OffSide
Learning to Identify Mistakes in Boundary Conditions

Arnar Briem, Jón; Smit, Jordi; Sellik, Hendrig; Rapoport, Pavel; Gousios, Georgios; Aniche, Maurício

DOI
10.1145/3387940.3391464

Publication date
2020

Document Version
Final published version

Published in
ICSEW'20

Citation (APA)
Arnar Briem, J., Smit, J., Sellik, H., Rapoport, P., Gousios, G., & Aniche, M. (2020). OffSide: Learning to Identify Mistakes in Boundary Conditions. In *ICSEW'20: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (pp. 203-208). Association for Computing Machinery (ACM). https://doi.org/10.1145/3387940.3391464

OffSide: Learning to Identify Mistakes in Boundary Conditions

Jón Arnar Briem, Jordi Smit, Hendrig Sellik, Pavel Rapoport, Georgios Gousios, Maurício Aniche
Delft University of Technology
The Netherlands
j.a.briem,j.smit-6,h.sellik,p.rapoport@student.tudelft.nl,G.Gousios,M.F.Aniche@tudelft.nl

ABSTRACT

Mistakes in boundary conditions are the cause of many bugs in software. These mistakes happen when, e.g., developers make use of '<' or '>' in cases where they should have used '<=' or '>='. Mistakes in boundary conditions are often hard to find and manually detecting them might be very time-consuming for developers. While researchers have been proposing techniques to cope with mistakes in the boundaries for a long time, the automated detection of such bugs still remains a challenge. We conjecture that, for a tool to be able to precisely identify mistakes in boundary conditions, it should be able to capture the overall context of the source code under analysis. In this work, we propose a deep learning model that learn mistakes in boundary conditions and, later, is able to identify them in unseen code snippets. We train and test a model on over 1.5 million code snippets, with and without mistakes in different boundary conditions. Our model shows an accuracy from 55% up to 87%. The model is also able to detect 24 out of 41 real-world bugs; however, with a high false positive rate. The existing state-of-the-practice linter tools are not able to detect any of the bugs. We hope this paper can pave the road towards deep learning models that will be able to support developers in detecting mistakes in boundary conditions.

KEYWORDS

software engineering, software testing, boundary testing, machine learning for software engineering, machine learning for software testing, deep learning for software testing.

1 INTRODUCTION

Software systems commonly suffer from defects that are caused by mistakes in boundary conditions. In simple words, these mistakes happen when developers use '<' or '>' in cases where they should have used '<=' or '>=' (or vice-versa).

As an example, see the two code snippets, extracted from JPacMan [4], an educational Java PacMan game used to teach software testing, in Listing 1. The method `withinBorders()` decides whether the coordinates (i.e., `x` and `y`) are within the game map. To that aim, the method performs four comparisons: whether `x` and `y` are positive numbers, and whether `x` and `y` are within the

```
// Incorrect code: x > 0
public boolean withinBorders(int x, int y) {
    return x > 0 && x <= getWidth() && y >= 0 &&
        y <= getHeight();
}

// Correct code: x >= 0
public boolean withinBorders(int x, int y) {
    return x >= 0 && x <= getWidth() && y >= 0 &&
        y <= getHeight();
}
```

Figure 1: An example of a mistake in a boundary condition. Code extracted from JPacman[2].

width and the height of the map. Although each of these conditions might seem simple in isolation, when put together, the expression becomes complex and prone to mistakes. A less attentive developer might, for example, write `x > 0` instead of `x >= 0`, which would cause a bug in the game. The same type of mistake might also happen inside loops. *Off-by-one errors*[1] happen when a loop executes one time too many or too few.

These mistakes are particularly difficult to find in source code. After all, the result of the program is not always obviously wrong, as it is "merely off by one". In most cases, the mistake will lead to an "out of bounds" situation, which will then result in an application crash. In worst scenarios, such bugs can even lead to security breaches, as they might not crash an application, but lead to arbitrary code execution or memory corruptions, which are potentially exploitable by adversaries [3].

