Are Neural Bug Detectors Comparable to Software Developers on Variable Misuse Bugs?

Cedric Richter  
University of Oldenburg  
Oldenburg, Germany  
cedric.richter@uol.de

Jan Haltermann  
University of Oldenburg  
Oldenburg, Germany  
jan.haltermann@uol.de

Marie-Christine Jakobs  
Technical University of Darmstadt  
Darmstadt, Germany  
jakobs@cs.tu-darmstadt.de

Felix Pauck  
Paderborn University  
Paderborn, Germany  
fpaucek@mail.upb.de

Stefan Schott  
Paderborn University  
Paderborn, Germany  
stefan.schott@upb.de

Heike Wehrheim  
University of Oldenburg  
Oldenburg, Germany  
heike.wehrheim@uol.de

ABSTRACT
Debugging, that is, identifying and fixing bugs in software, is a central part of software development. Developers are therefore often confronted with the task of deciding whether a given code snippet contains a bug, and if yes, where. Recently, data-driven methods have been employed to learn this task of bug detection, resulting (amongst others) in so called neural bug detectors. Neural bug detectors are trained on millions of buggy and correct code snippets.

Given the "neural learning" procedure, it seems likely that neural bug detectors – on the specific task of finding bugs – have a performance similar to human software developers. For this work, we set out to substantiate or refute such a hypothesis. We report on the results of an empirical study with over 100 software developers, targeting the comparison of humans and neural bug detectors. As detection task, we chose a specific form of bugs (variable misuse bugs) for which neural bug detectors have recently made significant progress. Our study shows that despite the fact that neural bug detectors see millions of such misuse bugs during training, software developers – when conducting bug detection as a majority decision – are slightly better than neural bug detectors on this class of bugs. Altogether, we find a large overlap in the performance, both for classifying code as buggy and for localizing the buggy line in the code. In comparison to developers, one of the two evaluated neural bug detectors, however, raises a higher number of false alarms in our study.

CCS CONCEPTS
• General and reference → Empirical studies; • Software and its engineering → Software testing and debugging; • Computing methodologies → Neural networks; • Human-centered computing → User studies.

KEYWORDS
Bug detection, variable misuse bugs, empirical study.

1 INTRODUCTION
Debugging, i.e. finding and repairing bugs in software, is a frequent task during software development [3, 12, 42, 44]. Consequently, major efforts have been made to support software developers in this task or – even better – to automate bug localization and repair. In the past, static or dynamic analysis tools have been employed for this purpose (e.g. SpotBugs, the successor of FindBugs [9], ErrorProne [1] or Infer [13]). Studies have however shown that (static) tools can only find certain types of bugs [15]. This motivated the usage of data-driven methods for bug detection.

Data-driven techniques learn bug patterns from millions of training samples of correct and buggy code snippets. Recent advances in machine learning, in particular in natural language processing, have boosted the development of so called neural bug detectors [6, 16, 21, 31, 33, 34, 45]. Designed to be complementary to existing static tools, these bug detectors often focus on a single type of bug such as variable misuses [16, 21, 45] or binary operator bugs [31] which are hard to identify with a static rule. While at first requiring a long training phase, the tools are then extremely fast in classifying a code snippet as buggy or correct, finding the location of the bug and making repair suggestions. They do so on the source code only, without any execution.

Given that these tools seem to “learn” bug patterns and employ some “neural” architecture for this, the question comes up whether they actually achieve a performance similar to human software developers with respect to (a) classification (determining whether a code snippet is buggy or non-buggy) and (b) localization (finding the buggy line of code). We here consider a developer to be a person who implements, reviews, or tests code.

In this work, we investigate this question for variable misuse bugs [6], a class of bugs commonly addressed by existing neural bug detectors. In a variable misuse bug the code implementer
unintentionally uses the wrong variable. Variable misuses are frequent \cite{6,22} and easy to fix (as they require only a single identifier change) but often missed by the original implementer \cite{6}. In fact, around 6% of all Java bugs identified in the Google build system \cite{43} were identified to be related to variable misuse bugs.

While neural bug detectors become increasingly more effective in detecting variable misuses, it is unclear how software developers perform on the same task. Our hypothesis for this work was that developers have indeed comparable classification and localization performance on variable misuse bugs.

To be able to substantiate or refute this hypothesis, data on the performance of (a) neural bug detectors and (b) developers in the task of variable misuse bug detection was required. For the first, we reimplemented two different approaches for neural bug detection, representing the two main strands of research in this area: graph neural networks \cite{7} (GNNs) representing graph-based and transformers \cite{34} representing text-based techniques. Both of these were then trained on more than 2M examples derived from method implementations found in public Java GitHub projects. Afterwards, we evaluated them on our detection tasks. To get the required data about developers, in particular for the same detection tasks, we performed an empirical study with over 100 developers and faced them with the problem of detecting variable misuse bugs in code snippets.

Listing 1 exemplifies one such variable misuse detection task. Here, the task is derived from a real world bug fix taken from the ManySStuBs4J benchmark \cite{22}. The developer and similarly the neural bug detector would be confronted with this code snippet, however without highlighting line 4 and without the comment in line 5. For a correct answer, developers and neural bug detectors would then need to classify the code snippet as "buggy" and choose line 4 as the bug location.

In our empirical study, every developer received eight such bug detection tasks. The tasks were selected out of the ManySStuBs4J data set \cite{22}, a collection of single statement bugs from open-source Java projects. We filtered the data set for variable misuse related bugs, and manually inspected all resulting code snippets for its unambiguity (i.e., the possibility of uniquely attributing a bug to the change of a single variable in a single line of code). This finally resulted in 313 bug detection tasks, three of which we used as examples in the introductory explanation of the study. The two neural bug detectors were evaluated on the same (remaining) 310 tasks.

As the developers did neither always agree on the results of classification (buggy or not buggy) nor on the location of the bug, we used a majority decision to determine the outcome on a particular task for developers as a whole. Our comparison then showed that humans and neural bug detectors indeed have a comparable performance on variable misuse bugs, developers even slightly surpassing GNN as well as transformer on both classification and localization. We however also found that in particular one of the two) neural bug detectors raises many more false alarms than developers.

