Learning Off-By-One Mistakes: An Empirical Study on Different Deep Learning Models

Master's Thesis

Hendrig Sellik

Learning Off-By-One Mistakes: An Empirical Study on Different Deep Learning Models

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Hendrig Sellik

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Adyen
Simon Carmiggeltstraat 6-50
1011DJ
Amsterdam, the Netherlands
https://www.adyen.com/

Learning Off-By-One Mistakes: An Empirical Study on Different Deep Learning Models

Author:	Hendrig Sellik
Student id:	4894502
Email:	`h.sellik@student.tudelft.nl`

Abstract

Mistakes in binary conditions are a source of error in many software systems. They happen when developers use ‘<’ or ‘>’ instead of ‘<=’ or ‘>=’. These boundary mistakes are hard to find for developers and pose a manual labor-intensive work. While researches have been proposing solutions to identify errors in boundary conditions, the problem remains a challenge.

In this thesis, we propose deep learning models to learn mistakes in boundary conditions and train our model on approximately 1.6M examples with faults in different boundary conditions. We achieve an accuracy of 85.06%, a precision of 85.23% and a recall of 84.82% on a controlled dataset. Additionally, we perform tests on 41 real-world boundary condition bugs found from GitHub and try to find bugs from the Java project of Adyen. However, the false-positive rate of the model remains an issue. We hope that this work paves the way for future developments in using deep learning models for defect prediction.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. Aniche, Faculty EEMCS, TU Delft
Company supervisor:	O. van Paridon, Adyen B.V.
Committee Member:	Dr. C. Hauff, Faculty EEMCS, TU Delft

Preface

This thesis has been submitted for the degree of Master of Science in Computer Science at the Delft University of Technology. The journey from the beginning of my studies to the end of my thesis has been full of passion and devotion to the area I love the most. I have grown both as a person and an academic. However, this all would have not been possible without the people I would like to thank in this preface.

Maurício Aniche, thank you for the time you have dedicated to me. Your support during the thesis was unparalleled. You let me discover the domain in the beginning and helped me with your guidance and enthusiasm where it was needed the most. Special thanks to Georgious Gousious and Uri Alon for taking their time and helping me with the questions I had for them. I would also like to thank the members of my thesis committee, Arie van Deursen and Claudia Hauff, for their time and effort.

Colleagues at Adyen, thank you for your warm welcome at the company and for providing me with a pleasant milieu even in the unprecedented times due to the outbreak of the COVID-19 virus. I could always count on your constructive feedback and insightful remarks in your area of expertise. Onno van Paridon, thank you for the time you dedicated to me as a supervisor.

Finally, I would like to thank my friends and family for their support. You listened to my monologues about the thesis and even though the technical details did not make much sense to you, I could always rely on you to hear me out.

Hendrig Sellik
Delft, the Netherlands
September 7, 2020

Contents

List of Figures

Chapter 1

Introduction

Software projects have been getting larger with regards to the size of the codebase, now reaching millions on lines of code. At rapid and continuous development cycles, the defects which prevent programs from compiling are caught, but there is an array of bugs that might only appear at runtime. One of them is an off-by-one mistake, which happens when developers use ‘>’ or ‘<’ in cases where they should have used ‘=>’ or ‘<=’ or vice versa.

Early studies such as [28] [32] and [45], have been conducted to specifically research methods to automate the testing of boundary conditions. However, the work either fails to solve the issue automatically or requires significant manual effort on behalf of the user.

In response, the industry has adopted static analysis tools, such as SpotBugs and PVS Studio, which address the problem in some specific scenarios. However, they fail to capture most of the off-by-one errors that are present and require manual work to define and fine-tune the rules. At the same time, the research community has seen success in many software code tasks that were not possible before, such as method naming and method description generation, thanks to the rapid developments in the area of machine learning.

The lack of reliable tools to capture off-by-one mistakes leaves an excellent opportunity to try applying machine learning to this domain which would benefit both the scientific community and the industry.

This study attempts to contribute to the knowledge base of using state-of-the-art machine learning techniques to detect bugs in software code. We trained several binary classification models on likely correct methods and their automatically mutated likely incorrect counterparts. Later, we tested the models in a cross-project manner on both open-source software and a company project. The main research flow is depicted in Figure 3.1 and the main contributions of this thesis are:

1. An empirical study on the performance of different deep learning models to detect off-by-one mistakes.

2. A quantitative and qualitative evaluation of deep off-by-one detection models in real-world open source and proprietary bugs.

The models are trained on over 1.6M examples and the best results are obtained with the Code2Seq [8] model achieving 85.23% precision and a recall of 84.82% on a controlled balanced testing set. When introducing imbalance to the testing set, the metrics fall to 64.65% and 41% respectively. Testing the model on known bugs from open-source projects yields a 55.88% precision and 46.34% recall on a balanced set consisting of 82 methods. When evaluating the model on Adyen Java repository, the approach did not reveal any bugs per se, but pointed to code considered to deviate from good practices.

Chapter 2

Background

Bug-detection is a well-explored domain in computer science. In this chapter, we discuss technologies related to the thesis and give background information to the problem at hand. It is divided into 5 main sections. Firstly, bugs in software are discussed in Section 2.1, next, in Section 2.2, the structure of Abstract Syntax Trees is explained, in Section 2.3 the current static analysis tools used in the industry are discussed, in Sections 2.4.1 and 2.4.2 we explain the Code2Vec and Code2Seq models which this work heavily relies on, in Section 2.4.3 the Graph Relational embedding Attention Transformer Model is described and in Section 2.5, the ML models related to bug detection are discussed.

2.1 Bugs in Software

Professional software developers are capable of producing complex systems and the industry has adapted many ways, such as unit testing, pair programming, static and dynamic analysis, code reviews, etc, to prevent bugs from finding their way into production software. Still, defects appear even in very critical software.

This is caused by many factors, such as changing the software environment or the hardware that the program is being run on, users using the software beyond the specified non-functional parameters, the dependent software having a bug, hasty development to meet a deadline or the software just running for long enough time to have issues which were not present before, such as the famous millennium bug and the very similar defects that appear every year when the time is changed to adjust for the daylight saving.

As a software system grows, the number of execution branches grows to a level in which it is almost impossible to cover all the edge cases manually. At the same time, some kinds of errors are very hard to detect automatically with current methods. In this thesis, we focus on one such error, the off-by-one error.

Off-by-one errors are a type of logic errors which allow the program to compile but result in errors during runtime. They are caused by the binary expressions ”<”, ”<=”, ”>” and ”=>” being *off by one*. This means that ”<” is swapped with ”<=” and ”>” by ”=>” or vice versa. These errors can be silent, meaning that they might go unnoticed or non-silent, meaning that they produce a program crash.

```
// 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 2.1: An example of a mistake in a boundary condition. Code extracted from JPacman[1].

Silent errors are represented by cases where the program does not crash but outputs a result that is not originally intended. For example, when one wants to check if an element is inside the scope of x and y coordinates but checks one of the variables inclusively and the other one not. This is illustrated in a real-world example (Figure 2.1) where the variable *x* should be checked to be zero inclusive.

These types of errors also occur in more mature and widely used projects. For example, see a bug found in Apache JMeter in Figure 3.4. An untraditional loop was made where a list was traversed in reverse order. However, the author of the code forgot to include the first element of the list while iterating, resulting in skipping the element. This was later found by the developers and fixed.

The most common scenario for a non-silent error is when an element is accessed in an array which is out of bounds, producing an *IndexOutOfBoundsException* in Java.

2.2 Abstract Syntax Trees

Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of the source code. It preserves the original structure of the parse tree but eliminates nonessential nodes such as parenthesis and semicolons, which

[1] `https://github.com/serg-delft/jpacman`
[2] `https://github.com/apache/jmeter/`

```
// Incorrect code: i > 0
public void removeHeaderNamed(String name)
{
    Vector removeIndices = new Vector();
    for (int i = getHeaders().size() − 1; i > 0; i−−)
    {
        Header header = (Header) getHeaders().get(i).getObjectValue();
        if (header == null)
        {
            continue;
        }
        if (header.getName().equalsIgnoreCase(name))
        {
            removeIndices.addElement(new Integer(i));
        }
    }

    for (Enumeration e = removeIndices.elements(); e.hasMoreElements();)
    {
        getHeaders().remove(((Integer) e.nextElement()).intValue());
    }
}

// Correct code: i >= 0
public void removeHeaderNamed(String name)
{
    Vector removeIndices = new Vector();
    for (int i = getHeaders().size() − 1; i >= 0; i−−)
    {
        Header header = (Header) getHeaders().get(i).getObjectValue();
        if (header == null)
        {
            continue;
        }
        if (header.getName().equalsIgnoreCase(name))
        {
            removeIndices.addElement(new Integer(i));
        }
    }