Not surprisingly, many software engineering researchers have been proposing (manual) boundary testing techniques (e.g., [7, 9, 15, 19, 20]) since the early days. However, manually inspecting code for off-by-one errors is very time-consuming since determining which binary operator is actually the correct one is usually heavily context-dependent. The industry has been relying on static analysis tools, such as SpotBugs or PVS-Studio. SpotBugs promises to identify possible infinite loops, as well as array indexes, offsets, lengths, and indexes that are out of bounds. PVS-Studio also tries to identify mistakes in conditional statements and indexes that are out of bounds in array manipulation. And while they can indeed find some of them, static analysis tools are known for providing a high number of false positives [5, 10, 11].[3]

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3391464

[1]https://en.wikipedia.org/wiki/Off-by-one_error
[2]https://github.com/serg-delft/jpacman
[3]Although the papers we cite did not focus specifically on the mistakes we are discussing here, we will later show in our study that these static analysis tools are not enough in detecting mistakes in boundary conditions.

In this work, we conjecture that a model based on distributed path representations of methods [2] is able to learn syntactical patterns, including code context, that can lead to off-by-one errors. To examine this, we train and test *OffSide*, a Code2Vec-like model on over 1.5 million code snippets, mutated to include errors in boundary conditions. Our model shows an accuracy from 55% up to 87% (depending on the type of statement where the boundary error was in). We also mine and manually examine 41 real-world bugs attributable to off-by-one errors and show that our model is able to identify the locations of 24 of them. Existing state-of-practice linter tools are not able to detect any of them.

The main contributions of this paper are.

- The proposal of a deep learning model for detecting of mistakes in boundary conditions.
- A quantitative and qualitative evaluation of our model in a training set of more than 1.5 million code snippets and a testing set of 28k code snippets.
- A research agenda that aims at improving the model.

2 APPROACH

Path representations of graphs are a general technique for embedding graphs into lower dimensions [17]. In the context of source code, path representations work by walking the AST and encoding the sequence of nodes encountered by the walks. Those walks usually start from random terminal nodes and end at other terminal nodes. By taking various random walks on the AST and embedding the result, models, such as Code2Vec, can learn the *context* in which specific token sequences are found as well as an overall representation of code, given a training label.

OffSide is based on the premise that the attentional layer in Code2Vec will learn the appropriate weights to enable it to differentiate between cases where path embeddings encode errors in boundary conditions. Intuitively, to achieve maximum discriminatory power, it should be trained with path embeddings of pairs of correct and non-correct examples of the same code. Such pairs can be easily generated by introducing artificial mutation operations that inject bugs, similarly to Pradel and Shen [18].

3 DATA COLLECTION

To train *OffSide*, we need both positive (containing bugs) code and negative code (bug-free). We use Code2Vec's dataset as a starting point, and create bugs by performing specially-designed mutations, inspired by Pradel and Shen [18]. In the following, we discuss each of these steps in detail.

3.1 Datasets

To train and validate *OffSide*, we use, as basis, the same `java-large` dataset collected by Alon et al. [22]. It consists of 1,000 top-starred projects from GitHub and contains about 4 million code snippets. From that dataset, we filter out (1) all methods not containing a '<', '<=', '>' or '<=' (hereafter referred to as *comparators*), (2) constructors, and (3) files that the parser we used, *JavaParser*[4], is unable to parse. This results in a base dataset of 1,357,210 methods containing a total of 2,736,573 comparators.

[3]https://code2vec.org/
[4]https://javaparser.org/

Table 1: Distribution of the type of statements containing comparators in our base dataset.

