# Using Distributed Representation of Code for Bug Detection

**Analytcs and Machine Learning for Softare 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
@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 has been shown to be very good at method name prediction and claimed to be efficient at other tasks. 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 tasks such as mixed binary or logical operators. We define bug detection as a binary classification problem and train our model on a 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 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 project [5]. 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 helping with the 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 which make the developers ignore the results of the tools [3]. Recently, a lot of Artificial Intelligence solutions have emerged which help with the issue [1, 2, 6, 7]. 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 work in particular, code2vec [2] model (see section 2 for description) delivers state-of-the-art performance on method naming. However, the authors do not test the model on other tasks while theorizing that it should also yield good performance.

In this work, we use code2vec model with pre-trained embeddings for code and change the deep learning model layer used for method naming with a binary classification layer for other tasks such as mixed binary or logical operators. Our intuition is that the change in the AST attention upon introducing a bug as in figure **??** is learnable by a model. Hence the system should be able to identify bugs.

Hence, the main contributions of this paper are.

- Replicating work done by authors of Alon et al. [2]
- Quantitative and qualitative evaluation on code2vec's performance on tasks other than method naming.

The paper is divided into following sections. In section 2 relevant literature used to create this paper is discussed, in section 3 data origins and preprocessing is explained, in section 4 the architecture of the model and hyper-parameters are discussed. Finally, in section 5, the model is evaluated which is followed by a conclusion in section 6.

## 2 RELEVANT LITERATURE

Code2Vec by Alon et al. [2] is a state-of-the-art deep neural network model 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 using Abstract Syntax Tree (AST) paths of the code. While the idea of using AST paths to generate code embedding is also used by other authors such as Hu et al. [4], Code2Vec is superior due to the novel way of using AST paths. Instead of linearizing paths as [4], 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.

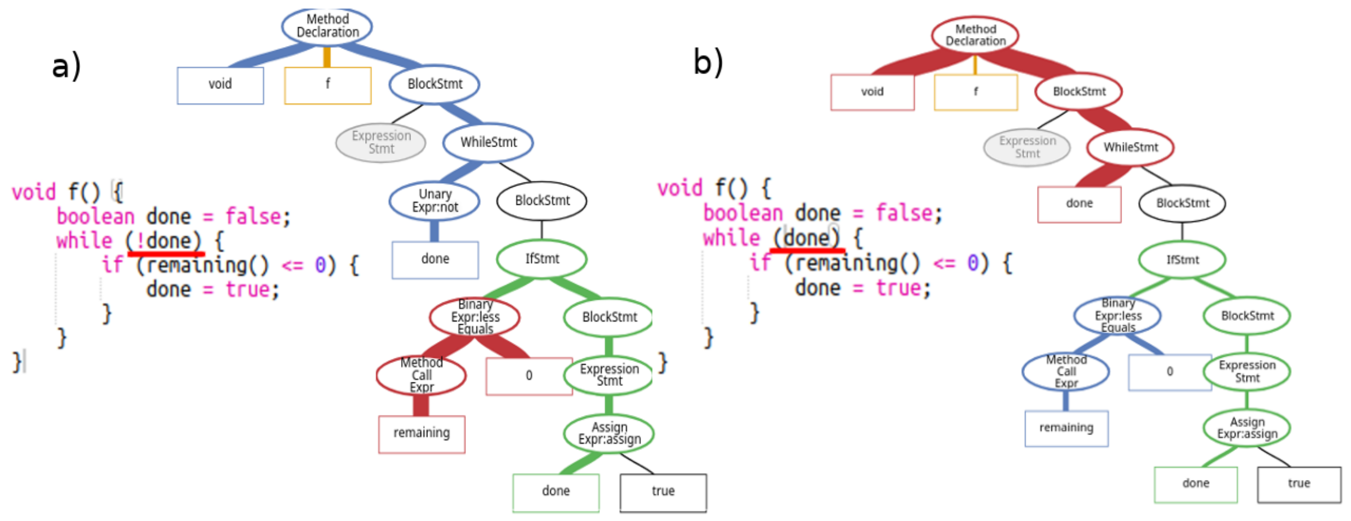
The authors make use of embedding similarity between similar code to predict method names. The model is trained on a dataset of 12 million Java methods and compared to other competitive approaches. It significantly outperforms them by having approximately 100 times faster prediction rate at the same time having a better F1 score of 59.5. 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. [6] 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 code, but instead of using AST-s like code2vec, embedding is created from specific parts of 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.

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

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

[2]https://code2vec.org/



```
void f() {
    boolean done = false;
    while (!done) {
        if (remaining() <= 0) {
            done = true;
        }
    }
}
```

```
void f() {
    boolean done = false;
    while (done) {
        if (remaining() <= 0) {
            done = true;
        }
    }
}
```

**Figure 1: Example change of AST after introducing bug using code2vec web interface**[2]

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

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt.

### 3.1 Type 1

### 3.2 Type 2

### 3.3 Dataset

## 4 MODEL

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt.

## 5 EVALUATION

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt.

## 6 CONCLUSION

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt.

## 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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.