    for (Enumeration e = removeIndices.elements(); e.hasMoreElements();)
    {
        getHeaders().remove(((Integer) e.nextElement()).intValue());
    }
}
```

Figure 2.2: An example of a mistake in a boundary condition. Code extracted from Apache Jmeter[2].

can be derived from the structure of the tree without being explicitly stated [20].

For the implementations in this thesis, we use the JavaParser library which parses the Java code and produces an AST. An example of the AST structure is given in Figure 2.3. The gray nodes on the image represent the nodes that are defined by the JavaParser library and represent code blocks, method declarations, binary expressions, method call expressions, etc.

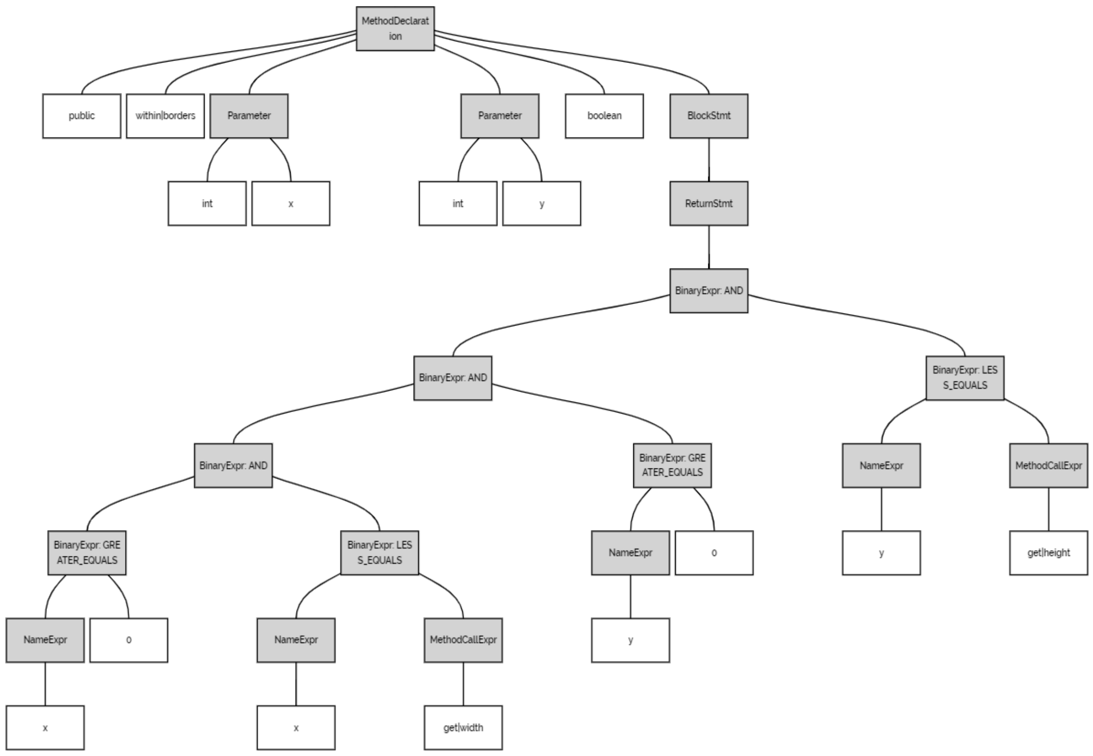

Figure 2.3: Example AST of the code in Figure 2.1.

The white nodes mostly represent user-defined values or types declared on the language level. If observed from left to right, these nodes will form a string similar to the original code. In this thesis, we define a path along the AST as the closest route between two leaf nodes. It has a vital role in creating method embeddings which are used during classification for for downstream tasks.

2.3 Static Analysis Tools

As the codebase sizes of industry projects reach millions of lines of code, it is very difficult to manually examine or cover all the execution branches with unit tests. Hence, static analysis tools are used which analyze source code to automatically find bugs and ease the labor-intensive code inspection efforts. The static nature of the tools means that they do not execute the source code but rather infer the possible erroneous states, usually from Abstract Syntax Trees or Control Flow Graphs.

SpotBugs[3] (formerly known as FindBugs [29]) developed by D. Hovemeyer et al. is a static code analysis tool to find bugs in code. During its development, the authors found that such tools are not widespread and pointed out several trivial, yet critical bugs from mature systems such as Apache Tomcat. Today, however, the SpotBugs and other static analysis tools are widely adopted in the industry projects and their continuous integration pipelines [12].

SpotBugs uses the Apache Bytecode Engineering Library[4] (BCEL), a bytecode analysis library to create a detector for each type of bug. The detectors use different strategies such as looking at the structure of classes and methods or creating and analyzing a data flow graph, also allowing interprocedural bug detection.

Error Prone [2], developed at Google, works by extending the OpenJDK Java compiler and performing an error checking after the flow phase of the compiler. A number of predicates are manually created to match AST nodes of the source code for errors and additional checks on the dataflow graph of the code are made. If a violation is found, Error-Prone will fail the compilation with an error message and can propose fixes to the errors or even automatically fix them. Giving errors during compilation time will ensure that the defective code does not get ignored and is caught during development. It also leverages parsing and type checking already done by the compiler and therefore avoids redundant work.

Nullaway [11], developed at Uber, is a popular plugin to Error Prone specifically designed to robustly find *NullPointerExceptions* (NPE-s) with minimal performance overhead. It relies on the developers annotating methods with *@Nullable* and making assumptions about unannotated code, after which a control flow graph is made to analyze the dataflow and find NPE-s.

Infer [16] is a tool developed at Facebook and aims to verify programs by building their compositional proofs. Essentially, it uses an assembly-like intermediate language which is derived from Java bytecode and builds a mathematical model of the source code that represents all possible execution paths. The call graphs of the model can be updated for functions/procedures individually while keeping the ability to find interprocedural defects, such as nullpointer dereferencing errors seen in Figure 2.4. This allows to cache the call graph and incrementally update the parts that were affected by a code change [17]. In addition, groups of procedures can be analyzed independently, which allows the

[3]SpotBugs GitHub page - `https://github.com/spotbugs/spotbugs/`

[4]Apache BCEL - `http://commons.apache.org/proper/commons-bcel/`

process to be run in parallel.

PMD[5] is a static analysis tool that works on predefined rulesets. For different rules, different data is processed which might include the AST, data flow graph and type resolution information. PMD also supports incremental analysis by storing the processed data in a cache and loading it if the file under analysis has not changed. However, PMD only supports one source file at the time and hence cannot capture more complicated bugs spanning multiple files.

```
public class CodeSample {
  String computeSomething(boolean flag) {
    if (flag) {
      return null;
    }
    else {
      return 'something';
    }
  }

