Advancing Code Readability: Mined & Modified Code for Dataset Generation

Lukas Krodinger

Master Thesis in M.Sc. Computer Science
Faculty of Computer Science and Mathematics
Chair of Software Engineering II

Matriculation number	89801
Supervisor	Prof. Dr. Gordon Fraser
Advisor	Lisa Griebl

3rd March 2024

Abstract

Deep learning-based models are achieving increasingly superior accuracy in classifying the readability of code. Recent research focuses mostly on different model architectures to further improve code readability classification. The models mostly use (parts of) the same labeled dataset, consisting of 421 code snippets. However, it is known that deep learning-based approaches improve with a large amount of data. Consequently, a larger labeled dataset could greatly advance the research field of code readability classification. In this work, we investigate the use of a new dataset consisting of 69k code snippets together with its novel generation approach. The generation approach involves the extraction and modification of code snippets from public GitHub repositories. The generated dataset is evaluated using a survey with 200 participants and by training a state of the art code readability classification model both with and without the new dataset. In the future, our dataset might increase the accuracy of all readability classification models.

Contents

1. INTRODUCTION

Code readability is of utmost significance in the domain of software development. In the domain of software development, the significance of code readability cannot be overstated. Together with understandability, it serves as the foundation for efficient collaboration, comprehension, and maintenance of software systems [27, 1]. Maintenance alone will consume over 70 % of the total lifecycle cost of a software product and for maintenance, the most time-consuming act is reading code [7, 9, 29, 5]. Therefore, it is important to ensure a high readability of code. In order to archive this, we need to measure readability. In the last years, researchers have proposed several metrics and models for assessing code readability with an accuracy of up to 81.8 % [7, 27, 10, 30]. In recent years, deep learning-based models are able to achieve an accuracy of up to 88.0 % [19, 20, 33, 22, 16, 17].

However, a major limitation of these models is not their architecture, but the amount of available data for Java code readability classification, which comprises 421 code snippets [7, 10, 30]. The current training data originates from questionnaires, where humans have manually labeled the code snippets. This has two drawbacks: Firstly, manual labeling requires a lot of effort. Secondly, the dataset is too small for deep learning, as those require To address those drawbacks, we aim to automatically generate more training data.

Deep learning-based models perform better the more training data they get [12]. Therefore, one approach to further improve existing models is to gather more training data. This requires, as it was done previously, a lot of effort and persons willing to rate code based on their readability.

The main idea of this work is to investigate whether it is possible to achieve higher accuracy in code readability classification using automatically generated data.

In a first step, following the approach of Allamanis et al. [2], we download GitHub[1] repositories with high code quality. Our criteria for high code quality are an elevated number of stars, forks, method comments the use of and compliance with a checkstyle[2] specification. For example, developers prefer high code quality and therefore star or fork repositories with high code quality more likely. We select Java files from the repositories that meet our criteria and extract methods from the Java classes in these files and label them as well readable (Assumption 1).

[1] `https://github.com/`, accessed: 2024-02-29
[2] `https://checkstyle.sourceforge.io/`, accessed: 2024-02-09

In a second step, all selected Java files are manipulated so that its code is subsequently less readable. You can find an exemplary result of this in TODO. We extract methods from the Java classes in these files and label them as poorly readable (Assumption 2).

After both steps, we have a new, automatically generated dataset for code readability classification. We call it minded-and-modified dataset.

How can we manipulate code so that it is less readable afterwards? We introduce a tool called Readability Decreasing Heuristics. This is a collection of heuristics that, when applied to Java files, lower the readability of it. Such heuristics are replacing spaces with newlines or increasing the indentation of a code block by a tab or multiple spaces. Most changes also decrease readability when applied in reverse (replacing newlines with spaces, decreasing indentation).

Methods in Java are syntactically the same, before and after applying Readability Decreasing Heuristics. Functionality does not change either. However, if various modifications are applied many times, those changes are capable of lowering the readability of source code, as TODO suggests.

We conducted a user study to validate our assumptions (Assumption 1 and 2) and to thereby verify the mined-and-modified dataset. Additionally, we evaluate the performance improvement on the towards model of Mi et al. [22] using the automatically generated dataset.

