Survey. *Information* 10, 4 (2019), 150–218.

[33] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering (TSE)* 41, 12 (2015), 1236–1256.

[34] Bin Lin, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. On the Impact of Refactoring Operations on Code Naturalness. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. IEEE, 594–598.

[35] P Majumder, M Mitra, and BB Chaudhuri. 2002. N-gram: A Language Independent Approach to IR and NLP. In *International Conference on Universal Knowledge and Language*.

[36] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 457–468.

[37] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical Statistical Machine Translation for Language Migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. 651–654.

[38] Tien N Nguyen. 2016. Code migration with statistical machine translation. In *Proceedings of the 5th International Workshop on Software Mining*. 2–2.

[39] Vu H Nguyen, Hien T Nguyen, Hieu N Duong, and Vaclav Snasel. 2016. n-Gram-based Text Compression. *Computational Intelligence and Neuroscience* (2016).

[40] Fuchun Peng and Dale Schuurmans. 2003. Combining Naive Bayes and N-gram Language Models for Text Classification. In *European Conference on Information Retrieval (IR)*. Springer, 335–350.

[41] Musfiqur Rahman, Dharani Palani, and Peter C. Rigby. 2019. Natural Software Revisited. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 37–48. https://doi.org/10.1109/ICSE.2019.00022

[42] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the " Naturalness" of Buggy Code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)*. IEEE, 428–439.

[43] Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. 2019. Does BLEU score work for code migration?. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 165–176.

[44] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the Localness of Software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. 269–280.

[45] Xinwei Zhang and Bin Wu. 2015. Short Text Classification based on Feature Extension Using the N-gram Model. In *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*. IEEE, 710–716.

[46] Hao Zhong and Hong Mei. 2020. Learning a graph-based classifier for fault localization. *Science China Information Sciences* 63, 6 (2020), 1–22.