# Using Distributed Representation of Code for Bug Detection

**Analytics and Machine Learning for Software Engineering**
**IN4334 - Group 5**

Jón Arnar Briem
4937864
TU Delft
j.a.briem@student.tudeft.nl

Jordi Smit
4457714
TU Delft
j.smit-6@student.tudeft.nl

Hendrig Sellik
4894502
TU Delft
h.sellik@student.tudeft.nl

Pavel Rapoport
4729889
TU Delft
p.rapoport@student.tudeft.nl

## ABSTRACT

Recent advances in neural modeling for bug detection have been very promising. More specifically, using snippets of code to create continuous vectors or *embeddings* has been shown to be very good at method name prediction and claimed to be efficient at other tasks, such as bug detection. However, to this end, the method has not been empirically tested for the latter.

In this work, we use the Code2Vec model of Alon et al. to evaluate it for detecting off-by-one errors in Java source code. We define bug detection as a binary classification problem and train our model on a large Java file corpus containing likely correct code. In order to properly classify incorrect code, the model needs to be trained on false examples as well. To achieve this, we create likely incorrect code by making simple mutations to the original corpus.

Our quantitative and qualitative evaluations show that an attention-based model that uses a structural representation of code can be indeed successfully used for other tasks than method naming.

## 1 INTRODUCTION

The codebases of software products have increased yearly, now spanning to millions making it hard to grasp the knowledge of the projects [12]. All this code needs to be rapidly developed and also maintained. As the amount of code is tremendous, it is an easy opportunity for bugs to slip in.

There are multiple tools that can help with this issue. Most of the time, working professionals rely on static code analyzers such as the ones found in IntelliJ IDEA[1]. However, these contain a lot of false positives or miss the bugs (such as most off-by-one errors) which make the developers ignore the results of the tools [6]. Recently, a lot of Artificial Intelligence solutions have emerged which help with the issue [1, 2, 13, 15]. Although they are not a panacea, they provide enhanced aid to developers in different steps of developing processes, highlighting the potentially faulty code before it gets into production.

One particular solution by Alon et al., the Code2Vec model [2] (see Section 2 for description), delivers state-of-the-art performance on method naming. The authors do not test the model on other tasks, however, they theorize that it should also yield good performance.

In this work, we use the Code2Vec deep learning model and replace the layer for method naming with a binary classification layer. The aim is to repurpose the model for detecting off-by-one logic errors found in boundary conditions (see Section 3). Our intuition is that the change in the Abstract Syntax Tree (AST) weights upon introducing a bug as seen in Figure 1 is learnable by our model. Hence the system should be able to learn to identify off-by-one bugs.

The main contributions of this paper are.

- Replicating work done by authors of Alon et al. [2]
- Quantitative and qualitative evaluation of a Code2Vec's performance on a task other than method naming.

The paper is divided into the following sections. In Section 2 the relevant literature used to create this paper is discussed, in Section 3 the data origins and preprocessing is explained, in Section 4 the architecture and training of the model are discussed. Finally, in Section 5, the model is evaluated which is followed by some reflections regarding our work in Section 6 and a conclusion in Section 7.

## 2 RELEVANT LITERATURE

*Code2Vec* by Alon et al. [2] is a state-of-the-art deep neural network model used to create fixed-length vector representations (*embeddings*) from code. The embeddings of similar code snippets encode the semantic meaning of the code, meaning that similar code has similar embeddings.

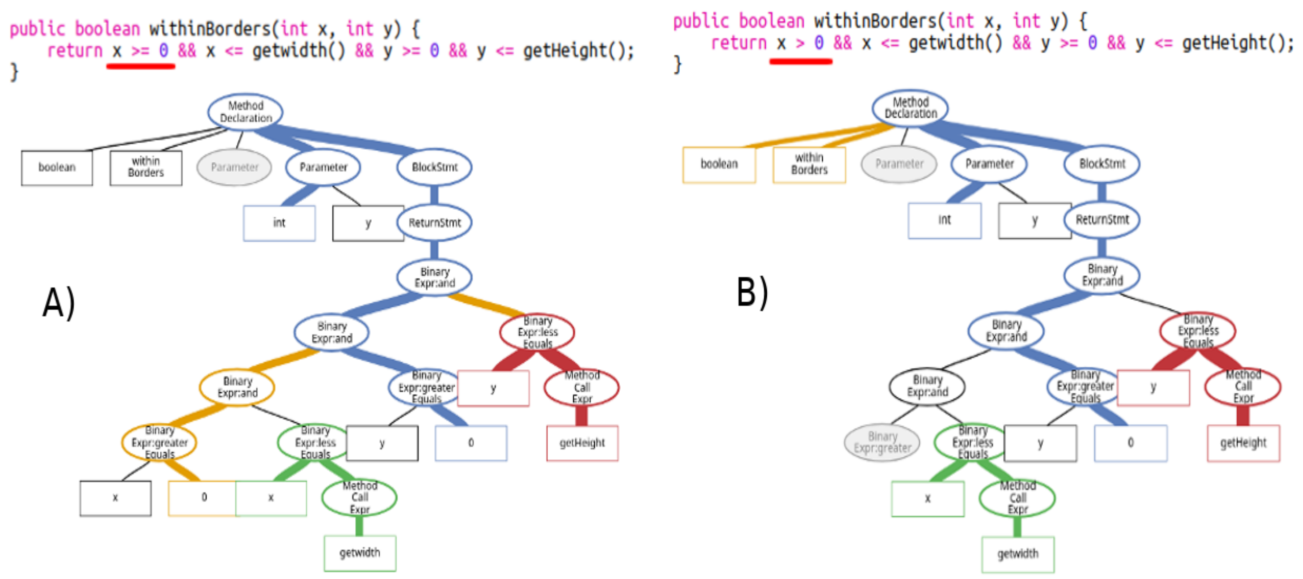
The model relies on the paths of the Abstract Syntax Tree (AST) of the code. While the idea of using AST paths to generate code embeddings is also used by other authors such as Hu et al. [8], Code2Vec is superior due to the novel way of using AST paths. Instead of linearizing paths as [8], Code2Vec authors use a path-attention network to identify the most important paths and learn the representation of each path while simultaneously learning how to aggregate a set of them.