The main findings of our study can be summarized as follows.

- Looking at developers individually, we find that (a) their detection capabilities on variable misuse bugs are rather good, the rate of correct answers significantly exceeding simple guessing, and that (b) with increasing size of code snippets their detection rate deteriorates.
- Comparing developers and the two neural bug detectors on variable misuse bugs, we observe (a) a large overlap of humans and tools in correct answers, with a slight advantage on the side of developers, (b) a higher number of false alarms for in particular one neural bug detector, and (c) a similar degradation in the results of neural bug detectors when increasing code size.
- Interestingly, there are also a number of tasks on which developers succeed but neural bug detectors fail, as well as vice versa.

All data of the study as well as the experiments with the two neural bug detectors can be found in our artifact at Zenodo \cite{8}.

2 METHODOLOGY

To be able to compare the performance of developers and neural bug detectors, we first of all need data about developer performance in bug finding. To this end, we performed a study with over 100 developers. This section gives an overview on the study design and execution, the resulting data and the two neural bug detectors used for comparison. In the study design, we followed the recommendations given in \cite{23,24} as far as applicable. The design has been approved by the ethical review board of our institution.

2.1 The Variable Misuse Detection Task

In this work, we evaluate both developers and neural bug detectors on the same task: The detection and localization of variable misuse bugs. Variable misuse bugs occur when an implementer unintentionally uses the wrong variable identifier. For example, a variable might be named ambiguously and hence may mistakenly be used instead of the intended one, or a code passage may be duplicated and incompletely edited afterwards such that a certain identifier is not replaced (copy-and-paste bug). For our work, we chose such variable misuses, not only because the bug type is frequent in open source projects \cite{6,22} (which provides us with enough examples for our study) but also because there exist a variety of bug detectors \cite{16,21,33,34,45} specialized on this sort of bugs (which we aim to compare with).

Example. An example of a variable misuse bug can be found in Listing 1 in Section 1. Line 4 contains the variable misuse bug as the comment in the line below states.

When confronted with the task of finding variable misuse bugs, one first has to determine all variable identifiers. In the example all identifiers are highlighted in blue. Next, the usages must be distinguished from the definitions. A variable usage occurs whenever

```java
private List<T> toList(Enumeration<T> first, Enumeration<T> second) {
    List<T> list = emptyListField.copy();
    while (first!=null & first.hasMoreElements()) {
        list.add(second.nextElement());
    }
    while (second!=null & second.hasMoreElements()) {
        list.add(second.nextElement());
    }
    return list;
}
```

Listing 1: Variable misuse example
Are Neural Bug Detectors Comparable to Software Developers on Variable Misuse Bugs? ASE ’22, October 10–14, 2022, Rochester, MI, USA

We designed a study to get data about the performance of developers. The task is to achieve outputs of developers which can be compared to the outputs given by neural bug detectors, i.e. for some given code snippet, this assumption cannot be definitely validated. Such bug detection tasks thus require some sense of what is likely correct or likely buggy code.

2.2 Design of the Developer Study

We designed a study to get data about the performance of developers on variable misuse bug detection. The key requirement on this study was to achieve outputs of developers which can be compared to the outputs given by neural bug detectors, i.e. for some given code snippet we needed answers for the classification and localization question. We designed the study in two steps. First, we created an initial design, which we evaluated in a pre-study with a selected group of participants. The group consisted of three Java experts, two beginners and an expert in study design. We then used the feedback gathered in the pre-study to improve our design. This resulted in the following structure.

At the beginning, a participant is asked to fill in a survey about some demographic data (gender, age, continent of education, development experience, development context, team size). Afterwards, before starting the actual study, the participant receives a textual and a video explanation regarding the variable misuse task itself. To further illustrate the task and familiarize participants with the interface, three example code snippets and their solutions are presented to every participant. During the study every participant furthermore has continuous access to an explanation of the variable misuse detection task. Afterwards, every participant gets eight randomly chosen tasks shown in succession (a trial). The tasks are randomly chosen out of our benchmark set so that we get results for a representative number of different tasks. After completing all eight tasks, the participant receives a score referring to the number of correctly solved tasks. No feedback regarding individual tasks is displayed, so that the same participant can potentially complete another trial without the possibility of learning from previous trials. In addition, while the participants were allowed to skip the remaining tasks after 20 minutes, we did not limit the overall time needed for solving the eight tasks.

The study was performed via a web-application that participants accessed via their web-browsers. Figure 1 shows the interface for the detection tasks. The code snippet in the center of the figure represents the current task – in this case the same as in Listing 1. As shown, each task was presented to the user in the form of a single Java method without any additional context – no documentation or additional source code was given. This exactly matches the input given to neural bug detectors. While working on our study online, users still had access to other resources like StackOverflow or Google.

For every task, participants were asked to answer the classification and (potentially) the localization question. Naturally, the latter question must only be answered when the answer to the first question classifies the snippet as buggy. To be able to proceed to the next task, a participant has to either mark the code snippet as correct, or select a potentially buggy line. Once a line of code is clicked on, it becomes highlighted and counts as selected. By clicking on another line before submitting the result (which is done by pressing the Next button), participants can still change the selection.

2.3 Recruitment of Study Participants

We decided for an anonymous and unpaid participation in the study. Developers were recruited via posts on social media (Twitter, Slack and Discord channels), Java forums, student forums and by personal invitation. The study run for six weeks and at the end we obtained 1146 answers from 111 participants in 134 participations (multiple participations were allowed).

As mentioned in Section 2.2, participants were asked about some demographic data as well as data referring to their work and experience in software development. We had 82.9% male and 13.5% female participants (rest: undisclosed); 31.5% developed software in a commercial context and 37.8% worked on open source projects (rest: others); 57.7% were younger than 30, 38.7% between 30 and 50, rest above 50; participants were predominantly european (92.8%). To get a feeling for the representativeness of our participants (at least wrt. experience), we compared our data with the 2021 Developer Survey of Stack Overflow\(^1\). Figure 2 exemplarily shows the ratios for coding experience in our study and the Stack Overflow survey. We see that our participants are a bit less experienced compared to the Stack Overflow users, which might indicate that the results might have been even better for typical Stack Overflow users.

