Chair of Software Engineering II

Scratch Bug Detection Using the N-gram Language Model

Bachelor Thesis by

Eva Gründinger

Advisor
Prof. Dr. Gordon Fraser

October 25, 2020

Contents

Contents ii

Abstract iv

List of Figures vi

List of Tables vii

1 Introduction 1

2 Background 4

2.1 Analysing Scratch Programs 4

2.1.1 The Structure of Scratch 4

2.1.2 The Static Scratch Code Analyser LitterBox 6

2.2 N-gram Language Model 7

2.2.1 Probability Calculation of N-grams 7

2.2.2 N-gram Model Configuration 8

3 N-gram Bug Detection in Scratch 10

3.1 Tokenization of Scratch Code 10

3.2 N-gram Model Building in Scratch 14

3.2.1 Calculating and Adding N-grams 14

3.2.2 Smoothing of Probability Distribution 15

3.3 Bug Detection in Scratch 16

3.3.1 Configurations 16

3.3.2 Pruning False Bugs 17

3.4 Implementation 18

3.4.1 Tokenizer 18

3.4.2 NGramTrainer 19

3.4.3 NGramBugFinder 19

4 Evaluation **20**
4.1 Data Sets and Models 20
4.1.1 Big Data Set 21
4.1.2 Pupils' Projects 21
4.2 RQ1: Comparison to Litterbox 25
4.3 RQ2: Violation Classification 25
4.4 RQ3: Open Solutions 28
4.5 RQ4: General vs Project-specific Model 29
4.6 Threats to Validity 30

5 Related Work **31**
5.1 Analysing Scratch Programs 31
5.2 Object Usage Anomalies 31
5.3 N-gram Language Models 32
5.4 Bugram 32

6 Conclusion **33**

Bibliography **36**

Eidesstattliche Erklärung **39**

Abstract

In contrast to rule-based methods which often rely on special patterns that appear rather frequently in source code, and detect violations based on the inferred rules, the n-gram language model is another approach to bug detection. In this bachelor's thesis, this new way of software bug detection is proposed in order to improve software reliability and the quality of Scratch programs. After tokenization, *token* sequences are assessed with their calculated probabilities which are based on the existing model. If a detected *token* sequence has a rather low probability, it gets reported as a potential bug because the assumption is that these kinds of sequential tokens are unusual and should be highlighted to the programmer as a bad practice or programming mistake that affects the program. The n-gram language model gets evaluated in the following ways. First, Scratch projects are analysed to find bugs, code smells or unusual use cases. Then the result is compared to the reported bugs by LitterBox which assesses the same projects. The n-gram language model reported sequences for each task although Whisker tests may have passed and no code smells were found by LitterBox which shows the different kind of violations n-gram models are able to detect. Second, detected bugs are categorized to identify if n-gram language models are suitable for bug detection in Scratch projects and if this implementation is able to compete with already existing approaches like LitterBox. The project-specific n-gram language model can detect extensions to an original task, which LitterBox is not able to do, as well as dead code and empty scripts/bodies, whereas unused variables or long scripts are not in its reported violations. Third, it is analysed if the n-gram language model is able to get valid results with a set of tasks that were openly completed. The analysis demonstrates that this approach of bug detection is even effective in circumstances where solutions of tasks are only fairly related to each other. The project-specific model is still able to identify potential bugs with the exception of the possibility to categorize unusual use cases due to missing reference solutions. Fourth, a comparison between a general n-gram language model and project-specific models showed that the general model

out of a big data set is not as useful than project-specific ones. For instance, unusual use cases cannot be detected because of missing reference solutions. To have exact numbers that demonstrate the accuracy of the estimations, further research is needed to give the calculated negative probabilities more meaning. The results suggest that the implementation this bachelor's thesis is referring to is complementary to the above mentioned methods of software analysis.

List of Figures

1.1 Example bug detection with n-gram language model 2

2.1 A representative Scratch script 5
2.2 Script code controlling a sprite 6
2.3 A sprite's reaction after an executed event 6

3.1 Overview of the n-gram model building process. 10
3.2 Example AST structure of a simple Scratch project. 10
3.3 Tokenization of a Scratch script 11
3.4 Scratch block sequences 14

4.1 Boxplot for distribution of reported sequences 23
4.2 Boxplot for distribution of reported sequences with probability thresholds 23
4.3 Histogram for probability distribution of *Fruit Catching* task 24
4.4 Histograms of *Monkey* and *Elephant* tasks 24
4.5 Histograms of *Cat* and *Horse* tasks 24
4.6 Example *script* for detected unusual use case in Scratch code 27
4.7 Example for detected bug and false positive in *Fruit Catching* project . . 27
4.8 Example for detected bug and false positive in open project 29

List of Tables

3.1 The from tokenization excluded information-based AST nodes 12
3.2 The from tokenization excluded Scratch blocks 13
3.3 The tool chain for n-gram bug detection in Scratch 18

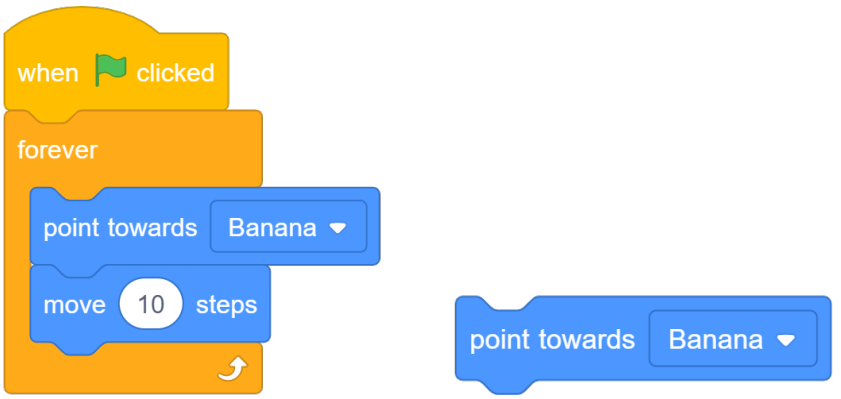
4.1 General model statistics 21
4.2 Project-specific model statistics 22
4.3 Representative projects of each task for bug detection 22
4.4 The number of reported bugs found by LitterBox and n-gram model . 25
4.5 The categorization of all reported bugs 26
4.6 An excerpt of the report for the *Fruit Catching* task 26
4.7 Open projects report 28
4.8 An excerpt of the report for the *Zauberei* task 28
4.9 General vs project-specific model 30

1 Introduction

As one of the most popular block-based programming languages Scratch [Mal+10] has established a growing community over the years that surrounds a database of almost 60 million created projects[1]. Because of the fun and easy programming environment that supports novice programmers particularly in their creative process, Scratch is preferred by students over text-based languages when it comes to studying about the world of programming. The drag-and-drop principle of puzzle pieces helps beginners to avoid common syntax mistakes and encourages children in a fun way to take control over the *stages* and *sprites* and create their own world and stories. Memorization instead of recall is an important part of the Scratch system because by picking your favourite blocks from a palette instead of having to know all commands right away is helping programming novices to get fast and viewable results.

Though Scratch makes programming seem easy, there are also many mistakes that programmers can make if they are unaware of common programming practices. The frustration of finding these bugs in a huge number of code blocks is even more apparent as a teacher who has to examine each project manually because of missing tools in the Scratch editor that usually are a part of standard programming frameworks, like Intellij[2] or Eclipse[3]. In order to support Scratch users and improve their code quality and accuracy, there are many researchers who found methods to detect bug patterns [Frä+20] as well as code smells [Var+19; MR14; Boe+13] or even implemented automated approaches for testing [SKF19] for the Scratch environment. As a result, tools like Hairball [Boe+13], Dr. Scratch [MRR15] or LitterBox [Frä+20] were created to further the analysis of Scratch programs and assist programmers in their bug detection process.

[1] `https://scratch.mit.edu/statistics/`, last accessed September 01, 2020
[2] `https://www.jetbrains.com/idea/`, last accessed September 01, 2020
[3] `https://www.eclipse.org/`, last accessed September 01, 2020



(a) First script of monkey sprite

(b) Second script of monkey sprite

Figure 1.1: Example bug detection with N-GRAM LANGUAGE MODEL

In order to improve code quality and reliability of programs, many rule-based techniques have been researched to detect bugs and code smells. However, these rule-based approaches are relying on highly frequent patterns in the code. Another approach is the usage of N-GRAM LANGUAGE MODELS for code analysis. This way of detecting defective code is already successfully applied by BUGRAM [Wan+16] on JAVA code. The assumption is that low probability *token* sequences are unusual, which may indicate bugs, bad practices or special uses of code.

