Detecting Logical Errors in Programming Assignments Using code2seq

Kevin Chapman

Anton Lückner

Department of Computer and Systems Sciences

Degree project 15 credits
Computer and Systems Sciences
Degree project at the bachelor level
Spring term 2023
Supervisor: Mateus de Oliveira Oliveira
Swedish title: Användingen av code2seq för att upptäcka logiska fel i programmeringsuppgifter

Stockholm University

DVK Kandidatuppsats Program

This thesis was written within the Spring 2023 edition of the DVK Kandidatuppsats Program.

Coordination

Mateus de Oliveira Oliveira

Organization

Henrik Bergström
Peter Idestam-Almquist
Mateus de Oliveira Oliveira
Beatrice Åkerblom

Abstract

The demand for new competent programmers is increasing with the ever-growing dependency on technology. The workload for teachers with more and more students creates the need for more automated tools for feedback and grading. While some tools exist that alleviate this to some degree, machine learning presents an interesting avenue for techniques and tools to do this more efficiently.
Logical errors are common occurrences within novice code, and therefore a model that could detect these would alleviate the workload for the teachers and be a boon to students.
This study aims to explore the performance of the machine learning model code2seq in detecting logical errors. This is explored through an *empirical experiment* where a data-set consisting of real-world Java code that is modified to contain one specific logical error is used to train, validate and test the code2seq model. The performance of the model is measured using the metrics: *accuracy, precision, recall* and *F1-score*. The results of this study show promise for the application of the code2seq model in detecting logical errors and have the potential for real-world use in classrooms.

Keywords: Automated feedback, code2seq, machine learning, neural machine translation.

Synopsis

Background

Automated feedback is the idea of giving feedback to novice programmers without requiring an instructor to get involved, reducing the workload on teachers and thus making the entry level of programming more accessible. This area of study has found several solutions with varying levels of success. One of the areas that shows promise but is not widely explored is the use of machine learning to automate the feedback process. Studies in machine learning have shown that it has potential but needs more refined models to catch the proper errors and give appropriate feedback.

Problem

With the growth of computer science and programming education, there is an increasing need for automated feedback and correction to alleviate the workload for teachers. Though several tools and techniques exist, there is still a gap in the uses of machine learning in this area. This study strives to make strides towards filling this gap.

Research Question

The machine learning model that will be explored in its ability to be used for automated feedback is code2seq. More specifically, it will be tested on its performance at detecting logical errors with the research question:
How well does code2seq perform at detecting logical errors?

Method

To address the research question, we will perform two different *empirical experiments* where the data-set will be altered to contain one commonly occurring logical error. The original data-set will be altered separately for each experiment. The code2seq model will then be trained and tested on these data-sets and evaluated on the metrics: *accuracy, precision, recall* and *F1 score*.

Result

The results of the *for-loop iteration* experiment were 90.77% *accuracy*, 91.96% *precision*, 89.89% *recall* and a *f1-score* of 90.92%. The *cascading if-statement* experiment gave a result of 80.73% *accuracy*, 85.04% *precision*, 78.5% *recall* and 81.64% *f1-score*. These results imply that code2seq can be effective at detecting logical errors.

Discussion

The main contribution of this thesis is showing the suitability of using code2seq to detect logical errors in novice code. The results in this study are limited to proving the viability of code2seq in detecting logical errors in novice code using separately trained models for each type of error. These findings could be used for further research into these types of models or be used as a basis to implement tools based on code2seq in a real-world setting.

Acknowledgement

Special thanks to Folke Bylund for providing the hardware used in our experiments.

Contents

List of Figures

List of Tables

List of Abbreviations

ANN - Artificial neural network
AST - Abstract syntax tree
FP - False positive
FN - False negative
ML - Machine learning
NMT - Neural machine translation
TP - True positive
TN - True Negative

Chapter 1

Introduction

With growing dependencies on technology, the need for good programmers keeps increasing. With the growing demand for programmers, more and more people are enrolling in courses that require the assessment of programming assignments and exercises. This places stress on teachers and teacher-aides to give qualitative feedback to students. To alleviate this, studies on automatic assessment techniques and methods that could give similar results as manual evaluation are warranted.

According to Aldriye et al. (2019) a lot of progress has been made to improve automatic assessment but there is still room for improvement. They discuss three types of errors that are relevant when assessing programming assignments, syntax errors, logical errors and run-time errors. Of these, syntax errors are the easiest to correct since the compiler can find these automatically. The hardest error to catch is run-time errors since these errors can occur at any time during the execution of the program. Logical errors can be difficult to catch, and Aldriye et al. (2019) discusses several techniques that can be used to catch these errors and give feedback such as *Unit testing, Peer-To-Peer Feedback, Pattern Matching* etc. None of these techniques are fully automatic. Therefore it can be of interest to explore the usage of automated machine learning models, such as the code2seq model put forward in Alon et al. (2018). This study will explore the effectiveness of this model in detecting logical errors in programming assignments.

1.1 Background

According to Fradkov (2020), machine learning was first introduced by Frank Rosenblatt with the creation of the perceptron which used an input of analogue and discrete signals and was modelled to learn similarly to animal and human brains. This is considered the prototype for modern artificial neural networks. After the introduction of artificial neural networks and machine learning many

studies have been conducted in several areas of research. One of these areas being neural machine translation which is the use of artificial neural networks to translate from one natural language to another.

Of importance here is the sequence-to-sequence model (seq2seq) which was put forward by Sutskever et al. (2014), which takes full sequences of the original language as input and encodes it into an embedded representation. It then decodes the embedded representation into a desired output language. Seq2seq models have widely been used for natural language processing such as translation, but have also seen uses for processing sequences of program code. The code2seq model takes the idea of processing code, but instead of directly using sequences of code as written, it uses an alternative approach which uses the structural paths through the code. Alon et al. (2018) claims that this creates better encodings of source code.

1.2 Problem formulation

When it comes to automated feedback there are several established techniques that are commonly used. All of these techniques have their advantages and disadvantages and have varying degrees of automation. Some of the most common approaches are *Unit testing, Peer-To-Peer feedback* and *Pattern matching*.

Unit testing builds on the use of test cases and test methods that detect if the software contains errors, produce the right outputs and follow the specifications given by the assignment. A major benefit of *Unit testing* is that it is fully automatic once it is implemented and can be reused however many times necessary. According to Aldriye et al. (2019), a downside with this approach is that every test needs to be implemented by hand. This can take a lot of time and effort on behalf of the instructor depending on how large the assignment is.

Peer-To-Peer Feedback is an approach that builds on the idea of students grading each other's answers. The benefits of this approach can be that students learn to identify errors and what the causes may be. However, there are multiple problems with this approach. First and foremost, it's not an automatic grading technique. It can be slow and the feedback runs the possibility of being incomplete or wrong since the limited knowledge of the students may be insufficient to give correct feedback in an acceptable amount of time. Aldriye et al. (2019) says that this feedback has the risk of creating more work for an instructor as it might need to be looked over.

Pattern Matching is an approach where the instructor provides the specification of the output that a correct assignment would generate. The system then requests another compiler that creates a program that verifies if the submitted solution meets the provided specification. According to Aldriye et al. (2019) there are many disadvantages with this technique since it only gives a grade to

perfectly matching submissions.

All these techniques take up precious resources and time from instructors and therefore it would be of interest to further automate the feedback process. Some machine learning models have been proposed to achieve this such as the models shown in Gupta et al. (2019) which provide feedback on syntactic errors using reinforcement learning techniques. Bhatia & Singh (2016) similarly finds syntactic errors using recurrent neural networks. Piech et al. (2015) proposes a model that provides feedback on style and functionality using recursive neural networks. As we can see there is interest in finding appropriate applications of different machine learning models to detect and provide feedback on different types of errors. Therefore it is relevant to explore already existing models that have been proven effective in other use cases, in new areas.
As such our problem can be formulated as follows.