2.4 Benchmark Selection

In parallel to the study design, we selected the variable misuse detection tasks, which we obtained from the ManySStuBs4J data set [22]. The ManySStuBs4J data set contains 153,652 Java bug fixes that represent candidates for the tasks of our study. The bug fixes originate from 1,000 popular open-source Java projects and each fix changes a single statement. Since ManySStuBs4J does not directly

\(^1\)https://insights.stackoverflow.com/survey/2021
include a category for variable misuses, we filtered the set for all bug fixes which repair a variable misuse by mining the original repositories of the respective projects. This includes bug fixes from the categories Change Identifier Used, Change Operand, and Same Function Wrong Caller. From the remaining bug fixes, we aimed to select those fixes as tasks that do not only represent variable misuses but are also manageable by both humans and neural bug detectors. To this end, we selected all fixes that change a method that

- is parsable before and after application of the fix,
- contains at most 1024 tokens (the limit of most existing neural bug detectors [16]), and
- consists of at most 30 lines of code (larger methods are candidates for refactoring [36]).

In total, we ended up with 387 task candidates. Since the bug fixes in ManyStu$\textsuperscript{4}$ are identified based on commit messages and may fix multi-line statements, for which the buggy line is ambiguous, we inspected each of these 387 bug fixes manually by two authors and checked whether they are unambiguous variable misuse fixes. Candidates with differing assessments were reviewed by a third author. We then excluded all candidates that were rejected by at least two authors, and kept the remaining ones. In general, the reasons for exclusion of bug fixes were

- changes across multiple lines, so that there is no unique buggy line,
- (same) variable misuses applying to multiple lines (hence, no unique bug location),
- multiple changes in a single line (i.e., fixing the variable misuse is not sufficient for repair),
- non-functional fixes, e.g., refactorings which do not change the method behavior.

After manual review, we ended up with 342 tasks. From these tasks, we selected three as introductory representative examples$^2$ for our study (see Section 2.2). For the remaining 339 tasks we randomly selected half of them to become non-buggy tasks – tasks that do not include a variable misuse bug. These tasks consist of the methods after application of the bug fix. The other half is used as buggy tasks, i.e. they show the methods prior to application of the fix.

As an outcome of our pre-study, we decided to auto-format the tasks with the tool google-\texttt{java-format}$^3$ to ensure a uniform presentation across tasks. Error locations were realigned by computing the AST difference [14] between the now formatted version of the method before the bug fix and its unformatted fixed counterpart after the bug fix. This process further excluded 29 buggy tasks where a realignment was not possible.

Finally, this leaves us with a benchmark of 310 tasks including 134 buggy and 176 non-buggy tasks. After formatting, tasks contain between 3 and 45 lines of code (between 30 and 348 tokens). While on average a task contains 18 lines of code (and 123 tokens), standard deviation (7.9 for lines of code) and variance (61.8 for lines of code) are high and indicate that our tasks differ significantly in size. Note that our benchmark tasks are large in comparison to the typical method implementation found in the open source project of the bug. On average, the tasks contain four times more code lines than the median method implementation and twice the number of code lines of the average of method implementations in the respective project.

### 2.5 Data Cleaning

Before evaluating the 1 146 answers, we performed some data cleaning. We excluded all trials that were aborted by the user through closing the questionnaire before the end. Thereby, 30 trials with 82 answers were removed. In six trials, we observed that the users answered all eight tasks in less than 60 seconds, which strongly indicates that random answers were selected. To put this into perspective, the average participant required around 13 minutes per trial and all but the six trials required at least 3 minutes. We therefore used the overall time taken for a trial as another exclusion criterion and excluded these six trials. As already mentioned, we did not provide solutions or detailed feedback on trials within or after finishing the study (except for the score that only reflects the number of correctly classified tasks). Hence, users were allowed to participate multiple times. Six users participated twice, three users three times and one user four times. These users gave 72.2% (52/72) correct answers in their first trial and 71.2% (74/104) correct answers during retries. As we see that there is indeed no learning effect in multiple participations, we thus did not exclude their answers and used in total 1 016 answers for 134 buggy and 176 non-buggy tasks$^4$ in 127 trials for the evaluation.

### 2.6 Neural Bug Detectors

For comparison, we reimplemented two popular types of neural network based bug detectors: a graph neural network (GNN) based detector and a detector based on a transformer model. GNNs have been shown to be highly effective in detecting variable misuses by incorporating graph level information. Here, we employed a recent GNN model [7] that learns to identify bugs based on structural-, control-flow and data-flow information. Since the GNN benefits from analytical information provided by a static analyzer – which also includes information about potential variable misuse locations (which a human developer would first of all have to identify herself) – we also included a text based transformer model for bug detection [34]. Similar to our participants, the transformer directly operates on source code without having access to any additional annotations.

To be consistent with previous work, the bug detectors are jointly trained on three problems$^5$: (a) classifying whether a given method

---

$^2$The resulting benchmark and the introductory examples are available in our artifact.

$^3$https://github.com/google/google-java-format

$^4$For one non-buggy task, all answers were excluded during data-cleaning.

$^5$We observed that including the repair problem also improves the detection performance.
implementation contains a variable misuse, (b) localizing the variable misuse at token-level and (c) proposing a repair for the identified variable misuse bug. During our evaluation, however, we ignore (c), the proposed repair, and focus on the capability of the detector to (a) classify a method as buggy and (b) find the misuse location. Since our focus is on line-level bug localization during our study, for (b) we use the line of code containing the token that is most likely incorrect (if any) according to the detector. The computed likelihood in form of a probability score is reported as the confidence of the detector.

For training the bug detectors, we employ mutants (of methods implementations from public projects) instead of real bugs, as it is standard in neural bug detection. Mutants are generated via a context based mutation operator [34], as this was shown to improve the detection performance. In contrast to mutants employed in previous works [45], a contextual mutator produces more realistic mutants (i.e., bugs more closely resembling real code) by mutating a variable usage at a random location dependent on the given context. In total, the bug detectors are thereby trained on over 2M examples obtained by mutating existing method implementations from popular GitHub projects [17]. We removed all method implementations from the training set which are close duplicates [4] to the methods occurring in our benchmark. We employed the technique described by Allamanis et al. [4] to identify duplicate implementations.