  public int doSTuff() {
    String s = computeSomething(true);
    return s.length();
  }
}
```

Figure 2.4: Example of an interprocedural null value dereferencing when flag is *true*.

The above-mentioned tools and research all use predefined rules which require exhaustive manual labor to produce but fail to capture some types of bugs. A machine learning model might automatically learn more complicated rules/-patterns that are more accurate.

2.4 Machine Learning for Software Engineering

Machine Learning for Software Engineering has seen rapid development in recent years inspired by the successful application in the Natural Language Processing field [27]. It is applied in many tasks related to software code such as code translation [18], [40], [41], code completion [27],[44], [49], [33], identifier and method naming [4], [5], [9], type inference [25], API Usage [22], [36] and code refactoring [10]. Many of those approaches allow to change the downstream task while making minor modifications to the underlying model.

[5]PMD official web page `https://pmd.github.io/latest/pmd_devdocs_how_pmd_works.html`

The next subsections will be explaining the techniques and models we used in our thesis and give an overview of the similar work done in the field.

2.4.1 Code2Vec Model

The Code2Vec model created by Alon et al. [9] is a Neural Network model used to create embeddings from Java methods. These embeddings were used in the original work to predict method names.

The architecture of this model requires Java methods to be split into *path contexts* based on the AST of the method. A path context is a random path between two nodes in the AST and consists of 2 terminal nodes x_s, x_t and the path between those terminal nodes p_j which does not include the terminals. The embeddings for those terminal nodes and paths are learned during training and stored in 2 separate vocabularies. During training, these paths are concatenated to a single vector to create a *context vector* c_i which has the length l of $2 \cdot x_s + x_p$ where the length of x_s is equal to x_t. More formally:

$$c_i = embedding\left(\left\langle x_s, p_j, x_t \right\rangle\right) = [value\_vocab_s; path\_vocab_j; value\_vocab_t]$$

The acquired context vectors c_i for paths are passed through the same fully connected (dense) neural network layer (using the same weights). The network uses hyperbolic tangent activation function and dropout in order to generate a *combined context vectors* $\tilde{c}_i$. The size of the dense layer allows controlling the size of the resulting context vector. The calculation for the context vector is the following:

$$\tilde{c}_i = tanh(W \cdot c_i)$$

Where $W \in \mathbb{R}^{h \times l}$ are dense layer weights where the height h determines the context vector size and l depends on the embeddings as shown previously.

The attention mechanism of the model works by using a global attention vector $a \in \mathbb{R}^h$ which is initialized randomly and learned with the rest of the network. It is used to calculate attention weight a_i for each individual combined context vector $\tilde{c}_i$:

$$a_i = \frac{exp(\tilde{c}_i{}^T \cdot a)}{\sum_{j=1}^{n} exp(\tilde{c}_j{}^T \cdot a)}$$

It is possible, that some methods are not with a large enough AST to generate the required number of context paths. For this dummy (masked) context paths are inputted to the model which get a value of 0 for attention weight a_i. This enables the model to use examples with the same shape.

After calculating weight for context vectors, their linear combination provides a code vector v:

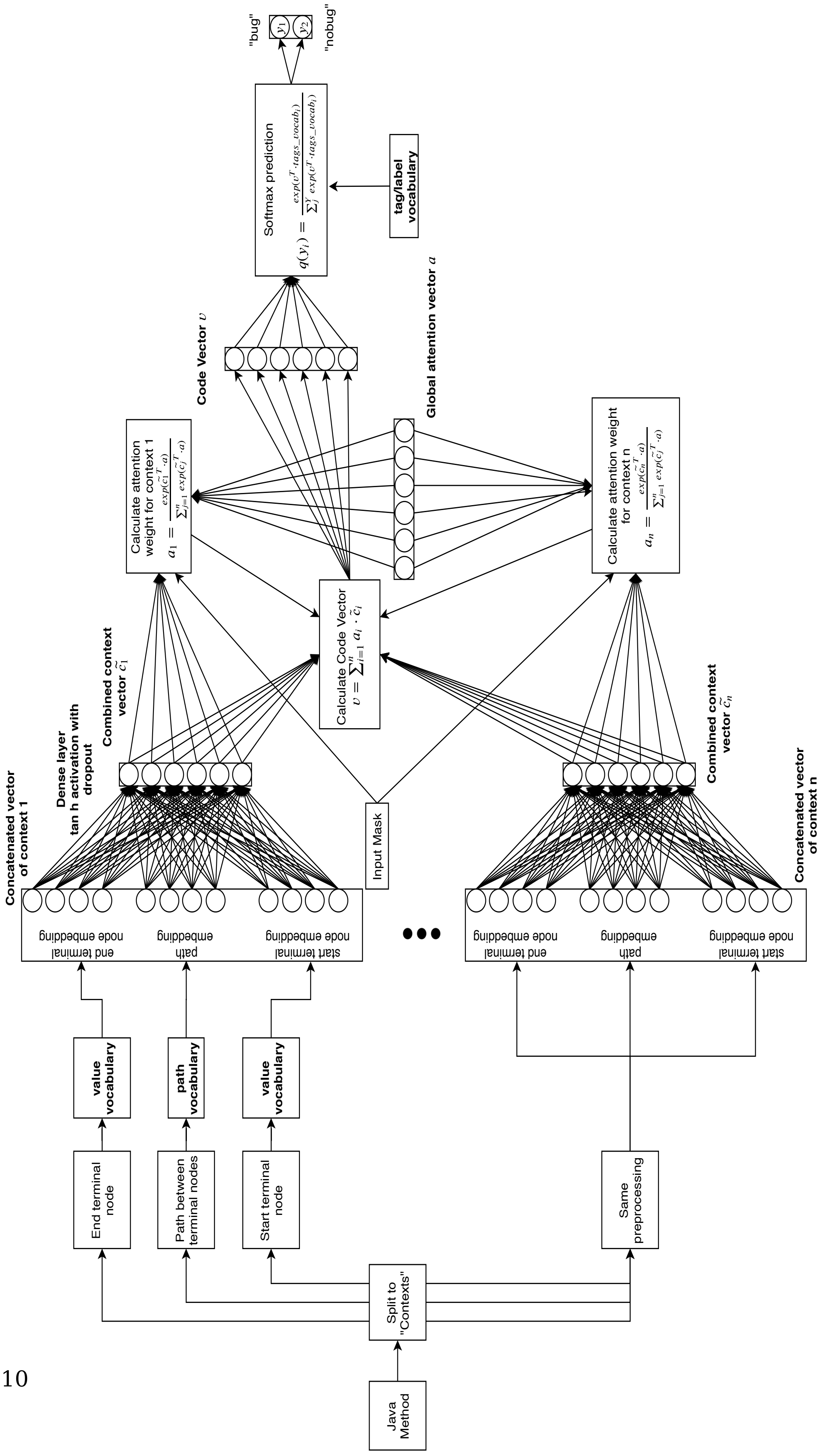
Java Method
Split to "Contexts"
End terminal node
Path between terminal nodes
Start terminal node
Same preprocessing
value vocabulary
path vocabulary
value vocabulary
end terminal node embedding
path embedding
start terminal node embedding
Concatenated vector of context 1
Dense layer tan h activation with dropout
Combined context vector $\tilde{c}_1$
Concatenated vector of context n
Combined context vector $\tilde{c}_n$
Input Mask
Calculate attention weight for context 1
$a_1 = \frac{exp(\tilde{c}_1^T \cdot a)}{\sum_{j=1}^n exp(\tilde{c}_j^T \cdot a)}$
Calculate attention weight for context n
$a_n = \frac{exp(\tilde{c}_n^T \cdot a)}{\sum_{j=1}^n exp(\tilde{c}_j^T \cdot a)}$
Calculate Code Vector
$v = \sum_{i=1}^n a_i \cdot \tilde{c}_i$
Global attention vector a
Code Vector v
Softmax prediction
$q(y_i) = \frac{exp(v^T \cdot tags\_vocab_i)}{\sum_j^Y exp(v^T \cdot tags\_vocab_j)}$
tag/label vocabulary
y_1
y_2
"bug"
"nobug"

Figure 2.5: Code2Vec model architecture.

$$v = \sum_{i=1}^{n} a_i \cdot \tilde{c}_i$$

During training, a tag vocabulary $tags\_vocab \in \mathbb{R}^{|Y| \times l}$ is created where for each tag (label) $y_i \in Y$ corresponds to an embedding of size l. The tags are learned during training and in the task proposed by the authors, these represent method names.

A prediction for a new example is made by computing the normalized dot product between code vector v and each of the tag embeddings $tags\_vocab_i$:

$$\text{for } y_i \in Y : q(y_i) = \frac{exp(v^T \cdot tags\_vocab_i)}{\sum_j^Y exp(v^T \cdot tags\_vocab_i)}$$

This results in a probability for each tag y_i. The higher the probability, the more likely the tag belongs to the method.

2.4.2 Code2Seq Model

Code2Seq model created by Alon et al. [8] is a Sequence-to-Sequence Deep Learning model used to create embeddings from Java methods from which method descriptions are learned. The original work was used to generate sequences of natural language words to describe methods.

Similarly to the Code2Vec model, the model works by generating random paths from the AST with a specified maximum length. Each path consists of 2 terminal tokens x_s, x_t and the path between those terminal nodes p_j which, in Code2Seq, includes the terminal nodes p_s, $p_t \in p_j$, but not tokens.

It is important to make a difference between terminal tokens and path nodes. The former are user-defined values, such as a number *4* or variable called *string-Builder* while the latter come from a limited set of AST constructs such as Name-Expr, BlockStmt, ReturnStmt. There are around 400 different node types that are predefined in the JavaParser implementation [6].

During training, the path nodes and the terminal tokens are encoded differently. Terminal tokens get partitioned into subtokens based on the camelCase notation, which is a standard coding convention in Java. For example, a terminal token *stringBuilder* will be partitioned into *string* and *Builder*. The subtokens are turned into embeddings with a learned matrix $E^{subtokens}$ and encoding is created for the entire token by adding the values for subtokens:

$$\text{Encode terminal node } (x_s) = \sum_{s \in split(x_s)}^{len(x_s)} E_s^{subtokens}$$

[6]JavaParser Node types `https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.15.9/com/github/javaparser/ast/Node.html`

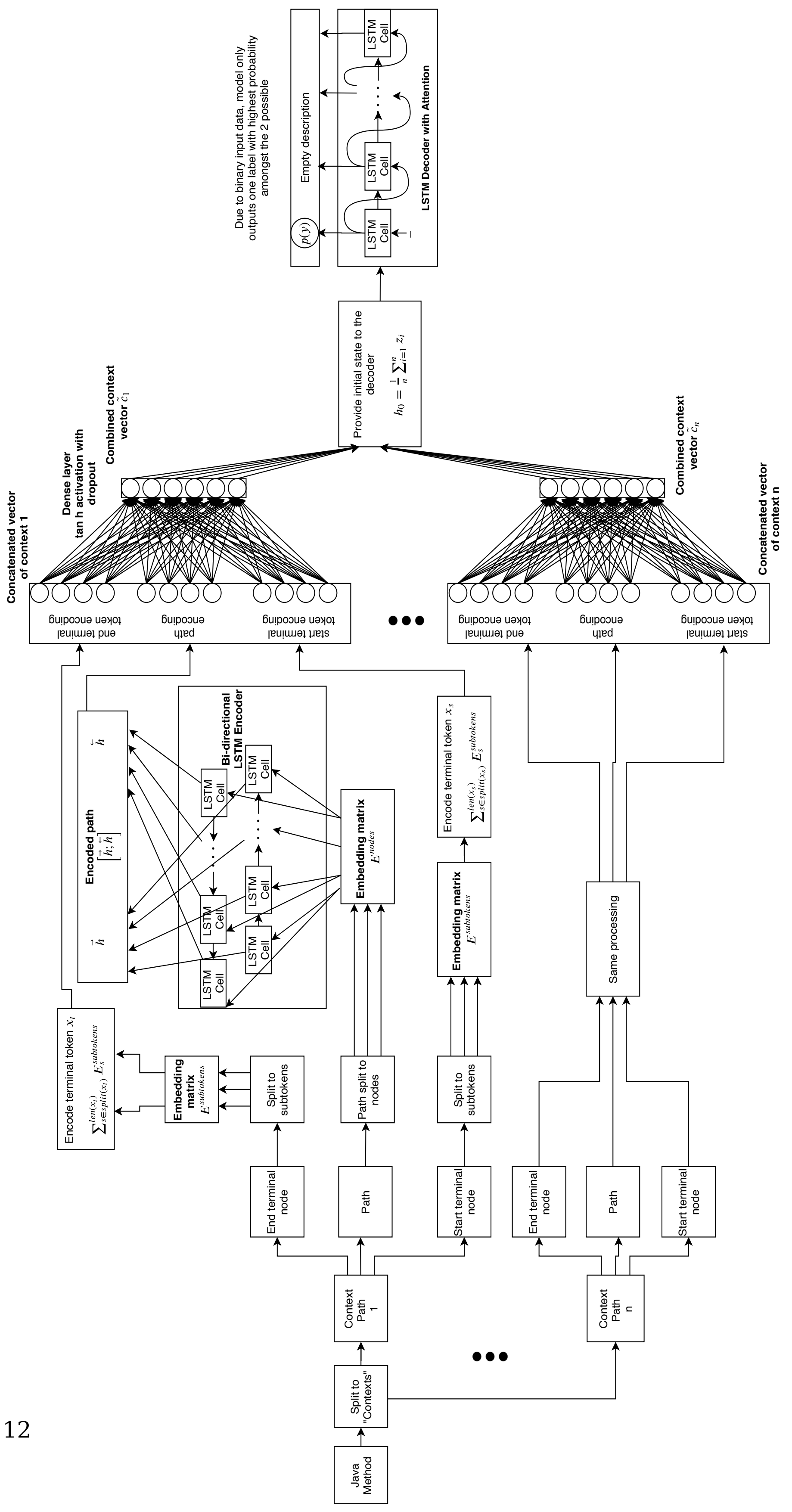
Java Method
Split to "Contexts"
Context Path 1
Context Path n
End terminal node
Path
Start terminal node
Split to subtokens
Path split to nodes
Embedding matrix $E^{subtokens}$
Embedding matrix E^{nodes}
Encode terminal token x_t $\sum_{s \in split(x_t)}^{len(x_t)} E_s^{subtokens}$
Encode terminal token x_s $\sum_{s \in split(x_s)}^{len(x_s)} E_s^{subtokens}$
Bi-directional LSTM Encoder
LSTM Cell
Encoded path $[\vec{h}; \overleftarrow{h}]$
Same processing
end terminal token encoding
path encoding
start terminal token encoding
Concatenated vector of context 1
Concatenated vector of context n
Dense layer tan h activation with dropout
Combined context vector $\tilde{c}_1$
Combined context vector $\tilde{c}_n$
Provide initial state to the decoder $h_0 = \frac{1}{n}\sum_{i=1}^{n} z_i$
LSTM Decoder with Attention
Empty description
$p(y)$
Due to binary input data, model only outputs one label with highest probability amongst the 2 possible

Figure 2.6: Code2Seq model architecture.

Paths of the AST are also split into nodes and each of the nodes corresponds to a value in a learned embedding matrix E^{nodes}. These embeddings are fed into a bi-directional LSTM which final states result in a forward pass output $\overrightarrow{h}$ and backward pass output $\overleftarrow{h}$. These are concatenated to produce a path encoding.

$$\text{Encode path}(p_s, ..., p_t \in p_j) = LSTM\left(E^{nodes}_{p_s}, ..., E^{nodes}_{p_t}\right) = \left[\overrightarrow{h}; \overleftarrow{h}\right]$$

As with the Code2Vec model, the encodings of the terminal nodes and the path are concatenated and the resulting encoding is an input to a dense layer with *tanh* activation to create a *combined context vector* $\tilde{c}_i$.

$$c_i = encoding\left(\langle x_s; p_s, ..., p_t; x_t \rangle\right) = [Encode(x_s); Encode(p_s, ..., p_t); Encode(x_t)]$$

$$\tilde{c}_i = tanh(W \cdot c_i)$$

Where $W \in \mathbb{R}^{d \times [2d_{path} + 2d_{token}]}$ is a matrix for dense layer weights where the height d determines the combined context vector size and the width depends on the starting/terminating token context size $2d_{token}$ and concatenated forward and backward pass result of the LSTM (Long Short-Term Memory) encoding of the path $2d_{path} = \left[\overrightarrow{h}; \overleftarrow{h}\right]$. The matrix W was defined wrong in the original work, however, the authors were notified of this.[7]

Finally, to provide an initial state to the decoder, the representations of all n paths in a given method are averaged.

$$h_0 = \frac{1}{n} \sum_{i=1}^{n} z_i$$

The decoder uses the initial state h_0 to generate an output sequence while attending over all the combined context vectors $\tilde{c}_1$, ..., $\tilde{c}_n$. The resulting output sequence represents a natural language description of the method.

The advantage of the Code2Seq model is in the way the context vectors $\tilde{c}_i$ are created. In particular, due to splitting terminal nodes. the vocabulary of the terminal nodes yields greater flexibility towards different combinations of subtoken combinations. In addition, while Code2vec embeds entire AST paths between terminals, the Code2Seq model only embeds subtokens. This results in fewer out-of-vocabulary examples and a far smaller model size. The model also has an order-of-magnitude fewer parameters compared to the Code2Vec model.

[7] `https://github.com/tech-srl/code2seq/issues/40`

2.4.3 Graph Relational Embedding Attention Transformers

Instead of traversing the AST of a Java program and using different paths as an input to the model, some researchers have started to use Graph Neural Networks (GNN-s) instead to leverage the structural nature of the AST as proposed by Allamanis et al. [6]. In addition, the AST is enhanced with control flow information. For example, the usages of the variables are connected to emphasize that it is the same object as opposed to only having the same name.

This work has been iterated by Hellendoorn et al. [26], who propose Graph Relational Embedding Attention Transformers (GREAT). The idea of enhancing the AST with additional information remains the same, but the authors use the initial graph to bias the attention of a transformer model instead of performing several message passing steps with a GNN.

2.5 Machine Learning for Defect Prediction

Nowadays, a lot of projects use static analysis tools such as the ones described in Section 2.3. These tools, however, rely on bug pattern detectors manually crafted and fine-tuned by static analysis experts. The huge amount of different bug patterns makes it very difficult to cover more than a fraction of them. Solving this problem via machine learning techniques instead has been a promising and popular research topic.

Nam et al. [39] use different metrics such as coupling between source code objects, number of lines in code, the entropy of code changes, etc. to train a logistic regression classifier that predicts file-level defects in source code. The novelty of their work is in the introduction of Transfer Component Analysis (TCA) where the authors take the distributions of the features from a project and make it match to the project on which the model was trained on. This enables to make better cross-project defect predictions.

Yang et al. [51] analyze 6 large open-source projects consisting of 137,417 changes to detect defect prone commits. The authors use a Deep Belief Network model and commit level features such as the number of modified directories, files, lines of code added and deleted, number of unique changes to modified files, etc. The model achieves 0.36 precision, 0.69 recall and 0.45 F1-Score on average.

Madeyski et al. [37] use popular Travis Continuous Integration environment to extract file-level metrics from builds, such as number of commits for a given file, number of unique committers, modified lines since the last build, local timestamp of the commit that caused the build, last build status, etc. Also, the authors use additional information such as commit hash, file location, etc. for creating models that predict CI build failures.

Pradel et al. [43] use a technique similar to Word2Vec [38] to learn embeddings for JavaScript code tokens extracted from the AST. These embeddings are used to train two-layer feedforward binary classification models to detect bugs. Each trained model focuses on a single bug type and the authors test it on problems such as wrong binary operator, wrong operand in binary operation and swapped function arguments. These models do not use all the tokens from the code, but only those specific to the problem at hand. For example, the model that detects swapped function arguments only uses embeddings of the function name and arguments with a few other AST nodes as features.

Allamanis et al. [6] use Gated Graph Neural Network [35] to detect variable misuse bugs on a token level. As an input to the model, the authors use an AST graph of the source code and augment it with additional edges from the control flow graph.

Pascarella et al. [42] show that defective commits are often composed of both defective and non-defective files. They also train a model to predict defective files in a given commit. Habib et al. [24] create an embedding from methods using a one-hot encoding of tokens such as keywords (for, if, etc.), separators (;, (), etc.), identifiers (method, variable names and literals (values such as "abc" and 10). The embeddings for the first 50 tokens are then used to create a binary classification model. The oracle for training data is a state-of-the-art static analysis tool and the results show that neural bug finding can be highly successful for some patterns, but fail at others.

Li et al. [34] use method AST in combination with a global Program Dependency Graph and Data Flow Graph to determine whether the source code in a given method is buggy or not. The authors use Word2Vec to extract AST node embeddings with a combination of GRU Attention layer and Attention Convolutional Layer to build a representation of the method's body. Node2Vec [21] is used to create a distributed representation of the data flow graph of the file which the inspected method is in. The results are combined into a method vector which is used to make a softmax prediction.