Problem 1. *How can existing machine learning models be used for automated programming assignment feedback?*

1.3 Research Question

An important precursor to enabling the automatic feedback of programming assignments is the ability to detect errors that we want to provide feedback on. To explore our stated problem 1:*How can existing machine learning models be used for automated programming assignment feedback?*, this study will examine the machine learning model code2seq. Alon et al. (2018) has shown that Code2seq is effective at deriving natural language sentences from snippets of code in the tasks of predicting method names and generating code descriptions. To our knowledge, the model has not been tested in the application of detecting logical errors in code, and therefore it creates an area of application that is of interest to explore. To evaluate the effectiveness of code2seq at detecting logical errors we have chosen the following research question:

Research Question 1. *How well does code2seq perform at detecting logical errors?*

To answer the research question we will introduce the error we intend to detect into a public data-set of Java projects and train code2seq on this processed data-set.

As the variety of logical errors that can occur in beginner code is immense, we limit our logical errors to the most commonly occurring ones. Ettles et al. (2018) brings up several recurring errors. Among the most commonly occurring, as Ettles et al. (2018) puts forward, are those that occur due to misconceptions regarding indexing and initialization of arrays and the use of control flow using *if* and *else* statements. With this, our technical questions can be formulated as:

Technical Question 1. *How well does Code2Seq perform in detecting the logical error of using if-if-else flow structure instead of if-else if-else?*

Technical Question 2. *How well does Code2Seq perform in detecting the logical error of initializing the wrong start index in for-loop iterations of arrays?*

We will approach our questions through an *empirical experiment* where the same data-set used in code2seq original study is filtered to only contain methods that have the possibility of containing the logical error. Half of the examples in the data-set are edited to contain the logical error. This processed data-set will then be used to train and test code2seq's model and measured using the metrics: *accuracy, precision, recall* and *F1-score*.

1.4 Related Work

Multiple studies have explored the area of providing feedback or grading using different techniques and programs.
Singh et al. (2013) proposes a method for automated assignment evaluation by using a known correct solution and an error model consisting of known possible errors and corresponding corrections. The method uses constraint-based synthesis to compute the minimal amount of corrections necessary to improve a student's solution and was able to give feedback to 65% of the incorrect assignments.

Piech et al. (2015) presents a machine learning model based on Hoare Triples, which is a representation of a program before and after execution. These are traditionally used to prove the correctness of code as presented in Hoare (1969). Piech et al. (2015) creates embedded Hoare triple through an encoder-decoder model using a recursive neural network in order to propagate feedback on programming assignments. Their model showed effectiveness in providing feedback on the functionality, style and strategy of the student's work. In one of their experiments, they achieved 33% recall at 90% precision.

1.5 Results and Considerations

Our experiments achieved promising results with the highest *f1-score* at 90.92% in the *for-loop iteration* experiment. These results imply that code2seq can be well suited for detecting the types of logical errors included in our experiments. Code2seq shows promise to be implemented in a real-world setting and would be a beneficial automated feedback tool to be used in programming courses. This encourages further study of other possible applications of code2seq and similar machine learning models.

Chapter 2

Background

The idea of processing natural language to model grammar or to translate from one language to another has been studied since the 1950s and has slowly been developed over time. Liddy (2001) states that the general consensus is that Warren Weaver brought the idea of machine translation to general notice in 1949 and inspired many studies that furthered our understanding of natural language processing. In modern-day computer science, many techniques have been developed and are used in multiple areas such as AI or language translator programs. Alon et al. (2018) takes this concept but applies it to a programming language instead of a natural language. They do this by breaking down the code into an abstract syntax tree and using the paths between terminals of the tree as representations of code. Meaning is extracted from these representations by using the techniques of neural machine translation in their model code2seq.

2.1 Abstract Syntax trees

An abstract syntax tree (AST) is a representation of the structure and contents of a program. It is a rooted tree where each node contains a constituent part of the program, such as an if statement or a variable. By creating and examining an AST of a program we can study its structure, flow and function. As the name suggests ASTs are abstracted by representing the structure or content of a program that are logically meaningful. A node in an AST could for example be a while loop, if-statement or a variable assignment. See figure 2.1.

Neamtiu et al. (2005) suggests an implementation of using two different ASTs of the same program at different points in time in order to evaluate changes in the code over time. It is pointed out that ASTs provide a useful abstraction in analysis since low-level changes are ignored, such as changes in type or variable name (ibid.). Piech et al. (2015) makes use of ASTs generated from programs made by students to create Hoare triples for each sub-tree. This is captured by recording the state of the program before the active node is executed and

afterwards, so that the precondition is the state before execution, the instructions is the node that will be executed and the postcondition is the state after execution of the sub-tree.

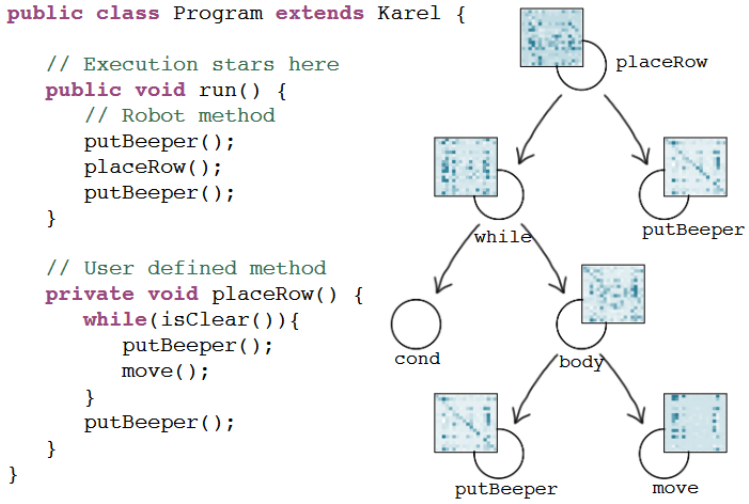

Figure 2.1: An Example of an abstract syntax tree Piech et al. (2015)

2.2 Machine learning

Machine learning (ML) is a field of artificial intelligence that is continually growing and has proliferated many areas of research and practical application. With the ever-growing availability of computational power and data, the use of machine learning to extract meaning from data has become prevalent in business settings and computer science research among others. According to Jordan & Mitchell (2015) machine learning seeks to answer two fundamental questions:

How can one construct computer systems that automatically improve through experience?

and

What are the fundamental statistical-computational-information-theoretic laws that govern all learning systems, including computers, humans and organizations?

They point out that for many use cases, such as natural language processing and computer vision, it is preferential to provide desired input-output behaviour rather than manually programming desired outputs for any predicted input. At the center of this, is what is called a learning problem, which Jordan & Mitchell

(2015) describes as the problem of improving some metric of performance when executing a task, through some type of training experience. This is typical in classification problems where a program has to assign a positive or negative label on some input such as purchase transactions or medical patient records, and the metric to be evaluated and improved could be the accuracy to which the algorithm correctly labels examples, meaning that the amount of correctly labelled examples is divided by the total amount of examples to be labelled. Jordan & Mitchell (2015) poses a few goals to theoretically find out the characteristics of an algorithm and the difficulty of the learning problem, of main interest are:

How accurately can the algorithm learn from a particular type and volume of training data?

and

how robust is the algorithm to errors in its modelling assumptions or to errors in the training data?