Statement	Count	Percent	Distribution of the comparators			
			>	>=	<	<=
If	1,382,841	50.53%	41.9%	17.5%	30.0%	10.5%
For	921,946	33.69%	1.4%	3.4%	90.6%	4.6%
While	104,412	3.79%	31.4%	11.8%	49.1%	7.7%
Ternary	100,567	3.65%	44.1%	13.0%	34.5%	8.4%
Return	87,804	3.21%	31.2%	25.5%	23.5%	19.8%
Method	70,268	2.55%	45.4%	19.6%	22.4%	12.5%
Assert	23,708	0.87%	27.6%	31.1%	17.9%	23.4%
Var. decl.	18,500	0.68%	46.2%	18.3%	24.3%	11.2%
Assign	11,129	0.41%	43.1%	21.6%	23.4%	12.0%
Do	9,839	0.36%	35.6%	12.5%	44.8%	7.1%
Expression	4,965	0.18%	41.1%	15.7%	31.3%	11.9%
Obj. creation	558	0.02%	48.6%	25.3%	15.4%	10.8%
Array init.	26	0.00%	38.5%	42.3%	7.7%	11.5%
Cast	10	0.00%	20.0%	30.0%	30.0%	20.0%

For further analysis of our base dataset and the performance of *OffSide* with different sub-types of boundary mistakes, we define two classifications for each comparator. The type of operator (*less*, *lessEquals*, *greater*, or *greaterEquals*) and the type of statement it occurs in (there are 14 different types: *if*, *for*, *while*, *return*, *ternary operator*, *method*, *assert*, *do*, *variable declaration*, *assign*, *expression*, *object creation*, *array initialization*, and *cast*). The type of statement containing the comparator is defined by the class of the parent node in the AST.

The distributions of types of comparators and the types of statements containing those comparators in the base dataset can be seen in Table 1. We observe that the distributions of comparators and statement types are not uniform (i.e., the data set is imbalanced).

3.2 Generating Positive and Negative Instances

For each code snippet m in our dataset, we first identify all statements S_t of a given type t, $t \in \{if, for, while, ...\}$ (see the full list in Table 1). For each type of statement t that exists in code snippet m, we randomly select one $s \in S_t$. Let $C \in \{>, \geq, <, \leq\}$ be all the comparators that exist in the statement s. We randomly select one $c \in C$, and mutate it (i.e., '<' is replaced by '<=' and vice versa, same goes for '>' and '>='). Both the original method (bug-free) and all its mutations (buggy) are added to our final preprocessed dataset.

This process results in a 54:46 proportion of positive (containing a bug) and negative (bug-free) instances during training. Overall, our training set consists of 1,512,785 data points (bug-free and buggy methods), our validation set of 28,086 data points, and our test set of 104,958 data points. The split into training, test and validation datasets is based on the split in the original java-large dataset collected by Alon et al. [2]. Since that split is done on a project level, this leads to the test dataset only containing methods from projects that are not included in the other two datasets and vice versa.