The N-GRAM LANGUAGE MODEL works by assessing *token* sequences of programs by their probability in the learned model while low probability ones are marked as bugs. For bug detection the five main components that need to be configured for the model are: *Gram size*, *sequence length*, *reporting size*, maximum *probability threshold* and *minimum token occurrence*. After adjusting these settings, the n-gram Markov model is able to obtain the probabilities of all *token* sequences. The probability of each *token* in a sequence is only determined by its previous n - 1 tokens. Using a 3-gram model the probability of the sequence s is calculated like it is shown in the Equation 1.1. Then the language model ranks the outcome based on its probability in descending order and reports the sequences with the lowest probabilities as potential bugs.

$$P(s) = P(b_1) \cdot P(b_2 \mid b_1) \cdot P(b_3 \mid b_1 b_2) \cdot P(b_4 \mid b_2 b_3) \tag{1.1}$$

In this bachelor's thesis a technique that uses N-GRAM LANGUAGE MODELS to automatically detect bugs in SCRATCH programs is proposed. For instance, Figure 1.1 shows

SCRATCH scripts that were created as a solution for the *Monkey* task and control the monkey sprite in the project.

The block sequence [Never, GoToPos], shown in Subfigure 1.1b, can basically never be executed because of a missing *hat block*. Consequently, it is a rather unusual sequence and has only a probability of 0.12%, whereas the sequence [GreenFlag, RepeatForever-Stmt], which can be found as the first two blocks in Subfigure 1.1a, is a common use case with an occurrence probability of 7.18%. Therefore, the sequence [Never, GoToPos] is suspicious and indicates the existence of a code smell or bug.

This is an example of how to use the N-GRAM LANGUAGE MODEL to find low probability sequences and mark them as potential bugs that should be further analysed by the programmer. The accuracy of the probability calculation depends on many configuration parameters of the model and calculation methods like *smoothing* that will be explained further in this bachelor's thesis.

2 Background

As this thesis merges known concepts from Scratch analysis and n-gram language model, we introduce the concepts our work is based on in the following sections. Section 2.1 introduces Scratch and its functionality as a block-based programming language as well as the Scratch analyser LitterBox. Section 2.2 introduces the concept of a n-gram language model. It is necessary to understand the difference between n-grams and an n-gram model and the way of using it in order to obtain valuable information about Scratch code.

2.1 Analysing Scratch Programs

The functionality of Scratch is important to understand in order to follow the main research of this thesis. The following methods and analysis are based on the programming language Scratch and LitterBox which is the basis of the evaluation and used its AST as a guideline for the structure of the n-gram model.

2.1.1 The Structure of Scratch

Scratch[1] is a block-based programming language designed for kids that was created by the *Massachusetts Institute of Technology*[2]. With over 50 million registered users and shared projects[3] it is one of the most popular tools for teaching children from a young age how to develop computational thinking skills to solve problems programmatically.

[1] `https://en.scratch-wiki.info/wiki/Scratch`, last accessed August 19, 2020
[2] `https://web.mit.edu/`, last accessed September 06, 2020
[3] `https://scratch.mit.edu/statistics/`, last accessed August 19, 2020

The 'drag-and-drop-programming' style that SCRATCH is utilizing allows the programmer to pick from a pallet of existing blocks and build scripts by combining these code pieces like a jigsaw puzzle. *Blocks*[4] are separated into different types that include *hat, stack, reporter, boolean,* and *cap*. Each data type has a specific shape that indicates their intended usage in order to avoid errors in the syntax of the code. In contrast to text-based programming languages, the *blocks* help children to memorize the commands and avoid structure errors that otherwise would very likely occur in the beginning.

After combining different *blocks*, *scripts*[5] are created which resemble code methods in JAVA. *Scripts* are found in the code of the actors that perform the actions, called *sprites*[6] as well as the *stage*[7] which visualises the background of the created program.

Figure 2.1 shows the typical structure of a SCRATCH *script* that uses most of the before mentioned data types. Starting the *script* with a *hat* and ending with a *cap* block, there are also 10 different categories that each block is assigned to: *Motion, Looks, Sound, Event, Control, Sensing, Operators, Variables, List,* and *My Blocks*.

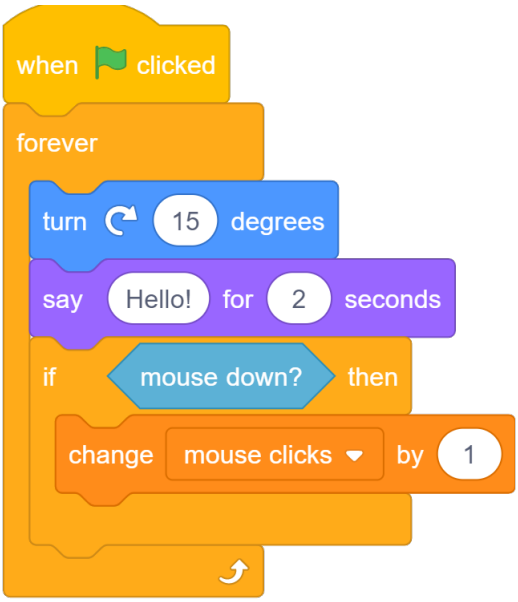


Figure 2.1: A representative script in SCRATCH.

For instance, Figure 2.2 of an example SCRATCH project[8] shows a *script* that controls

[4] `https://en.scratch-wiki.info/wiki/Blocks`, last accessed August 19, 2020
[5] `https://en.scratch-wiki.info/wiki/Script`, last accessed August 19, 2020
[6] `https://en.scratch-wiki.info/wiki/Sprite`, last accessed August 19, 2020
[7] `https://en.scratch-wiki.info/wiki/Stage`, last accessed August 19, 2020
[8] `https://scratch.mit.edu/projects/308718195/`, last accessed August 20, 2020

the picture of Jungkook, a member of the band BTS, which is shown in Figure 2.3. Once the user clicks on the picture of Jungkook, the 'click' event of the *script* is executed and the text "Saranghae <3" is shown for 2 seconds.

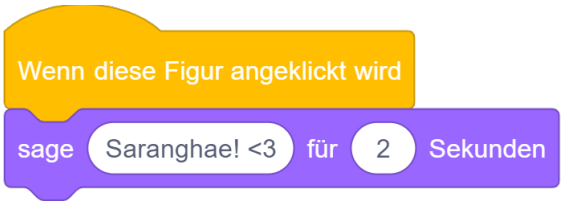


Figure 2.2: A SCRATCH script controlling a corresponding sprite

(a) Sprite in front of a background

(b) Sprite responding to an event

Figure 2.3: A SCRATCH stage and sprite that respond to events

2.1.2 The Static Scratch Code Analyser LitterBox

LITTERBOX[9] is a static code analysis tool for detecting bugs in SCRATCH projects where SCRATCH code is saved as a JSON file and downloaded, for example, using the SCRATCH REST API[10]. LITTERBOX then creates an abstract syntax tree (AST) for a SCRATCH project. The occurrence of bugs in code is mostly the consequence of recurring bad code writing habits. With the help of LITTERBOX these bug patterns and code smells can be filtered and analysed in order to correct the found bugs and avoid similar mistakes in the future. LITTERBOX is developed at the Chair of Software Engineering II and the Didactics of Informatics of the University of Passau and is the foundation for the analysis of SCRATCH programs with n-gram models in this bachelor's thesis.

[9] `https://gitlab.infosun.fim.uni-passau.de/se2/litterbox`, last accessed August 19, 2020
[10] `https://projects.scratch.mit.edu`, last accessed August 20, 2020

2.2 N-gram Language Model

Statistical language models, in its essence, are the type of models that assign probabilities to sequences of words. The N-GRAM LANGUAGE MODEL is the simplest model that assigns probabilities to sentences and sequences of words. You can think of an n-gram as the sequence of N words. A 2-gram or bigram is a two-word sequence of words like 'please read', 'read very', or 'very carefully', and a 3-gram or trigram is a three-word sequence of words like 'please read very', or 'read very carefully'.

2.2.1 Probability Calculation of N-grams

N-GRAM LANGUAGE MODELS usually consist of sentences s of words w that are all based on a specific dictionary D. Using *Markov chains* the language model calculates a probabilistic distribution of all possible sequences that can occur in the language based on D.

Definition 1 (Markov chain) *"A usually discrete stochastic process (such as a random walk) in which the probabilities of occurrence of various future states depend only on the present state of the system or on the immediately preceding state and not on the path by which the present state was achieved." [Mera]*

The probability of s is then calculated by partitioning it into its individual words w. But the probability of w is only dependent on the n - 1 previous words. Given a sentence $s = w_1 w_2 w_3 \cdots w_m$ the probability is estimated with the Formula 2.1 where $h_i = w_{1-n} \cdots w_i$ is the history sequence for the conditional probability of w_i:

$$P(s) = \prod_{i=1}^{m} P(w_i | h_{i-1}) \tag{2.1}$$

For instance in the case of a 3-gram, each word is dependent on the probability of the last $n - 1 = 3 - 1 = 2$ words. Therefore the probability of the sentence s = 'Would you like to go outside?' can be estimated with the Formula 2.2:

$$\begin{aligned} P(s) = P(Would) \cdot P(you|Would) \cdot P(like|Would\ you) \\ \cdot P(to|you\ like) \cdot P(go|like\ to) \cdot P(outside|to\ go) \end{aligned} \tag{2.2}$$

2.2.2 N-gram Model Configuration

There are five key factors that affect the number of bugs that a language model can find. These are the important *configuration parameters* that need to be set in order to achieve optimal results:

Definition 2 (Gram Size) *"The size of an n-gram model. Different gram sizes enable the model to use different internal probabilities to calculate the probabilities of token sequences." [Wan+16]*

The *gram size* n is an essential part of building an N-GRAM LANGUAGE MODEL and calculating the probabilities of a token sequence by utilizing n - 1 predecessors. Existing studies found that 3 to 6-gram models gave the best results, although the optimal *gram size* for bug detection is still unknown.