While there exist many types of machine learning algorithms, Jordan & Mitchell (2015) suggests three main paradigms that describe the differences in approaches that different algorithms take, these being: *Supervised learning, unsupervised learning* and *reinforcement learning.*

Supervised learning is an approach that uses labelled data-sets to train the algorithms to classify or predict the accurate result. This approach requires an outside agent that labels the input and outputs with the correct classification. Supervised learning can be separated into two types when data mining, regression and classification. Regression aims to understand the relationship between dependent and independent variables. For example, predicting numerical values based on different independent variables and determining the strength of the relationship. This can for example be used for weather analysis and prediction. Classification is the task of assigning data into specific classes that represent some aspect of what that data is or contains. For example, if an email is spam or not. In spam filters where the program lets the spam through at first and when the user classifies if the email is spam or not the program readjusts its algorithm to not make the same mistake in the future when similar emails are received.

Unsupervised learning on the other hand does not use labelled data, it instead uses machine learning algorithms to analyse and cluster data-sets to discover patterns without human intervention. According to Mahesh (2020), it does this by learning features of the data and using this learned information to recognize the class of the data. This is mainly used for clustering and feature reduction.

Reinforced learning uses intelligent agents that through feedback find their own way to solve a problem. This is done by training the program to make its own decisions in every state of the program and implementing the agent in such a way that it finds a balance between exploration of the unknown and exploitation of what it already knows. This is commonly used when developing AI.

2.2.1 Metrics

As previously mentioned, metrics play a major role in optimizing a classifier algorithm, whether that is a statistical algorithm like logistic regression or something machine learning based. The choice and modelling of the metrics that are used shape how data is interpreted and what conclusions can be drawn from it. According to Hossin & Sulaiman (2015), classification techniques fall into one of the categories: binary, multiclass or multi-label classification. Of most relevance here is binary classification which is what will be focused on in this paper. A metric is typically used in both the training stage and testing stage of a classifier. In the former, it is used to optimize the algorithm by being the measure that the algorithm uses to select between solutions. In the latter, it is used to evaluate the effectiveness of a classifier when exposed to unseen data (ibid.). When it comes to binary classification the evaluation of a metric can be described using a confusion matrix, which is a table where the row denotes the predicted labels and the column denotes the actual true labels, see figure 2.2. The confusion matrix gives us the true positives (TP) and true negatives (TN) which are the instances that have been labelled correctly, also false positives (FP) and false negatives (FN) which are the instances that have been labelled incorrectly (ibid.).

	Predicted Positive	Predicted Negative
Actual Positive	TP	FP
Actual Negative	FN	TN

Figure 2.2: A binary confusion matrix

Hossin & Sulaiman (2015) points out that the most commonly used metric in practice is accuracy for both binary and multi-class classification problems. This is complemented by the error rate metric which measures the amount of incorrect predictions divided by the total amount of predictions. Based on the confusion matrix we get the following that are discussed here and will be of importance to the results of this study:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

$$Error - rate = \frac{FP + FN}{TP + FP + TN + FN}$$

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

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

$$F1 - score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Accuracy and error rate have the benefits of being easy to compute and are easily understood by humans, but provide less distinctive values that make it harder to evaluate a model and discriminate an optimal solution. Depending on the goal it can be better to use other metrics, such as recall, which is the measure of positive examples that are correctly classified. For example, it can be used in cases where avoiding false negatives is preferable. Lastly Hossin & Sulaiman (2015) points out that it is imperative that the choice of metric is not done haphazardly, since the impact of any given metric is tangible on the solution that is reached by a classifier.

2.2.2 Artificial Neural network

An artificial neural network is a computing system that tries to emulate the problem-solving process of a human brain. According to Guresen & Kayakutlu (2011), this can be done by implementing artificial neurons that emulate neurons in the natural brain. These artificial neurons are connected by so-called edges and can transmit a signal to other neurons in the network. This signal is often a binary number (0 or 1), and usually has a weight that tells the next neuron how important this signal is to the result. Note that a neural network can modify these weights on on its own over time to get more desirable outputs (ibid.). The neurons are often implemented in layers, as can be seen in figure 2.3, that perform different transformations of the input to get to a final result. This however is not enough for an neural network to be functional for its intended use, it also needs to be trained. The training process can be done by supervised learning as explained in section: 2.2. The way a neural network learns is by adjusting the weight of the edges between the neurons to find what part of the problem has the most effect on the final classification.

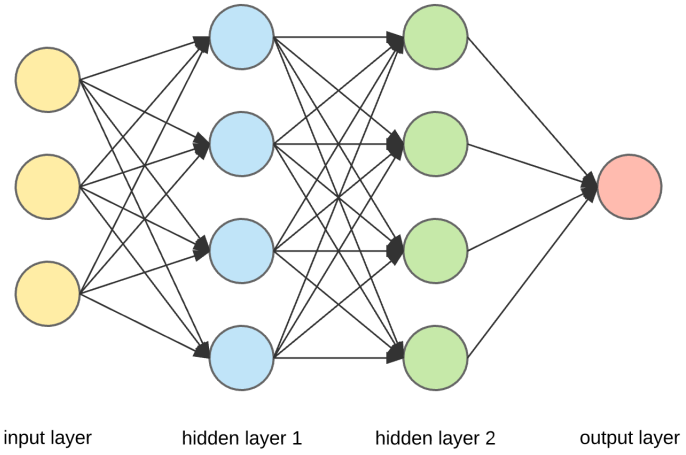

Figure 2.3: An Example of a recurrent neural network

According to Guresen & Kayakutlu (2011), a neural network is a directed graph but not all directed graphs are neural networks. Guresen and Kayakutlu argue that for a directed graph to be labelled as a neural network it needs to fulfill the following criteria:

- at least one start node (or Start Element; SE)
- at least one end node (or End Element; EE)
- at least one Processing Element (PE)
- all the nodes used must be Processing Elements (PEs), except start nodes and end nodes
- a state variable n_i associated with each node i
- a real valued weight w_{ki} associated with each link (ki) from node k to node i
- a real valued bias bi associated with each node i
- at least two of the multiple PEs connected in parallel
- a learning algorithm that helps to model the desired output for given input
- a flow on each link (ki) from node k to node i, that carries exactly the same flow which equals to n_k caused by the output of node k
- each start node is connected to at least one end node, and each end node is connected to at least one start node
- no parallel edges (each link (ki) from node k to node i is unique)

2.2.3 Neural Machine Translation

Neural Machine Translation uses an artificial neural network as an approach to machine translation to predict a sequence of words in a natural language. Bahdanau et al. (2014) say that this process is usually done by the use of a recurrent neural network as an encoder to encode a source sentence into a fixed-length vector to be run through a second recurrent neural network as a decoder that from that vector predicts the translation to the desired language. To maximize the probability of a correct translation, the encoder-decoder system is jointly trained (ibid.). According to Bahdanau et al. (2014) this *Encoder-Decoder framework* is set up such that the encoder reads the input sentence as a sequence of vectors $x = (x_1, ...x_{T_x})$ into a vector c^2 so that

$$h_t = f(x_t, h_{t-1})$$

and

$$c = q(\{h_1, ..., h_{T_x}\}$$

where $h_t \in R^n$ is a hidden state at time t, and c is a vector generated from the sequence of hidden states. f and q are non linear functions.
The decoder tries to predict the next word y_t given the vector c in combination with all the previous predicted words. To do this the decoder defines a probability over the translation y by decomposing the joint probability into the ordered conditionals:

$$p(y) = \Pi_{t=1}^{T} p(y_t|\{y_1, ..., y_{t-1}\}, c)$$

where $y = (y_1, ..., y_{T_y}$. With a recurrent neural network, each conditional probability is modelled as

$$p(y_t|\{y_1, ..., y_{t-1}\}, c = g(y_{t-1}, s_t, c)$$

where g is a nonlinear, potentially multi-layered function that outputs the probability ofy_t, and s_t is the hidden state of the recurrent neural network Bahdanau et al. (2014).