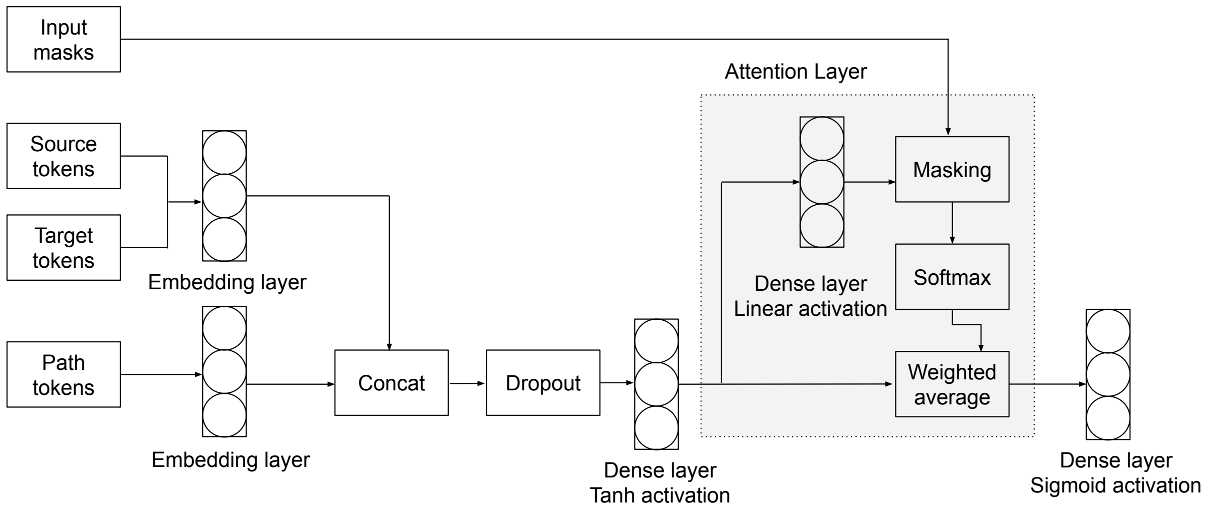

Figure 2: The neural network architecture to identify mistakes in boundary conditions.

3.3 Source code representation

After a method has been extracted, it passes through the same preprocessing pipeline as used in the original Code2Vec paper [2]. The source code of each method is again turned into an AST using the modified JavaExtractor from [22]. We then select at most 200 paths between 2 unique terminals in the AST of the method. We encode these terminals into integer tokens using the dictionary used by Code2Vec [2] and hash the string representation of the paths with Java hashcode method. This means that each method in Java code is turned into a set of at most 200 integer tuples of the format $(terminal_i, path, terminal_j)$ whereby $i \neq j$ and $path$ is an existing path in the AST between source $terminal_i$ and target $terminal_j$.

4 MODEL

4.1 The Neural Network Architecture

OffSide is an attention-based model based on Code2Vec, whereby the overwhelming majority of the weights are in the embedding layer of the network. The architecture of the model is shown in Figure 2. The model takes a set of at most 200 integer token tuples of the format $(terminal_i, path, terminal_j)$ as an input. It embeds these inputs into a vector with 128 parameters, whereby the terminal tokens and the path tokens each have their own embedding layer. These embeddings are concatenated into a single vector and passed through a dropout layer. These 200 vectors are then combined into a single context vector using an attention mechanism. The context vector will be used to make the final prediction. We modify the output of the Code2Vec model, by adding a Sigmoid activation with a single (binary) output unit; this enables Code2Vec to classify input code as buggy or non-buggy.

4.2 Training

We use binary cross-entropy as the loss function and Adam [14] as the optimization algorithm. The training process is halted after the accuracy on the validation set did not increase for 2 epochs and the weights with the lowest validation loss were kept.

The authors of Code2Vec speculate that their pre-trained weights could be used for transfer learning [2]. We experiment with applying transfer learning in two ways. Initially, we attempt "feature extraction" whereby the pre-trained weights of the Code2Vec model were frozen and only the final layer was replaced and made trainable. Then, we try "fine-tuning" with pre-trained weights of the Code2Vec model as the initial value of our model and allow the model to update all the values as it sees fit, expect the embeddings weights. Finally, we also train a model with "randomly initialized weights" as a baseline. We base our further results on the "fine-tuning" model, as it proved best in terms of F1-score (see results in the appendix [13]).

4.3 Reproducibility

All the source code and datasets can be found in our code repository.[5] The dataset we used can be found on Zenodo.[6]

5 EVALUATION

We define two research questions to evaluate *OffSide*'s performance:

RQ$_1$. What is the accuracy of *OffSide* in detecting errors in boundary conditions?

RQ$_2$. How does *OffSide* perform on real-world errors in boundary conditions, when compared against the state-of-the-practice linting tools?

To answer RQ$_1$, we apply *OffSide* on the test set composed of **104,958 data points** (see Section 3.2). We evaluate the performance of the model by means of traditional binary classification performance metrics, such as precision, recall, accuracy, and F1.