Our contributions are as follows:

- We combine and unify existing datasets [7, 10, 30]
- We propose a approach to mine well readable methods
- We create a tool to decrease the readability of Java class files
- We propose a novel dataset generation approach and introduce a new readability classification dataset, namely the mined-and-modified dataset
- We evaluate our generation approach with a user study
- We evaluate our generation approach by comparing the model performance of the towards model of Mi et al. [22] trained with and without the mined-and-modified dataset

The survey confirms both, the well-readable-assumption (Assumption 1) and the poorly-readable-assumption (Assumption 2). Although our approach for creating a dataset works in principle, we are not able to address enough aspects of code readability with the proposed heuristics. Thus, our dataset probably only addresses a partial problem of code readability.

2. BACKGROUND AND RELATED WORK

In the following subsections you find an overview of the background and related work on code readability and our approach to dataset creation.

2.1. CODE READABILITY

We start with an overview over definitions of code readability.

Buse and Weimer provides one of the first definitions: "We define readability as a human judgment of how easy a text is to understand."

Tashtoush et al. combines numerous other aspects from various definitions. According to them code readability can be measured by looking at the following aspects [34]:

- Ratio between lines of code and number of commented lines
- Writing to people not to computers
- Making a code locally understandable without searching for declarations and definitions
- Average number of right answers to a series of questions about a program in a given length of time

Recent definitions of code readability are shorter, trying to focus on the key aspects. Oliveira et al. defines readability as "what makes a program easier or harder to read and apprehend by developers" [25].

Also Mi et al. summarizes code readability as "a human judgment of how easy a piece of source code is to understand" [21]. This comes close to the definition of Buse and Weimer [7].

There are various related terms to readability: Understandability, usability, reusability, complexity, and maintainability [34]. Among those especially complexity and understandability are closely related to readability.

Readability is not the same as complexity. Complexity is an “essential” property of software that arises from system requirements, while readability is an “accidental” property that is not determined by the problem statement [7, 6].

Readability is neither the same as understandability, as the key aspects of understandability are [30, 15, 37, 4]:

- Complexity
- Usage of design concepts

- Formatting
- Source code lexicon
- Visual aspects (e.g., syntax highlighting)

Posnett et al. states that readability is the syntactic aspect of processing code, while understandability is the semantic aspect [27].

Based on Posnett et al., Scalabrino et al. writes about readability: "Readability measures the effort of the developer to access the information contained in the code, while understandability measures the complexity of such information" [30, 27].

For example, a developer can find a piece of code readable but still difficult to understand. Recent research gives evidence that there is no correlation between understandability and readability [31].

Comparing the definitions of code readability in literature we can see, that there are some common aspects in most definitions. These are:

- Ease/complexity of understanding/comprehension/apprehension
- Human judgment/assessment
- Effort of the process of reading (differentiation to understandability)

Based on this, we come up with the following definition:

Code readability is the human assessment of the effort required to read and understand code.

In the last years, researchers have proposed several metrics and models for assessing code readability with an accuracy of up to 81.8 % [7, 27, 10, 30]. In recent years, deep learning-based models are able to achieve an accuracy of to 88.0 % [19, 20, 33, 22, 16, 17]on available datasets. Examining these works more closely in the following, we delve into their intricacies.

2.2. CONVENTIONAL CALCULATION APPROACHES

A first estimation for source code readability was the percentage of comment lines over total code lines [1]. Then researchers proposed several more complex metrics and models for assessing code readability [7, 27, 10, 30]. Those approaches used handcrafted features to calculate how readable a piece of code is. They were able to achieve up to 81.8 % accuracy in classification [30].

```java
/**
* This method determines the sign of a given number and prints a corresponding message.
*
* @param number The input number to be checked.
*/
public static void printSign(int number) {
        if (number > 0) {
                System.out.println("Number is positive");
        } else if (number < 0) {
                System.out.println("Number is negative");
        } else {
                System.out.println("Number is zero");
        }
}
```

(a) An example of a simple and well readable Java method.

```java
public static void foo(int bar) {if ((bar>0)) {
                        System.out.println("Number is positive");
                                } else if (bar < (0 + 0)) {
                        System.out.println("Number is negative");
        } else {
                System.out.println("Number is zero");
        }
}
```

(b) The same example as in Listing 1a but modified for poor readability.

Listing 1.: Well readable (Listing 1a) vs. poorly readable (Listing 1b) code.