2.3 Code2Seq

The following section describes the code2sec model and its components with regard to the previously described concepts.

Code2seq is a machine learning model that generates natural language sequences from parts of code, that was put forward in Alon et al. (2018). The model leverages the syntactic structure of code to generate sequences from encodings of the compositional paths that make up a piece of code's AST. The model has shown success in two main tasks: Code summarization from Java code where a method name is derived from the code body, and code captioning where a descriptive natural language sentence is generated for C# code. See Figure 2.4.

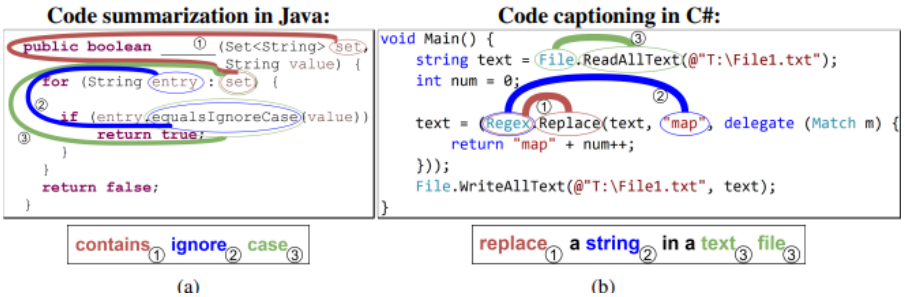

Figure 2.4: Examples of code and the corresponding output that code2seq generates for the task of code summarization and code captioning. Source: Alon et al. (2018)

According to Alon et al. (2018) the code2seq model represents a given code snippet by taking all pairwise paths between terminal nodes of the code snippet's AST and representing these as a sequence of terminal and non-terminal nodes. This has some benefit over directly taking a sequence of tokens as written in the code, also known as a seq2seq (sequence to sequence) approach. The result of this is that it makes it possible to find structural paths that are equivalent in pieces of code that are differently written on the surface. Take for example an iterative function that can be implemented with a for or while loop. Structurally these would be the same and would be captured as thus when using encoded AST paths, but not when using sequential token-based representations. See figure 2.5 as an example.

Code2seq uses an NMT encoder-decoder structure such as that used in Bahdanau et al. (2014) where k amounts of paths are sampled from any given AST, and encodes them as separate fixed-length vector representations. The average of all vectors that represent an AST, derived using mean pooling, is used as the starting state for the decoder. The decoder creates an output sequence by computing a context vector which is created by going over each vector representation. Since each path is encoded separately and their vectors are averaged, the decoder is not affected by the order that it is fed each input.

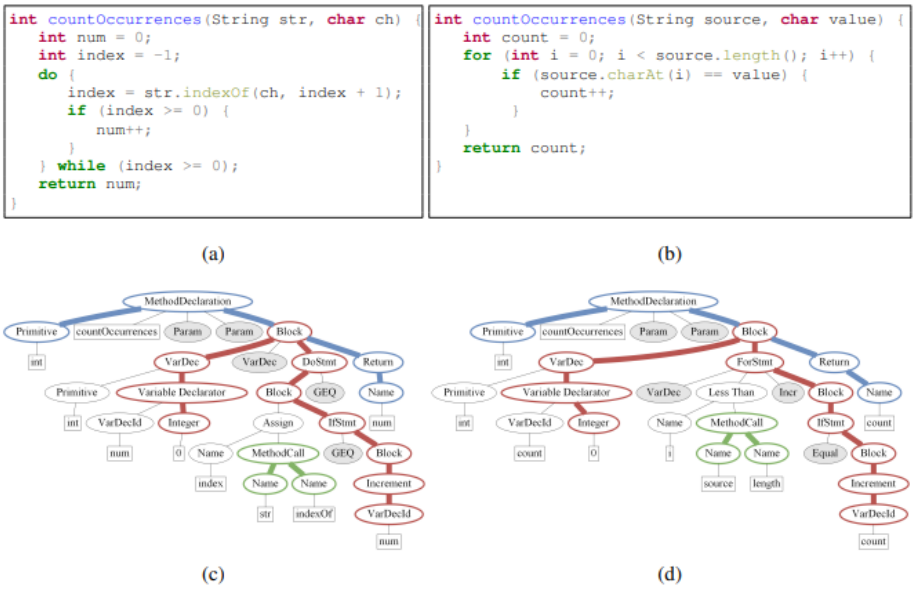

Figure 2.5: Example of two different implementations of the same method that are written differently, one using a do-while loop and one with a for-loop. These look different, but the normalized AST representation shows similar paths. Source: Alon et al. (2018)

Chapter 3

Methodology

3.1 Research Strategy

This study aims to explore how well code2seq performs in the novel application of detecting logical errors in code. More specifically this study will try to answer Research Question 1:

How well does code2seq perform at detecting logical errors?

To answer this question we will use an *empirical research* strategy *experiment* using *quantitative* data. (Johannesson & Perjons 2014, p 40-42) describe experiments as an empirical investigation of cause and effect relationships that aims to prove or disprove the causal relationship between a factor and an observed outcome. This relates to this study's goal of finding out how code2seq performs in the novel application of finding logical errors. Denscombe (2017) writes that experiments are well suited to quantitative research since the data that is produced from controlling variables and measuring the changes lends itself well to statistical analysis. As such this study's experiment will test the predictive performance of the code2seq model for logical errors, and how the choice of logical error affects the performance of the model.

To answer Technical Question 1:

How well does Code2Seq perform in detecting the logical error of using if-if-else flow structure instead of if-else if-else?

and Technical Question 2:

How well does Code2Seq perform in detecting the logical error of initializing the wrong start index in for-loop iterations of arrays?

Two experiments will be performed. Both will be based on the same data-set but each experiment will use a separate altered version to allow the model to be trained to predict their respective error. This creates two different data-sets with potential overlap.

3.2 Software Use and Implementation

For this study, a TensorFlow 2.1 implementation of the code2seq model proposed by Alon et al. (2018) is used. This is provided by `https://github.com/Kolkir/code2seq`. This version is used for compatibility reasons with available hardware. For the purpose of this study, there are two main parts of this software that are of interest. There is the code2seq model itself and the preprocessor for Java code, named JavaExtactor. This preprocessor parses through each method contained within the projects of the data-set, and converts them to separate abstract syntax trees. It then takes these and extracts the paths between terminal nodes of the AST. These are labelled with the methods name and aggregated within a file, note that the label does not have to be a method name for the model to work. This data can then be used to train and validate the code2seq model. The code2seq model predicts an output sequence by finding similarities between different ASTs by comparing their paths. To configure the model to perform a binary classification, of whether a method contains an error or not, we change it to only target a sequence as one part, instead of six. Before running the data-set through the JavaExtractor program, we need to filter out any ineligible methods, introduce the error we want to predict and label the methods accordingly.

In order to mutate the Java code within the data-set to contain the error we desire to predict, we implement our own preprocessor that filters out any ineligible methods. We introduce the error to approximately 50% of the eligible methods and change the method's name to reflect if the error is introduced or not. The method name is changed to a label that represents if the method contains the error or not, which the model will use for its predictions. To select eligible methods we apply a policy which discriminates between methods that have the possibility of containing the error or not. For example, any method that does not contain a for-loop that iterates over an array is removed from the Java file. We apply this policy to minimize potential bias in the data-set and to make sure that the predictions are based on the proper feature. It also makes it easier to detect and correct bias in the preprocessing. After finding all eligible methods, any Java file within the data-set that does not contain any methods is removed entirely for optimization purposes.

The altered data-set is then run through code2seq's preprocessor to be compatible with the machine learning model and is then used to train, validate and test the model.