To answer RQ$_2$, we apply *OffSide* on **41 real-world bugs in boundary conditions** that we extract from the 500 most starred GitHub Java projects. The analyzed projects are not part of our

[5]https://github.com/SERG-Delft/ml4se-offside
[6]https://zenodo.org/record/3606812

training and test sets before, and thus are never seen by our model. By means of a Pydriller script [21], we extract a list of candidate commits where authors made a change in comparators (e.g., a '>' by a '<='). This process return a list of 1,571 candidate commits. We then 1) randomly select one candidate commit, 2) manually analyze the change and the commit message to identify whether it is a bug fix, 3) apply our model to the buggy (where we expect the model to identify an error) and to the bug-free version (where we expect our model to not predict an error) of the code, and 4) apply state-of-the-practice linting tools. We stop this process after identifying 41 bugs, which we deem satisfactory for this analysis.[7]

We use three different static analyzers as a baseline for our evaluation. **SpotBugs** (v.4.0.0-beta1)[8], formerly known as FindBugs[8], is an open-source static code analyzer for Java. It analyzes Java bytecode for occurrences of different patterns that are likely containing a bug. At the time of writing this report, SpotBugs is able to identify over 400 of such patterns[9], out of which six we consider to be relevant for the type of bug we are detecting: IL_INFINITE_LOOP (an apparent infinite loop), RpC_REPEATED_CONDITIONAL_TEST (repeated conditional tests), RANGE_ARRAY_INDEX (array index is out of bounds), RANGE_ARRAY_OFFSET (array offset is out of bounds), RANGE_ARRAY_LENGTH (array length is out of bounds), and RANGE_STRING_INDEX (string index is out of bounds). We also use **PVS-Studio** (v.7.04.34029)[10] which is a proprietary static code analysis tool for programs written in C, C++, C# and Java. Out of 75 possible patterns for Java code analysis, two were suitable for our evaluation: V6003 (potential error in a construct consisting of conditional statements), and V6025 (index value is outbound the valid range). Finally, we use the static analyzer integrated into **IntelliJ IDEA** Ultimate[11] (v. 2019.2.3), a popular IDE among Java developers.

5.1 RQ1: What is the accuracy of *OffSide* in detecting errors in boundary conditions?

In Table 2, we show the performance metrics of our models, grouped by statement type and comparator, respectively. In the appendix [13], we discriminate the performance of each statement type and its all possible comparators.

The average accuracy of the model is around 0.65 to 0.75. Overall, we observe that our models provided a median accuracy of 0.67 (when looking at the accuracy numbers per statement, Table 2) and 0.75 (when looking at the accuracy numbers per comparator, see Appendix). Moreover, the F1 measures (which, different from the accuracy metric, takes both false positives and false negatives into account) present similar numbers.

The accuracy per statement type is correlated to the amount of training data. We observe that the precision is correlated with the total amount of data points available for each context type and the types which have the highest number of occurrences also tend to produce a higher F1 score. For example, our model achieves an F1 score of 0.87 when detecting bugs in *for loops*, which are well represented in our dataset. However, our model only achieves an F1 score of 0.52 detecting bugs when *assigning* a boolean value to a variable with a logical condition, a case that is severely underrepresented in the training data.

[7]We specifically focused on *for loops* in the last three bugs, as to increase its presence in our dataset.
[8]SpotBugs official GitHub page: https://github.com/spotbugs/spotbugs
[9]https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html
[10]PVS-Studio official home page: https://www.viva64.com/en/pvs-studio/
[11]https://www.jetbrains.com/idea/

Table 2: Performance metrics of our model per statement type.