2.3. DEEP LEARNING BASED APPROACHES

In recent years code readability classification is dominated by machine learning, especially deep learning approaches. As the quality of the models increased, so did their accuracy (see Table 1).

IncepCRM was the first introduced deep learning model called for code readability classification. It automatically learns multi-scale features from source code with minimal manual intervention [19].

In a follow up paper Convolutional Neural Networks (ConvNets) were introduced to code readability classification in a model called DeepCRM. Other than previously, DeepCRM employs three ConvNets with identical architectures and was trained on differently preprocessed data [20].

Table 1.: Accuracy scores of two-class readability classification models.

Model	Type	Accuracy
Buse [7]	Conventional	76.5 %
Possnet [27]	Conventional	71.7 %
Dorn [10]	Conventional	78.6 %
Scallabrino [30]	Conventional	81.8 %
Mi_IncepCRM [19]	Deep Learning	84.2 %
Mi_DeepCRM [20]	Deep Learning	83.8 %
Sharma [33]	Deep Learning	84.8 %
Mi_Towards [22]	**Deep Learning**	**85.3 %**
Mi_Ranking [16]	Deep Learning	83.5 %
Mi_Graph [17]	Deep Learning	88.0 %

Another study proposes an approach using Generative Adversarial Networks (GANs). The proposed method involves encoding source codes into integer matrices with multiple granularities and utilizing an EGAN (Enhanced GAN) [33]. It was able to surpass the accuracy of previous readability classification models as shown in Table 1.

The limitation of previous deep learning-based code readability models was to focus primarily on structural features. This was addressed by proposing a method that extracts features from visual, semantic, and structural aspects of source code. Using a hybrid neural network composed of BERT, CNN, and BiLSTM, the model processes RGB matrices, token sequences, and character matrices to capture various features [22].

Up to this point, code readability classification was considered mainly as a task that is applied to a single code snippet at once. A new approach was introduced that frames the problem as a ranking task. The proposed model employs siamese neural networks to rank code pairs based on their readability [16].

All previous accuracy scores in two class classification were surpassed by the introduction of a graph-based representation method for code readability classification. The proposed method involves parsing source code into a graph with abstract syntax tree (AST), combining control and data flow edges, and converting node information into vectors. The model, comprising Graph Convolutional Network (GCN), DMoNPooling, and K-dimensional Graph Neural Networks (k-GNNs) layers, extracts syntactic and semantic features [17].

You can find an overview over the accuracy scores for the models mentioned in Table 1.

Until now many deep learn architectures and components were introduced with the goal to surpass previous classification accuracy scores. Their common limitation is a dataset consisting of 36077 code snippets for training and evaluation. The main contribution of this work is not a model that outperforms a state of the art model but rather a new dataset (generation approach). For evaluation we opted for the *Mi_Towards* model (hereinafter referred to as towards model) from Mi et al. [22]. We did not choose the best performing one, *Mi_Graph*, as its main contribution is to use the AST representation of the code, while our dataset generation approach includes features that are not represented in the AST [17].

2.4. DATA AUGMENTATION

All the mentioned models were trained with (a part of) the data from Buse, Dorn and Scalabrino consisting of a total of 421 Java code snippets. The data was generated with surveys. They therefore asked developers several questions, including how well readable the proposed source code is [7, 10, 30]. We will refer to this dataset as *merged* dataset.

The problem that there is little data in the area of code readability classification for machine learning models has been recognized.

A recent paper addressed the challenge of acquiring a larger amount of labeled data using augmentation. The researchers proposed this to artificially expand the training set instead of the time-consuming and expensive process of obtaining labels manually. They employ domain-specific transformations, such as manipulating comments, indentations, and names of classes/methods/variables, and explore the use of Auxiliary Classifier GANs to generate synthetic data. They advance to a classification accuracy of 87.3 % [21]. Lately researchers successfully enhanced the data augmentation approach by incorporating domain-specific data transformation and Generative Adversarial Networks (GANs) [18]. The results of both show, that more data has a significant impact on the reached classification accuracy. However, they artificially augment data based on the 36077 code snippets of the merged data set. Therewith their augmented data is based on the small dataset. The minded-and-modified dataset is not.