3.3 Dataset Compilation / Collection / Synthesis

For this study, a data collection method is used which involves collecting existing *documents* in the form of Java files. We will use the java-large data-set that was put forward in Alon et al. (2018). This data-set consists of 9500 open-source Java projects from GitHub. The same distribution of projects for training, validation and testing as Alon et al. (2018) will be used. Initially, this data-set contains 17843598 methods in the training set, 461623 in the test set and 357136 in the validation set. This provided approximately 18m methods in total, which will be reduced when we perform our alterations to the data-set. A potential concern with this data-set is that it consists of real-world code and not code related to student assignments. However, we reason that real-world code is often more complex than code made for programming assignments. We argue that if the model is able to predict errors in complex code accurately, it can reasonably be assumed to work for simple code as well. This minor concern is outweighed by the benefit of being able to acquire a large repository of code, which would be more difficult to acquire with student code.

For the purpose of this study, all data points that do not have the possibility of containing the error we want the model to predict are excluded in the preprocessing stage. For example, if a method does not contain a for-loop that iterates over an array, it will not be valuable for prediction since it can never contain the bug and would create *noise* and possible *bias*. Therefore it is removed from the data-set to reduce noise that could affect the results. Gupta & Gupta (2019) puts forward that the existence of noise within a data-set can lead to drastically worse predictive performance and less meaningful information from a machine learning model. This causes decreased classification accuracy and worse prediction results. Another point of concern that can affect the performance and bias of a trained model is *overfitting*, which Ying (2019) describes as an inevitable issue when training a machine learning model, but one that can be alleviated. *Overfitting* is a phenomenon with complex causes, such as a model being trained on noise within the data-set. *Overfitting* can lead a model to have good results during training but have difficulties to generalize on unseen data. To combat this we implement a randomization of which part in the method is changed if there are multiple places the error could possibly be inserted.

After removing ineligible methods for the *for-loop iteration* experiment, we are left with 112548 methods for training, 2777 for validation and 5450 for testing. For the *cascading if-statement* experiment, we are left with 202907 methods for training, 3573 for validation and 7357 for testing.

To be able to train the model we need to have examples of code that contain the error we desire to detect. Since the data is sourced from open-source GitHub projects, they can be assumed to be correct and must be edited to con-

tain the error. To introduce the logical errors into the data-set, we assume that the projects are initially logically correct and do not contain the type of error we are trying to predict. This is a reasonable assumption since the projects used are highly rated on GitHub and can therefore be assumed to have been vetted by other developers. With this as a baseline, the preprocessor will randomly select half of the data points and introduce the desired error into those methods. Half of the examples are selected to give the model ample examples of erroneous code from which to learn the identifiers of error-containing code. This is not an accurate representation of the distribution of correct and incorrect code in a real-world situation but as Batista et al. (2004) points out, a minority class can be difficult for a model to learn the pattern of from an imbalanced data-set. Therefore the data-set is modelled to have a balanced distribution of correct and incorrect code.

Due to not being able to fully guarantee that all the examples do not already contain the error, there is a possibility of labelling incorrect code as correct or vice versa. This is a possible source of *bias*. However, the risk of this occurring is small and should have a negligible effect on the results.

3.4 Data analysis

The data analysis for this study will be a *quantitative analysis* of *nominal data* using *descriptive statistics*. Johannesson & Perjons (2014) describes descriptive statistics as a form of data analysis for quantitatively describing a sample of data. The data analysis will be performed with the built-in evaluation function of code2seq. This function implements standard machine learning metrics calculations to evaluate how correctly the model labelled examples. Since the output of the model in this application is a set of predictions where a given function is predicted to contain or not contain an error, the metrics used help quantify the amount of true positives, false positives, true negatives and false negatives. As such the metrics that are used are *accuracy, precision, recall* and *f1-score* as explained in section 2.2.1.

Several metrics are used in order to reduce bias and get a clearer picture of the performance of the model as a whole. Individually each metric can give meaningful insight to the performance but not a full description. Therefore a combination of metrics is used to allow for more accurate insights. For example, precision and recall are a complement to each other since a high score in one of these metrics and low in the other means the model makes too liberal predictions or vice versa. This balance between *precision* and *recall* is measured by the *F1-score* which provides the harmonic means between the two metrics. The use of multiple metrics helps reduce the *bias* of each singular metric that may be present during data analysis.

As previously mentioned in Section: 2.2.1 the choice of metrics has an important role when training a classifier and there is a possibility for bias when using simpler metrics such as accuracy. In terms of introducing bias, the effect of metric choice here is not as impactful as how the data-set is modelled. Due to the aim of finding how code2seq performs in a new application, we choose to use the same metrics as Alon et al. (2018) in order to have comparable results.

3.5 Alternative Research Strategies

Case study could be a reasonable research strategy to evaluate the effectiveness of code2seq. Johannesson & Perjons (2014) describes case study as a strategy to investigate a phenomenon by deeply focusing on one or a few occurrences. By holistically looking at all factors of an occurrence in detail, a deeper understanding of why something happened is achieved. This could be applied to our research question. However, the nature of the results would instead focus on why the model produces a specific prediction for a given method. For example, one could examine the paths of the AST that the model used to make a certain prediction and analyze this to derive an understanding of the model. This would instead be a *qualitative* approach as it focuses on a small amount of in-depth examples and would lend itself to an *inductive analysis* which Johannesson & Perjons (2014) describes as the derivation of general statements from the analysis of specific cases and situations.

Another relevant research strategy is *action research*. This would mean testing and evaluating this application of the code2seq model in a real-world setting such as a programming course. According to Johannesson & Perjons (2014) action research aims to solve important issues people experience in their practices. The subject of study with this approach would be to observe the effects the implementation of the model would have on students learning and their results. Denscombe (2017) points out the cyclical nature of action research which would make such a study especially viable since this would allow for comparison between different rounds of a course when the model is implemented. This could use both *quantitative* data in the comparison of grade results and *qualitative* data by capturing the experience of the students in using the model through interviews or surveys.

3.6 Validity and Reliability

To ensure the validity of our study we use a large data-set that is trustworthy and contains code from several authors. This means it should consist of a representative variety of different code styles and should be free of logical errors so that when the logical error is introduced, it should be the only variable that affects the results. This ensures that we are training the model to predict logical errors and nothing else, which Heale & Twycross (2015) points out as a crucial

aspect of validity, being whether the instrument covers that which it claims to cover, in this case, logical errors.

Heale & Twycross (2015) also state that to achieve reliability you need stability and equivalence. This means that the experiment needs to be repeatable and yield comparable results across multiple iterations of the experiment. To achieve this, our experiment uses a pre-established distribution of training, validation and test data. This ensures that the preconditions of the experiment across different runs are similar and should yield comparable results. There is slight randomness introduced when altering the code to contain errors which might negatively affect reliability, however, this should be negligible and conversely has a positive effect on validity since it avoids introducing artificial patterns into the data-set.

3.7 Ethical Considerations

This study does not have any human participants as subjects of the experiments and no personal information is handled. All data is gathered from open-source repositories and freely given for anyone to use. As such no informed consent is necessary and the only information about the authors of the data is what is already provided by them publicly, no further data is collected. In addition to this, all programs are run locally and no data is uploaded to any place the authors of this paper do not control. Anonymity is also preserved since the processed data does not contain any information outside of the structure of the code. In terms of bias, there is a limited risk of human bias since the calculations of the results are formulaically generated and not interpreted by the researchers, and appropriate steps have been taken to ensure that the risk of bias within the result is reasonably minimized. There are no obvious applications of this research that could pose a risk of harm to any specific person or society as a whole. With this in mind, we deem that there are no reasonable harmful ethical considerations to be drawn from this research.