Definition 3 (Sequence Length) *"The length of token sequences to be considered when building n-gram models and detecting bugs. Different sequence lengths enable the model to capture different program scenarios and further affect the performance of the bug detection." [Wan+16]*

The *length* of analysed sequences of a project is essential to get a detailed analysis of the project's structure. The number of code blocks and scripts in a SCRATCH project can vary from 1 to a few hundred. By partitioning a program into smaller parts, more bugs can be obtained.

Definition 4 (Reporting Size) *"The number of sequences, in the bottom of the ranked list, which will be reported as bugs. An appropriate number may reduce the amount of false positives." [Wan+16]*

In contrast to the usual rule-based bug finding techniques which use a probability threshold to distinguish potential bugs from normal sequences, the N-GRAM LANGUAGE MODEL detects token sequences of low absolute probabilities. A larger number of bugs that get reported from the found list will definitely lead to a bigger size of found bugs but also means a higher risk of reporting sequences that are normal occurrences.

Definition 5 (Minimum Token Occurrence) *"The minimum number of times a token must occur in the code to be included in an n-gram model. An appropriate value helps to filter out token sequences that use unusual/special methods, thus have low probabilities but are not bugs." [Wan+16]*

Taking out tokens in a project that are not frequently used by using a *minimum token occurrence* is common practice for natural language processing (NLP) techniques. It reduces the chances of falsely reporting uncommon token sequences that are not bugs but very rarely used.

Definition 6 (Probability Threshold) *"If a token sequence does not reach this probability threshold, it can be considered a bug or unusual occurrence."*

After the analysis of a project, a number of sequences with the least calculated probabilities are getting reported as bugs. But in order to minimize the number of false positives in the bug set, it is common practice to determine a *probability threshold* that differentiates a bug from a rare occurrence of a sequence.

3 N-gram Bug Detection in Scratch

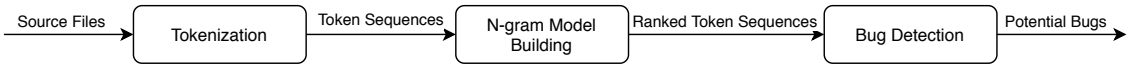


Figure 3.1: Overview of the n-gram model building process.

Figure 3.1 visualises the N-GRAM LANGUAGE MODEL building process which is explained in this chapter in more detail. In Section 3.1 the tokenization process is described with the way the SCRATCH project is parsed and converted into tokens. The next step is to use the created tokens to build the N-GRAM LANGUAGE MODEL like it is shown in Section 3.2. Finally, bugs can be detected with the help of the calculated model according to Section 3.3.

3.1 Tokenization of Scratch Code

Definition 7 (Token) *"A token is a single fragment of SCRATCH code that is used to partition code into smaller pieces in order to obtain information about its syntax on a specific granularity level."*

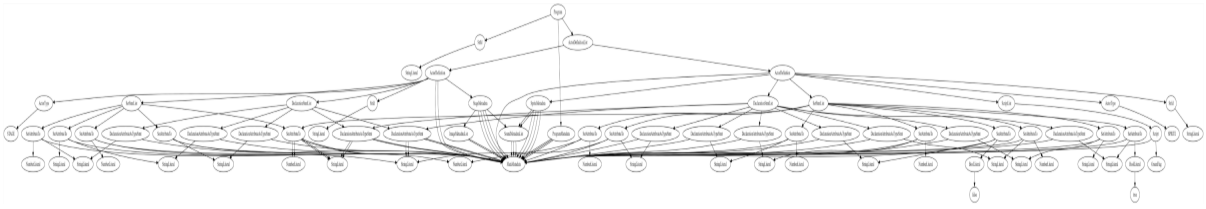
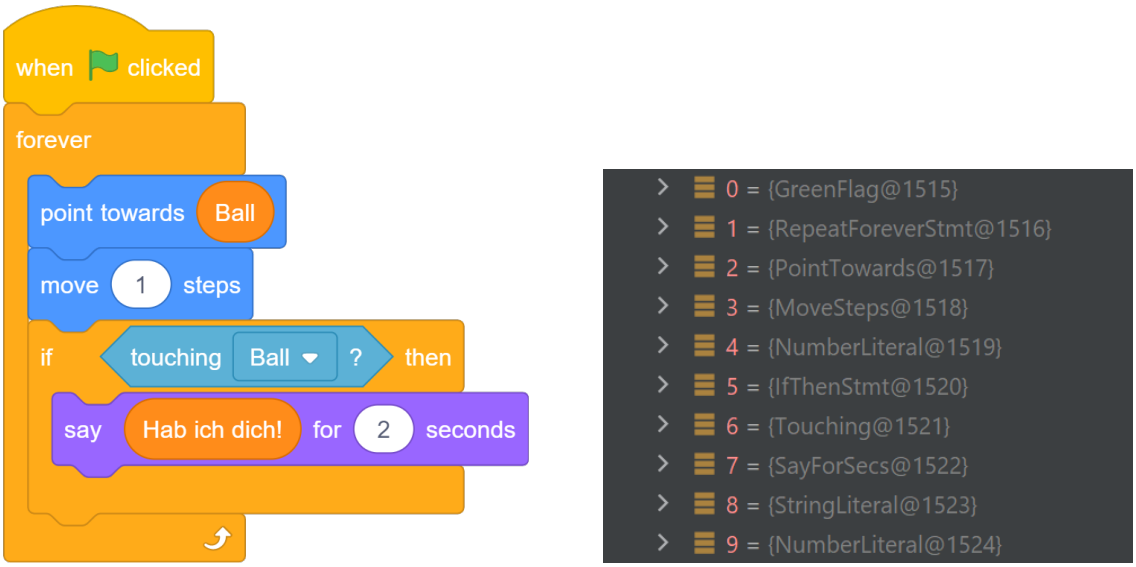


Figure 3.2: Example AST structure of a simple SCRATCH project.

In order to identify unusual block sequences in a SCRATCH project, each project should be represented in the same form and structure. To visualise a SCRATCH project like

it is shown in Figure 3.2, the tokenization process is based on the abstract syntax tree (AST) structure of LitterBox.

For a simple Scratch project with only one [GreenFlag] block, the tree structure is already pretty big. This is caused by the number of additional information nodes that are created by the LitterBox AST but are not necessary or even part of the essential tokens that are needed for an n-gram language model. In Table 3.1 is a quick overview of all AST-only nodes that are for information purposes, whereas in Table 3.2 are real Scratch blocks that exist in the AST that were excluded because of their usage in *drop-down lists* only. It is assumed that a programmer cannot choose falsely in a *drop-down list* and because the list itself can not be altered, all *drop-down boxes* are seen as one *token* with their corresponding Scratch blocks. These nodes are all excluded for the tokenization in order to keep the n-gram model as small as possible and increase bug detection efficiency.



(a) The script to tokenize.

(b) The tokens of the script.

Figure 3.3: Tokenization of a Scratch script.

As an example to show how a tokenized Scratch script looks like in the Tokenizer program, Figure 3.3 demonstrates how the partitioning into tokens works. While Subfigure 3.3a visualises a solution script that was created for the *Cat* task, the screenshot in Subfigure 3.3b is a picture of the calculated tokens which do not always equal the original Scratch blocks because of their foundation in the LitterBox AST.

Table 3.1: The excluded information-based AST nodes for tokenization of Scratch projects.

Class	Type	Example
`ActorDefinition`	`AbstractNode`	Information storage of actor
`ActorDefinitionList`	`AbstractNode`	List of `ActorDefinition`
`ActorType`	`ASTLeaf`	`STAGE, SPRITE`
`DeclarationStmt`	`Stmt`	All types of attribute declarations
`Metadata`	`ASTNode`	Additional information of the `Program, Sprites, Stages, Resources, Blocks`
`ProcedureDefinition`	`AbstractNode`	Definition of custom procedures
`ProcedureDefinitionList`	`AbstractNode`	List of `ProcedureDefinition`
`Program`	`AbstractNode`	Placeholder for a project
`Script`	`AbstractNode`	`Event` and `StmtList` of an actor
`ScriptList`	`AbstractNode`	List of `Script`
`SetAttributeTo`	`AbstractNode`	Sets attributes at start of program
`SetStmtList`	`AbstractNode`	List of attributes to set
`StmtList`	`AbstractNode`	List of `Stmt`
`StrId`	`LocalIdentifier`	Project ID

Table 3.2: The excluded Scratch blocks for tokenization.