Statement type	Total	Accuracy	Recall	Precision	F1
If	49,418	0.72	0.6666	0.7462	0.7042
For	39,018	0.8723	0.8565	0.8844	0.8702
While	5,718	0.7272	0.6691	0.757	0.7104
Return	3,558	0.7476	0.6931	0.7779	0.7331
Ternary	3,114	0.6574	0.5466	0.7021	0.6147
Method	1,954	0.6592	0.5916	0.684	0.6345
Assert	608	0.6135	0.5164	0.6408	0.5719
Do	598	0.689	0.5886	0.7364	0.6543
Var. decl.	544	0.6581	0.5368	0.7087	0.6109
Assign	336	0.5982	0.4464	0.641	0.5263
Expression	72	0.6111	0.4167	0.6818	0.5172
Obj. creation	20	0.55	0.4	0.5714	0.4706
Total	104,958	0.7733	0.7303	0.799	0.7631

We observe that the model can also perform well with boundary mistakes in moderately underrepresented classes such as *return statements* (F1 score of 0.73) and *while loops* (F1 score of 0.71). This might indicate that the problems were similar enough for the errors in *if statements* and *for loops* for the model to generalize. We conjecture that the most underrepresented classes like *assigning value* to a variable are noisy and the model was not able to generalize towards those classes.

The model is biased towards *for* loops. A detailed analysis shows that the classical *for loop* (with < operator) scores are significantly higher than others (86-89% accuracy). This might be due to *for loops* with comparators such as *(int i = 0; i < number; i++)* being considered as a boilerplate in Java code. We also observe that the model is biased towards predicting <= in a for a loop as a bug and < not as a bug. This can be explained by the balance of the training set where the majority of the for loops contain a < operator. Hence, we conjecture that the model learns to classify our mutated code with <= operator as faulty.

Training on all types of statements seems to benefit specific statements. We were also surprised by the positive results in *if statements*, as the model achieved an accuracy and F1 score of 0.72 and 0.70, respectively. *If statements* have no default structure, as *for loops* have, and thus, knowledge of the context is needed in order to make a good prediction. To better understand our results, we trained a model specifically for *if conditions* to see how it would perform on the test data. [12] Interestingly, the results are not better than the model trained with all mutations and the model performs worse when judging *if statements* that contain a > operator (F1 score 0.63 vs 0.34). One possible reason for this might be that the model can generalize the relationship better with more data, independent of the contexts like *if conditions* or *for loops*.

[12]The results of this specific model can be seen in our appendix [13].

RQ$_1$ summary: Deep learning models can learn how to identify mistakes in boundary conditions. Our initial models provides an average accuracy of 65% to 75%. The amount (and quality) of training data plays an important role in the process.

5.2 RQ2: How does *OffSide* perform on real-world errors in boundary conditions, when compared against the state-of-the-practice linting tools?

We present the performance of *OffSide* in the 41 real-world boundary mistakes (both in the buggy version as well as in the fixed version of the code snippet) in Table 3.

The model is able to detect real-world bugs, but with a high false positive rate. We observe that our tool was correctly able to identify the bug in 24 out of the 41 cases, or 58% of the cases. Out of these 24 correct predictions, the tool was able to also correctly label the non-buggy version of the code as bug-free 16 times (66% of the 24 correct predictions, or 39% of the 41 total cases).

The state-of-the-practice linter tools did not find any of the real-world bugs. We also note that these bugs are indeed not trivial to be identified. In fact, they were not identified by any of the state-of-the-practice linting tools. Therefore, we did not include their results in the table.

RQ$_2$ summary: The model presents a reasonable performance in real-world bugs. The current false positive rate, however, is high. The state-of-the-practice linters, on the other hand, do not identify such bugs.

6 DISCUSSION AND FUTURE WORK

We consider our initial exploration on whether deep learning models can learn to identify mistakes in boundary conditions to be successful. Our models had an average accuracy of 0.75 and, in real-world bugs from projects never seen before by the model, an accuracy of almost 60%. However, much still needs to be before this model can be used in the real-world:

A better, larger, and more balanced dataset. In the first version of the model, we had a training dataset of around 1.5M methods. Other similar papers make use or larger datasets, e.g., around 12M snippets in Code2Vec's paper [2] and around 5M snippets in DeepBugs [18]. Moreover, we can see in Table 2 that elements with more data present higher accuracy. We therefore expect that a larger dataset will naturally give us better accuracy.