Wang et al. [50] define bug prediction as a binary classification problem and train three different graph neural networks based on control flow graphs of Java code. They use a novel interval-based propagation mechanism to more efficiently generalize a Graph Neural Network (GNN). The resulting method embedding is fed into a feed-forward neural network to find nullpointer dereference, array index out of bounds and class cast exceptions. For each bug type, a separate bug detector is trained.

Overall, there are different ways to use deep learning for bug detection. Earlier work used code metrics (code complexity, file length, method length, etc) or information from version control systems (developer age on the repository, commit time, number of files edited, etc.). The works used traditional machine learning models suitable for structured data such as SVM, Naive Bayes or Random Forest models.

Recent work has moved towards using source code tokens, AST-s or Control Flow Graph (CFG) representations of code as features. The premise is that these features provide more context to identify bugs that otherwise might not be completely captured by only analyzing structural metrics [47]. Some works even combine the representations of AST-s and CFG [6], [34]. These models use different architectures such as random forest model, feed-forward neural network, Convolutional Neural Network (CNN), Long-Short Term Memory network (LSTM), etc.

Chapter 3

Approach

In order to detect Off-By-One errors in Java code, we aim to create a *hypothesis function* that will calculate output based on the inputs generated from an example. More specifically, we train and compare different binary classification machine learning models to classify Java source code methods to one of the two possible output labels which are ”defective” and ”non-defective”. If a method is considered as ”defective”, it is suffering from an off-by-one error, otherwise, it is deemed to be clear from errors.

These models are based on the Code2Vec [9] and Code2Seq [8] models, state-of-the-art deep learning models originally developed for generating method names and descriptions. They use Abstract Syntax Tree paths of a method as features and create an embedding by combining them with the help of an attention mechanism. In addition, a Transformer model by Hellendoorn et al. [26] is used and a baseline model based on source code tokens and Random Forest model. We acquired the datasets necessary for the training of these models from the work of Alon et al. [9] which results in an imbalanced dataset of 920K examples (1 to 10 ratio) and a balanced dataset of 1.6M examples when combined with our automatically mutated methods.

We train on both imbalanced and balanced data to see the difference in performance. In addition, further training is done with data from a company project to fine-tune the model and find bugs from that project specifically. In the following, we provide a more detailed description of the approach. This includes the problem definition, description of the datasets, model architecture and hyperparameter optimization. A visualization of the approach can be seen in Figure 3.1.

The natural distribution of Java projects has considerably more non-defective methods than defective ones [23] which means that the dataset is imbalanced. For asymmetric binary classification problems such as bug or cancer detection, the minority class is generally marked as positive. Hence, in this thesis, all the positive examples are equal to defective examples and the aim is to find as many

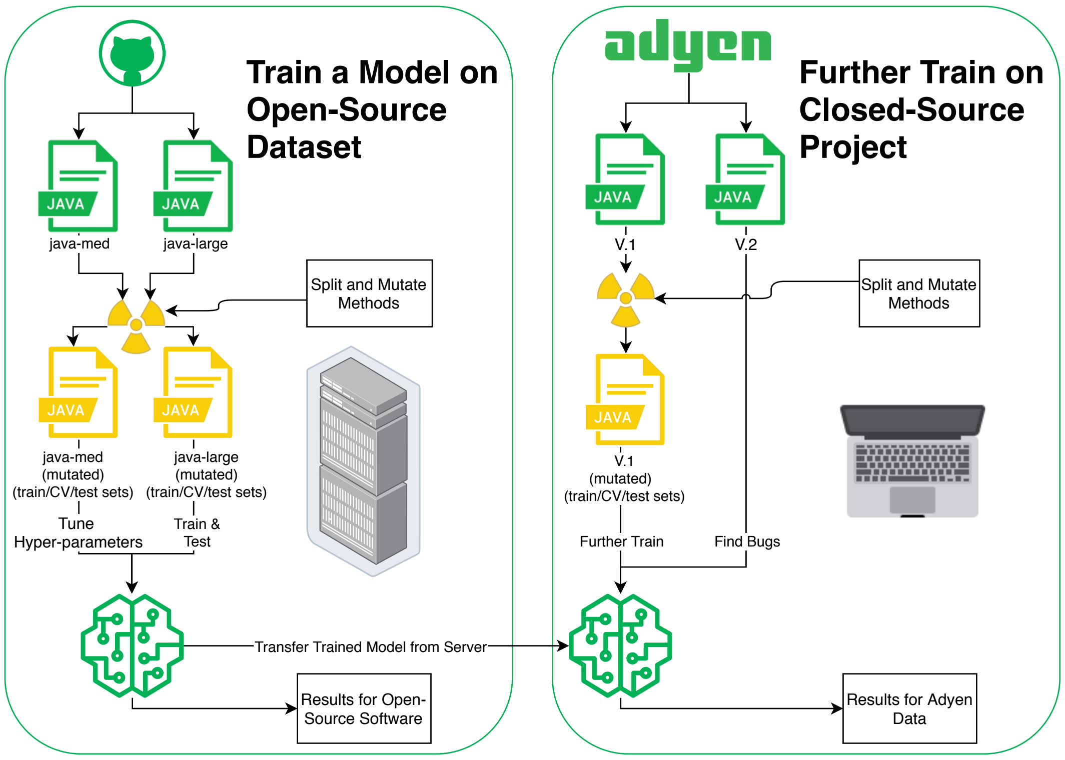
Train a Model on Open-Source Dataset
JAVA
java-med
JAVA
java-large
Split and Mutate Methods
JAVA
java-med (mutated) (train/CV/test sets)
Tune Hyper-parameters
JAVA
java-large (mutated) (train/CV/test sets)
Train & Test
Transfer Trained Model from Server
Results for Open-Source Software
adyen
Further Train on Closed-Source Project
JAVA
V.1
JAVA
V.2
Split and Mutate Methods
JAVA
V.1 (mutated) (train/CV/test sets)
Further Train
Find Bugs
Results for Adyen Data

Figure 3.1: The flow of the research including data collection, mutation, training and testing.

positive examples without misclassifying the negative examples (creating false positives).

3.1 Datasets

We used different datasets to tune hyperparameters, train, evaluate and test the model. Java-large and java-medium datasets provided by Alon et al. [8] were used for model training and hyper-parameter selection respectively. We used Adyen's production Java code to further train and test the model with project-specific data. We used an additional real-world-bugs dataset to evaluate models on real-world bugs. A summary of the datasets can be seen in Table 3.1.

1. *Java-med-balanced* dataset consists of 1000 top-starred Java projects from GitHub. Out of those 1000 projects, 800 were randomly selected for the training set, 100 for validation set and the remainder 100 were used for the testing set. Originally, this dataset contained about 4M methods, but 170,295 were candidates for off-by-one errors (e.g. methods with loops and if conditions containing binary operator $<$, $<=$, $>$ or $>=$). This re-

sulted in a balanced dataset of 340,590 methods, 170,295 of them assumed to be correct and 170,295 assumed to be buggy.

2. *Java-large-balanced* dataset consists of 9500 top-starred Java projects from GitHub created since January 2007. Out of those 9500 projects, 9000 were randomly selected for the training set, 250 for the validation set and the remainder 300 were used for the testing set. Originally, this dataset contained about 16M methods, but 836,380 were candidates for off-by-one errors (e.g. methods with loops and if conditions containing binary operator $<$, $<=$, $>$ or $>=$). After mutating the methods, the final balanced dataset consisted of 1,672,760 methods, 836,380 of them assumed to be correct and 836,380 assumed to be buggy.

3. Additional imbalanced datasets *java-med-imbalanced* and *java-large-imbalanced* were constructed to emulate more realistic data, where the majority of the code is not defective. A 10-to-1 ratio between non-defective and defective methods was chosen since it resulted in a high precision while having a reasonable recall. It was empirically observed that upon increasing the ratio of non-defective methods even further, the model did not return possibly defective methods when running on Adyen's codebase. Meaning that if the ratio was higher than 10-to-1, the recall of the model became too low to use it.

4. *Adyen's code* is a repository containing the production Java code of the company. It consists of over 200,000 methods out of which 7,435 contain a mutation candidate to produce an off-by-one error. After mutating the methods, this resulted in a balanced dataset containing 14,870 data points.

5. *41 real-world bugs* in boundary conditions were used for manual evaluation. We extracted the bugs from the 500 most starred GitHub Java projects. The analyzed projects were not part of the training and evaluation sets and thus are not seen by a model before testing. Using a Pydriller script [48], we extracted a list of candidate commits where authors made a change in a comparator (e.g., a ">" to "=>"; "<=" to "<", etc.). This process returned a list of 1,571 candidate commits which were analyzed manually until 41 were confirmed to be off-by-one errors and added to the dataset. The manual analysis was stopped due to it being a very labor-intensive process.

The java-medium and java-large datasets only contained Java files from open-source projects and are shown to contain very little duplicates between training, validation and testing sets by Allamanis et al. [3] due to the split of the

Table 3.1: Datasets used in the paper

Dataset name	**Split Ratio nobug:bug**	**Number of extracted examples**			
		Train	**Validation**	**Test**	**Total**
Java-med-balanced	1:1	266,934	41,786	31,870	**340,590**
Java-med-imbalanced	10:1	146,814	22,982	17,529	**187,325**
Java-large-balanced	1:1	1,593,610	30,634	48,516	**1,672,760**
Java-large-imbalanced	10:1	876,485	16,849	26,684	**920,018**
Adyen code	1:1	11,032	690	3,148	**14,870**
Real-life dataset	1:1	0	0	100	**100**

dataset occurring at the project level. A near duplicate remover was also explored (available in GitHub [1]), but it was ultimately not used due to time constraints.

In order to train a supervised binary classification model, defective examples are also needed. To get defective samples, we modified the existing likely correct code to produce likely incorrect code. We stripped the dataset of the methods which did not contain binary expressions ("<", "<=", ">" or "=>"). Next, for each method, we found a list of possible mutation points and selected a random one. After this, we altered the selected binary expressions using JavaParser[2] in a way to generate an off-by-one error (see Section 2.1). The flow of the feature extraction is depicted in Figure 3.2.

Due to changing only one of the expressions, the equivalent mutant problem does not exist[3] for the training examples in this thesis unless the original code was unreachable at the position of the mutation. It is also important to note that the datasets are split on a project level for the java-med and java-large datasets and on a submodule level for Adyen code. This means that the positive and the negative examples both end up in the same training, validation or test set. We did this to avoid evaluating model predictions on a code that only had one binary operator changed compared to the code that was used during training.

To provide input to the model, the AST of the previously processed code had to be converted to features. The GREAT approach (see Section 2.4.3) allows to feed the code tokens and their connections to the model together with the label. However, some extra preprocessing is required for Code2Vec and Code2Seq based models. This was done by traversing the paths of the AST with a *for loop* until the desired count of paths was reached or all of the paths were traversed. A path of the AST consists of two terminal (leaf) nodes, one at the start and one

[1] Near duplicate Java code remover: `https://github.com/SERG-Delft/near-duplicate-code-remover`

[2] JavaParser GitHub page `https://github.com/javaparser/javaparser/`

[3] Equivalent mutant problem may exist, for example, if we mutate "dead code". However, we conjecture that this is a negligible problem and will not affect the results.

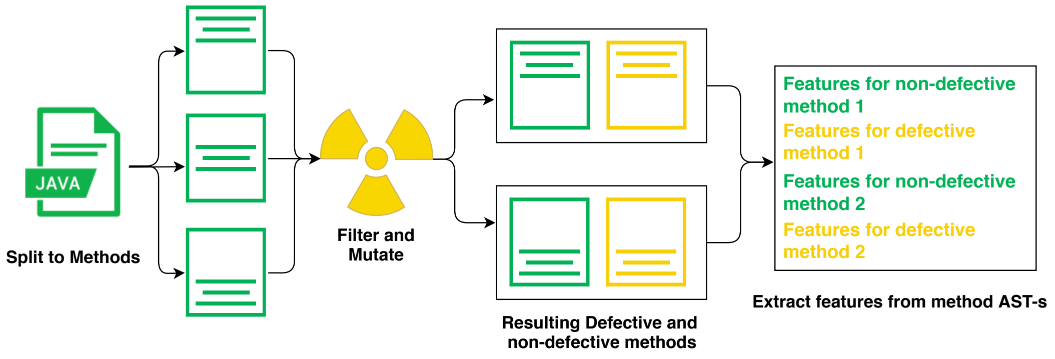
JAVA
Split to Methods
Filter and Mutate
Resulting Defective and non-defective methods
Features for non-defective method 1
Features for defective method 1
Features for non-defective method 2
Features for defective method 2
Extract features from method AST-s

Figure 3.2: The flow of feature extraction for Code2Vec and Code2Seq models.

at the end, and the path between them. The terminal nodes consist of literals and identifiers defined by users, such as a value of 1000 or an integer variable called *myInteger* whereas the nodes between the terminals are from a limited set of AST constructs such as BinaryExpression, BlockStmt, etc. defined by the implementation of JavaParser.

The feature extraction for Code2Vec or Code2Seq models is very similar, the main difference is in the path representation and Code2Seq preprocessing terminal values as subtokens when there is a variable with CamelCase notation (more details in Sections 2.4.2 and 2.4.1). An example of a single path in the AST is provided in figure 2.3.

3.2 Analysis

As mentioned at the beginning of this chapter, the real-world dataset is imbalanced. This means that the metrics to analyze the model have to be chosen carefully. For example, if accuracy is used on a dataset where 90% of the examples are correct, then a model that only predicts examples as non-defective will also achieve an accuracy of 90%. This is seemingly excellent but does not reflect the actual usefulness of the model. To better understand the performance of the model, we evaluate it with precision and recall.

Precision helps to evaluate the models' proneness to classify negative examples as positive. The latter is also known as false positive. This means that a model with high precision has a low false-positive rate and a model with low precision has a high false-positive rate. More formally, precision is the number of true positive (TP) predictions divided by the sum of true positive and false positive (FP) predictions:

$$Precision = \frac{TP}{TP + FP}$$

For a bug detection model, low precision means a high number of false positives, making the developers spend their time checking a large number of errors reported by the model only to find very few predictions that are defective. This means that in this work, we prefer high precision for a bug-detection model.

Monitoring precision alone is not enough since a model that is precise but only predicts a very few bugs per thousands of bugs is also not useful. Hence, recall is also measured. It measures the models' ability to find all the defective examples from the dataset. A recall of a model is low when it does not find many of the positive examples from the dataset and very high if it manages to find all of them. More formally, it is the number of true positive predictions divided by the sum of true positive and false negative predictions.

$$Recall = \frac{TP}{TP + FN}$$

Ideally, a bug prediction model would find all of the bugs from the dataset and have a high recall score. However, deep learning networks usually do not achieve perfect precision and recall at the same time. For more difficult problems with a probabilistic model, there can be a tradeoff. When increasing the threshold of the model confidence for the positive example, the recall will decline. For this reason, a sci-kit learn package was used to also make a precision-recall curve to observe the effect of the change in precision and recall upon changing the confidence of the model needed to classify an example as positive (defective).

3.3 Model Architecture

The models we used in this work are based on the recent Code2Vec model [9] and its enhancement Code2Seq [8]. We describe the models in more detail in Sections 2.4.1 and 2.4.2, but the modifications we made to the models are described in this section. Additionally, a Transformer model by Hellendoorn et al. [26] and a baseline model are described, which were used as a comparison to the former models.

3.3.1 Code2Vec

Code2Vec model generates code embeddings from a given snippet of source code. The use of embeddings is inspired by natural language processing, where analogous embeddings generated using different techniques such as Word2Vec [31] started a rapid development in various NLP tasks.

However, these embeddings have not been thoroughly tested for the task of bug detection. This represents an interesting direction to look into. The

[4] `https://github.com/serg-delft/jpacman`

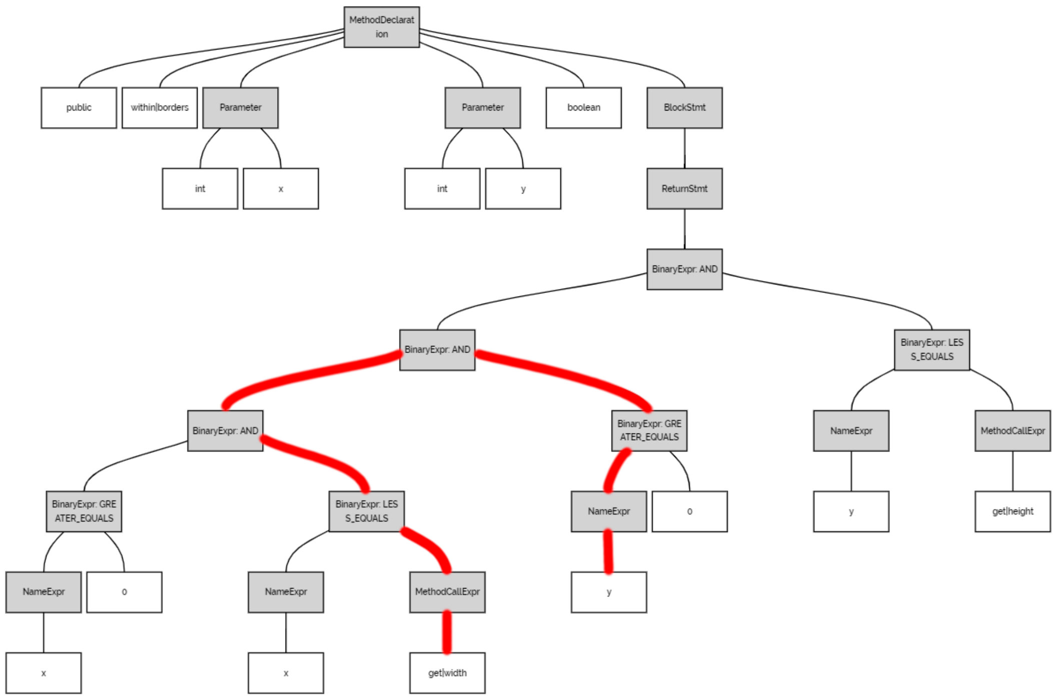

Figure 3.3: Example AST of the code in Figure 3.4. The red path traverses AST path from method call expression getWidth() to variable y.

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

Figure 3.4: An example of a boundary condition. Code extracted from JPacman[4].

possible tags in the original task proposed for Code2Vec were method names, but in this thesis, the input data contains only binary labels for bug indication. As an example, the code represented in Figure 3.4 becomes input to the model in Figure 3.5. The input consists of a label followed by a series of hashed context paths that traverse the AST path given in Figure 3.3. The first context path traversed is colored in red.