Class	Type	Examples
Position	ASTNode	MousePos, RandomPos
Backdrop	AbstractNode	Backdrop1, Backdrop2,...
Costume	AbstractNode	Costume1, Costume2,...
DragMode	ASTLeaf	Not_DRAGGABLE, DRAGGABLE
ElementChoice	ASTNode	Next, Prev, Random, WithExpr
ForwardBackwardChoice	ASTLeaf	FORWARD, BACKWARD
GraphicEffect	ASTLeaf	COLOR, GHOST, BRIGHTNESS,...
Key	AbstractNode	Spacebar, Arrow keys, Any,...
LayerChoice	ASTLeaf	FRONT, BACK
Loudness	NumExpr, ASTLeaf	Integer
Message	AbstractNode	Message1, Message2,...
NumFunct	ASTNode, ASTLeaf	ABS, ACOS, ASIN, ATAN,...
Rotation-Style	ASTLeaf	Don't rotate, Left-right, All around
Size	NumExpr, ASTLeaf	Integer
SoundEffect	ASTLeaf	PAN, PITCH
TimeComp	ASTNode, ASTLeaf	DATE, DAY_OF_WEEK, HOUR,...
Timer	NumExpr, ASTLeaf	Integer
Touchable	ASTNode	Color, MousePointer, SpriteTouchable, Edge
Variable	DataExpr	My variable
Volume	NumExpr, ASTLeaf	Integer

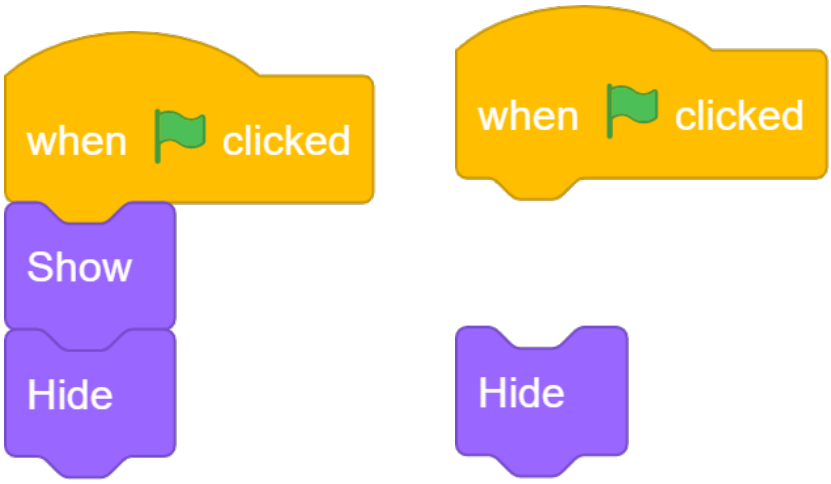
3.2 N-gram Model Building in Scratch

In the following sections the focus is on the building process of the n-gram language model, specifically in Scratch. Subsection 3.2.1 goes in detail on how to calculate the probabilities of the found *token* sequences and add them to the growing model. The method of *Smoothing* is then explained in Subsection 3.2.2 as well as its importance in bug detection.

3.2.1 Calculating and Adding N-grams

First of all, the language model has to extract all possible *token* sequences in a Scratch project and calculate their probabilities like it is described in Subsection 3.2.1. This way a reliable probability distribution is created that will be the basis for later bug detection.

For example, given the sequence [GreenFlag, Show, Hide] that is shown in Figure 3.4a, all its consecutive subsequences are added to the model. In this case the subsequence [GreenFlag, Hide] that is shown in Figure 3.4b would be the only sequence that is ignored because [Hide] does not immediately follow [GreenFlag].



(a) A Scratch block sequence. (b) Ignored subsequence.

Figure 3.4: Scratch block sequences.

If Figure 3.4a would be the sequence we want to calculate the probability of, the estimation based on the *Markov chain* is executed by solving the Equation 3.1. We are setting the *gram size* to 3 in this example. All internal probabilities are added by consulting the

existing model and searching for the already calculated estimations. Therefore, the token sequence [GreenFlag, Show, Hide] can be calculated based on internal probabilities of its existing subsequences.

$$P([GreenFlag, Show, Hide]) = P([GreenFlag]) \cdot P([Show]|[GreenFlag]) \cdot P([Hide]|[GreenFlag, Show]) \tag{3.1}$$

3.2.2 Smoothing of Probability Distribution

Definition 8 (Smoothing) *"In mathematics 'smoothing' means to free a graph, a collection of data, etc. from irregularities." [Merb]*

If the analysed project is not part of the training data set, it is important to smooth the probability distribution in order to avoid probabilities of zero. In this implementation, *Add-One-Smoothing* was utilized which prevents non existent sequences to be shown by adding an additional count to each n-gram. All the counts that used to be zero will now have a count of 1, the counts of 1 will be 2, and so on. This algorithm is also called *Laplace Smoothing*. This way there are no sequences with a probability of zero stored in the model that could affect the calculation of the sequence probabilities.

In the case of the unigram [GreenFlag], its maximum likelihood to appear in a Scratch project is estimated by counting the occurrences c during the model training process and divide it through the total number of tokens N. Therefore, the Equation 3.2 looks like this:

$$P([GreenFlag]) = \frac{c}{N} \tag{3.2}$$

Add-One-Smoothing then, like the name suggests, only adds one to each count. Since there are a total of D tokens in the model vocabulary and we added one more sighting to each one of them, we also have to adjust the denominator of the fraction accordingly. One more observation of each *token* means we have to increase the denominator by D. If we apply the mathmatic rules accordingly, the equation results in the following calculation in Equation 3.3:

$$P_{Laplace}([GreenFlag]) = \frac{c+1}{N+D} \tag{3.3}$$

3.3 Bug Detection in Scratch

The specific algorithm to find bugs in Scratch is implemented the following way. At first, the probabilities of all sequences of the analysed project have to be calculated with help of the model, like it is described in Subsection 3.2.1. After that, the sequences are ranked based on their probability and only the ones with the lowest probability get reported as potential bugs. In the next Subsection 3.3.1 all important parameters are set to ensure the optimal model analysis results. In Subsection 3.3.2 we found a method to minimize the number of false positives in the reported bug set.

3.3.1 Configurations

For the Scratch model implementation we added five parameters that have to be set accordingly in order to achieve the best evaluation results that are discussed later in Chapter 4.

Gram Size. For probability calculations a *Markov chain* is used to get the conditional probability of a sequence using its n - 1 *token* predecessors as context information. In this work, the *gram size* is adopted from the Bugram [Wan+16] paper where the value of 3 had the best results.

Sequence Length. In order to analyse Scratch projects on a rather small granularity level, we utilized *sequence lengths* of 2 and 3 in our implementation after analysing the results of multiple combinations of *sequence lengths*. Together with the *gram size* of 3, these *sequence lengths* are effective in partitioning the Scratch programs into pieces that give back useful probabilities. The usage of two different *sequence lengths* also ensures that the reported bugs are at the bottom of at least two analysing n-gram models. This method reduces the number of false positives that would otherwise occur more frequently.

Reporting Size. We did not limit the *reporting size* for the small data set that is further introduced in Subsection 4.1.2. The number of reported bugs is very small in the case of the data set because the projects as well as the model itself are not very big. If we would have restricted the number of reported bugs even more, there would not be enough output created to analyse.

Minimum Token Occurrence. In this bachelor's thesis the *minimum token occurrence* is chosen as 1. Because of the size difference between usual JAVA projects compared to normal SCRATCH projects, it would not be helpful to filter out any tokens. SCRATCH programmers do not have as much freedom in their implementation because of the restrictions of a block-based language and usually choose a different purpose for their programs than text-based programming languages would, which leads to much smaller projects with fewer usages of specific tokens. So, the *minimum token occurrence* parameter would only make it harder to find low probability sequences in projects which is why we decided not to raise this threshold for the analysis.

Probability Threshold. The *probability threshold* is determined individually for each N-GRAM LANGUAGE MODEL by examining the reported sequences of a few example projects, categorizing the true bugs and choosing a probability value that is low enough to detect the most true bugs with a minimum number of false positives. Every sequence with a higher probability is then considered a normal sequence that can not be reported as a bug because its occurrence rate is too high. In this implementation, the threshold is fixed at 0.05% for the big data set of Subsection 4.1.1 and at 0.5%, 0.6% or 1.6% for the pupils' projects of Subsection 4.1.2.

3.3.2 Pruning False Bugs

Token sequences with low probabilities are at the bottom of an n-gram reporting list for potential bugs. But there is always the chance that the found sequence is not actually wrong but just a very unusual or special use case in which its probability would also rank rather low. To find and filter out this kind of false bugs and reduce the candidate bug set, *token* sequences can only be reported when they are at the bottom of at least two ranked lists of N-GRAM LANGUAGE MODELS with the same *gram size* but different

sequence lengths. This way, sequences are sorted out that just appear on one of these lists and the chance to report false positives gets drastically reduced.