Moreover, as we show in Section 3.1, our initial dataset contains four times fewer usages of >= or <= compared to usages of > or <. We also have a larger number of $i < x$, rather than <=, >, or >=, inside *for loops*. These differences can lead to biased training and, as a result, models tending to give false positive results in case of >= or <=. One way to mitigate the issue is to create a balanced dataset with a more equal distribution of binary operators, as well as the distribution of the places of their occurrence (if-conditions, for- and while-loops, ternary expressions, etc.) In addition, while the amount of method duplication is small in Code2Vec's dataset [1], removing duplicates might improve our accuracy; and while our approach does not currently suffer from equivalent mutants [6], this is indeed a point of attention for the future when we will make use of more advanced mutation techniques.

Finally, while we identified 41 real-world off-by-one-bugs, a larger dataset of real bugs might be of great importance in evaluating the accuracy of our model. The development (or even the reuse) of datasets like Defects4J [12] is, thus, an important future work.

Table 3: Results of applying *OffSide* in 41 real-world boundary bugs.The value 1 means that the tool marked the analyzed snippet as buggy. A ✓ indicates where *OffSide* made the right decision.

#	Bug fix	Statement Type	Bug Prediction	No-Bug Prediction
1	>= to >	Var. decl	✓	✓
2	<= to <	If	✓	✓
3	>= to >	Method	✓	✓
4	<= to <	If	✓	✓
5	> to >=	Method	✓	X
6	>= to >	If	✓	✓
7	>= to >	If	X	✓
8	< to <=	For	X	✓
9	> to >=	Var. decl	X	✓
10	>= to >	If	X	✓
11	> to >=	If	✓	✓
12	>= to >	If	✓	✓
13	> to >=	If	X	X
14	> to >=	If	X	✓
15	> to >=	If	✓	X
16	>= to >	If	✓	✓
17	>= to >	If	✓	✓
18	> to >=	If	✓	✓
19	<= to <	While	✓	✓
20	<= to <	If	✓	X
21	> to >=	If	✓	X
22	> to >=	Method	✓	X
23	> to >=	Method	X	X
24	> to >=	Method	✓	✓
25	> to >=	If	X	✓
26	> to >=	If	X	✓
27	>= to >	Assign	X	X
28	> to >=	If	✓	✓
29	< to <=	Ternary	✓	X
30	> to >=	If	X	X
31	>= to >	If	X	✓
32	> to >=	Assert	✓	X
33	>= to >	If	X	X
34	< to <=	For	X	✓
35	> to >=	While	✓	✓
36	> to >=	While	X	X
37	> to >=	While	✓	X
38	< to <=	If	X	X
39	<= to <	For	✓	✓
40	> to >=	For	✓	✓
41	< to <=	For	X	X

Support inter-procedural analysis. Currently, we only analyze one method at a time. To detect more complex bugs in boundary conditions, we conjecture that this knowledge is essential, and omitting the contents of called methods might lead to unpredictable results. We plan to explore techniques that would allow us to expand the context of the code representation, as in Li et al. [16].

Better understand the differences in RQ_1 and RQ_2. Interestingly, our accuracy was higher in the results of our RQ_1 than in RQ_2. By manually analyzing the 41 bugs we selected for RQ_2, we see no reason for the model to have a lower accuracy. We intend to better understand the root cause of this difference in future work.

Fine-grained bug identification. Our model currently predicts boundary mistakes at method-level. That is, the model predicts whether a method contains a boundary mistake or not. It is not able to detect exactly which boundary is buggy. While we argue that this is already helpful for developers (developers could, for example, increase their code review efforts in methods marked as buggy by our model), future work should focus on identifying, in a fine-grained manner, where the bug is.