Finally, all bug detectors are evaluated on the same 310 tasks as provided to our participants. Unlike the developers, the bug detectors have no access to information beyond the given code snippets.

3 RESULTS

Next, we report on the results of our study. Overall, the evaluation of data has the objective of answering our one overarching research question:

**RQ: How do neural bug detectors compare to software developers with respect to their bug detection capabilities?**

First, to validate our study design, we evaluate how well the human participants performed in our study. Afterwards we make the comparison with neural bug detectors, separately looking at classification and localization. To achieve comparable results at a task level, we aggregate the developer decisions per task (via a majority vote). We then compare the aggregated decision of the developers with the decision of each neural bug detector on the same task.

During this section, we use the following terminology (partially explained before). A trial is one pass through the study and consists of eight bug detection tasks. A task is considered buggy if the code snippet of the task contains a bug, otherwise it is non-buggy. An answer is the response of a developer to a task. Answers can be correct or incorrect, and hence tasks can be correctly or incorrectly classified. Answers can further be correct or incorrect localizations, depending on whether the right line in the code of a buggy task has been selected or not.

3.1 Developer Performance

We start by reporting about the classification and localization performance of individual developers, and then also take a look at the influence of code snippet size on solving tasks in general.

**Classification.** For classification, we received in total 1016 answers for 310 tasks, out of which 176 tasks were non-buggy and 134 tasks buggy. For buggy tasks, developers gave the correct answer in 63.7% of the cases, the incorrect answer in 36.3% of the cases. For non-buggy tasks, we obtained correct answer in 77.4% of the cases, incorrect ones in 22.6%. Figure 3 depicts the distribution of correct task classifications during a trial (average 5.70, median 6). We see that developers perform significantly better than what would be achieved by simply guessing a solution.

We have also evaluated what the influence of demographic data on classification is. We found the programming experience (in years) to slightly improve classification performance, the average (for correctly classified tasks in a trial) moving up from 5.27 (again out of eight) for less than two years experience over 5.77 for two to five years up to 5.85 for more than five years. The other factors (age, gender, team size, etc.) have shown no significant influence.

**Localization.** For localization, there are (naturally) more possibilities for giving incorrect answers. We received 455 answers to the 134 buggy tasks. Out of these, 43.5% chose the correct line; in 56.5% of the cases localization is incorrect. Again, localization abilities slightly improve with the number of years of programming experience (the average going up from 31.7% for less than two years of experience, over 46.0% for two to five years, to 48.0% for more than five years). As before other factors have no influence.

**Influence of Code Size.** Besides demographic factors, we also investigated what the influence of code size is on developer performance. Here, we do not look at developers individually (by giving results per trials), but aggregate over all developers (and thus look at tasks) to see how the performance of developers in general is affected by code size.

Here, we have actually found a significant influence (which interestingly is the same for the neural bug detectors, see Section 3.2 below). The upmost boxplot in Figure 4 shows the impact of the number of lines of code (LOC) on classification (left part) and localization (right part). On the x-axis we see three classes of sizes (1-10 LOCs, 11-20 LOCs and >20 LOCs). The y-axis gives the confidence which here is the percentage of correct classifications and localizations, respectively, in the interval [0, 1]. We see that both for classification and localization the mean (green triangle) and median (blue line) deteriorates with an increase in code size. The descending median values with respect to the three classes are 1.00,
Finally, our last evaluation concerned the agreement of developers on their given answers. This is of interest for the comparison with neural bug detectors, as we define the answer of “a developer in general” (on a task) as the answer occurring in the majority of answers of our study participants.

Figure 5 gives the agreement in the number of tasks (x-axis). Here, we do not distinguish between correct and incorrect answers; the developers could also (and sometimes do) agree on incorrect answers. Not all tasks received enough answers to achieve an agreement between participants. Hence, we only considered tasks where an agreement was theoretically possible. For classification, this excluded six tasks where an agreement was not possible because at most one answer was given per task. For localization, the agreement is computed by looking at answers for buggy tasks only, and only for tasks where the majority has classified the task as buggy (so that a localization is actually provided by the majority of answers). We excluded 44 tasks in which this was not possible. In the figure, we distinguish between “no majority decision” (no majority in answers to one task, i.e. either buggy/non-buggy or a line in the code) and “majority decision”, which we furthermore split into “unanimous” and “non-unanimous”. We see that the participants reach an agreement in most cases (around 90% for classification and 77% for localization).

### 3.2 Comparison wrt. Classification

Next, we evaluate how developers perform in comparison to our two baseline neural bug detectors wrt. classification. First of all, as already mentioned, to have some way of comparing developers “in general” with neural bug detectors, we aggregate the answers of developers using a majority decision. This means, the answer of “the developer in general” is the answer the majority of developers have given. As we cannot compute a majority if only a single answer is given, we excluded all such tasks. This results in 304 tasks out of which 133 tasks are buggy and 171 tasks are non-buggy.

In the comparison, we are not only interested in the percentage of correct and incorrect answers, but would like to more specifically see on how many tasks developers and neural bug detectors agree. Therefore, we next present the results in number of tasks, and look at (a) the number of correct bug detections on buggy tasks, and (b) the number of false alarms, i.e. falsely attributing a non-buggy task as buggy.

Detection of Bugs. Figure 6a shows the overlap of bugs correctly detected by the (majority of) developers and the two neural bug detectors. The diagram also states the number of tasks with no majority decision (nine). One can see that out of the 124 tasks that contain a variable misuse, only 14 have neither been detected by the developers nor by the bug detectors (box “undetected”). Furthermore, the biggest observable overlap consists of buggy tasks detected by the developers as well as both bug detectors. The developers (with 82 detections) and the GNN-based bug detector (with 85) found almost the same amount of total bugs. In contrast, the transformer-based bug detector (with 62 detections) performed 24.4% worse than the developers. There are in total 17 bugs which are uniquely found by the developers, i.e. not detected by neither of the bug detectors. However, there were also 28 bugs exclusively found by the bug detectors, with the GNN-based bug detector having most of them uncovered. This shows that while a big share of...
bugs is uncovered by each of the three compared entities, there are still multiple unique findings that are exclusively detected by the developers or the neural bug detectors.