3.4 Implementation

This section explains the tool chain of our approach for n-gram language model in Scratch and clarifies implementation decisions. Table 3.3 gives an overview over the model building process in Scratch, the bug detection system, the parameters which can be adjusted, and the input and output of each step.

Table 3.3: The tool chain for n-gram bug detection in Scratch.

Steps	Input	Tool	Output format	Parameters
Model Building	Scratch data set, `model.csv`	NGram-Trainer, Tokenizer	`.csv`	gram size
Bug Detection	Scratch bug set, `model.csv`, `report.csv`	NGram-BugFinder, Tokenizer	`.csv`	report size, gram size, sequence length, probability threshold, with/without smoothing

3.4.1 Tokenizer

The basis of the n-gram building process is to tokenize each project and structure it in the same way. The NGramTrainer as well as NGramBugFinder both use LitterBox to create an AST for a given program. The AST then maps every command block to a corresponding AST node. In this process every *hat block* is implementing the interface `Event` and all remaining *command blocks* are associated with the `Stmt` interface. For instance, the [Turn Right] block is represented by an object of the `TurnRight` class which implements the `Stmt` interface. A visitor then visits the AST and partitions the project into its tokens.

3.4.2 NGramTrainer

NGramTrainer, a feature added to LitterBox, creates an n-gram language model based on a set of Scratch projects. The model stores a probability distribution for all *token* sequences that were found in the training set. It is then utilized as a foundation for bug detection in Scratch.

The model creation process consists of three steps: At first, all possible sequences of tokens that can be created from the analysed training set are calculated, counted, and stored in a `Map` structure. Then we go through the `Map` of sequences and estimate the probability of each sequence, like it is shown in Equation 3.2, by counting the amount a sequence appeared during the n-gram calculation and dividing it by the total number of sequences. After the calculation a finished model is printed as a `.csv` file and is ready for its usage in further analysis and bug detection procedures.

3.4.3 NGramBugFinder

NGramBugFinder, another extension that was added to LitterBox, continues the analysis process of the NGramTrainer by using its calculated model. First, to find bugs in a project, the project itself is partitioned into sequences of a specific length that are stored in a `Map`. Then the probability of each sequence is estimated with help of the given probability distribution of the n-gram model. The probabilities are then ranked and the sequences with the lowest scores are reported as potential bugs. The final `.csv` file then shows the reported sequences with their location information and calculated probabilities.

4 Evaluation

The main research points for this bachelor's thesis are answers to the following questions:

RQ1 How effective are n-gram language models for bug detection in comparison to LitterBox?

RQ2 What kinds of violations were detected by the n-gram language model?

RQ3 Does the n-gram language model work in the case of open task solutions?

RQ4 Efficiency of general n-gram language model vs project-specific model?

In the following analysis, parameters for the n-gram language model configuration were determined, like it is described in Subsection 3.3.1. The *gram size* of 3 is adopted from the Bugram [Wan+16] paper. For the *sequence length* we got the best results with the lengths 2 and 3. Furthermore, the *probability threshold* was manually estimated for each n-gram language model. The value that distinguished the most true bugs from false positives was chosen.

4.1 Data Sets and Models

RQ1, RQ2 and RQ3 are all answered by using project-specific models (Subsection 4.1.2) and analysing representative projects of different pupil tasks. In RQ4 the general model (Subsection 4.1.1) which is based on a bigger data set is compared to the reports of the project-specific models.

4.1.1 Big Data Set

In order to have a sufficient number of sequences to calculate a probability distribution from and as comparison to the smaller project-specific models, we decided to build an n-gram language model on a large data set with its statistics shown in Table 4.1. The dataset consists of 75,277 Scratch projects. From December 2019 to January 2020 we downloaded the most recent projects with the Scratch REST API[1]. We did exclude remixes from the data set. LitterBox could not parse 114 projects, thus the n-gram language model was created of 74,914 projects without any exceptions and consists of 138,298 calculated n-grams. The creation of the model took 17 days and was conducted on machines equipped with Intel Xeon E5-2650 v2 @ 2.60 GHz CPUs with 256 GiB of RAM.

Table 4.1: General model statistics.

File	Size	#Projects	#Ngrams
`model.csv`	7,865 KB	74,914	138,298

4.1.2 Pupils' Projects

The creation of the project-specific models as well as bug detection with the pupils' projects was done in a few seconds for every set of solutions and did not throw any exceptions. All experiments with the small data sets were conducted on a Swift SF314-57 with an Intel i5 core and 8 GB RAM.

Project-specific Models. The project-specific models with further statistics shown in Table 4.2 are build on a set of correct as well as defective solutions of five small coding tasks for students. Based on the task the pupils had to implement, we call these sets *Monkey*, *Elephant*, *Cat*, *Horse* and *Fruit Catching* task. We used solutions of pupils which originate in primary programming education [GFH16] for the *Monkey*, *Elephant*, *Cat* and Horse tasks. For the *Fruit Catching* task we used the same data set as Stahlbauer et al. in their work about testing Scratch programs automatically [SKF19]. In addition, a set of open task solutions that pupils could experiment with was used for Section 4.4.

[1] `https://github.com/LLK/scratch-rest-api/wiki`, last accessed May 8, 2020

Table 4.2: Project-specific model statistics.

Task	File	Size	#Projects	#Ngrams
Fruit Catching	`fruit_model.csv`	226 KB	42	3,257
Monkey	`monkey_model.csv`	33 KB	120	477
Elephant	`elephant_mode.csv`	39 KB	130	559
Cat	`cat_model.csv`	49 KB	129	715
Horse	`horse_model.csv`	30 KB	73	440
Open	`open_model.csv`	101 KB	295	1,897

Bug Detection. For bug detection we chose one representative project of each task which are listed in Table 4.3 and analysed them with LitterBox as well as the created n-gram models for the research questions.

Table 4.3: Representative projects for each task used for bug detection.

Task	Project	Size	#Sprites	#Blocks
Fruit Catching	`K7_S02.sb3`	176 KB	3.0	81.0
Monkey	`ID_065_Aufgabe-Affenjagd.sb3`	181 KB	3.0	18.0
Elephant	`ID_118_Aufgabe-Elefant.sb3`	456 KB	4.0	23.0
Cat	`ID_005_Aufgabe-Katze.sb3`	86 KB	2.0	14.0
Horse	`ID_080_Aufgabe-Pferd.sb3`	6 KB	1.0	14.0
Open	`m043.sb3`	304 KB	3.0	26.0
Open	`m052-Zauberei.sb3`	869 KB	7.0	18.0
Open	`m067-Apfelsuche.sb3`	330 KB	3.0	31.0
Open	`m074-Sternenjaeger.sb3`	58 KB	12.0	65.0
Open	`m091 2.sb3`	53 KB	7.0	70.0
Open	`w002-Weltraumangriff.sb3`	703 KB	7.0	57.0

Project-specific Model Reports. The created boxplot in Figure 4.1 visualises a distribution of the reported sequences for all pupil projects. We can see that each task has a different number of sequences that were detected as potential bugs with the *Fruit Catching* task being the one with the most reported sequences. The huge difference between the *Fruit Catching* task and the rest of the tasks is probably related to its big block counts in the projects which ultimately leads to a higher number of reported sequences.

As a comparison to Figure 4.1, Figure 4.2 shows a distribution of all reported sequences that have an occurrence probability below the estimated *probability thresholds* and therefore are more likely to be true bugs or unusual use cases than false positives.

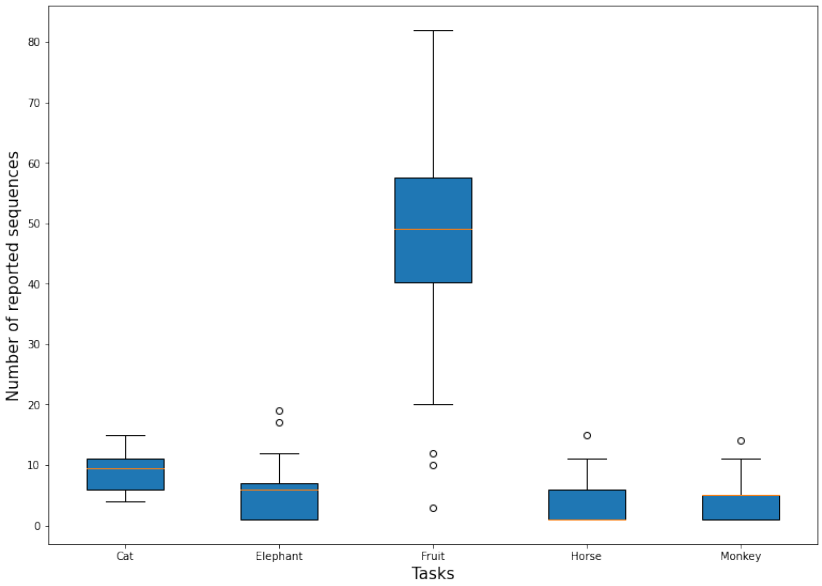


Figure 4.1: Boxplot for distribution of reported sequences.