```
nobug get|width,−961791548,y y,−2020533971,boolean y,1380883547,0
    ↪ y,−1237204701,y y,576687632,get|height boolean,−1974650020,0
    ↪ boolean,470433203,y boolean,613922707,0 ...
```

Figure 3.5: An example of the AST paths extracted for the Code2Vec model.

This effectively means that the last part of the model which was previously used for name prediction becomes a binary classification model for off-by-one errors and results in a probability for each tag y_i. The higher the probability, the more likely the tag (”bug” / ”nobug”) belongs to the method. The source code to the model is available in GitHub[5].

3.3.2 Code2Seq

We selected the Code2Seq model for very similar reasons as the Code2Vec model. It also uses AST paths for code related tasks. However, as explained in Section 2.4.2, it preprocesses the AST terminal values as subtokens, which enables to have less learnable parameters and is more flexible towards out-of-vocabulary cases. Moreover, the Code2Seq does not rely on code embeddings, but encoding and decoding.

The structure of the input is similar, with the main difference being the terminal nodes being split at camel cases and the path itself being a series of tokens instead of a hashed version of the path. As an example, the code represented in Figure 3.4 becomes input to the model in Figure 3.6. The input consists of a label followed by a series of hashed context paths that traverse the AST path given in Figure 3.3. The first context path traversed is colored in red.