The generalization to other languages and problems. Mistakes in boundary conditions might happen in source code of any programming language. We wonder how much our approach would generalize to other (not only static, but also dynamic) languages and, whether deep learning models would be able to learn such bugs. Moreover, right now we solely focus on *off-by-one* errors. Future work can explore the effectiveness of our models in detecting any types mistakes in boundaries (e.g., not only detect a > that was mistakenly a >=, but also mistakenly a <, <=, ==, or ≠).

6.1 Threats to Validity

Internal validity. We consider a boundary to happen in "any existing inequality expression in the source code", i.e., any source code construct that makes use of >, >=, <, and <=. Given that our method makes use of mutations to generate buggy instances, and therefore, a clear limitation of our model is that it never learns to detect the *absence* of a boundary check. We expect future researchers to work on models that detect not only "bugs in boundaries that are already expressed in the source code", but also "missing boundary checks".

External validity. More replications of this work (including industry systems) are required before we claim generalizability.

7 RELATED WORK

DeepBugs by Pradel et al. [18] uses a deep learning model to identify bugs related to swapped function arguments, wrong binary operators and wrong operands in binary operation. DeepBugs feeds method and argument names into Word2Vec to calculate identifier embeddings and then aggregats those in a feed-forward neural network that predicts whether a code is buggy or not. Interestingly, the model is trained on artificially created bugs.

8 CONCLUSION

Developers are prone to make mistakes in boundary conditions. Offering them mechanisms to detect such mistakes before they become bugs in their software systems is fundamental. Given that such mistakes are highly dependent on context, traditional static analysis tools currently do not yield accurate results. This paper explores the idea of identifying such mistakes by means of learning deep models. Our initial exploration shows promising results.

REFERENCES

[1] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, 2019.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.

[3] Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities.* Pearson Education, 2006.

[4] M Finavaro Aniche, FFJ Hermans, and A van Deursen. Pragmatic software testing education. In *SIGCSE 2019-Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* Association for Computing Machinery (ACM), 2019.

[5] Katerina Goseva-Popstojanova and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015.

[6] Bernhard JM Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 192–199. IEEE, 2009.

[7] Daniel Hoffman, Paul Strooper, and Lee White. Boundary values and automated component testing. *Software Testing, Verification and Reliability*, 9(1):3–26, 1999.

[8] David Hovemeyer and William Pugh. Finding bugs is easy. *Acm sigplan notices*, 39(12):92–106, 2004.

[9] Bingchiang Jeng and Elaine J Weyuker. A simplified domain-testing strategy. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(3):254–270, 1994.

[10] Brittany Johnson. A study on improving static analysis tools: Why are we not using them? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1607–1609. IEEE, 2012.

[11] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.

[12] René Just, Darioush Jalali, and Michael D Ernst. 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*, pages 437–440, 2014.

[13] Hendrig Sellik Pavel Rapoport Georgios Gousios Maurício Aniche Jón Arnar Briem, Jordi Smit. Offside: Learning to identify mistakes in boundary conditions (appendix). Complementary tables: https://figshare.com/articles/Offside_Learning_to_Identify_Mistakes_in_Boundary_Conditions_Appendix_/11689227; Source code: https://github.com/SERG-Delft/ml4se-offside; Dataset: https://zenodo.org/record/3606812, 2020.

[14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from z and b. In *International Symposium of Formal Methods Europe*, pages 21–40. Springer, 2002.

[16] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA): 1–30, 2019.

[17] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 701–710, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2956-9. doi: 10.1145/2623330.2623732. URL http://doi.acm.org/10.1145/2623330.2623732.

[18] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA): 147, 2018.

[19] Stuart C Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Proceedings Fourth International Software Metrics Symposium*, pages 64–73. IEEE, 1997.

[20] Philip Samuel and Rajib Mall. Boundary value testing based on uml models. In *14th Asian Test Symposium (ATS'05)*, pages 94–99. IEEE, 2005.

[21] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 908–911. ACM, 2018.

[22] Shaked Brody Uri Alon, Omer Levy and Eran Yahav. Code2seq: Generating sequences from structured representations of code. *ICLR*, 2019.