Chapter 4

Main Results

4.1 Technical Development

The technical development of this study consisted of three primary parts: introducing bugs into the data-set and labelling it, pre-processing the data-set to be compatible with the code2seq model and training, validating and testing code2seq on the data-set. Our experiments were done on a computer with an 8-core CPU, an NVIDIA RTX 2080 Super GPU and 32GB RAM using Ubuntu 18.04. For reproducibility purposes, our implementation is made publicly available at `https://github.com/kevinchappy/exarb`. A permanent link for our repository is also available at Software Heritage.

4.1.1 Data-set preparation

This step was done using the FileVisitor.java program. This program first reads the path to the root directory of the data-set and visits each Java file within. The path to the root directory is stored in the variable *path* and can be changed to any local directory that contains Java files or directories with Java files that should be altered. The program iterates over each file in parallel and enacts changes to each method as described in the methodology section 3.3. For each type of error, we use a different class that implements the ModifierVisitor interface from JavaParser. Important here is the part where it finds eligible methods that can contain the error. This is done by finding statements that match specific criteria for each error. Note that this alteration process is done in place and directly makes changes to the data. It is therefore recommended to perform this on a copy of the original data-set.

The following code filters out any method that does not contain the prerequisite for the *for-loop iteration error*. Crucially it checks if the for-loop uses a field access expression as its end condition.

```
for (Statement statement : method.getBody().
                get().getStatements()) {
    if (statement instanceof ForStmt) {
        ForStmt forStmt = (ForStmt) statement;
        if (forStmt.getCompare().isPresent()) {
            Expression expr = forStmt.getCompare().get();
            if (expr instanceof BinaryExpr) {
                BinaryExpr bExpr = (BinaryExpr) expr;
                if (forStmt.getInitialization().size()==1 &&
                    (bExpr.getRight().isFieldAccessExpr())){
                    list.add(forStmt);
                }
            }
        }
    }
}
```

Examples of an eligible and ineligible method during the data-set filtering process for the *for-loop iteration experiment* are shown below:

```
eligibleMethod(){
    for(int i = 0;
    i < arr.length; i++){
       //do something
    }
}
```

Code 4.1: Example of eligible for-loop method

```
ineligibleMethod(){
    for(int i = 0;
    i < 10; i++){
        //do something
    }
}
```

Code 4.2: Example of ineligible for-loop method

In the example Code 4.1 and Code 4.2 we can see that the former uses the field access expression *arr.length* to determine the end condition of i which indicates that it is likely to iterate over an array and is therefore eligible. Conversely the latter method uses an integer expression for its end condition, which makes it unlikely to iterate over an array and is therefore ineligible. This implementation does not consider the body of the for-loop.

The if-else control flow filtering determines eligibility with this code segment which checks if a method contains any if-statements and if that statement is a cascading if-statement. A cascading if-statement is an if-statement whose else-statement in of itself is an other if-statement. This means that any eligible method contains at least one else-if clause. If-statements that are cascading are considered eligible, non-cascading if-statements as well as methods that does not contain any if-statement are not.

```
for (Statement statement : methodBody) {
    if (statement.isIfStmt() &&
            statement.asIfStmt().hasCascadingIfStmt()) {
    return true;
    }
}
```

Examples of what an eligible and ineligible method during the data-set filtering process for the *cascading if-statement experiment* is shown below:

```
eligibleMethod(int cond){
    if(cond < 0){
        //do something
    }else if(cond < 10){
        //do something
    }else{
        //do something
    }
}
```

Code 4.3: Example of eligible if-else method

```
ineligibleMethod(int cond){

    if(cond < 0){
        //do something
    }else{
        //do something
    }

}
```

Code 4.4: Example of ineligible if-else method

In the examples Code 4.3 and 4.4 we can see that the former method contains an else-if clause and is therefore cascading and is considered eligible. The latter contains an if-statement but is not cascading as its else statement is not an instance of an if-statement. Therefore it is not eligible.

If the program finds that a method does not contain any eligible statements, it entirely deletes that method from the file. When the program finds an eligible method, it randomly selects to introduce the desired bug or to leave the method unchanged. If it does not introduce the bug it changes the method name to *noBug* and writes that to the file. If it decides to introduce the bug it randomly selects one of the eligible statements and alters it accordingly to contain the desired error.

The following code segment introduces the for-loop iteration error by accessing the initialization variable of the loop counter and creates a new integer variable that is higher than the original initialization by one. The original variable is then replaced by the new incremented variable. *for-loop iteration error*:

```
IntegerLiteralExpr initVal = varDec.getInitializer().get();
int newInitVal = initVal.asInt() + 1;
varDec.setInitializer(new IntegerLiteralExpr(
                String.valueOf(newInitVal)));
method.setName("bug");
```

After the alteration of the code is done and the for-loop iteration error is introduced the method will change in the way depicted in the before and after examples below:

```
forLoopMethod(){
    for(int i = 0;
    i < arr.length; i++){
       //do something
    }
}
```

Code 4.5: Before introduction of error in for-loop iteration experiment.

```
bug(){
    for(int i = 1;
    i < arr.length; i++){
       //do something
    }
}
```

Code 4.6: After introduction of error in for-loop iteration experiment.

As depicted in the examples above the initialization value in Code: 4.5 is initially zero and is incremented by one, as shown in Code: 4.6. The method name is also changed from *forLoopMethod* to *bug* to provide a label that represents if the code contains the desired bug. If the method is selected to not contain a bug, the method body is not changed as depicted above but the method name is changed to *nobug*.

For the *cascading if-statement experiment*, the following code segment is used to introduce the desired error. This is done by first finding the else-if clause that we want to change into an if clause. To understand the workings of this change we need to understand how Javaparser structures a cascading if-statement. As a cascading if-statement is structured as a linked list where the else statement is the next node in the list, and an else-if clause is where that node is itself an if-statement. As such we can change this else-if clause by removing its parent's reference to itself and adding the given if-statement to the main method body at the index after the root of the original cascading if-statement.

```
if (statement != null) {
    statement.removeElseStmt();
    body.add(index + 1, newRoot);
    m.setBody(method.getBody().get().setStatements(body));
}
```

The two code snippets below illustrate an example of what a given piece of code could look like before and after the alteration to contain the cascading if-statement error:

```
elseIfMethod(int cond){
    if(cond < 0){
        //do something
    }else if(cond < 10){
        //do something
    }else{
        //do something
    }
}
```

Code 4.7: Before introduction of error in cascading if-statement experiment

```
bug(int cond){
    if(cond < 0){
        //do something
    } if(cond < 10){
        //do something
    }else{
        //do something
    }
}
```

Code 4.8: After introduction of error in cascading if-statement experiment

The depiction above shows the alterations that are made. Firstly, the else-if clause in Code: 4.7 is replaced with an if clause with the exact same condition and body as shown in code 4.8. This means that the original if-statement is no longer cascading and the bug is introduced. Similarly, as with the *foor-loop iteration error*, we label the method according to whether a bug was introduced or not.