The authors make use of embedding similarity of similar code to predict method names. The model is trained on a dataset of 12 million Java methods and compared to other competitive approaches. It significantly outperforms them by having approximately 100 times faster prediction rate at the same time having a better F1 score of 59.5 at method naming. While they only tested their model for method naming, the authors believe that there are a plethora of programming language processing tasks the model can be used for.

*DeepBugs* by Pradel et al. [13] uses a deep learning model to identify bugs related to swapped function arguments, wrong binary operators and wrong operands in binary operation. The model creates an embedding from the code, but instead of using ASTs like Code2Vec, the embedding is created from specific parts of the code. For example, embeddings for identifying swapped function arguments are created from the name of the function, names and types of the first and second arguments to the function with their parameter names, and name of the base object that calls the function.

[1]https://www.jetbrains.com/idea/

[2]https://code2vec.org/



```
public boolean withinBorders(int x, int y) {
    return x >= 0 && x <= getwidth() && y >= 0 && y <= getHeight();
}
```

```
public boolean withinBorders(int x, int y) {
    return x > 0 && x <= getwidth() && y >= 0 && y <= getHeight();
}
```

**Figure 1: Example change of AST weights after introducing bug using Code2Vec web interface**[2]
(Option *A* being correct and *B* incorrect code)

They use a Javascript dataset containing likely correct code. In order to generate negative examples for training, they use simple transformations to create likely incorrect code. The authors formulate bug detection as a binary classification problem and evaluate the embeddings quantitatively and qualitatively. The approach yields an effective accuracy between 89.06% and 94.70% depending on the problem at hand.

## 3 DATA

In this section, we describe the data used to train our model. In Section 3.1 we first describe the type of bug we aim to detect, in Section 3.2 we describe the data we used. In Section 3.3 we describe how we mutated the data to generate positive code (containing bugs) and finally we describe how we represent source code as a vector using the approach defined by Alon et al. [2] in Section 3.4.

### 3.1 Off-by-one errors

Off-by-one errors (in Java terms '<' vs '<=' and '>' vs '>=') are generally considered to be among the most common bugs in software. They are particularly difficult to find in source code because there exist no tools which can accurately locate them and the result is not always obviously wrong, it is merely off-by-one. In most cases, it will lead to an "Out of bounds" situation which will result in an application crash. However, it might not crash an application and might lead to arbitrary code execution or memory corruptions, potentially exploitable by adversaries [5].

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. This is also why we chose to base our approach on the work done by Code2Vec [2], this allows the model to discover code context heuristics which static code analyzers are not able to capture.

| Comparator | Count | Percentage |
|---|---|---|
| greater | 755,078 | 27.43% |
| greaterEquals | 35,9455 | 13.06% |
| less | 1,389,789 | 50.48% |
| lessEquals | 248,848 | 9.04% |

**Table 1: Distribution of comparators found in the java-large dataset collected by Alon et al.[14]**

### 3.2 Datasets

To train and validate our project we use the java-large dataset collected by Alon et al. [14]. It consists of 1000 top-starred projects from GitHub and contains about 4 million examples. Furthermore, since the Code2Vec model was already trained on this data we reduce the transfer learning needed to adapt the weights between the two models.

The distributions of '<', '<=', '>' or '<=' (hereafter referred to as *comparators*) and the types of statements containing those comparators in the original dataset can be seen in tables 3.2 and 3.2. As can be seen, the distributions of both comparators and statement types is far from uniform with < accounting for over half of the comparators and *if* statements containing a similar share of all comparators. We considered trying to balance the dataset and see what effect that would have on the accuracy of the model but were unable to do so due to time constraints.

| Statement Type | Count | Percentage |
|---|---|---|
| If | 1,399,436 | 50.83% |
| For | 9,219,48 | 33.49% |
| While | 104,412 | 3.79% |
| Ternary | 100,567 | 3.65% |
| Method | 70,268 | 2.55% |
| Other | 68,735 | 2.50% |

**Table 2: Distribution of the type of statements containing comparators found in the java-large dataset collected by Alon et al.[14]**

For our training data we need both positive (containing bugs) code and negative code (bug-free), to do this we take the aforementioned dataset and create bugs by performing specially designed mutations. First, we parse the code using Java Parser [3] to generate an AST based on the source code. We then search the AST for methods that might contain the bug in question, generate a negative example by performing the mutation, which will be further explained in the next section. Then we take both the negative and positive examples and send them through the preprocessing pipeline used in [2] before passing them into our model. This process is described in detail in the next sections.