**False Alarms.** Figure 6b shows the overlap of false alarms (i.e., non-buggy tasks incorrectly classified as buggy) given by the developers and the two bug detectors. Here, we have 20 (non-buggy) tasks for which the developers did not give a majority decision and 82 (non-buggy) tasks without false alarms. In comparison to the bug detection rate, there is much less overlap. While developers and the GNN-based bug detector have a nearly equal bug detection rate, developers (with 18 tasks) produce 69.0% less false alarms than the GNN-based bug detector (with 58 tasks). The transformer-based bug detector’s false alarm number (16) is slightly lower than the one of the developers, however, since the bug detection rate with 24.4% is also lower, the ratio of bug detection to false alarms is still worse than that of the developers. Since the number of false alarms has a non-negligible influence on the usability (and thus usage) of bug detection tools, we see this as a significant difference. Bug detection done via static analysis often suffers from a high number of false alarms and this has been observed to be a major obstacle to the usage of such tools [19, 37, 46].

**Influence of Code Size.** Similar to the developers, we inspect the influence of code size on the confidence of the neural bug detectors in having given the right classification answer. Recall that confidence here is the probability computed by the bug detector that the given answer is correct. Figure 4 depicts the detection confidence of developers (already discussed) and neural bug detectors with respect to the program size in lines of code (LOC). The left hand side of each chart shows classification. Similar to the developer’s classification confidence, we observe that the classification confidence deteriorates with the increase in size. In comparison to the developers the program size has a stronger effect on the confidence of the transformer as visible in the median values for the three size classes (classification: 0.62, 0.44, 0.29, localization: 0.98, 0.08, 0.00). The GNN-based bug detector seems to be less affected by low to medium sized programs (classification: 0.93, 0.92, 0.53). Only large sized programs (>21 LOC) decrease the classification confidence of the GNN-based bug detector. In case of the localization task the median values drop similarly as for the transformer (localization: 1.00, 0.11, 0.00).

### 3.3 Comparison wrt. Localization

Next, we evaluate and compare the performance of developers and neural bug detectors with respect to localization. First of all, we need to fix what we mean by a majority decision for localization and which tasks we can consider during the comparison. We say that the localization of the developers is *correct* if the majority of the answers are correct localizations, and *incorrect* if the majority of the answers are incorrect localizations. In all other cases, e.g. when two answers are correct localizations and two are incorrect localizations, we observe no majority decision (this occurs for 16 tasks) and do not consider the task during evaluation. This leaves us with 117 tasks in total. For comparison, we again evaluate the neural bug detectors individually on all benchmark tasks.

**Localization.** Figure 6c depicts the overlap in correctly localized bugs by developers, GNN and transformer. We observe that developers correctly localized 49 of the 117 buggy lines (41.9%), the GNN 48 buggy lines (41.0%) and the transformer 44 (37.6%), which is a relatively high localization rate. For 42 tasks, neither developers nor neural bug detectors correctly localized the buggy line. Most importantly, we again observe that developers, GNN and transformer mark the same line as buggy for 23 tasks, being the highest overlap, just as for classification. A high bug detection rate coupled with a high false alarm rate, as we have observed for GNNs during classification, could indicate that the bug detector simply guesses. However, we found that the GNN bug detector still localizes most of the found bugs correctly and that the number of correctly localized bugs is comparable to developers and transformers.

**Influence of the Code Size.** Similar to classification, program size has an impact on the confidence in finding the right buggy line, as depicted in the right half of Figure 4. Recall that, for localization, confidence of the bug detectors means the probability of the error line containing a variable misuse. Whereas all three have a high confidence for selecting the buggy line for small programs, the confidence significantly decreases for larger programs: For programs with up to ten lines of code, the median of the confidences is 0.63 for developers, 1.00 for the GNN and 0.98 for the transformer. For larger programs, the median of the confidences decreases to 0.50 for developers, 0.11 for the GNN and 0.14 for the transformer for programs with eleven to 20 lines of code and to 0.33 for developers and 0.00 for both bug detectors for larger programs. Moreover, both...
bug detectors have a high variation in the confidence, as the boxes that include the median for the lower and upper half of the answers mostly ranges from zero to a confidence (close to) one. For example, the transformer assigns a probability below 10% to the buggy line for the vast majority of all tasks with more than 21 lines of code.

Summarizing all the results with respect to our research question, we get:

The performance of neural bug detectors is comparable to that of software developers, both for classification and localization. However, when studying the ratio of bug detection and false alarms, the advantage is on the side of the developers. In addition, the size of the program has significant impact on both developer and neural bug detector performance.

4 DISCUSSION

Besides pure numbers, we also wanted to get a feeling for the sort of code examples on which developers and neural bug detectors agree and disagree. To this end, we manually inspected over 100 code snippets again after the study, with an eye on finding commonalities with the developers and improvement options for neural bug detectors. In this section, we discuss insights gained from this inspection and give some ideas for how to address potential limitations of neural bug detection in the future.