```
nobug get|width,SmplNm0|Cal|Leq|And|And|Geq|Nm|SmplNm0,y
    ↪ public,Mdfr0|Mth|SmplNm1,within|borders ...
```

Figure 3.6: An example of the AST paths extracted for the Code2Seq model.

The downstream task with Code2Seq was predicting natural language description for a method and the original model used a decoder of a sequence-to-sequence model to generate final labels. However, in this work, we only consider the first token of the sequence as the output. Moreover, since the network has never seen a sequence longer than one token (’bug”/”nobug”), it will not output a longer sequence and the model becomes a probabilistic binary classification model. The source code to the model is available in GitHub[6].

3.3.3 Graph Relational Embedding Attention Transformer

We used Graph Relational Embedding Attention Transformers (GREAT for short) by Hellendoorn et al. [25] to compare the performance to our existing models. Originally, the model was used for a popular VarMisuse task, with the aim of

[5]Source code to the Code2Vec model used in this thesis: `https://github.com/hsellik/thesis/tree/master/code2vec`

[6]Source code to the Code2Seq model used in this thesis: `https://github.com/hsellik/thesis/tree/master/code2seq`

pointing to a misused variable in code and to another variable, which should be there instead. However, in this work, we disregarded the pointer to the correct variable and trained the first pointer to locate the wrong usage of binary operators instead.

The model takes a graphical representation inspired by Hellendoorn et al. [26] and Allamanis et al. [7] as its input. More precisely, the input is a list of code tokens, followed by edge information which includes the connection type. The edges allow to add more information regarding the data flow, for example, it allows to connect a variable token to its next usage in the code. The input generator for the model is open-sourced in a GitHub repository [7] and the model is also available in GitHub[8].

3.3.4 Baseline Model

We developed a baseline model to assess the performance of a simpler architecture. For this, we used a Random Forest model [14] and compared the performance with the same datasets. The implementation is freely available on a git repository[9].

For preprocessing, we used the same JavaParser library to generate tokens out of Java code. This time, we did not obtain any information regarding the AST and we used the library only to more robustly generate tokens from the methods compared to other approaches, such as using regular expressions. For example, the *computeSomething* method in Figure 2.4 was tokenized to the following input to the model.