Recently researchers developed a methodology to identify readability-improving commits, creating a dataset of 122k commits from GitHub's revision history. This dataset was used to automatically identify and suggest readability-improving actions for code snippets. They trained a T5 model to emulate developers' actions in improving code readability, achieving a prediction accuracy between 21 % and 28 %. The empirical evaluation shows that 82-91 % of the dataset commits aim to improve readability, and the model successfully mimics developers in

21 % of cases [36]. This shows the potential of a large dataset. However, the approach dataset and model results are hardly comparable with previous studies due to the usage of commits instead of code snippets. We, on the other hand, keep to use code snippets.

2.5. DIVERSE PERSPECTIVES

There is also other important research in the field of readability classification that does not directly affect this work, but could have implications for future work.

Fakhoury et al. showed based on readability improving commit analysis that previous models do not capture what developers think of readability improvements. They therefore analyzed 548 GitHub[1] commits manually. They suggest considering other metrics such as incoming method calls or method name fitting [11].

Oliveira et al. conducted a systematic literature review of 54 relevant studies on code readability and legibility, examining how different factors impact comprehension. The authors analyze tasks and response variables used in studies comparing programming constructs, coding idioms, naming conventions, and formatting guidelines [25].

In a recent study participants demonstrated a consistent perception that Python code with more lines was deemed more comprehensible, irrespective of their level of experience. However, when it came to readability, variations were observed based on code size, with less experienced participants expressing a preference for longer code, while those with more experience favored shorter code. Both novices and experts agreed that long and complete-word identifiers enhanced readability and comprehensibility. Additionally, the inclusion of comments was found to positively impact comprehension, and a consensus emerged in favor of four indentation spaces [28].

Choi, Park et al. introduced an enhanced source code readability metric aimed at quantitatively measuring code readability in the software maintenance phase. The proposed metric achieves a substantial explanatory power of 75.7 %. Additionally, the authors developed a tool named Instant R. Gauge, integrated with Eclipse IDE, to provide real-time readability feedback and track readability history, allowing developers to gradually improve their coding habits [8].

Mi et al. aim to understand the causal relationship between code features and readability. To overcome potential spurious correlations, the authors propose a causal theory-based approach, utilizing the PC algorithm and additive noise models to construct a causal graph. Experimental results using human-annotated readability data reveal that the average number of comments positively impacts code readability, while the average number of assignments, identifiers, and periods has a negative impact [23].

Segedinac et al. introduces a novel approach for code readability classification using eye-tracking data from 90 undergraduate students assessing Python code snippets [32].

2.6. DATA GENERATION

In addition to related work on models and datasets, there is also related work that uses some of the ideas that we employ in our proposed approach for data generation.

Loriot et al. created a model that is able to fix Checkstyle[3] violations using Deep Learning. They inserted formatting violations based on a project specific format checker ruleset into code in a first step. They then used a LSTM neural network that learned how to undo those injections. Their approach is working on abstract token sequences. Their data is generated in a self-supervised manner [14]. A similar idea has been explored by Yasunaga and Liang [38]. We will use the idea of intentional degradation of code for data generation.

Another concept we will employ is from Allamanis et al. They cloned the top open source Java projects on GitHub[1] for training a Deep Learning model. Those top projects were selected by taking the sum of the z-scores of the number of watchers and forks of each project. The projects have thousands of forks and stars and are widely used among software developers and thus the authors assumed the code within to be of good quality [2]. We will also use fork and star counts as criteria for well readable code (Assumption 1).

3. MINED AND MODIFIED CODE FOR DATASET GENERATION

In the following subsections we will describe our approach.

3.1. WORK ON EXISTING DATASETS

Most of the related work (see Section 2) uses a combination of the data of Buse and Weimer, Dorn and Scalabrino et al. The raw data from their surveys can be downloaded [4], but their data is not uniformly formatted, including ratings

[3]`https://checkstyle.org/`, accessed: 2023-07-25
[4]`https://dibt.unimol.it/report/readability/`, accessed: 2024-02-18

that are not Java code snippets, as well as the individual ratings rather than the mean of the ratings used for training machine learning models. Other than our mined and modified code snippets theirs do not all have the scope of a method, but instead consist of a few lines of code.

We converted and combined the three datasets into one: code-readability-merged. In recent years, Huggingface[5] established as the pioneer in making models and datasets available. Therefore we decided to publish the merged dataset on Huggingface[6].

We refer to this dataset as the *merged* one.