4.1.2 Code2Seq

The altered data-set is then run through code2seq's pipeline for pre-processing data and training the model on that data. First, the data is preprocessed in the JavaExtractor which extracts AST paths and labels them according to their method name. Note that the method name in our altered data-set represents if the given method contains a bug or not.
In order to preprocess the data-set we edit the *preprocess.sh* the preprocess.sh script using the instructions contained within, so that it points to the training, validation and testing sets we want to use for training. The script is then run using the following command:

bash preprocess.sh

This yields three files, one for each set, that contain each sample from our data-set. Each sample is a list of fields, where the first field is the method's target label, in our case *bug* or *nobug* and each subsequent field is a context that represents a path through the method's AST and contains three components that are separated with a comma. The first and third of these components are tokens that represent the start and end nodes of a path. These are separated into subtokens using |. The second component is the path that connects the tokens. For example, these can look as follows:

bug my|key,StringExression|MethodCall|Name,get|value
noBug my|key,StringExression|MethodCall|Name,get|value

With our dataset preprocessed, we can now train code2seq with it. First, we have to change some parameters in the file *config.py* within code2seq. In order to make code2seq perform binary classification we change the parameter MAX_TARGET_PARTS as below:

```
config.MAX_TARGET_PARTS = 6
```

to

```
config.MAX_TARGET_PARTS = 1
```

This was changed from 6 to 1 so that the output sequence consisted of one subtoken instead of many. To illustrate the change made with this parameter, this can look like:

$$Label = nobug$$

instead of

$$Label = no|bug$$

This means that the model is more likely to find patterns that predict if the error is present, rather than other irrelevant patterns.

Another parameter that needed to be kept in mind was BATCH_SIZE. This determines the amount of samples used in each cycle during training and the size of this is dependent on the capability of the GPU. The following line was set according to the VRAM of the GPU used during training:

```
config.BATCH_SIZE = 128
```

We used 128 samples for each batch with 8 GB of VRAM.

The code2seq model was then trained on the preprocessed data across several epochs, with the goal of improving at each iteration. The training was performed by calling the file *train.sh*. This was edited to point to the location of our training and validation sets. It was then run using the following command:

bash train.sh

After each epoch of training the model was evaluated on the validation set to see if any improvements were been made to the model, this was measured using *accuracy, precision, recall and F1-score*. When evaluating a prediction

is made for each method in the validation or test set depending on what step in the process we were. This set of predictions is used to calculate the above-mentioned metrics. The confusion matrix below illustrates the structure of the predictions.

	Predicted bug	Predicted nobug
Actual bug	true bug	false bug
Actual nobug	false nobug	true nobug

When no improvements were been detected for 10 iterations, the model was considered to be trained to completion and could be tested. Testing was done on the testing set and measured using the aforementioned metrics.

The evaluation was done by calling the *evaluate.sh* script. This script was edited to contain the paths to the trained model and test set. It was called using the following command:

```
bash evaluate.sh
```

This yielded the final results for the performance of the model when predicting over the test set. This process was done in the exact same way for each of our experiments.

4.2 Experimental Results

For our experiments, we performed the steps as laid out in section4.1 on the java-large data-set. We performed two different experiments that separately filtered and altered the java-large data-set using different policies and trained the code2seq model on these altered data-sets separately. The java-large data-set contains 9500 open-source Github projects which consist of 2092810 Java files that together contain 18662357 methods. For the preparation of the *for-loop iteration* experiment, we applied the filtering method described in 4.1.1 and we were left with 117665 methods in total, of which 57523 had the bug introduced and 60142 were left unchanged. In the preparation of the *cascading if-statement*experiment we applied the filtering method also described in 4.1.1. Following this we were left with 216006 in total, of which 107728 had the bug introduced and 108278 were left unchanged. In the table below we show the results of this step.

Experiment	**Total methods**	**Bugged**	**Unchanged**
`For-loop iteration`	117665	57523	60142
`Cascading if-statement`	216006	107728	108278

Table 4.1: Total amount of methods in the data-set per experiment and the distribution of methods containing bugs and unchanged methods.

The java-large data-set has a pre-configured distribution of projects divided into a training-set, validation-set and test-set. Originally This distribution comes to approximately the following distribution of methods in each set: 95.6% in the training-set, 1.9% in the validation set, and 2.5% in the test set.

In our *for-loop iteration* experiment we have 93.2% of methods in the training set, 2.3% om the validation set and 4.5% in the test set.
In the *cascading if-statement* experiment we have 94.9% of samples in the training set, 1.7% in the validation set and 3.4% in the test set. As the distribution of sets is done on a project basis, the variety of percentages is due to the original distribution of projects. The amount of eligible methods can vary between projects and lead to varying distributions after filtering. However, the variety should not be large enough to have a tangible effect on the model and therefore results. The total amount of methods in each set for both experiments is displayed in the table below:

Experiment	**Training set**	**Validation set**	**Test set**
`For-loop iteration`	112548	2777	5450
`Cascading if-statement`	202907	3573	7357

Table 4.2: Distribution of amounts of methods in the training-set, validation-set and test-set for each experiment.

When the model finished training, the test set was used to evaluate the performance of the model's predictive power for these logical errors. The evaluation of the model consists of letting it perform predictions on unseen methods and determine whether they are labelled as bug or nobug. The measurements used to evaluate this performance are *accuracy, precision, recall* and *f1-score*. Each of these measures different aspects of the performance.

Accuracy measures the amount of correct predictions of whether a method contains a bug or not against the total amount of predictions made.
Precision measures what fraction of predictions for a given label are actually that label.
Recall measures how many of all the methods that should have been predicted as the given label was predicted as such.
F1-score measures the trade-off between precision and recall using the harmonic mean between the two measurements.

With these metrics in mind, the two experiments performed yielded the following results as shown in the table below:

Experiment	**Accuracy**	**Precision**	**Recall**	**F1-score**
`For-loop iteration`	90.77%	91.96%	89.89%	90.92%
`Cascading if-statement`	80.73%	85.04%	78.50%	81.64%

Table 4.3: Final results for each experiment

With these results, we can see that the overall performance was quite good but varied in some aspects for each of the two experiments. The overall performance in the *for-loop iteration* experiment had better results than the *cascading if-statement* experiment which could indicate that code2seq is better at detecting this type of error. However, it could also be that the error is simpler and therefore easier for the model to predict. Since it may be that it is more common for these kinds of for-loops to be initialized with zero as starting index and the model only needs to look for initialization values that are not zero. This simple pattern might be easier for the model to learn. Whereas the patterns in the *cascading if-statement* experiment were possibly not as simple or rigid. However, we can see that the model is still well-suited for detecting such an error. Of interest is that the model achieved higher precision than recall in both experiments but particularly in the *cascading if-statement* experiment, which might indicate that it was on the conservative side when predicting this error. The F1-score for both experiments is high and indicates a good overall balance between recall and precision which in turn solidifies the good performance of the model.

Overall these are promising results for the application of code2seq to detect these kinds of logical errors.

Chapter 5

Conclusion

In this study, we explored the viability of using the code2seq machine learning model to detect logical errors within novice code. More specifically we have considered this through the Research Question 1.3:

How well does code2seq perform at detecting logical errors?

In order to answer the research question we implemented a program that filtered out any methods that did not contain the prerequisites to contain a specific type of error in our data-set, and introduced errors to approximately half of them. These were labelled accordingly. We performed two experiments to answer our technical questions: Technical Question 1:

How well does Code2Seq perform in detecting the logical error of using if-if-else flow structure instead of if-else if-else?

and Technical Question 2:

How well does Code2Seq perform in detecting the logical error of initializing the wrong start index in for-loop iterations of arrays? .

For our data-set we used a repository of existing Java code, originally consisting of 2092810 files which after filtering for each experiment separately consisted of 129411 files and 69592 files respectively. Code2seq was then separately trained on the corresponding data-set for each experiment and measured on its predictive performance using the metrics: *accuracy, precision, recall* and *f1-score*.

The most remarkable conclusion from our study is that the model got high scores even with the relatively small training set which implies that code2seq is well suited to be trained at finding logical errors such as those used in our experiments, as the models were able to learn relevant patterns with a limited data-set.