## 3.3 Mutations

To generate our mutations we consider each method and search for any occurrences of comparators. If the method contains a comparator it is extracted from the code for further manipulation, any methods not containing a comparator are ignored. Once we have extracted all methods of interest from the code, the method is parsed for different *context types* that comparators occur in. This is done so that we can account for different sub-types of off-by-one errors and also evaluate if our model is better at detecting bugs in some contexts than others.

```
public void setContents(List<Content>
    contentsBefore, List<Content> contentsAfter) {
    for (int i = 0; i < contentsAfter.size();
        i++) {
        Content content = contentsAfter.get(i);
        if (content instanceof PathContent) {
            paths.add((PathContent) content);
        }
    }
}
```

**Listing 1: An example of a method before mutation. The context type of this comparator is FORless**

We define a context type as the combination of the operator (less, lessEquals, greater or greaterEquals) and the type of statement it occurs in (For loop, If statement, While loop, etc), more specifically this is the class of the operator's parent node in the AST. For a full list of all context types refer to Appendix B. For every method, a random comparator is selected, mutated and turned into its respective off-by-one comparator (for example, '<' will be mutated into '<='). Finally both the original and the mutated methods are added to the dataset.

```
public void setContents(List<Content>
    contentsBefore, List<Content> contentsAfter) {
    for (int i = 0; i <= contentsAfter.size();
        i++) {
        Content content = contentsAfter.get(i);
        if (content instanceof PathContent) {
            paths.add((PathContent) content);
        }
    }
}
```

**Listing 2: An example of a method after mutation. The context type of this comparator is FORlessEqueals**

## 3.4 Source code representation

After a method has been extracted, it is passed through the same preprocessing pipeline as was used in the original Code2Vec paper [2]. The source code of each method is again turned into an AST using the modified JavaExtractor from [2]. 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[4]. 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 $terminal_j$.

In this paper, we want to apply transfer learning from the original Code2Vec model [2] to our problem. Allowing us to reuse the pre-trained weights from the original Code2Vec model [2]. However, we can only do this if our model preprocesses and encodes the source code in the same way as described in the original Code2Vec paper [2]. That is why our preprocessing code is based on the Java Extractor Jar and dictionaries as published on the repository [5] of Code2Vec and tailored to our needs.

## 4 MODEL

In this section, we describe our model in detail. Section 4.1 describes the architecture of the neural network. Section 4.2 describes how the network was trained. Finally, we cover the precision of our model in Section 4.4.

## 4.1 Neural Network Architecture

The model is an attention-based model based on Code2Vec [2] whereby the overwhelming majority of the weights are in the embedding layer of the network. The architecture of the model has been depicted in Figure 2. The model takes a set of at most 200 integer token tuples of the format $(terminal_i, path, terminal_j)$ as

[3]https://javaparser.org/

[4]https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#hashCode--
[5]https://github.com/tech-srl/code2vec

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.

Our model's architecture is similar to the architecture of the original Code2Vec model [2]. The only difference is in the final layer where our model uses Sigmoid activation with a single output unit. This is because we have a binary classification problem instead of a multi-classification problem. We chose to use the same architecture as Code2Vec to allow the use of transfer learning using the pre-trained weights of Code2Vec [2]. This also allows us to verify the claim made by the Code2Vec authors that there are a plethora of programming language processing tasks that their model can be used for [2].

## 4.2 Training

For the training and validation process, we preprocessed the raw Java training and validation dataset collected by Alon et al. [14]. The generation process resulted in a training set of 1, 512, 785 data points (mutated or correct Java methods) and a validation set of 28, 086 data points that were used to train and test our model respectively. We used large over the medium dataset [14] because it resulted in slightly higher overall scores (Table 4). For this training process, we used binary cross-entropy as our loss function and Adam [10] as our optimization algorithm. The model was also trained using early stopping on the validation set. The training process was 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 have speculated that their pre-trained weights could be used for transfer learning [2]. That is why we experimented with applying transfer learning in two ways. Firstly, we attempted Feature Extraction whereby the pre-trained weights of the Code2Vec model were frozen and only the final layer was replaced and made trainable. Secondly, we tried Fine-Tuning with pre-trained weights of the Code2Vec model as the initial value of our model and allowed the model to update all the values as it saw fit, expect the embeddings weights. Finally, we also trained a model with randomly initialized weights as a baseline. The resulting accuracies are displayed in table 3.

| Model | Feature Extraction | Fine-Tuning | Randomly Initialized | Transformer model |
|---|---|---|---|---|
| Accuracy | 0.732 | 0.788 | **0.790** | 0.717 |
| F1 | 0.709 | **0.781** | 0.777 | 0.674 |
| Precision | 0.776 | 0.809 | **0.831** | 0.794 |
| Recall | 0.653 | **0.756** | 0.730 | 0.585 |

**Table 3: Evaluation comparison of different architectures on an unseen test of** 88, 228 **examples based on the mutated test set from Alon et al. [14]**

| Model | Fine-Tuning medium dataset | Fine-Tuning large dataset |
|---|---|---|
| Accuracy | 0.758 | **0.788** |
| F1 | 0.757 | **0.781** |
| Precision | 0.762 | **0.809** |
| Recall | 0.751 | **0.756** |

**Table 4: Evaluation comparison of the medium data set (**482, 138 **training examples) and the large data set (**1, 512, 785 **training examples) [14] .**