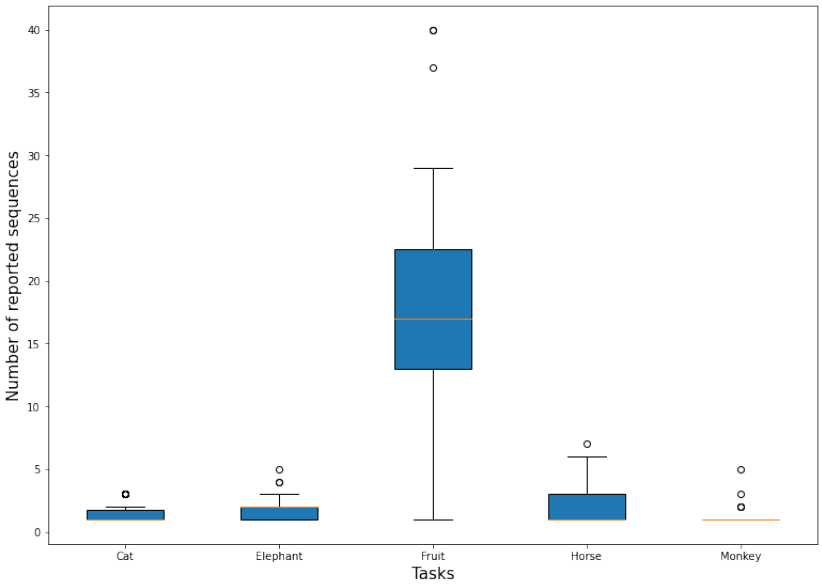


Figure 4.2: Boxplot for distribution of reported sequences with probability thresholds.

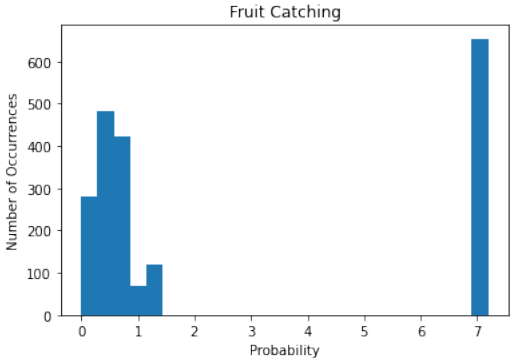


Figure 4.3: Histogram of *Fruit Catching* task with a *probability threshold* of 0.6%.

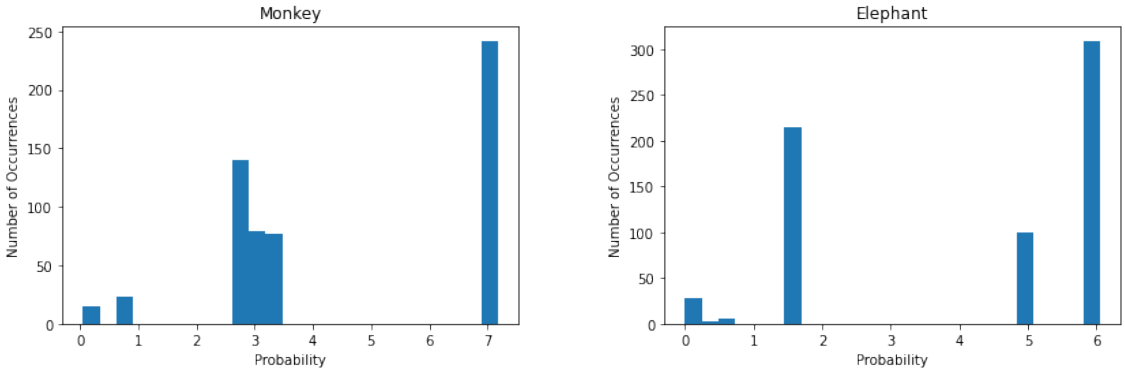


Figure 4.4: Histograms of *Monkey* and *Elephant* tasks with a *probability threshold* of 1.6%.

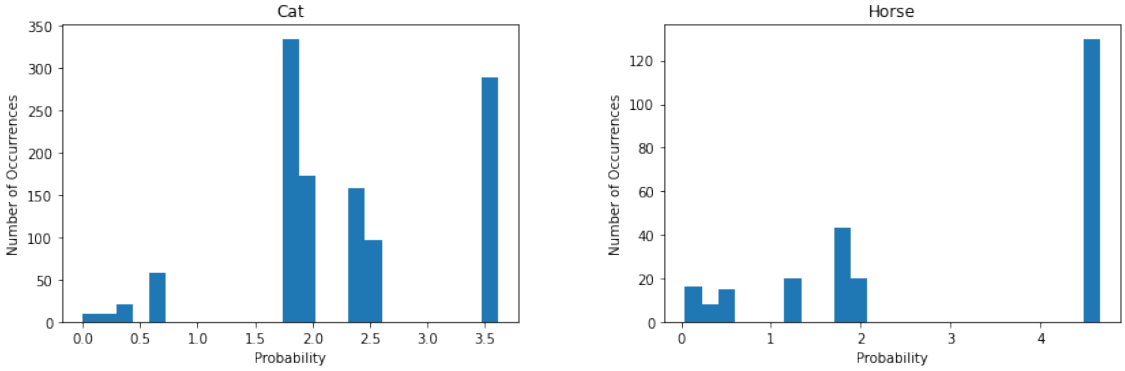


Figure 4.5: Histograms of *Cat* and *Horse* tasks with a *probability threshold* of 1.6%.

The histograms of Figures 4.3, 4.4 and 4.5 show the probability distributions of the reported sequences for each task. The estimated *probability thresholds* can be seen in the figures as separations of low probability sequences to usual occurrences. A manual examination confirmed that below the chosen *probability thresholds* the probability of false positives is very small.

4.2 RQ1: Comparison to Litterbox

After setting the *gram size* to 3 and *sequence lengths* to 2 for the first analysis and 3 for the second bug detection analysis, project-specific models for each pupil task from Table 4.2 are used for a comparison between LitterBox [Frä+20] and the n-gram language model approach of Bugram [Wan+16]. The same projects from Table 4.3 are analysed by LitterBox and the n-gram language model to test how many sequences are reported after knowing the failed tests of each project through Whisker [SKF19]. The *reporting size* is unlimited in this case. Table 4.4 shows the number of reported smells or sequences of each method.

Table 4.4: The number of reported bugs found by LitterBox and n-gram model.

Task	#FailedTests	#LitterboxSmells	#NgramSequences
Fruit Catching	-	113	42
Monkey	0	2	8
Elephant	0	1	5
Cat	0	0	6
Horse	1	0	8

The analysis shows that even though in some cases LitterBox did not find smells and Whisker tests all passed, the n-gram language model reported sequences that pointed out potential bugs or unusual use cases. This shows that the n-gram language model has a different range of violations it can detect in Scratch projects.

4.3 RQ2: Violation Classification

The analysis procedure is continued by analysing specifically the bugs that are reported by the n-gram language model. For each task one project is manually analysed to

estimate the rate of false positives. The probability threshold varies from the size of the model. Table 4.3 shows the projects that were chosen for assessment as well as their further information. After all by the n-gram language model detected potential bugs were collected in the candidate bug set, the defective code is manually classified into the following categories: *True Bugs*, *Unusual Use Cases* or *False Positives*. Table 4.5 displays the numbers for each category.

Table 4.5: The categorization of all reported bugs.

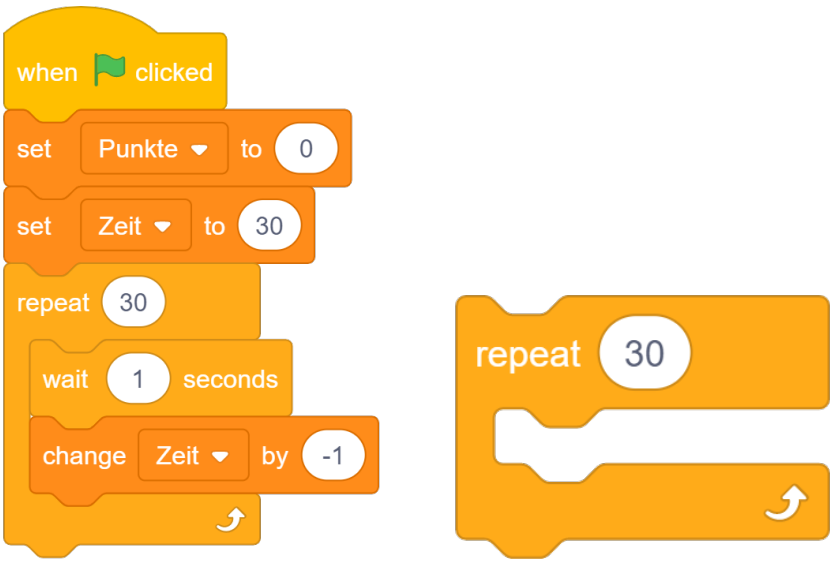
Task	Probability Threshold	#Reported	#True	#UnusualUse	#FalsePositive
Fruit Catching	0.6%	23	15	3	5
Monkey	1.6%	3	2	1	0
Elephant	1.6%	1	0	1	0
Cat	1.6%	1	0	1	0
Horse	1.6%	4	3	0	1

If a sequence is an unusual use case, it will not be detected by LitterBox because it is not a code smell. Most of the time an unusual use is a consequence of an extension of the original task and therefore not part of the usual used sequences that are needed for a project to pass the test.