3.2. CLASSIFICATION CONSIDERATIONS

When classifying the readability of code, the state of the art is to perform a binary classification into well readable and poorly readable code [19, 20, 33, 22, 16].

However, code readability classification is not a binary classification task per se. Mi et al. introduced a third, neutral class to address this problem [17]. When rating code snippets, a Likert scale [13] from 1 (very unreadable) to 5 (very readable) was used [7, 10, 30]. While the amount of classes varies, one can encode the data internally as a single-value representation between 0 and 1 where a higher value means higher readability. The output of the model is well readable if the value after the last layer is above 0.5 and poorly readable otherwise.

Our evaluation model is the towards model of Mi et al. [22] and uses the single-value representation. We want to show how they transformed the rating scores into a binary classification problem. First, the mean values of all scores are calculated. In a second step, the snippets are ranked according to their mean score. Then, the top 25 % of the data is labeled as well readable (1.0) and the bottom 25 % is labeled as poorly readable (0.0). The 50 % of the data in between is not used at all [22].

While this transformation is fine in principle, especially with the argument that the data in the middle is neither well readable nor poorly readable, it has drawbacks that only 50 % of the available data is used for model training and evaluation:

[5] `https://huggingface.co/`, accessed: 2024-02-18

[6] `https://huggingface.co/datasets/se2p/code-readability-merged`, accessed: 2024-02-18

First, the available data is further reduced from 421 to 210 Java code snippets. Note that a bottleneck in readability classification is the small amount of available data. So this is a significant loss.

Secondly, evaluation is performed with only those 210 snippets as well. Thus, the model was only evaluated on 50 % of the available data. We suspect that this might be a thread to validity. It could be that the performance of the model is remarkably lower when the evaluation is performed with random, unseen data that also contains moderately readable code snippets.

However, we will both continue to use the binary classification approach as well as to the towards model [22] to make our results comparable to theirs.

3.3. DATASET GENERATION APPROACH

In contrast to previous datasets for readability classification, our dataset is generated using an automated approach. The aim is to mine and modify code from GitHub to obtain both well readable and poorly readable methods. This approach is novel to the best of our knowledge. You can find a visualization in Figure 1.

While we refer to the merged data set (Section 3.1) as the merged data set, the data set generated by our approach is referred to as the mined-and-modified dataset.

Since we ultimately extract methods from code, code snippets and methods are synonyms for our mine-and-modify approach. This is not the case with the merged data set, as there a code snippet is not necessarily an entire method.

The approach is divided into four parts. The first three steps are used to mine well readable Java code. In a final step, we will modify the well readable code to achieve our second goal, namely poorly readable source code.

We start by querying the GitHub REST API[7] for repositories that use checkstyle (query string: "checkstyle filename:pom.xml"). The repository informations (including the URL) are stored together with the main branches. We remove all repositories that do not fulfill these criteria:

- The repository is not a fork of another repository
- The repository is not archived
- The repository is not disabled
- The repository language is Java