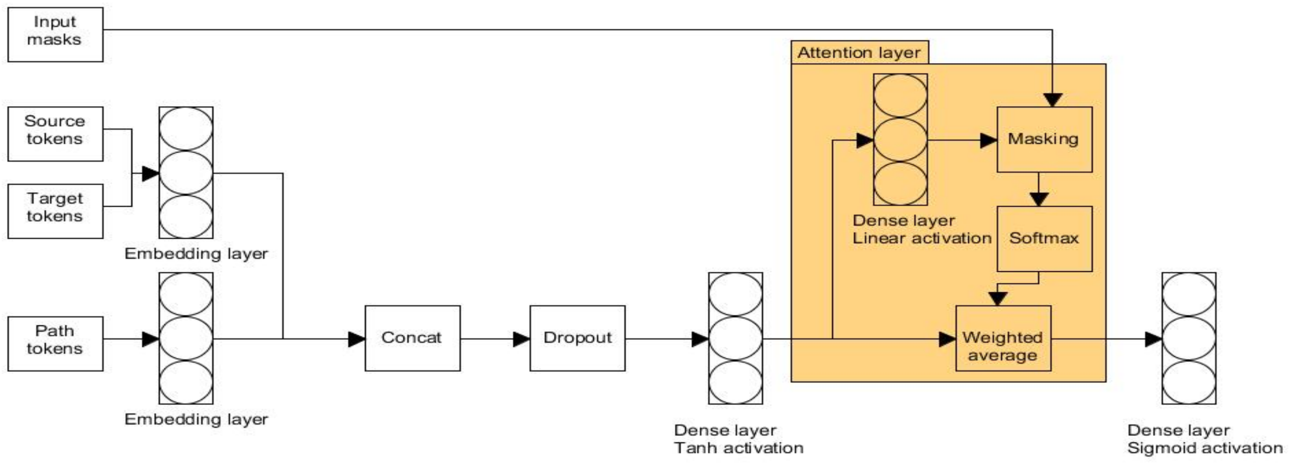


**Figure 2: Neural network architecture**

## 4.3 Alternative Architecture

Besides the original Code2Vec architecture, we also tried a more complicated architecture based on BERT [4]. The idea behind this architecture was that the power of the Code2Vec [2] architecture lays in the attention mechanism. To verify this idea we tried a transformer model based on BERT. This model encodes the source cod,e in the same manner as the Code2Vec model. It then embeds these input tokens using the frozen pre-trained embedding layers from Code2Vec. Finally, it passes these embedding through multiple BERT layers before making the final prediction. However as can be seen in Table 3, this architecture achieved a much lower overall score than the original Code2Vec architecture. This is most likely due to the observed Java code AST being limited to method scope as mentioned in Section 3. Since our preprocessing is highly depended on the Code2Vec preprocessing pipeline and the overall scores for this model were lower, we did not pursue this architecture any further.

## 4.4 Predicting Bug Location

Our model looks for bugs with a method-level precision, that is, it predicts whether a method contains an off-by-one bug or not. It is not able to detect where exactly inside that method the bug is. This is due to our effort to capture the context of the whole method. Looking at a smaller section of code would yield more precisely located bugs, but possibly at the cost of context which would reduce the accuracy of the model.

## 5 EVALUATION

In this section, we present the results for our quantitative analysis to better understand our model and select the best performing architecture. We also perform a qualitative evaluation with the best performing architecture and compare it to static analyzers.

## 5.1 Quantitative Evaluation

Different model architectures were compared with regards to simple metrics such as accuracy, F1 score, precision and recall. Out of the 4 tested architectures, the fine-tuning Code2Vec architecture was selected for further evaluation based on the metrics (see Table 3). This architecture is also used in the qualitative evaluation.

To better understand the model and to analyze the context types in which it performs the best, we broke down the performance of the model by context type (see Table 8 in Appendix B). The test dataset is previously unseen code from the same data-gathering project and mutation method as used to train the model (see Section 3), hence the distribution of different bug opportunities remains the same.

From Table 8, it is evident 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.

It is notable that the model can also perform well with off-by-one errors in moderately underrepresented classes such as *return statements* (F1 score of 0.73) and *while loops* (F1 score of 0.71). This might mean that there was just enough data or the problems were similar enough for the off-by-one errors in *if statements* and *for loops* for the model to generalize. The most underrepresented classes like *assigning value* to a variable are noisy and the model was not able to generalize towards those classes.

In order to get more detailed insight, the 12 cases were further divided into 48 (see Table **??** in Appendix B). A total of 104 958 data points (Java methods) were created for testing. Each data point contains a method with a comparator and a possibility for an off-by-one error. The names in the first column of Table **??** indicate the class of the off-by-one error (for example *FOR* stands for *for loop*) and the comparator indicates which comparator was passed to the model (for example *less* stands for <). The comparator can be from the original code (hence likely correct), but it may also be introduced by a mutation (hence likely incorrect). Hence *FORless* is a method containing a *for loop* with a < operator which could have been mutated or not. We can then compare the model output with the truth.

The data points were passed into the model which gave a prediction for the data point to be a bug (true) or not a bug (false). The 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.

It can also be seen 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 model learns to classify our mutated code with <= operator as faulty.

The results with *if conditions* seemed fascinating from Table **??** because there is not a default structure as there is with *for loops* and knowledge of the context is needed to make a prediction. Hence we trained a model specifically with mutations in *if conditions* to see how it will perform on the test data.

The results of this specific model can be seen in Table 10 in Appendix B. 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*.

## 5.2 Qualitative Evaluation

While running our model to find bugs on Apache Tomcat[6], Apache Ant[7] and Apache Druid[8] projects, we were not able to find bugs that broke functionality, but problems that were related to code quality or false positives (see table 6 in appendix A). This also correlates with the results of Ayewah et al. [3] who found that most important bugs in production code are already fixed with expensive methods such as manual testing or user feedback. A much more realistic

[6]https://github.com/apache/tomcat
[7]https://github.com/apache/ant
[8]https://github.com/apache/incubator-druid

| Tool | Pattern Code | Meaning |
|---|---|---|
| **PVS-Studio** | V6003 | The analyzer has detected a potential error in a construct consisting of conditional statements. |
| | V6025 | When indexing into a variable of type 'array', 'list', or 'string', an 'IndexOutOfBoundsException' exception may be thrown if the index value is outbound the valid range. The analyzer can detect some of such errors. |
| **SpotBugs** | 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 |
| | RANGE_STRING_INDEX | String index is out of bounds |

**Table 5: Static analysis tools and their specific patterns used for comparison.**

use-case for such tools is in the developing stage, where the benefit is the largest [9]. Hence we analyze code snippets concerned with off-by-one errors and highlight the situations in which the model succeeds over static analyzers and the situations it does not.