Neural bug detectors are comparable to developers. A key outcome of our quantitative evaluation was the observation of a significant overlap between bugs detected and localized by a neural bug detector and those recognized by developers. Hence, we were interested in seeing what kind of bugs these are. We found that neural bug detection can be very effective for bugs that can be clearly identified from the surrounding context such as the bug shown in Listing 2. In this code snippet, the implementer (i.e. the software engineer who implemented the method) likely intends to retrieve an existing privateDataManager object, and – if it does not exist – to create a new privateDataManager based on the given connection. Therefore, it is quite likely that the condition in line 3 uses the wrong variable. This was actually recognized by all participants working on this task and both neural bug detectors. Even though the example code snippet represents an easily recognizable bug, it is still an instance of buggy code which is almost a duplicate of the line which contains the bug. In fact, an analysis of the training dataset revealed that this type of “copy-and-paste” bug is shown in Listing 3. Here, the bug was localized by all participants working on the task, while neither the graph neural network nor the transformer was able to identify the bug location. Since both neural bug detectors missed the bug, it can be suspected that this behavior is a consequence of the (common) training process. In fact, an analysis of the training dataset revealed that this type of copy-and-paste bug is heavily underrepresented with less than 1% of all training samples taking this form, i.e. containing a line of code which is almost a duplicate of the line which contains the bug. In the future, this can potentially be circumvented by increasing the data quality by more effective mutation strategies for the generation of the training set. For example, previous work has shown that the bug detection performance can be significantly improved by introducing more realistic mutants [7, 28, 33, 34]. Even though we already trained on such more advanced contextual mutants [34], we observe that the specific mutation location (e.g., a variable in a nearly duplicated statement) can have an impact. In fact, a more recent work [7] showed that these types of copy-and-paste bugs can be addressed in Python programs when the mutator is adversarially

Listing 2: Example of variable misuse detected by all participants and bug detectors

```java
protected static FrequencySet<String> combineMax(
    FrequencySet<String> a, FrequencySet<String> b) {
    FrequencySet<String> result = combineAndClip(a, b, 1);
    for (Map.Entry<String, MutableInt> entry : a.entrySet()) {
        result.get(entry.getKey()).v = entry.getValue().v;
    }
    for (Map.Entry<String, MutableInt> entry : a.entrySet()) {
        // BUG: a instead of b (Line 7)
        slot.v = Math.max(slot.v, entry.getValue().v);
    }
    return result;
}
```

Listing 3: Example of copy-and-paste bug missed by neural bug detectors

```java
public static synchronized PrivateDataManager
    getInstanceFor(XMPPConnection connection) {
    PrivateDataManager privateDataManager =
        instances.get(connection);
    if (connection == null) {
        // BUG: connection instead of privateDataManager (Line 3)
        privateDataManager =
            new PrivateDataManager(connection);
    }
    return privateDataManager;
}
```

4 DISCUSSION

Besides pure numbers, we also wanted to get a feeling for the sort of code examples on which developers and neural bug detectors agree and disagree. To this end, we manually inspected over 100 code snippets again after the study, with an eye on finding commonalities with the developers and improvement options for neural bug detectors. In this section, we discuss insights gained from this inspection and give some ideas for how to address potential limitations of neural bug detection in the future.

Neural bug detectors are comparable to developers. A key outcome of our quantitative evaluation was the observation of a significant overlap between bugs detected and localized by a neural bug detector and those recognized by developers. Hence, we were interested in seeing what kind of bugs these are. We found that neural bug detection can be very effective for bugs that can be clearly identified from the surrounding context such as the bug shown in Listing 2. In this code snippet, the implementer (i.e. the software engineer who implemented the method) likely intends to retrieve an existing privateDataManager object, and – if it does not exist – to create a new privateDataManager based on the given connection. Therefore, it is quite likely that the condition in line 3 uses the wrong variable. This was actually recognized by all participants working on this task and both neural bug detectors. Even though the example code snippet represents an easily recognizable bug, it is still an instance of buggy code which is almost a duplicate of the line which contains the bug. In fact, an analysis of the training dataset revealed that this type of “copy-and-paste” bug is shown in Listing 3. Here, the bug was localized by all participants working on the task, while neither the graph neural network nor the transformer was able to identify the bug location. Since both neural bug detectors missed the bug, it can be suspected that this behavior is a consequence of the (common) training process. In fact, an analysis of the training dataset revealed that this type of copy-and-paste bug is heavily underrepresented with less than 1% of all training samples taking this form, i.e. containing a line of code which is almost a duplicate of the line which contains the bug. In the future, this can potentially be circumvented by increasing the data quality by more effective mutation strategies for the generation of the training set. For example, previous work has shown that the bug detection performance can be significantly improved by introducing more realistic mutants [7, 28, 33, 34]. Even though we already trained on such more advanced contextual mutants [34], we observe that the specific mutation location (e.g., a variable in a nearly duplicated statement) can have an impact. In fact, a more recent work [7] showed that these types of copy-and-paste bugs can be addressed in Python programs when the mutator is adversarially
trained together with the bug detector in order to introduce hard to find mutants. Finally, it could also be interesting to more closely simulate the way human developers learn, e.g., by using real bugs for the training process, instead of mutations of real code.

**Longer code, more difficulties.** Our results indicate a clear trend: The performance of developers and neural bug detectors alike decreases as the length of the code snippets increases. While this is in line with prior research in neural bug detection [16], it is interesting that this trend is also visible for developers on modest size programs (20-40 LOC). Although prior studies on code readability [30] and code understandability [40] found no strong correlation (of readability/understandability) with program size, we hypothesize that the understandability of the implementers’ intention and hence the ability to argue about the existence of a variable misuse is much more difficult for larger code snippets. In fact, a related study [27] showed that identifying the purpose of a method implementation with an appropriate method name is much more difficult for human participants when the method is larger. We leave a more in-depth study of this phenomenon open for future research.

**GNNs can be self-inconsistent.** In comparison to developers and the transformer, the GNN produces a high number of false alarms. For this reason, we manually reviewed the output of the GNN on the 58 false alarms. Recall that the considered bug detectors do not only identify a bug but also propose a potential repair. In this case, we analyzed the potential repair to get a better understanding why the GNN produces a false alarm. Surprisingly, we found that the GNN is self-inconsistent for 12 of the 58 false alarms. In other words, after applying the proposed fix and running the GNN on the code again, the GNN still classifies the changed code as buggy and proposes to revert the change. An example for a non-buggy task, where this inconsistent behavior can be observed, is shown in Listing 4. The GNN identifies the variable a as a bug that can be fixed by replacing it with b. If we perform the false fix and provide the result to the GNN, it proposes to replace b with a. While this problem is known for GNNs [5], it is interesting that we could not reproduce a similar behavior with the transformer-based bug detector. Since there is no clear trend visible for the remaining 46 false alarms and the transformer generally produced a smaller number of false alarms, we could imagine that a hybrid approach (e.g., by using a transformer to validate alarms produced by the GNN) could mitigate this problem.