5.1 Related Work

When comparing our results to the related work mentioned in section 1.4 we can see that code2seq might potentially have matching viability for providing automated programming assignment feedback. At least when it comes to logical error detection. Singh et al. (2013) study introduced a model that could catch 65% of faulty assignments and give some specific feedback on how to correct the errors. They used a model-based feedback approach that checks the students' code by comparing it to a specification of a solution of the assignment. This approach works well in smaller assignments where the workload of providing such a specification doesn't require too much time. However the larger the assignment the greater the workload. This is an area where our implementation with code2seq is more flexible and has the potential to find logical errors in any assignment without the need to specify the inner workings of the assignment. However, our implementation would not provide any feedback on if the assignment itself is correct, only if it contains any logical errors. Therefore Singh et al. (2013) model is more suitable to give feedback on specific assignments where as our trained model is more appropriate for giving feedback on specific errors, not the assignment itself.

Similarly to Piech et al. (2015) our study shows promising results for these types of machine learning models to provide feedback for assignment submissions. However, a key difference between our trained model and theirs is that their model performs multi-classification across several different labels that represent feedback that might occur in a submission, whereas we performed binary classifications for specific logical errors. Our model performed similarly in the precision metric as we achieved up to 91.96% precision and they achieved 90% precision. At these similar precision scores, we got a recall score of 89.89% and Piech et al. (2015) got a recall of 33%. This might be due to the inherent complexity that comes with multi-classification which could force the model to be more conservative to uphold a high level of precision. In contrast, our models' usage of binary classification makes it simpler for the model to learn the pattern of a label since there is less noise. This is because the pattern that informs the prediction of one class in multi-classification is noise for the prediction of another class.

The model used by Piech et al. (2015) was trained to provide feedback for a specific assignment, whereas ours was trained to find general errors that are commonly found within beginner code. This means that their model can give more tailored feedback for the assignment it is trained on, such as functionality, style and strategy. Our trained model only gives feedback on the existence of specific logical errors in the code, but not the stylistic or strategic approach. Similarly to the approach in Singh et al. (2013), the approach in Piech et al. (2015) is more suitable to a specific assignment, whereas ours is trained on an error and can therefore be applied more generally on submissions for different assignments.

5.2 Delimitations

The delimitations of our work are that the logical errors that our models are trained on are errors that are confined within methods using simpler structures such as for-loops and cascading if-statements. As such the performance of code2seq for detecting more complex logical errors that can occur within multi-class systems and the interaction between different classes and methods was not captured. This delimitation is due to the goal of detecting errors in beginner code where the more complex logical errors are not as common as simpler ones. As such, the results of this study are only applicable for consideration when regarding the question of detecting simpler logical errors. Additionally, our results only show the effectiveness of code2seq when performing binary classification in this use case.

5.3 Ethical implications

The results of our experiments show promise for these kinds of machine learning models to detect simpler logical errors, as such there is a possibility for these to be applied in real-world settings, such as automated feedback tools for programming courses. One might argue that one dangerous possibility is that this could endanger the job safety of teachers. However we see this as a low-risk possibility and that these kinds of tools are not capable of entirely replacing teachers, but could rather help alleviate teacher workloads and let them focus on giving more qualitative feedback on the more complex problems the students might encounter. Conversely, there is a reasonable risk that using this kind of tool might give erroneous feedback which can result in students receiving misleading feedback. However, this should not be a problem as long as teachers do not over-rely on such a tool as it is not a replacement for assignment grading, but a tool that can flag possible issues.

5.4 Considerations for future work.

This study shows encouraging results towards the applicability of machine learning models to help provide feedback on programming assignments. As such, this study could be used as a base for future work. Such as exploring the potential for code2seq and other machine learning models in detecting other types of errors. This could include the detection and classification of multiple logical errors in one model or be done similarly to our study but testing predictive performance on more complex errors.

Another potential future study could evaluate the effectiveness of applying a similarly trained model in a real-world setting such as a programming course. Such a study could possibly evaluate the difference in the grade results. Another viable option is to see how it would affect the experience of students or how the tool affects the workload for the teachers. Due to the general nature of our implementation, this work could also be extended to applications in different areas such as program repair.

Bibliography

Aldriye, H., Alkhalaf, A. & Alkhalaf, M. (2019), 'Automated grading systems for programming assignments: A literature review', *International Journal of Advanced Computer Science and Applications* **10**(3).

Alon, U., Brody, S., Levy, O. & Yahav, E. (2018), 'code2seq: Generating sequences from structured representations of code', *arXiv preprint arXiv:1808.01400* .

Bahdanau, D., Cho, K. & Bengio, Y. (2014), 'Neural machine translation by jointly learning to align and translate', *arXiv preprint arXiv:1409.0473* .

Batista, G. E., Prati, R. C. & Monard, M. C. (2004), 'A study of the behavior of several methods for balancing machine learning training data', *ACM SIGKDD explorations newsletter* **6**(1), 20–29.

Bhatia, S. & Singh, R. (2016), 'Automated correction for syntax errors in programming assignments using recurrent neural networks', *arXiv preprint arXiv:1603.06129* .

Denscombe, M. (2017), *EBOOK: The good research guide: For small-scale social research projects*, McGraw-Hill Education (UK).

Ettles, A., Luxton-Reilly, A. & Denny, P. (2018), Common logic errors made by novice programmers, *in* 'Proceedings of the 20th Australasian Computing Education Conference', pp. 83–89.

Fradkov, A. L. (2020), 'Early history of machine learning', *IFAC-PapersOnLine* **53**(2), 1385–1390.

Gupta, R., Kanade, A. & Shevade, S. (2019), Deep reinforcement learning for syntactic error repair in student programs, *in* 'Proceedings of the AAAI Conference on Artificial Intelligence', Vol. 33, pp. 930–937.

Gupta, S. & Gupta, A. (2019), 'Dealing with noise problem in machine learning data-sets: A systematic review', *Procedia Computer Science* **161**, 466–474.

Guresen, E. & Kayakutlu, G. (2011), 'Definition of artificial neural networks with comparison to other networks', *Procedia Computer Science* **3**, 426–433.

Heale, R. & Twycross, A. (2015), 'Validity and reliability in quantitative studies', *Evidence-based nursing* **18**(3), 66–67.

Hoare, C. A. R. (1969), 'An axiomatic basis for computer programming', *Communications of the ACM* **12**(10), 576–580.

Hossin, M. & Sulaiman, M. N. (2015), 'A review on evaluation metrics for data classification evaluations', *International journal of data mining & knowledge management process* **5**(2), 1.

Johannesson, P. & Perjons, E. (2014), *An introduction to design science*, Vol. 10, Springer.

Jordan, M. I. & Mitchell, T. M. (2015), 'Machine learning: Trends, perspectives, and prospects', *Science* **349**(6245), 255–260.

Liddy, E. D. (2001), 'Natural language processing'.

Mahesh, B. (2020), 'Machine learning algorithms-a review', *International Journal of Science and Research (IJSR).[Internet]* **9**, 381–386.

Neamtiu, I., Foster, J. S. & Hicks, M. (2005), Understanding source code evolution using abstract syntax tree matching, *in* 'Proceedings of the 2005 international workshop on Mining software repositories', pp. 1–5.

Piech, C., Huang, J., Nguyen, A., Phulsuksombati, M., Sahami, M. & Guibas, L. (2015), Learning program embeddings to propagate feedback on student code, *in* 'International conference on machine Learning', PMLR, pp. 1093–1102.

Singh, R., Gulwani, S. & Solar-Lezama, A. (2013), Automated feedback generation for introductory programming assignments, *in* 'Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation', pp. 15–26.

Sutskever, I., Vinyals, O. & Le, Q. V. (2014), 'Sequence to sequence learning with neural networks', *Advances in neural information processing systems* **27**.

Ying, X. (2019), An overview of overfitting and its solutions, *in* 'Journal of physics: Conference series', Vol. 1168, IOP Publishing, p. 022022.