We used 3 different static analyzers as a baseline for our evaluation. **SpotBugs** (v.4.0.0-beta1)[9], formerly known as FindBugs[7], 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[10], out of which 6 are relevant for this report (see table 5).

Secondly, we used **PVS-Studio** (v.7.04.34029)[11] which is a proprietary static code analysis tool for programs written in C, C++, C# and Java. It is aimed to be run right after the code has been modified to detect bugs in the early stage of developing process[12]. Out of 75 possible patterns for Java code analysis[13], 2 were suitable for our evaluation which are listed in table 5. Thirdly, we used the static analyzer integrated into **IntelliJ IDEA** Ultimate[14] (v. 2019.2.3).

The results show that the static analyzer of IntelliJ and SpotBugs are not able to detect off-by-one errors even if the size of the iterated array is explicitly stated. It is possible to observe this in the first example of table 6, where our model and PVS-Studio are able to detect the issue.

However, from the same table, it can be seen, that our model has learned to identify <= operator in for loops as a bug which confirms the results of quantitative analysis in Section 5.1. This was also the case when applying the model to GitHub projects. This could be considered as a styling issue as it is considered a best practice to loop with a < comparator.

It can be also be seen, that the false positives with <= do not affect other bugs, such as code analyzed in Table 7 example 2. In the example, the <= operator is replaced with < by mistake. Our model reports it as a bug while none of the static analyzers are able to detect this mistake. This is a case where our model outperforms static analyzers.

## 6 REFLECTIONS

From the work conducted, several points can be singled out as discussing topics.

### 6.1 Advantages

- **Better performance in non-trivial cases**: static analyzers that we used for evaluation can detect only very specific cases of off-by-one error. Our model allows to predict a much greater variety of off-by-one error cases.
- **Versatility**: The presented model can be trained to detect not only off-by-one errors but many other kinds of bugs with relatively small changes during the preprocessing stage. Only the algorithm used to create mutations should be changed in order to train our model to detect a different bug as long as a suitable training dataset exists.

### 6.2 Issues and potential improvements

- **Using untraditional coding style leads to false positives**: using a for-loop of type *for (i = start; i < end; ++i)* is very popular and a commonly used style. As a result, our model has 'learned' that any case of a for loop using <= is most likely a bug which in reality has a high chance of being a false positive.
- **Unbalanced dataset** As stated in Section 3.2 our initial dataset contains 4 times fewer usages of >= or <= compared to usages of > or <. This difference can lead to biased training and as a result, our model is more tending to give false-positive results in case of >=/<= usage. One of the ways to reduce this influence is the creation of balanced dataset with 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.)

[9]SpotBugs official GitHub page: https://github.com/spotbugs/spotbugs
[10]https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html
[11]PVS-Studio official home page: https://www.viva64.com/en/pvs-studio/
[12]https://www.viva64.com/en/t/0046/
[13]PVS-Studio Java patterns: https://www.viva64.com/en/w/#GeneralAnalysisJAVA
[14]https://www.jetbrains.com/idea/

- **Unknown behavior on long methods**: we currently consider at most 200 context paths. This is acceptable for our dataset where most of the methods are not very long. However, if input methods will be longer this might not be enough to provide decent predictions. The severity of this issue should be checked via further experiments on an appropriate dataset. However, it should be noted that increasing the size of context paths will also increase the computing time.
- **Current method constraints**: The AST paths extracted are constrained to the current method only. This issue is an artifact of the Code2Vec data extraction method. This approach is valid for their purpose since they do not need to know what is happening inside of the child methods which are called from the parent method to predict the name of the latter. However, for bug detection this knowledge is crucial and omitting contents of called methods might lead to unpredictable results. This could be solved by expanding the AST with the content of some of the inner method calls. However, as with the previous case, the maximum number of AST paths should be kept in mind.
- **Bug Creation**: Our dataset was created by inserting bugs into code using mutations similar to the approach used by [13]. This approach allows us to have plenty of data for training but depends on the quality of the original dataset. If that dataset contains many off-by-one errors, the model will interpret those errors as correct code resulting in lower accuracy. It is also possible that off-by-one errors that happen 'naturally' when developers write code are in some way different from the errors we created with mutations. If so, our model may not be able to correctly detect those 'natural' off-by-one bugs.

## 6.3 Future work

As future research, the same method could be applied to other languages. The model should benefit more with languages with dynamic typing, such as Javascript or Python. For the latter, a context path extractor[15] was recently created by Kovalenko et al. [11].

It might also be interesting to see if it is possible to achieve a higher overall score by not limiting the AST paths to the method scope. This should be possible since the model with the randomly initialized weights achieved similar overall scores to the model with fine-tuned weights. Hence, one is free to use altered encodings that span more than one method.

In addition, the method could be evaluated on different kinds of bugs. There is also some room for improvement by balancing the training set with regards to the comparators in specific parts of code, like *for loops*.

## 7 CONCLUSION

We used encouraging results of Code2Vec model to test it on detecting off-by-one errors in arbitrary sized Java methods. The core idea of using a soft-attention mechanism to gain vector representations from AST paths remained the same. However, we modified the final layer to repurpose the model for detecting bugs instead of naming a method.