Table 4.6: An excerpt of the report for the *Fruit Catching* task.

Actor	Position	Sequence	Probability
Bowl	Script@1608d14c	RepeatTimesStmt, NumberLiteral	0.06948304613674264
Apple	Script@1ff8da	Never, UntilStmt	0.44469149527515284
Apple	Script@3a4e11a	IfElseStmt, UnspecifiedBoolExpr	0.023346303501945526
Apple	Script@e3e2beb8	ChangeYBy, NumberLiteral	0.39355197331851033

Figure 4.6 from the project `K7_S02.sb3` shows an example for an unusual use case with the sequence [RepeatTimesStmt, NumberLiteral] that was reported with a probability of 0.07%, according to the report excerpt in Table 4.6. This means that in the solutions of the *Fruit Catching* task, the usage of a [RepeatTimesStmt] block was very limited and therefore it is despite its low probability not a bug.



(a) Example Scratch code.

(b) RepeatTimesStmt, NumberLiteral

Figure 4.6: Example *script* for detected unusual use case in Scratch code.

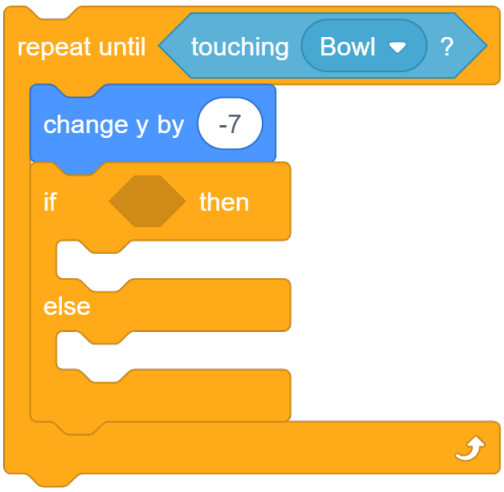


Figure 4.7: Example for detected bug and false positive in *Fruit Catching* project.

Another example would be the *script* in Figure 4.7 where everyone can see at first glance that there is a bug that is caused by missing *blocks*. The reported sequences, that are also shown in Table 4.6, are [Never, UntilStmt] with 0.44%, [IfElseStmt, UnspecifiedBoolExpr] with a probability of 0.02% and [ChangeYBy, NumberLiteral] with 0.39% although the last sequence is classified as a false positive.

When we compare the code smells that LitterBox is able to find with the bugs that

can be detected by the n-gram language model, then we see that the n-gram language model is not able to detect long scripts or unused variables in the Scratch code. But the n-gram language model is capable of finding dead code, empty scripts as well as empty bodies in if-else statements.

4.4 RQ3: Open Solutions

For a wider perspective on the usage of the n-gram language model approach to bug detection we selected a set of pupils' solutions where the children could try out all the things they had newly learned. This means that there is no unique solution for these tasks, just open interpretations of simple projects. Therefore, there is no method to determine if a reported sequence is an unusual use case. The project-specific model, we created out of the solutions, reported the following bugs displayed in Table 4.7 for a set of example projects of the data set. In this analysis the *probability threshold* is set to 0.5% for the model.

Table 4.7: Reported bugs of open projects.

Task	#Reported	#TrueBug	#FalsePositive
`m043.sb3`	4	3	1
`m052-Zauberei.sb3`	2	1	1
`m067-Apfelsuche.sb3`	1	0	1
`m074-Sternenjaeger.sb3`	1	0	1
`m091 2.sb3`	4	0	4
`w002-Weltraumangriff.sb3`	2	2	0

The frequent occurrence of false positives could be a consequence of the missing unusual use case categorisation. A lot of sequences just were not used as often by the novice programmers and when they appear one time their probability is consequently pretty low.

Table 4.8: An excerpt of the report for the *Zauberei* task.

Actor	Position	Sequence	Probability
Rokkduck	Script@497b81c5	PlaySoundUntilDone, Never	0.16643473084111865
Rokkduck	Script@497b81c5	RepeatTimesStmt, NumberLiteral	0.3615592433603648

Figure 4.8: Example for detected bug and false positive in open project.

The SCRATCH *script* in Figure 4.8 is a part of the `m052-Zauberei.sb3` code and contains an example for a bug as well as false positive that are shown in the report excerpt in Table 4.8. The sequence [PlaySoundUntilDone, Never] with a probability of 0.17% is a bug because it hints at the [Hide] block that can never be executed because of a missing *hat* block. Furthermore, the sequence [RepeatTimesStmt, NumberLiteral] with 0.36% is falsely reported because it is not a programming mistake. If there was a reference solution of the *Zauberei* task, then it could also be an unusual use case depending on the number of occurrences of the [RepeatTimesStmt] block in the solutions.

4.5 RQ4: General vs Project-specific Model

The bigger model may have a wider range of potential n-grams but this does not correlate with better results. Like it is shown in Table 4.9 with the *Cat* task as an example, the reported probabilities of the general model are not usable in a way to identify potential bugs, code smells or unusual use cases in the SCRATCH code in comparison to project-specific models.

One reason are negative possibilities that appeared in the reports. The origin of the negative calculations are still unclear and have to be analysed in a more thorough way in the near future. A programming mistake or miscalculations could be the reason but even in the case of a correct probability estimation, project-specific n-gram models are more reliable in their usage.

Table 4.9: General vs project-specific model.

Sequence	General	Project-specific
GreenFlag, RepeatForeverStmt	-163.7176695714846	0.0052368762629570725
Touching, SayForSecs	-93.4693467460925	0.0025830339970531286
PointTowards, MoveSteps	-56.557244922395064	3.548409418312439E-4
NumberLiteral, IfThenStmt	72.48561394813275	0.0033964044507544403
NumberLiteral	72.48561394813275	2.3721578534715566
StringLiteral, NumberLiteral	-46.75499727794927	0.0027517806291449814

4.6 Threats to Validity

There is no guarantee that this bachelor's thesis is free of faults or miscalculations. Here are a few points that should be addressed about the here described and executed research as well as evaluation of the n-gram language model on Scratch code.

Implementation of N-gram Model. To compare the results of LitterBox with the n-gram model approach, we implemented the n-gram language model as close as possible to the information given from the Bugram [Wan+16] paper and based on the AST that is created by LitterBox out of Scratch projects in JSON format. For our implementation, we have tried our best to tune the configuration parameters to obtain the best results. Our comparison is fair because both LitterBox as well as n-gram language model are evaluated on the same projects.

Bugs are manually verified. Following the Bugram [Wan+16] paper, reported bugs were assessed manually in order to approve and classify all low probability *token* sequences and distinguish between, i.e., true bugs, unusual use cases and false positives. Since we are not the original programmers of the analysed projects that are featured in this bachelor's thesis, the examination of the code is not objective. But because of common practice, it is an acceptable approach for the assessment of the given code.

5 Related Work

The analysis of code in order to obtain information and improve its efficiency and readability is a common method of reducing bug patterns and code smells. These are a few examples of specific ways how researchers try to increase their knowledge about written code and how to get better results in the long term independent of the assessed programming languages.

5.1 Analysing Scratch Programs

The tools Dr. Scratch [MRR15] as well as Hairball [Boe+13] analyse Scratch programs to find bugs and code smells. These are then reported to the user in order to improve the computational thinking and coding skills of novice programmers. Bad programming habits were assessed in a preliminary study by Moreno et al. [MR14] and code smells that are very common in Scratch were analysed by Vargas-Alba et al. [Var+19]. Stahlbauer et al. [SKF19] introduced Whisker which is a formal testing framework for Scratch. LitterBox, a tool created by Frädrich et al. [Frä+20] that creates an AST of Scratch programs, is used for finding code smells as well as bug patterns.

5.2 Object Usage Anomalies

Interactions with objects are required to follow a specific procedure, for example by a sequence of method calls. But these standards are rarely documented and can lead to problematic behaviour in the code. Wasylkowski et al. [WZL07] infers legal sequences of method calls to code examples. The results can then be used to find anomalies in the

analysed implementation. As an automatic defect detection algorithm, it is the first of its kind that uses method call sequences to learn and detect anomalies.