**Limited by the given context.** Finally, we manually inspected the 14 bugs that can neither be identified by the participants nor by the neural bug detectors. Generally, we found that the failed detection of bugs is not correlated with the program size, as methods with undetected bugs range from 5 to 37 LOC. Therefore, it is much more likely that the method implementation lacks the necessary information (e.g., the variable naming) necessary to detect the variable misuse. In fact, we found that especially smaller code snippets with undetected variable misuses often contain variables with unconventional names or calls of functions with less expressive names. For example, given the buggy code snippet in Listing 5, it becomes only apparent from the actual bug fix that the implementer aims to store the length of the message after the method `_put` is executed (temp − not as before `s` start). However, because the implementation of `_put` is unknown it is difficult to identify that the method call actually modifies the buffer `_buf`, which then would explain why `temp` is different from `s` start. Here, both developers and bug detectors are likely limited by the available context and incorporating knowledge about e.g. the method implementation can potentially resolve this issue. For developers, this limitation is artificial and an intended part of our study design. It can easily be resolved by giving a developer access to the remaining source code of the project. In fact, in a realistic scenario, a developer would likely be familiar with the project, have access to an IDE or debugger, and would potentially make use of execution-based tests. In contrast, current neural bug detectors do not directly support the integration of external knowledge which potentially limits the performance of bug detection. In the future, this limitation could potentially be overcome by integrating techniques from static analysis [32, 38] or code retrieval [26].

**Summary and Implications.** When the task of detecting variable misuse bugs was originally introduced [6], it was believed that software developers can easily recognize this type of bug as an error from experience. Motivated by this, several subsequent works [16, 34, 45] proposed to automate the detection of variable misuse bugs with learning based neural bug detectors. However, surprising at that time, it was discovered that the detection of variable misuse bugs found in real world projects is highly difficult for neural bug detectors. Since then, neural bug detectors for variable misuse bugs have been significantly improved by enhancing both the underlying learning algorithm [16, 45] and the employed training process [7, 21, 34].

In this work, we showed that the performance of developers is actually comparable to neural bug detectors (even though slightly better in avoiding false alarms), while having similar problems in detecting variable misuses. For example, finding variable misuses on longer code is more difficult for both bug detector and software developer. In addition, we found that not all real world tasks provide enough context to detect the variable misuse which limits both the developer’s and bug detector’s performance.

These similarities lead us to the question whether developers could benefit from integrating neural bug detectors in their debugging process. Neural bug detectors are often quite efficient in analyzing huge code corpora which allows to analyze a complete code base in a short amount of time. In addition, our study shows that neural bug detectors find a number of bugs comparable to the developers and can even uncover bugs that are missed by the developers. This is promising as this shows that neural bug detectors could potentially be used as a cheap utility to uncover potential bug locations.

Still, we have to acknowledge that developers are artificially limited in our study setup by the task definition (i.e. reviewing the implementation of a single function in a foreign code base) – a

![Listing 4: Self-inconsistent false alarm of the GNN.](image)
protected int putCodeWScope(String name, _codeWScope code) {
    _put(CODE_WS_CODE, name);
    _putValueString(code, _code);
    putObject(code, _scope);
    _buf.putInt(temp, _buf.position() - start);
    // BUG: start instead of temp (Line 8)
    return _buf.position() - start;
}

Listing 5: Example of an undetected variable misuse


5 THREATS TO VALIDITY

We briefly discuss internal and external threats to the validity of our results.

Internal validity: Training of bug detectors vs. developers. Even though both bug detectors and developers learn to detect bugs, the qualification and training for developers and neural bug detectors still differ. Bug detectors are specifically trained for variable misuse bugs while developers are used to detect all kinds of bugs and may inadvertently report different kinds of bugs during their trials. As a countermeasure, our study explicitly explained the task and allowed the participant at any time to look up a description of variable misuse. In addition, we trained the neural bug detector on millions of examples generated via mutation to detect buggy tasks, localize and repair the bugs. In contrast, we presented only three real world examples to the participants and relied on their experience. We think that this difference in training data reflects the reality and is therefore uncritical. Finally, our participants localize bugs via selecting a code line while the bug detectors report the likelihood that a token is buggy. For our evaluation, we then localize the buggy line of a bug detector by choosing the line with the token with the highest likelihood. Since a developer also applies some kind of likelihood when choosing the variable misuse (i.e., the token for the misused variable) and then selects the line, this mapping is acceptable.

Internal validity: Participant heterogeneity. Due to the variety of developers participating in our study, the quality of the participants’ answers may be of mixed quality. Linking buggy classifications with bug localization may provoke participants to favor the answer “non-buggy” or to guess a buggy code line if they cannot localize a bug. To counteract this, we have no preselection of the classification or of a buggy line, thus, we enforce the participant to make an explicit decision. Still, participants may have guessed answers. While we cannot exclude this completely, we only included complete trials in our evaluation, i.e., we only used the trials when a participant was engaged until the end. In addition, we removed trials that were completed too quickly, i.e., which indicate that all answers were guessed.

Internal validity: Aggregation. Our aggregation of the participants’ answers may have influenced our results as has been observed by Siegmund et al. [41] for other studies. We chose the majority decision and filtered out tasks without a majority vote. Alternatively, we also explored other tie breaking strategies (e.g., optimistic or pessimistic) without any filtering, but found that this does not change the overall evaluation outcome. Still, more aggressive filtering strategies or an alternative handling of ties may result in different outcomes.

External validity: Participation group. While we tried to mitigate this threat by using different recruitment channels, by design we only recruited voluntary participants. In addition, the majority of the developers are from Europe. Our results may not generalize beyond European developers.