We tested different architectures of the model and tried to apply transfer learning. However, transfer learning proved only to be slightly better than a randomly initialized model and did not speed up the training process significantly. We trained the model on a large Java corpus of likely correct code to which we introduced simple mutations to get faulty samples.

The results of the quantitative and qualitative analyses show that the model has promising results and can generalize off-by-one errors in different contexts. However, the model suffers from the bias of the dataset and generates false positives for exotic code that deviates from standard style. In addition, the current model only analyses the AST of a single method, hence context is possibly lost which would allow detecting a bug.

We believe that this method could be tested with other bugs, hence all of our code and links to our training data are available at https://github.com/serg-ml4se-2019/group5-deep-bugs.

[15] https://github.com/vovak/astminer/tree/master/astminer-cli

## REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[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] Nathaniel Ayewah and William Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 241–252. ACM, 2010.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

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

[6] 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.

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

[8] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210. ACM, 2018.

[9] 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.

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

[11] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Pathminer: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 13–17. IEEE Press, 2019.

[12] Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):37, 2013.

[13] 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.

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

[15] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720*, 2019.

## A QUALITATIVE EVALUATION CODE

| Tool | Output |
|---|---|
| Java Code | `public static void example() {`<br>`    int[] array = new int[5];`<br>`    for (int i = 0; i <= 5; i++) {`<br>`        System.out.println(array[i]);`<br>`    }`<br>`}` |
| Gold | 1 (bug) |
| PVS-Studio | 1 |
| IntelliJ | 0 |
| SpotBugs | 0 |
| Our model | 1 |
| Java Code | `public static void example() {`<br>`    int array[] = new int[6];`<br>`    for (int i = 0; i <= 5; i++) {`<br>`        System.out.println(array[i]);`<br>`    }` |
| Gold | 0 (correct) |
| PVS-Studio | 0 |
| IntelliJ | 0 |
| SpotBugs | 0 |
| Our model | **1** (model wrong) |
| Java Code | `public static void example() {`<br>`    int array[] = new int[5];`<br>`    for (int i = 0; i < 5; i++) {`<br>`        System.out.println(array[i]);`<br>`    }`<br>`}` |
| Gold | 0 (correct) |
| PVS-Studio | 0 |
| IntelliJ | 0 |
| SpotBugs | 0 |
| Our model | 0 |
| Java Code | `// Apache Tomcat Code`<br>`private boolean isPong() {`<br>`    return indexPong >= 0;`<br>`}` |
| Gold | 0 (correct) |
| PVS-Studio | 0 |
| IntelliJ | 0 |
| SpotBugs | 0 |
| Our model | **1** (model wrong) |
| Java Code | `// Apache Tomcat Code`<br>`public long getIdleTimeMillis() {`<br>`    final long elapsed =`<br>`    System.currentTimeMillis()`<br>`    - lastReturnTime;`<br>`    return elapsed >= 0 ? elapsed : 0;`<br>`}` |
| Gold | 0 (correct) |
| PVS-Studio | 0 |
| IntelliJ | 0 |
| SpotBugs | 0 |
| Our model | **1** (style issue) |

**Table 6: Qualitative Evaluation Results**

| Tool | Output |
|---|---|
| Java Code | `public boolean inBorders(int x, int y) {`<br>`    return x >= 0 && x <= getWidth()`<br>`        && y >= 0 && y <= getHeight();`<br>`}` |
| Gold | 0 (correct) |
| PVS-Studio | 0 |
| IntelliJ | 0 |
| SpotBugs | 0 |
| Our model | 0 |
| Java Code | `public boolean inBorders(int x, int y) {`<br>`    return x > 0 && x <= getWidth()`<br>`        && y >= 0 && y <= getHeight();`<br>`}` |
| Gold | 1 (bug) |
| PVS-Studio | 0 |
| IntelliJ | 0 |
| SpotBugs | 0 |
| Our model | 1 |
| Java Code | `public boolean contains(float value) {`<br>`    if (value > from && value <= to)`<br>`        return true;`<br>`    else`<br>`        return false;`<br>`}` |
| Gold | 1 (bug) |
| PVS-Studio | 0 |
| IntelliJ | 0 |
| SpotBugs | 0 |
| Our model | 1 |
| Java Code | `public boolean contains(float value) {`<br>`    if (value >= from && value <= to)`<br>`        return true;`<br>`    else`<br>`        return false;`<br>`}` |
| Gold | 0 (correct) |
| PVS-Studio | 0 |
| IntelliJ | 0 |
| SpotBugs | 0 |
| Our model | 0 |