5.3 N-gram Language Models

Hindle et al. [Hin+12] first introduced the N-GRAM LANGUAGE MODEL to show that software source code is highly repetitive and the N-GRAM LANGUAGE MODEL can be used in code suggestion and completion. This work is the basis for using language models to model source code and demonstrated how they could be used in software tools. A very accurate algorithm by using a Hidden Markov Model for code completion was proposed by Han et al. [HWM09]. SLAMC by Nguyen et al. [Ngu+13], which incorporated semantic information into an n-gram model, presented a method to code suggestion. It demonstrated how tokens can be seen more semantically instead of just syntactically. Raychev et al. [RVY14] investigated the effectiveness of various language models for code completion, i.e., n-gram and recurrent neural networks. By combining program analysis and the n-gram model, they proposed SLANG which had the goal to predict the sequence of method calls in a software system.

5.4 Bugram

The effective usage of n-gram language models in the field of bug detection is also demonstrated by Wang et al. [Wan+16] with their tool BUGRAM that finds defective code with N-GRAM LANGUAGE MODELS in JAVA programs. Although there are other studies that covered the usage of n-grams for detecting clone bugs [HCN14], localizing faults [Nes+08] and code search [KMA13], these did not leverage n-gram models. In contrast to n-grams that are only *token* sequences, n-gram models are Markov models built on n-grams.

6 Conclusion

Writing code can be a frustrating experience when the tools given are not enough to ensure that the created program fulfils all the important requirements to run smoothly and without mistakes. Even more so when the programmers are pupils who are in the beginner state of understanding what coding really means. Scratch already is a good starting platform to learn the basics but the editor lacks essential debugging tools.

To improve the accuracy of Scratch projects and lessen the frustration that comes with bug detection, this bachelor's thesis proposed a method of bug detection that is based on n-gram language models. The Tokenizer uses the AST of LitterBox as a foundation for tokenizing each Scratch project. By calculating a probability distribution of all n-grams in a given data set, the NGramTrainer is then able to estimate the probability of the occurrence of a specific *token* sequence that is used in Scratch projects. Assuming that a low probability hints at a programming mistake or an unusual use case, the NGramBugFinder can analyse projects based on the n-gram language model that was calculated by the NGramTrainer and report these sequences as potential bugs.

We created a n-gram language model out of a big data set that consists of 75,277 as well as small sets of pupil's solutions for common tasks. At first we used project-specific n-gram language models and utilized them to compare the reported bugs with other bug detection approaches. The analysis showed that an n-gram language model has a different range of detected violations than for instance LitterBox or Whisker tests do. Further inspection showed that most unusual use cases of Scratch code that were found be the n-gram language model were extensions of the original task. Other found bugs include dead code and empty scripts or bodies, whereas long scripts or unused variable are code smells that are only reported by LitterBox.

As another test of the n-gram language model approach we analysed a set of projects with different kinds of tasks and solutions. We wanted to know if the model can detect

mistakes in absence of a reference solution. The reports show that the method is working with a low percentage of false positives. Most bugs are found except for unusual use cases which is due to a missing reference.

Lastly, the attempt at utilizing the big data set in comparison to project-specific models was not as successful because of an appearance of negative probabilities. But aside from a possible miscalculation the numbers already hint at a not as effective use of the general model. Because of missing reference solutions it is also impossible to categorize unusual use cases. To approve this hypothesis that project-specific models are more accurate than general models, further research with the big data set is needed.

Because this is the first time utilizing N-GRAM LANGUAGE MODELS for bug detection in SCRATCH, there are different aspects of the evaluation that can be used or improved for further research.

Visualisation. Currently detected bugs are only printed into a `.csv` report file which is not very efficient for frequent usage. A better solution would be to visualize the reported *token* sequences and, instead of just writing the `Map` of low probability sequences into a file, to convert the result back into common SCRATCH blocks. This way the output is easier to compare to the originally analysed code and manual examination of the SCRATCH projects can be executed faster.

Token Granularity. Because the basis of the tokenization process is the AST of LITTERBOX that was modified to only contain the tokens that are important for bug detection and exclude all information-based AST nodes, the granularity of the tokens is set to a specific level. But it would be interesting to explore different granularity levels and their effect on the N-GRAM LANGUAGE MODEL and therefore the bug detection results. Should SCRATCH blocks like literals, drop-down lists or extensions like *pens* be part of the *token* AST or are those not effecting the end results?

Sizes of Data Sets. In this bachelor's thesis the N-GRAM LANGUAGE MODELS were created by large as well as smaller data sets but the optimal size to create the best model was not part of the research questions. We already discovered that smaller and project-specific models are more precise and accurate in their results than larger ones. But the best number of project solutions that should be used for the model is unknown.

Parameter Boundaries. Another interesting part of N-GRAM LANGUAGE MODELs are the many different parameters that can be configured. Is it useful to set the *gram size* or *sequence length* to big numbers over 10? What would happen if we set the *minimum token occurrence* higher? How big should the *reporting size* be to keep the false positive rate at the minimum? All these questions can be used for future research and are good starting points for further analysis of the N-GRAM LANGUAGE MODEL bug detection approach in SCRATCH.

Bibliography

[Boe+13] Bryce Boe et al. “Hairball: Lint-Inspired Static Analysis of Scratch Projects.” In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE 13. Denver, Colorado, USA: Association for Computing Machinery, 2013, pp. 215–220 (cit. on pp. 1, 31).

[Frä+20] Christoph Frädrich et al. “Common Bugs in Scratch Programs.” In: *ITiCSE20: 25th annual conference on Innovation and Technology in Computer Science Education*. To appear. Association for Computing Machinery, 2020 (cit. on pp. 1, 25, 31).

[GFH16] Katharina Geldreich, Alexandra Funke, and Peter Hubwieser. “A programming circus for primary schools.” In: *ISSEP 2016*. 2016, pp. 49–50 (cit. on p. 21).

[HWM09] Sangmok Han, David R. Wallace, and Robert C. Miller. “Code Completion from Abbreviated Input.” In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE 09. USA: IEEE Computer Society, 2009, pp. 332–343 (cit. on p. 32).

[Hin+12] Abram Hindle et al. “On the Naturalness of Software.” In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE 12. Zurich, Switzerland: IEEE Press, 2012, pp. 837–847 (cit. on p. 32).

[HCN14] Chun-Hung Hsiao, Michael Cafarella, and Satish Narayanasamy. “Using Web Corpus Statistics for Program Analysis.” In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA 14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 49–65 (cit. on p. 32).

[KMA13] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. "Rendezvous: A Search Engine for Binary Code." In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR 13. San Francisco, CA, USA: IEEE Press, 2013, pp. 329–338 (cit. on p. 32).

[Mal+10] John Maloney et al. "The scratch programming language and environment." In: *ACM Transactions on Computing Education (TOCE)* 10.4 (2010), pp. 1–15 (cit. on p. 1).

[Mera] Merriam-Webster. "Markov Chain." In: *Merriam-Webster.com dictionary*. URL: `https://www.merriam-webster.com/dictionary/Markov%20chain` (visited on 08/23/2020) (cit. on p. 7).

[Merb] Merriam-Webster. "smooth." In: *Merriam-Webster.com dictionary*. URL: `https://www.merriam-webster.com/dictionary/smoothing` (visited on 10/20/2020) (cit. on p. 15).

[MR14] J. Moreno and G. Robles. "Automatic detection of bad programming habits in scratch: A preliminary study." In: *IEEE Frontiers in Education Conference (FIE) Proceedings*. IEEE, Madrid, 2014, pp. 1–4 (cit. on pp. 1, 31).

[MRR15] Jesús Moreno-León, Gregorio Robles, and Marcos Román-González. "Dr. Scratch: Automatic analysis of scratch projects to assess and foster computational thinking." In: *RED. Revista de Educación a Distancia* 46 (2015), pp. 1–23 (cit. on pp. 1, 31).

[Nes+08] Syeda Nessa et al. "Software Fault Localization Using N-gram Analysis." In: *Wireless Algorithms, Systems, and Applications*. Ed. by Yingshu Li et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 548–559 (cit. on p. 32).

[Ngu+13] Tung Thanh Nguyen et al. "A Statistical Semantic Language Model for Source Code." In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: Association for Computing Machinery, 2013, pp. 532–542 (cit. on p. 32).

[RVY14] 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*. PLDI 14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 419–428 (cit. on p. 32).

[SKF19] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. "Testing Scratch Programs Automatically." In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 165–175 (cit. on pp. 1, 21, 25, 31).

[Var+19] Angela Vargas-Alba et al. "Bad Smells in Scratch Projects: A Preliminary Analysis." In: *Proceedings of the 2nd Systems of Assessments for Computational Thinking Learning Workshop*. 2019 (cit. on pp. 1, 31).

[Wan+16] Song Wang et al. "Bugram: Bug Detection with n-Gram Language Models." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 708–719 (cit. on pp. 2, 8, 9, 16, 20, 25, 30, 32).

[WZL07] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. "Detecting object usage anomalies." In: *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Dubrovnik, Croatia, Sept. 2007, pp. 35–44 (cit. on p. 31).

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie, dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, October 25, 2020

Eva Gründinger