```
bug public String computeSomething ( boolean flag ) { if ( flag ) { return null ; } else
    ↪ { return "something" ; } }
```

Figure 3.7: An example of the tokenization for the baseline model.

The first token is the label indicating if the method is defective, followed by the complete tokenized method. Tokens are then passed through a TF-IDF vectorizer [10] to generate numerical input to the model. During training time, the vector size is restricted so at least 300 examples are available for each feature. Hence, different training sets use different vector sizes obtained by calculating

[7] https://github.com/SERG-Delft/j2graph

[8] Source code to the GREAT model used in this thesis: https://github.com/hsellik/thesis/tree/master/great

[9] Baseline model repository: https://github.com/hsellik/thesis/tree/master/baseline

[10] Sklearn TF-IDF vectorizer https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

$m/300$ where m is the number of examples. The source code to the model is available in GitHub[11].

3.4 Hyper-parameter Optimization

The machine learning model parameters manually selected by the user, also known as *hyper-parameters*, are defined before training the model and generally do not change during the training process. To get the best performance for a given model, the space of hyper-parameter combinations must be tested empirically. This process is known as *hyper-parameter optimization*. The best resulting hyper-parameters after optimization for Code2Vec and Code2Seq models can be seen in Appendix A. The time-consuming effort of hyper-parameter optimization was not used for the GREAT model due to the similarity of the original task and the time constraints while conducting the thesis.

For hyper-parameter optimization, we used Bayesian optimization [46]. We selected model precision as the optimization parameter since high precision is required to obtain a usable defect prediction model. We used a machine with Intel(R) Xeon(R) CPU E5-2660 v3 processor running at 2.60GHz with a Tesla M60 graphics card.

Table 3.2: Hyper-parameter optimization runs[1]

Model Type	**Dataset**	**Runs**	**Epochs**	**Total Runtime**	**Validation Precision**	**Validation Recall**
Code2Vec①	java-med-imbalanced	120	6	6 days	0.68	**0.43**
Code2Vec②	java-med-balanced	26	6	3 days	0.82	**0.82**
Code2Seq③	java-med-imbalanced	59	7	8 days	**0.85**	0.35
Code2Seq④	java-med-balanced	100	7	25 days	**0.83**	0.80

[1] Results of the actual training runs are logged in Weights & Biases environment. Access them by following the link in the circles or from appendix A

We ran optimization for four different scenarios. Two runs for the balanced java-medium dataset with Code2Vec model and Code2Seq models respectively and an additional 2 runs with the same models for imbalanced datasets. The total computation time and the number of completed runs can be seen in Table 3.2. The interactive hyper-parameter optimization runs and results are available as interactive graphs on Weights and Biases [13] website (see footnotes in Table 3.2).

[11] Source code to the baseline model used in this thesis: `https://github.com/hsellik/thesis/tree/master/baseline`

Overall, the Code2Seq model achieved better results with best hyper-parameter configurations in terms of precision (0.85 vs 0.68 and 0.83 vs 0.82 for imbalanced and balanced datasets respectively). However, for the Code2Vec model, hyper-parameter configurations were found which performed better in terms of recall (0.43 vs 0.35 and 0.82 vs 0.80 for imbalanced and balanced datasets respectively).

We used Bayesian optimization over other methods like random search or grid search because it enables us to generate a *surrogate function* that is used to search the hyper-parameter space based on previous results and acts as intuition for parameter selection. This results in saving significantly more time because the actual model does not need to run as much due to wrong parameter ranges being discarded early in the process.

Table 3.3: Model training times.[12]

	Training on B			Training on I			Further Training	
	java-large			java-large			Adyen Data	
Model	Time	#Epochs		Time	#Epochs		Time	#Epochs
Baseline	5h 33m	1	⑤	1h 59m	1	⑥	48s[13]	1
GREAT	4d 49m	-	⑦	4d 18h	-	⑧	4h 9m	53
Code2Vec	1d 2h 2m	52	⑨	11h 6m	52	⑩	1h 1m	53
Code2Seq	3d 18h 18m	14	⑪	2d 14h 41m	15	⑫	1h 8m	17

3.5 Threats to validity

In this section, we discuss the threats to the validity of this study and the actions we took to mitigate them.

Internal validity. Our method uses an automatic mutation to generate faulty examples from likely correct code by editing one of the binary condition within the method. This means that while the correct examples represent a diverse set of methods from open-source projects, the likely incorrect methods may not represent a realistic distribution of real-world bugs. This affects the model that is being trained with those examples and also the testing results conducted on this data.
In addition, the comparison between Code2Seq and GREAT model might be misleading due to the latter analyzing and being evaluated on token level, but

the former on method level.

External validity. While the work included a diverse set of open-source projects, the only closed-source project that was used during this study was Adyen's. Hence, the closed-source projects are underrepresented in this study.

Chapter 4

Results

In this section, we describe the main research questions. Additionally, we describe the methodology to answer them with the obtained results.

4.1 Research Questions

RQ_1 How do the models perform on a controlled dataset?

In order to obtain a vast quantity of data, we use a controlled dataset (see Section 3.1). We train the models on the dataset and use metrics such as precision and recall to assess the performance.

RQ_2 How well do the methods generalize to a dataset made of real-world bugs?

We mine a dataset of real-world off-by-one error bugs from GitHub issues of various open-source projects. Then we use a model to predict the error-proneness of a method before and after a fix. This will indicate how well the model works for real-world data. This evaluation will enable us to extract the precision metric and compare it to the one from RQ_1.

RQ_3 Can the approach be used to find bugs from a large-scale industry project?

One useful application to an error-detection model is to analyze the existing project and notify of methods containing off-by-one errors. We make several runs where the model is firstly trained on a dataset with mutated code and then tested on real code to find such errors. In addition, we further train the model with a different version of the industry project to find errors in the future versions of the project.

4.2 Methodology

4.2.1 Open Source Software

To answer *RQ_1*, we performed hyper-parameter optimization. After this, we selected the best hyper-parameter values and trained the model with randomly initialized parameters on the java-large dataset on the same machine as used for hyper-parameter optimization (see Section 3.4). We trained Code2Seq and Code2Vec models until there was no gain in precision for 3 epochs of training in the evaluation set. After this, we assessed the model on the testing set of java-large dataset.
The process was conducted for three different configurations of data. These were:

1. BB - the training data was balanced (B) with the cross-validation and testing data also being balanced (B).

2. BI - the training data was balanced (B) with the cross-validation and testing data being imbalanced (I).

3. II - the training data was imbalanced (I) with the cross-validation and the testing data also being imbalanced (I).

The data imbalance was inspired by the work of Habib et al. [24], who reported that a bug detection model trained on a balanced dataset will have poor performance when testing on a more real-life scenario with imbalanced classes.

4.2.2 Testing on Real-World Defects

To answer *RQ_2*, we selected the best-performing model on the controlled java-large testing set (see Table 4.1), which was the model based on the Code2Seq architecture. After this, the model was tested on the bugs and their fixes found from several real-world Java projects (real-life dataset in Table 3.1).
Firstly, we tested the model on the correct code that was obtained from the GitHub diff after the change to see the classification performance on non-defective code. To test the model performance on defective code, the example was reverted to the state where the bug was present using the git version control system. After this, we recorded the model prediction on the defective method.

4.2.3 Adyen Data

To answer *RQ_3*, we trained the Code2Seq model only on the data generated from the company project, but the training did not start with randomly initial-

ized weights. Instead, the process was started with the weights acquired after training on the java-large dataset (see Figure 3.1).
We selected the Code2Seq based model because it had the best performance on the imbalanced testing set of the controlled java-large set. We selected the performance on the imbalanced controlled set as a criterion since we assumed that the company project also contains more non-defective examples than defective ones.
We used the pre-trained model because the company project alone did not contain enough data for the training process. Additionally, due to the architecture of the Code2Seq and Code2Vec models, the embeddings of terminal and AST node vocabularies did not receive additional updates during further training with company data. We trained the model until there was no gain in precision for 3 epochs on the validation set and after this, we tested the model on the test set consisting of controlled Adyen data.
We conducted additional checking on Adyen data by trying to find bugs in the newer version of the project. More specifically, we updated the project to a newer version using the git version control system and without any modifications to the original code, every Java method from the project was analyzed with the model. We analyzed all bug predictions that were over a threshold of 0.8 to see if they contained bugs.

4.2.4 Adyen

Adyen is one of the world's largest payment service providers allowing customers from over 150 countries to use over 250 payment methods including different internet bank transfers and point of sales solutions. In the fiscal year of 2019, Adyen's processing volume was 239.6 billion euros with a net revenue of 496.7 million euros [1]. It is notable, that the amount of payments processed in euros saw a 51% increase compared to the previous year, characterizing the company's rapid development.
The company is working in a highly regulated banking industry and combined with the high processing volumes there is little to no room for errors. Hence, Adyen uses the industry-standard best practices for early bug detection such as code reviews, unit testing and static analysis.
In addition to traditional quality assurance methods, it is at Adyen's best interest to look into novel tools to prevent software defects finding their way into their large codebase, preferring methods that are scalable and do not waste the most expensive resource of the company, the developers' time. The need exists to research machine learning systems that are capable of discovering software faults that the current static analysis tools fail to capture.

4.3 RQ$_1$ How do the models perform on a controlled dataset?

In Table 4.1, we show the precision and recall of the different models. In Figures 4.1, 4.2 and 4.3, we show the ROC curve, precision-recall curve and the confusion matrix of the experiment with Code2Seq based model for the imbalanced java-large dataset.

Table 4.1: Model results in controlled testing sets.[1,2]

	Experiment BB			**Experiment BI**			**Experiment II**						
	Java-large			Java-large			Java-large			Adyen data (cross-project)		Adyen data (further trained)	
Model	Pr. %	Re. %		Pr. %	Re. %		Pr. %	Re. %		Pr. %	Re. %	Pr. %	Re. %
Code2Seq	**85.23**	**84.82**	13	36.08	**84.86**	14	83.04	**42.34**	15	**71.15**	**24.66**	66.66	**30.66**
Code2Vec	80.11	77.01	16	28.52	75.53	17	64.65	41	18	53.85	20.46	43.95	23.39
GREAT	48.69	80.9	19	**100.00**	0.31	20	**100.00**	1.92	21	00.00	00.00	**95.40**	0.23
Baseline	50	49.08	22	8.99	49.18	23	17.86	0.15	24	0	0	9.25[3]	0.92[3]
Offside	80.9	75.6		-	-		-	-		-	-	-	-

[1] Results of the actual training runs are logged in Weights & Biases environment. Access them by following the link in the circles leading to appendix A

[2] BB stands for balanced training and testing set, II stands for imbalanced training set and testing set

[3] The model was trained only on Adyen data because the framework did not provide means to train partially

Observation 1: Models present high precision and recall when trained and tested with balanced data. The results show that when training models on a balanced dataset with an equal amount of defective/non-defective code and then testing the same model on a balanced testing set, both Code2Vec and Code2Seq models achieve great precision and recall where the Code2Seq based model has better precision (85.23% vs 80.11%) and recall (84.82% vs 77.01%) compared to the Code2Vec based model. In addition, the balanced models' performance was compared to the one used in OffSide [15], our previous work exploring only the use of Code2Vec model, which was also tested on the identical java-large dataset using very similar preprocessing pipeline and training model (80.11% vs 80.9% precision and 77.01% vs 75.6% recall).
The GREAT model also achieves high recall (80.9%), however, it suffers from a lower precision (48.69%) compared to Code2Seq and Code2Vec models. The baseline model achieves a 50% precision and 50% recall, which is considerably lower compared to the other models.

Observation 2: The metrics drop considerably when tested on an imbalanced dataset. When simulating a more real-life scenario and creating an imbalance in the testing set with more non-defective methods, the recall of

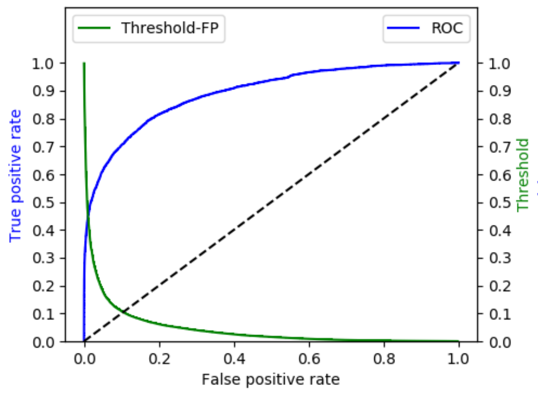

Figure 4.1: ROC curve for Code2Seq Experiment II with the java-large dataset.
Area under curve 0.89.

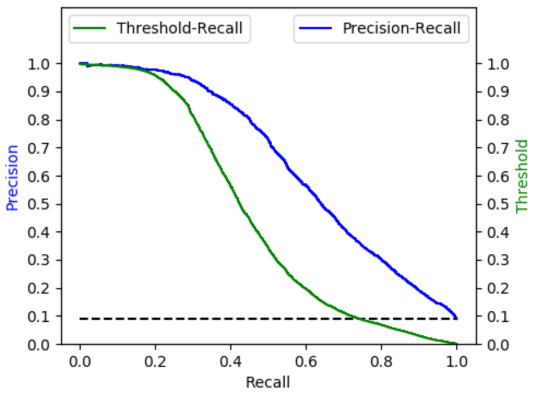

Figure 4.2: Precision-recall curve for Code2Seq Experiment II with the java-large dataset. Area under curve 0.65.

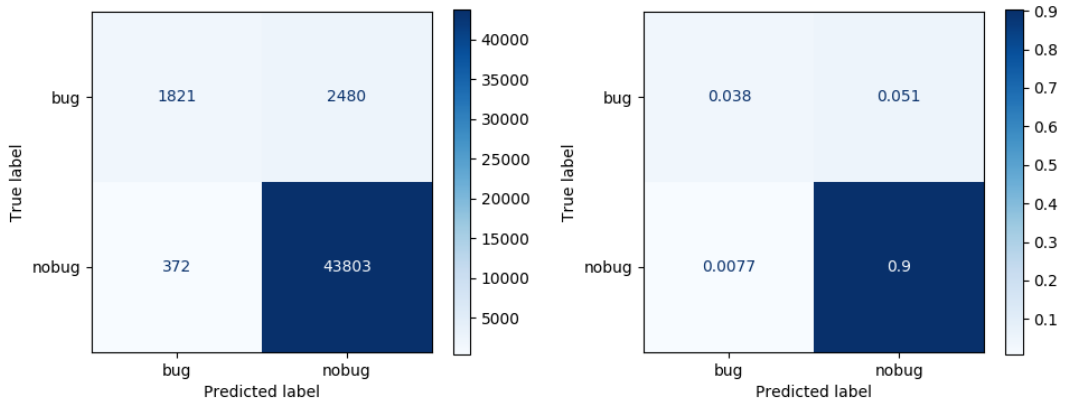

Figure 4.3: Confusion matrix and normalized confusion matrix for Code2Seq Experiment II with the Java-large dataset.

the models remained similar with recall increasing from 84.82 to 84.86 for the Code2Seq model and dropping from 77.01% to 75.53% for the Code2Vec model. However, the precision of the models reduced drastically with the Code2Seq model dropping from 85.23% to 36.08% and Code2Vec model from 80.11% to 28.52%.
The baseline model also drops in precision from 50% to 8.99% while keeping the same recall. interestingly, the GREAT model increases in precision from 48.69% to 100%, but the recall on the other hand decreases from 80.9% to 0.31%.

Observation 3: Low precision can be mitigated by training on an imbalanced dataset. We trained Code2Seq and Code2Vec models on an imbalanced dataset and results show that the precision score for imbalanced data returned

almost to the same level for the Code2Seq model (83.04% vs 85.23%), but remained lower for the Code2Vec based model (64.65% vs 80.11%). However, the recall declined drastically from 84.82% to 42.34% for the Code2Seq model and from 77.01% to 41.00% for the Code2Vec model. While the precision of the GREAT model increased to perfect 100%, the low recall indicates that only very few methods were marked as buggy and a substantial amount of faulty methods were labeled as correct.
In addition, we recorded the Receiver Operating Characteristic (ROC) curve, precision-recall curve and confusion matrices for each test with Code2Vec and Code2Seq models. As an example, see Figures 4.1, 4.2 and 4.3. As seen from Figure 4.2, the precision is ≈0.8 while recall remains ≈0.5 at a confidence threshold of 0.8. Moreover, it can also be seen that the model confidence is correlated where higher thresholds yield better precision but lower recall.

RQ_1 summary: Both Code2Seq and Code2Vec based models have a very good performance on a controlled dataset. Training model on an imbalanced dataset yields better results when also testing on an imbalanced dataset. This is not true for Random Forest based model and the GREAT model which already show poor results on the balanced dataset.

4.4 RQ_2 How well do the methods generalize to a dataset made of real-world bugs?

The performance of the model in the 41 real-world boundary mistakes and their non-defective counterparts are presented in Table 4.2 with the more detailed Table 4.3 for the Code2Seq(B) model with a threshold of 0.5 available. The Code2Vec and GREAT models were not used in this experiment given their inferior performance during quantitative testing.

Table 4.2: Results of applying the Code2Seq model to 41 real-world off-by-one bugs and their corrected versions. B stands for balanced training set, I stands for imbalanced training set.

Model	**Threshold**	**TP**	**TN**	**FP**	**FN**	**Precision**	**Recall**	**F1**
Code2Seq (B)	0.5	**19**	26	15	**22**	55.88	**46.34**	**50.67**
Code2Seq (B)	0.8	10	33	8	31	55.56	24.39	33.9
Code2Seq (I)	0.5	3	**41**	**0**	38	**100**	7.32	13.64
Code2Seq (I)	0.8	1	**41**	**0**	40	**100**	2.44	4.76

Observation 4: The model can detect real-world bugs, but with a high false-positive rate. Out of the 41 defective methods 19 (46.34%) were classi-

fied correctly and out of 41 correct methods, 26 (63.41%) were classified correctly.
The precision and recall scores of 55.88 and 46.34 were achieved while evaluating the model on real-world bugs with the Code2Seq model trained on balanced data using a threshold of 0.5. Compared to the results from the java-large testing set with augmented methods, the results are significantly lower with precision and recall being 29.35 and 38.08 points lower respectively (see metrics for Code2Seq model with *Experiment BB* in Table 4.1).

Table 4.3: Results of applying the Code2Seq model (trained on balanced data with a threshold of 0.5) to 41 real-world off-by-one bugs. A ✓indicates where the model made the right decision.

#	**Project**	**Bug fix**	**Statement type**	**Bug Prediction**	**No-Bug Prediction**
1	Glide	>= to >	Var decl.	✓	✓
2	Alluxio	<= to <	if	✓	✓
3	Presto	>= to >	method	X	✓
4	Netty	<= to <	If	X	✓
5	Orient DB	> to >=	Method	X	✓
6	Orient DB	>= to >	If	✓	✓
7	Apache Incubator Druid	>= to >	If	✓	X
8	Cat	< to <=	For	X	X
9	Sticky List Headers	> to >=	Var. Decl.	X	✓
10	Neo4j	>= to >	If	X	X
11	Neo4j	> to >=	If	X	X
12	Neo4j	>= to >	If	X	X
13	Apache Kafka	> to >=	If	X	X
14	Alluxio	> to >=	If	X	X
15	Alluxio	> to >=	If	✓	✓
16	Netty	>= to >	If	X	✓
17	Netty	>= to >	If	✓	X
18	Netty	> to >=	If	✓	✓
19	Netty	<= to <	While	✓	✓
20	Netty	<= to <	If	X	✓
21	Clojure	> to >=	If	✓	X

22	Presto	> to >=	Method	✓	✓
23	Presto	> to >=	Method	✓	✓
24	Presto	> to >=	Method	X	✓
25	Presto	> to >=	If	X	✓
26	My Bookshelf	> to >=	If	✓	X
27	Checkstyle	>= to >	Assign	X	✓
28	Gephi	> to >=	If	✓	✓
29	H2O-3	< to <=	Ternary	X	✓
30	H2O-3	> to >=	If	X	✓
31	H2O-3	>= to >	If	✓	X
32	H2O-3	> to >=	Assert	✓	✓
33	MapDB	>= to >	If	✓	✓
34	Google Gson	< to <=	For	X	X
35	Jenkins	> to >=	While	X	X
36	Jenkins	> to >=	While	X	✓
37	Jenkins	> to >=	While	X	X
38	Danmaku Flame Master	< to <=	If	✓	X
39	Platform Frameworks Base	<= to <	For	✓	✓
40	Apache Jmeter	> to >=	For	✓	✓
41	Oracle Graal	< to <=	For	X	✓

RQ$_2$ summary: The model represents a reasonable performance on the real-world dataset. If the model threshold is set high, the recall is very low and if the threshold is lowered, the false-positive rate becomes an issue.

4.5 RQ$_3$ Can the approach be used to find bugs from a large-scale industry project?

We present test results for the Adyen dataset with controlled bugs in Table 4.1.

Observation 5: Testing on a controlled dataset shows promising results.

Our empirical findings show that when a model is trained on an open-source dataset and then applied to the company project, it will have good precision and recall scores with 71.15% and 24.66% for Code2Seq and somewhat lower 53.85% and 20.46% for Code2Vec model respectively.

Observation 6: Further training on the Adyen project did not yield better results. It was hypothesized, that training the model further on Adyen codebase would give a boost in precision and recall scores. The recall of the models improved by 6.0 percentage points for Code2Seq based model and 2.93 for Code2Vec based model. However, the precision of both models dropped by 4.49 percentage points for Code2Seq and 9.9 for Code2Vec. A slight improvement was made to the GREAT model when further training on Adyen data, but the performance of the model was very low in terms of recall achieving only 0.23%.

Observation 7: Using the Code2Seq model to find new off-by-one bugs from the company repository did not reveal any bugs, but 20% of the methods reported by the model were considered suspicious by the developers. We only used Code2Seq model in this experiment given its better performance in the quantitative tests. Running the model on a newer version of the repository reported 36 potential bugs with a confidentiality threshold over 0.8. While no bugs were found after manual analysis, 7 of the methods could be marked as suspicious. When we showed these methods to the developers, they agreed that the methods deviate from good coding standards. These methods are categorized in Table 7.

Table 4.5: Exotic code types that were found by the model.

Exotic Code Type	Number of Methods
For loop initiated at wrong index	4
Constraints in binary expression not consistent	3

One example of the for loop case can be seen in Figure 4.4, where a *for loop* was initiated with an index value of one, but inside of the loop, one was subtracted so the loop could start at index 0. This is unnecessary as it is unconventional and requires additional comments to make the code understandable. There were similar situations, where the correct iterating value was used only once, but in the rest of the for loop, the value was still subtracted by one. While technically correct, it is difficult to read. For an example see Figure 4.5.
The second type of exotic code that surpassed the model threshold was related