**Table 7: Qualitative Evaluation Results**

## B QUANTITATIVE EVALUATION

| **Context Type** | **TP** | **TN** | **FP** | **FN** | **Total** | **Acc** | **Recall** | **Precision** | **F1** |
|---|---|---|---|---|---|---|---|---|---|
| FORlessEquals | **15906** | 177 | 573 | 2016 | **18672** | 0.8613 | **0.8875** | **0.9652** | **0.9247** |
| FORless | 214 | **16505** | 1417 | 536 | **18672** | **0.8954** | 0.2853 | 0.1312 | 0.1798 |
| IFgreaterEquals | 7835 | 3831 | **2253** | **3025** | 16944 | 0.6885 | 0.7215 | 0.7767 | 0.7480 |
| IFgreater | 3530 | 9290 | 1570 | 2554 | 16944 | 0.7566 | 0.5802 | 0.6922 | 0.6313 |
| IFlessEquals | 4498 | 900 | 735 | 1632 | 7765 | 0.6952 | 0.7338 | 0.8595 | 0.7917 |
| IFless | 605 | 5087 | 1043 | 1030 | 7765 | 0.7330 | 0.3700 | 0.3671 | 0.3686 |
| WHILEgreaterEquals | 813 | 237 | 150 | 251 | 1451 | 0.7236 | 0.7641 | 0.8442 | 0.8022 |
| WHILEgreater | 185 | 881 | 183 | 202 | 1451 | 0.7347 | 0.4780 | 0.5027 | 0.4901 |
| WHILElessEquals | 877 | 57 | 87 | 387 | 1408 | 0.6634 | 0.6938 | 0.9098 | 0.7873 |
| WHILEless | 33 | 1070 | 194 | 111 | 1408 | 0.7834 | 0.2292 | 0.1454 | 0.1779 |
| RETURNgreaterEquals | 553 | 284 | 104 | 245 | 1186 | 0.7057 | 0.6930 | 0.8417 | 0.7601 |
| RETURNgreater | 266 | 685 | 113 | 122 | 1186 | 0.8019 | 0.6856 | 0.7018 | 0.6936 |
| TERNARYgreaterEquals | 421 | 83 | 88 | 355 | 947 | 0.5322 | 0.5425 | 0.8271 | 0.6553 |
| TERNARYgreater | 77 | 627 | 149 | 94 | 947 | 0.7434 | 0.4503 | 0.3407 | 0.3879 |
| FORgreater | 488 | 129 | 122 | 98 | 837 | 0.7372 | 0.8328 | 0.8000 | 0.8161 |
| FORgreaterEquals | 107 | 515 | 71 | 144 | 837 | 0.7431 | 0.4263 | 0.6011 | 0.4988 |
| METHODgreaterEquals | 375 | 57 | 63 | 181 | 676 | 0.6391 | 0.6745 | 0.8562 | 0.7545 |
| METHODgreater | 42 | 441 | 115 | 78 | 676 | 0.7145 | 0.3500 | 0.2675 | 0.3032 |
| TERNARYlessEquals | 304 | 49 | 56 | 201 | 610 | 0.5787 | 0.6020 | 0.8444 | 0.7029 |
| TERNARYless | 48 | 437 | 68 | 57 | 610 | 0.7951 | 0.4571 | 0.4138 | 0.4344 |
| RETURNlessEquals | 302 | 120 | 79 | 92 | 593 | 0.7116 | 0.7665 | 0.7927 | 0.7794 |
| RETURNless | 111 | 338 | 56 | 88 | 593 | 0.7572 | 0.5578 | 0.6647 | 0.6066 |
| METHODlessEquals | 111 | 83 | 50 | 57 | 301 | 0.6445 | 0.6607 | 0.6894 | 0.6748 |
| METHODless | 42 | 129 | 39 | 91 | 301 | 0.5681 | 0.3158 | 0.5185 | 0.3925 |
| ASSERTgreaterEquals | 56 | 66 | 16 | 48 | 186 | 0.6559 | 0.5385 | 0.7778 | 0.6364 |
| ASSERTgreater | 36 | 70 | 34 | 46 | 186 | 0.5699 | 0.4390 | 0.5143 | 0.4737 |
| VARIABLEDECLARATORgreaterEquals | 104 | 26 | 7 | 47 | 184 | 0.7065 | 0.6887 | 0.9369 | 0.7939 |
| VARIABLEDECLARATORgreater | 7 | 124 | 27 | 26 | 184 | 0.7120 | 0.2121 | 0.2059 | 0.2090 |
| DOgreaterEquals | 86 | 13 | 10 | 44 | 153 | 0.6471 | 0.6615 | 0.8958 | 0.7611 |
| DOgreater | 8 | 111 | 19 | 15 | 153 | 0.7778 | 0.3478 | 0.2963 | 0.3200 |
| DOlessEquals | 78 | 2 | 4 | 62 | 146 | 0.5479 | 0.5571 | 0.9512 | 0.7027 |
| DOless | 4 | 110 | 30 | 2 | 146 | 0.7808 | 0.6667 | 0.1176 | 0.2000 |
| ASSIGNgreaterEquals | 51 | 18 | 12 | 59 | 140 | 0.4929 | 0.4636 | 0.8095 | 0.5896 |
| ASSIGNgreater | 11 | 87 | 23 | 19 | 140 | 0.7000 | 0.3667 | 0.3235 | 0.3438 |
| ASSERTlessEquals | 56 | 23 | 23 | 16 | 118 | 0.6695 | 0.7778 | 0.7089 | 0.7417 |
| ASSERTless | 13 | 57 | 15 | 33 | 118 | 0.5932 | 0.2826 | 0.4643 | 0.3514 |
| VARIABLEDECLARATORlessEquals | 25 | 20 | 11 | 32 | 88 | 0.5114 | 0.4386 | 0.6944 | 0.5376 |
| VARIABLEDECLARATORless | 11 | 42 | 15 | 20 | 88 | 0.6023 | 0.3548 | 0.4231 | 0.3860 |
| ASSIGNlessEquals | 9 | 4 | 2 | 13 | 28 | 0.4643 | 0.4091 | 0.8182 | 0.5455 |
| ASSIGNless | 0 | 17 | 5 | 6 | 28 | 0.6071 | 0.0000 | 0.0000 | 0.0000 |
| EXPRESSIONgreaterEquals | 10 | 2 | 1 | 13 | 26 | 0.4615 | 0.4348 | 0.9091 | 0.5882 |
| EXPRESSIONgreater | 0 | 22 | 1 | 3 | 26 | 0.8462 | 0.0000 | 0.0000 | 0.0000 |
| EXPRESSIONlessEquals | 5 | 1 | 2 | 2 | 10 | 0.6000 | 0.7143 | 0.7143 | 0.7143 |
| EXPRESSIONless | 0 | 4 | 3 | 3 | 10 | 0.4000 | 0.0000 | 0.0000 | 0.0000 |
| OBJECTCREATIONgreaterEquals | 3 | 1 | 0 | 4 | 8 | 0.5000 | 0.4286 | **1.0000** | 0.6000 |
| OBJECTCREATIONgreater | 0 | 5 | 2 | 1 | 8 | 0.6250 | 0.0000 | 0.0000 | 0.0000 |
| OBJECTCREATIONlessEquals | 1 | 0 | 0 | 1 | 2 | 0.5000 | 0.5000 | **1.0000** | 0.6667 |
| OBJECTCREATIONless | 0 | 1 | 1 | 0 | 2 | 0.5000 | 0.0000 | 0.0000 | 0.0000 |
| | | | | | | | | | |
| Total | 38317 | 42838 | 9641 | 14162 | 104958 | 0.7732 | 0.7301 | 0.7990 | 0.7630 |