External validity: Bug type selection. Our study evaluates the performance of developers and bug detectors on the common task of variable misuse bug detection. Therefore, our results might not generalize to other types of bugs found in open source projects [22]. However, among the limited types of bugs supported by neural bug detectors, variable misuses belong to the most frequent class of bugs [22, 43] supported by most neural bug detectors [16, 21, 45]. Therefore, we believe that our results and our comparison with software developers is representative for the state-of-the-art in neural bug detection for variable misuse bugs. Further, the bottlenecks we identified for neural bug detectors on variable misuse bugs (like missing context) are independent of the bug type. It is likely that these turn out to be bottlenecks for neural bug detectors designed for other bug types as well. Still, as our study is one of the first comparing neural bug detectors and developers on bug detection, it is an open question whether our results on the performance of software developers will generalize to other bug types.

External validity: Task selection. Our study concentrates small auto-formatted Java methods selected from the ManyStuBs4J benchmark. Therefore, our results might not transfer to larger methods / programs, other programming languages or benchmarks. However, our tasks result from a variety of open-source Java projects and real bug fixes. Thus, we believe that our results also apply to other (at least, small) Java methods. Moreover, code formatting is quite common among developers and used in many (Java) projects. We expect that our auto-formating has little to no influence on our results.

6 RELATED WORK

As learning-based techniques are getting increasingly more proficient at handling tasks traditionally done only by software engineers, it also becomes more interesting to see whether the learned behavior is actually similar to that of developers. In the following, we discuss and compare with related work that address (a) the comparison between developers and learning based models in tasks like program comprehension, code summarization and program repair, (b) studies that investigate debugging performance of developers and (c) neural bug detectors in general.

---

3Defects4J [20] or Bugs.jar [39], for example, focus more on various different kinds of bugs found in complete compilable projects.
Comparison with Developers. Several previous works [18, 27, 35, 47, 48] have addressed the question whether learning based models read and understand programs in a similar way to human developers. Rodeghero et al. [35], for example, compared whether Java developers attend to the same code elements as prioritized by a tf-idf model during code summarization. Itoh et al. [18] and Pal-tengahti and Pradel [27] addressed a similar question by comparing the eye movement of developers during code summarization with the attention mechanism of more sophisticated summarization models. Interestingly enough, it was found that code elements required by a developer for summarization and those found to be important by the more advanced models are highly similar. In addition, the later work showed that learning-based models and developers agree on methods that are difficult to summarize and that both developers and neural models especially struggle on longer code snippets. In our study, we made similar observations in the context of bug detection. For example, we found that bugs identified by developers and those found by neural bug detectors greatly overlap. In addition, we found that developers and neural bug detectors alike struggle to identify bugs when the method size increases.

However, not all prior work focused on the code summarization task or could identify a significant overlap between developers and the algorithms behavior. For example, Wainakh et al. [47] explored whether the learned similarity between identifiers actually correlate with the developers intuition of identifier similarity. They found that none of the seven tested algorithms accurately represent the similarity between identifiers. Further and probably most related to our study, Zhang et al. [48] investigated whether developers perform comparable or better than existing automated methods for program repair. They found that developers usually perform better than automated methods in terms of correct fixes while automated methods can further improve the performance of developers. In general, our study has a similar outcome for neural bug detection. Although there exists up to 28 bugs uniquely identified by bug detectors, developers still perform slightly better in bug detection when compared with an individual bug detector. Nevertheless, the existence of uniquely identified bugs indicates that neural bug detectors could supplement the manual code review. Further exploration in future work is needed.

Debugging studies. Debugging and software maintenance in general play an important role in the software development process [3]. Therefore, several studies were conducted to understand how developers can be effectively supported in the debugging process [2, 10, 11, 25, 29]. Böhme et al. [11], for example, explored how effective developers are in finding and repairing real bugs in open C projects. They found that developers often fail to produce correct patches and they identified a huge potential for automating this process. With our study, we can confirm that similar observations can be made for variable misuse in Java code snippets: we found that developers fail to identify all variable misuses, and automated bug detectors do not only have a performance comparable to developers, but there also exists a significant number of bugs uniquely identified by them. This shows that there is in fact potential for automation.

Neural bug detection. Research in neural bug detection mostly focuses on two types of learning-based algorithms: (a) text-based algorithms [16, 21, 45] that directly operate on source code without additional information and (b) graph-based algorithms [6, 7, 16] that exploit additional information gained through static analyses of the source code. Motivated by the variety of algorithms available, Hellenboorn et al. [16] evaluated and compared several algorithms on the variable misuse task on Python functions. They found that graph-based algorithms generally perform better in the detection of bugs than text based algorithms. In our study, we employed representative algorithms for both lines of work and made a similar observation for variable misuses in Java methods. Although the text-based transformer avoids more false positives, the graph-based neural network is more effective in identifying bugs.

7 CONCLUSION
In this study, we explored whether software developers have a performance comparable to neural bug detectors with respect to the identification of variable misuses in Java method implementations. Our study results reveal that there is a significant overlap of bugs found and localized by developers and those identified by neural bug detectors. In addition, when faced with the same tasks we found that developers suffer from similar difficulties as bug detectors such as bug detection in long, undocumented or inconveniently named methods. Even though the performance of developer and bug detector is comparable on variable misuses, bug detectors still have several limitations as compared to a developer. For example, by analyzing the training dataset we found that there exists at least one bug category (copy-and-paste errors) which is severely underrepresented and thus restricts the performance of the bug detector. Bugs of this category are often easily detectable by a developer. Finally, we found that (specific) bug detectors can still produce a significant number of false alarms which could hinder the adoption into practice and hence should be addressed in future work. Altogether, we believe that our study provides helpful insights for understanding the relation between developers and learning-based systems in the context of bug detection and code reviewing.

DATA AVAILABILITY STATEMENT
All our study results and implementations used for our evaluation are open source and available at Zenodo [8]. To allow the replication of our evaluation and facilitate future work, participant answers are also provided (in an anonymous form).

ACKNOWLEDGMENTS
This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (SFB 901). We thank Gözel Shakeri, Gerald Schwabe, Arnab Sharma, Lara Bargmann, Maik Wiesner, and Linghui Luo, who helped us to improve our study design during our pre-study. We also thank the participants of our study, who made this study possible, and the anonymous reviewers for their insightful comments and suggestions which have helped to improve this work.

REFERENCES