```
for (int i = 1; i <= something.length;i++) {
    char variable = callSomething(i−1); // we start with i=1, so subtract 1 for the
        ↪ index
    callFoo(variable);
}
```

Figure 4.4: A pseudocode of the situation the model predicted to be buggy.

```
for (int i = 1; i <= something.length;i++) {
    callFoo1(i−1);
    callFoo2(i−1);
    ..
    callFoo3(i−1);
    callFoo4(i); // real index only used once
    callFoo5(i−1);
    ...
    callFoo(i−1);
}
```

Figure 4.5: A pseudocode of the situation the model predicted to be buggy.

to the boundary conditions while comparing ranges. This is illustrated by Figure 4.6 where the first binary operator is larger or equal and the second one being *less* operator. This is compensated by the values that the expression uses and is tied to the business logic rather than an actual error. However, it is a possible source of error if such operators are not evaluated diligently during software development.

```
// check if value is between 1 (inclusive) and 10 (inclusive)
int a = callFoo("string");
if (a >= 1 && a < 11) {
        callFoo1();
} else {
        callBar();
}
```

Figure 4.6: A pseudocode of the situation the model predicted to be buggy.

Observation 8: Model can potentially be useful at commit time, however, the low recall makes it impractical. Fixing mistakes regarding good code practices for old pieces of software is not considered worthwhile at Adyen given the possible unwanted changes to the behavior of the software. However, if such a system were to be employed during automated testing, the alerts might help developers to adhere to better practices. However, the low recall makes it practical for very few commits.

RQ$_3$ summary: The model has reasonable performance on the controlled dataset. However, when it is tested on real production repository, the approach did not reveal any bugs per se, but pointed to code considered to deviate from good practices. Hence, it might add some additional value to existing static analysis checks, but can not replace them.

Chapter 5

Conclusion

Software development practices offer many techniques for detecting bugs at an early stage. However, these methods come with their challenges and are either too labor-intensive or leave a lot of room for improvement.
In this thesis, we studied the current state in the automated software testing related to Static Analysis and Machine Learning systems. We also adapted recent state-of-the-art deep learning models using Abstract Syntax Trees to find off-by-one errors in Java code which are very hard for traditional static analysis tools due to their high dependency on context.
We trained our model on approximately 1.6M examples and tested the results on 48K examples and conclude that the approach is effective on a controlled dataset having a precision of 85.23%, recall of 84.82% and an accuracy of 85.06%. However, the experiments also show that models trained on balanced data show very poor precision on imbalanced datasets, which we mitigated with training on imbalanced data.
Additionally, the model was tested on a large-scale Java project at Adyen. The model did not find any real errors but revealed methods with good code practice issues which shows that the model can capture the structure of the code. In order to use the model in production, the challenge of high false positive rate needs to be addressed. We believe that there is a potential in using Machine Learning models for defect detection and this thesis provides some exploratory work in that direction.

5.1 Future work

In this section, we discuss the possible improvements to the models and the additional experiments that could be executed with the model.

- **Analyzing model effects earlier in the development process.** In this thesis, the model was tested on code which was in the production environment from months to several years. In Adyen, the software application is in heavy use by millions of clients all over the world and the older code contains few defects. A more feasible way to discover bugs would be to apply them early in the process, such as running the model with parallel to the automated testing tools [30]. However, this kind of experiment will need a longer time frame to execute.

- **Supporting inter-procedural analysis.** Currently, our approach is only supporting the analysis of the AST of one method. However, the behavior of a method, and the possibility of the bugs thereof, also depends on the contents of the other methods. For example, in recent research by Compton et al. [19], the embedding vectors from the Code2Vec model are concatenated to form an embedding for the entire class. However, there could be better ways to achieve this.

- **Applying to other languages.** This work focuses solely on the Java programming language. However, this method could also be used on other languages such as Javascript or Python, for which the Abstract Syntax Tree can be generated.

- **Using different architectures.** In our work, we mainly looked at Code2Vec and Code2Seq models and briefly tested a Transformer model. Understanding, modifying and testing the former two models in different configurations took a significant time of this thesis. However, these models were originally constructed for method name and description generation respectively and although the authors claim that these models are good for any task, perhaps there are more optimal representations for bug detection specifically.

  One of the architectures, that looked very promising was recently made by Li et al. [34]. They used a program dependency graph and data flow graph in addition to the AST. However, their GitHub open-source implementation was not usable and they did not succeed to provide a working solution even when they were fully aware of the problem. The implementation of this model was too time-consuming to fit into the scope of this thesis.

- **Improving GREAT Architecture.** The original implementation of the model was meant for Python code and our preprocessing of the Java code to the model differed from the authors'. It may be possible, that the performance of the model would improve if the preprocessing is done differently.

Bibliography

[1] Adyen. *Adyen Shareholder Letter H2 2019*, 2020 (accessed July 2, 2020). URL `https://www.adyen.com/investor-relations/H2-2019#section0`.

[2] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. Building useful program analysis tools using an extensible java compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 14–23. IEEE, 2012.

[3] 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.

[4] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

[5] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015.

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

[7] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *PLDI*, 2020.

[8] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=H1gKYo09tX`.

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

[10] Maurício Aniche, Erick Maziero, Rafael Durelli, and Vinicius Durelli. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *arXiv preprint arXiv:2001.03338*, 2020.

[11] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical type-based null safety for java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 740–750, 2019.

[12] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481. IEEE, 2016.

[13] Lukas Biewald. Experiment tracking with weights and biases, 2020. URL `https://www.wandb.com/`. Software available from wandb.com.

[14] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[15] Jón Arnar Briem, Jordi Smit, Hendrig Sellik, Pavel Rapoport, Georgios Gousios, and Maurício Aniche. Offside: Learning to identify mistakes in boundary conditions.

[16] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[17] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–300, 2009.

[18] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in neural information processing systems*, pages 2547–2557, 2018.

[19] Rhys Compton, Eibe Frank, Panagiotis Patros, and Abigail Koay. Embedding java classes with code2vec: improvements from variable obfuscation [accepted]. In *MSR 2020*. ACM, 2020.

[20] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[21] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[22] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642, 2016.

[23] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 317–328, 2018.

[24] Andrew Habib and Michael Pradel. Neural bug finding: A study of opportunities and challenges. *arXiv preprint arXiv:1906.00307*, 2019.

[25] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 152–162, 2018.

[26] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations*, 2019.

[27] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

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

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

[30] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[31] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[32] 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.

[33] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*, 2017.

[34] 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.

[35] Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. Gated graph sequence neural networks. In *Proceedings of ICLR'16*, April 2016. URL `https://www.microsoft.com/en-us/research/publication/gated-graph-sequence-neural-networks/`.

[36] Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. Neural query expansion for code search. In *Proceedings of the 3rd acm sigplan international workshop on machine learning and programming languages*, pages 29–37, 2019.

[37] Lech Madeyski and Marcin Kawalerowicz. Continuous defect prediction: the idea and a related dataset. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 515–518. IEEE Press, 2017.

[38] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[39] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *2013 35th international conference on software engineering (ICSE)*, pages 382–391. IEEE, 2013.

[40] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 651–654, 2013.

[41] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 585–596. IEEE, 2015.

[42] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, 150:22–36, 2019.

[43] Michael Pradel and Koushik Sen. Deep learning to find bugs. *TU Darmstadt, Department of Computer Science*, 2017.

[44] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

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

[46] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[47] Tim Sonnekalb. Machine-learning supported vulnerability detection in source code. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1180–1183, 2019.

[48] 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, 2018.

[49] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014.

[50] Yu Wang, Fengjuan Gao, Linzhang Wang, and Ke Wang. Learning a static bug finder from data. *arXiv preprint arXiv:1907.05579*, 2019.

[51] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE, 2015.

Appendix A

Optimized Hyper Parameters

Best hyper-parameters selected for Code2Vec and Code2Seq models with balanced and imbalanced data respectively.

Table A.1: Best Hyper-parameters for Code2Vec model with imbalanced data

Hyper-parameter[1] [1]	**Final Value**
DEFAULT_EMBEDDINGS_SIZE	104
DROPOUT_KEEP_RATE	0.6046
MAX_CONTEXTS	210
MAX_PATH_VOCAB_SIZE	1,033,760
MAX_TOKEN_VOCAB_SIZE	1,775,529
TRAIN_BATCH_SIZE	1480

Table A.2: Best Hyper-parameters for Code2Vec model with balanced data

Hyper-parameter[1]	**Final Value**
DEFAULT_EMBEDDINGS_SIZE	106
DROPOUT_KEEP_RATE	0.6551
MAX_CONTEXTS	154
MAX_PATH_VOCAB_SIZE	629,050
MAX_TOKEN_VOCAB_SIZE	1,191,566
TRAIN_BATCH_SIZE	1551

[1]Hyper-parameter descriptions for Code2Vec model `https://bitbucket.org/DontSeeSharp/code2vec/`

[2]Hyper-parameter descriptions for Code2Seq model `https://bitbucket.org/DontSeeSharp/code2seq/`

Table A.3: Best Hyper-parameters for Code2Seq model with imbalanced data

Hyper-parameter[2]	**Final Value**
BATCH_SIZE	138
DECODER_SIZE	271
EMBEDDINGS_DROPOUT_KEEP_PROB	0.6506
MAX_CONTEXTS	293
MAX_NAME_PARTS	2
NUM_DECODER_LAYERS	1
EMBEDDINGS_SIZE	104
RNN_DROPOUT_KEEP_PROB	0.7168
SUBTOKENS_VOCAB_MAX_SIZE	141,556
MAX_PATH_LENGTH	11

Table A.4: Best Hyper-parameters for Code2Seq model with balanced data

Hyper-parameter[2]	**Final Value**
BATCH_SIZE	106
DECODER_SIZE	351
EMBEDDINGS_DROPOUT_KEEP_PROB	0.478
MAX_CONTEXTS	187
MAX_NAME_PARTS	7
NUM_DECODER_LAYERS	2
EMBEDDINGS_SIZE	114
RNN_DROPOUT_KEEP_PROB	0.68
SUBTOKENS_VOCAB_MAX_SIZE	161,513
MAX_PATH_LENGTH	22

Wandb experiment links

1. Code2Vec model sweep with an imbalanced dataset - `https://app.wandb.ai/SERG/code2vec/sweeps/mwvx6vzj`
2. Code2Vec model sweep with a balanced dataset - `https://app.wandb.ai/SERG/msc_thesis_hendrig/sweeps/97bj3ovb`
3. Code2Seq model sweep with an imbalanced dataset - `https://app.wandb.ai/SERG/msc_thesis_hendrig/sweeps/7ri9a20h`
4. Code2Seq model sweep with a balanced dataset - `https://app.wandb.ai/SERG/msc_thesis_hendrig/sweeps/3id66fcb`
5. Baseline model training run on a balanced training set: `https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/26ajk3jd/overview`
6. Baseline model training run on an imbalanced training set: `https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/2o79xhqj`
7. GREAT model training run on a balanced training set: `https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/16vr0eux`
8. GREAT model training run on an imbalanced training set: `https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/kaspx1in`
9. Code2Vec model training run on a balanced training set: `https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/qv7udsf6`
10. Code2Vec model training run on an imbalanced training set: `https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/730fzzou`
11. Code2Seq model training run on a balanced training set: `https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/6t066qxp/overview`
12. Code2Seq model training run on an imbalanced training set: `https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/w9vd78qn/overview`
13. Code2Seq model trained on balanced and tested on balanced data: `https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/hx9cotd0`
14. Code2Seq model trained on balanced and tested on imbalanced data: `https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/5lbsoj0u`

15 Code2Seq model trained on imbalanced and tested on imbalanced data: https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/4amyrlfn

16 Code2Vec model trained on balanced and tested on balanced data: https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/ku1l68mm

17 Code2Vec model trained on balanced and tested on imbalanced data: https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/payfbi3u

18 Code2Vec model trained on imbalanced and tested on imbalanced data: https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/gaue6g6p

19 GREAT model trained on balanced and tested on balanced data: https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/2ev2euek

20 GREAT model trained on balanced and tested on imbalanced data: https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/2f65jhip

21 GREAT model trained on imbalanced and tested on imbalanced data: https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/22zsp723

22 Random forest model trained on balanced and tested on balanced data https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/n6jq0hc6

23 Random forest model trained on balanced and tested on imbalanced data https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/1sc6rw7t

24 Random forest model trained on imbalanced and tested on imbalanced data https://app.wandb.ai/SERG/msc_thesis_hendrig/runs/2o79xhqj