**Quantitative evaluation results for all context types analyzed**

| **Statement Type for Bug** | **TP** | **TN** | **FP** | **FN** | **Total** | **Acc** | **Recall** | **Precision** | **F1** |
|---|---|---|---|---|---|---|---|---|---|
| IF | 16471 | **19108** | **5601** | **8238** | **49418** | 0.72 | 0.6666 | 0.7462 | 0.7042 |
| FOR | **16709** | 17326 | 2183 | 2800 | 39018 | **0.8723** | **0.8565** | **0.8844** | **0.8702** |
| WHILE | 1913 | 2245 | 614 | 946 | 5718 | 0.7272 | 0.6691 | 0.757 | 0.7104 |
| RETURN | 1233 | 1427 | 352 | 546 | 3558 | 0.7476 | 0.6931 | 0.7779 | 0.7331 |
| TERNARY | 851 | 1196 | 361 | 706 | 3114 | 0.6574 | 0.5466 | 0.7021 | 0.6147 |
| METHOD | 578 | 710 | 267 | 399 | 1954 | 0.6592 | 0.5916 | 0.684 | 0.6345 |
| ASSERT | 157 | 216 | 88 | 147 | 608 | 0.6135 | 0.5164 | 0.6408 | 0.5719 |
| DO | 176 | 236 | 63 | 123 | 598 | 0.689 | 0.5886 | 0.7364 | 0.6543 |
| VARIABLEDECLARATOR | 146 | 212 | 60 | 126 | 544 | 0.6581 | 0.5368 | 0.7087 | 0.6109 |
| ASSIGN | 75 | 126 | 42 | 93 | 336 | 0.5982 | 0.4464 | 0.641 | 0.5263 |
| EXPRESSION | 15 | 29 | 7 | 21 | 72 | 0.6111 | 0.4167 | 0.6818 | 0.5172 |
| OBJECTCREATION | 4 | 7 | 3 | 6 | 20 | 0.55 | 0.4 | 0.5714 | 0.4706 |
| | | | | | | | | | |
| Total | 38328 | 42838 | 9641 | 14151 | 104958 | 0.7733 | 0.7303 | 0.799 | 0.7631 |

**Table 8: Quantitative analysis by statement type**

| **Comparator** | **TP** | **TN** | **FP** | **FN** | **Total** | **Accuracy** | **Recall** | **Precision** | **F1** |
|---|---|---|---|---|---|---|---|---|---|
| LessEquals | **22172** | 1436 | 1622 | 4511 | **29741** | 0.7938 | **0.8309** | **0.9318** | **0.8785** |
| Less | 1081 | **23797** | **2886** | 1977 | **29741** | **0.8365** | 0.3535 | 0.2725 | 0.3078 |
| Greater | 4650 | 12472 | 2358 | 3258 | 22738 | 0.753 | 0.588 | 0.6635 | 0.6235 |
| GreaterEquals | 10414 | 5133 | 2775 | **4416** | 22738 | 0.6522 | 0.7022 | 0.7896 | 0.7434 |
| | | | | | | | | | |
| Total | 38317 | 42838 | 9641 | 14162 | 104958 | 0.7732 | 0.7301 | 0.799 | 0.763 |

**Table 9: Quantitative analysis by comparator type**

| **Context Type** | **TP** | **TN** | **FP** | **FN** | **Total** | **Accuracy** | **Recall** | **Precision** | **F1** |
|---|---|---|---|---|---|---|---|---|---|
| IFgreaterEquals | **7978** | 3647 | **2460** | 2845 | **16930** | **0.6867** | 0.7371 | 0.7643 | 0.7505 |
| IFgreater | 1744 | **8428** | 2395 | **4363** | **16930** | 0.6008 | 0.2856 | 0.4214 | 0.3404 |
| IFlessEquals | 4447 | 736 | 908 | 1688 | 7779 | 0.6663 | 0.7249 | **0.8304** | **0.7741** |
| IFless | 539 | 4825 | 1310 | 1105 | 7779 | 0.6895 | 0.3279 | 0.2915 | 0.3086 |
| | | | | | | | | | |
| Total | 14708 | 17636 | 7073 | 10001 | 49418 | 0.6545 | 0.5952 | 0.6753 | 0.6327 |

**Table 10: Quantitative evaluation results for a model trained only on bugs concerned with $if$ statements**