[7] `https://docs.github.com/en/rest`, accessed: 2024-02-15

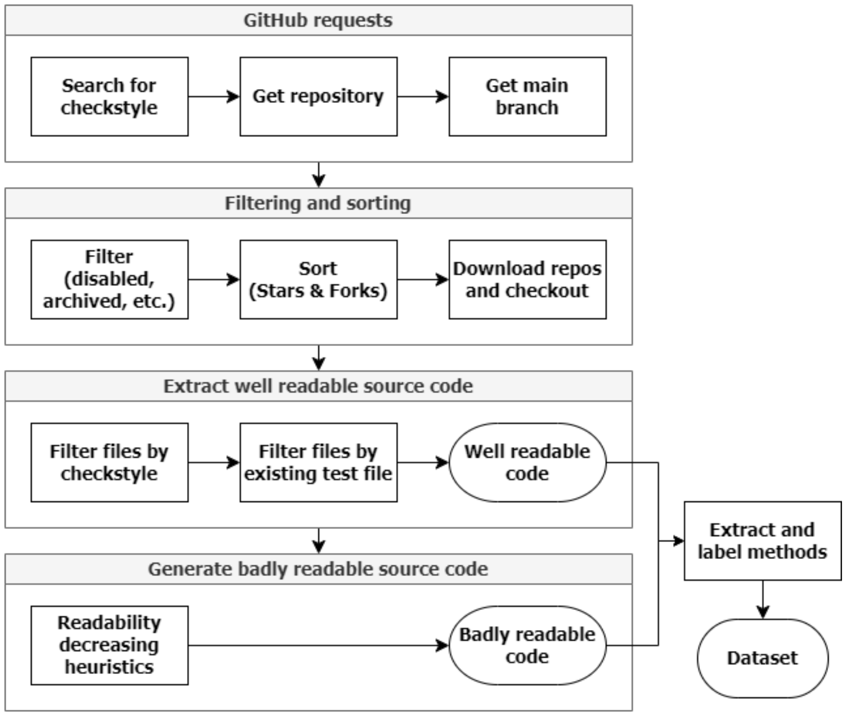


Figure 1.: The used dataset generation approach.

- The repository has at least 20 stars
- The repository has at least 20 forks

The remaining ones are sorted by their star and fork count (equally weighted). The 100 best are cloned and their main branch is checked out.

In a third step we run checkstyle[2] against the projects own checkstyle configuration to get all Java class files, that pass the own checkstyle test. A tool from Maximilian Jungwirth[8] was used for this purpose. From the Java classes that passed this filter we extract all methods that have a comment of any kind at the beginning of the method. This results in 36077 methods which we assume to be well readable.

The fourth and final step is to generate poorly readable code from the well readable one. Therefore we use the proposed Readability Decreasing Heuristics (see Section 3.4). Afterwards we again extract all methods with a comment at the beginning of the method. Initially we planned to not require comments for the poorly readable dataset part. However, it turns out that in this case all well

[8] `https://github.com/sphrilix`, accessed: TODO

readable methods have a comment while most of the poorly readable do not have one. This lead to shortcut learning, whether a method has a comment or not instead of learning to distinguishing the methods by all other criteria as well. After removing code snippets that are identical for the original and the variant with reduced readability (see Section 3.4) and balancing the data set using random sampling, the result is a data set consisting of 69k code snippets.

3.4. READABILITY DECREASING HEURISTICS

In this section we explain how we achieved to decrease the readability of code using Readability Decreasing Heuristics (RDH). The RDH are a set of code manipulation heuristics that are applied to Java files. One part is performed on the abstract syntax tree (AST) representation of the Java files using the spoon library[9] [26]. Another part is executed when pretty-printing the AST back into Java files. This part cannot be displayed in the AST and is therefore executed at source code level immediately after the reverse transformation.

The RDH tool initially converts the Java code of any well readable Java class file into an AST. In the end the AST is parsed back to Java code using an pretty printer. If nothing else is done, this results in the `none`. Note that the code produced by the tool in this way will be slightly different from the original input code, as the styling and formatting of the original code will be overwritten by the default formatting of the Java Pretty Printer of the spoon library[9] [26].

Various modifications can be made between the two steps and during pretty-printing (see Table 3). These modifications are performed on the AST representation of the code to ensure that the functionality stays the same (see Table 2a).

By default the new identifiers for the rename modifications (`renameVariable`, `renameField` and `renameMethod`) are generated in an iterating manner. For each class file we start with *v0* for variables, *f0* for fields and *m0*. We increase the index of each (0 at the beginning) by 1 whenever a name is used. We also added a mode that uses Code2Vec [3] for the generation of identifiers for `renameMethod` instead. With that we can predict more realistic method names. Code2Vec generates multiple method name predictions at once. By picking not the best one but instead the one with the longest name we aim to decrease readability while choosing realistic method names.

These modifications are performed on source code during pretty-printing as they cannot be displayed in the AST: Table 2b.

[9] `https://spoon.gforge.inria.fr/`, accessed: 2024-15-02

Table 2.: All Readability Decreasing Heuristics with explanation and example.

(a) The Readability Decreasing Heuristics that are performed on the AST.

RDH	Description	Example
`renameVariable`	Renaming a variable declaration and its usages	Listing 1b, Line 1
`renameField`	Renaming a field declaration and its usages	Listing 2b, Line 6
`renameMethod`	Renaming a method declaration and its usages	Listing 1b, Line 1
`add0`	Adding zero to numbers	Listing 1b, Line 3
`insertBraces`	Inserting superfluous braces	Listing 1b, Line 1
`starImport`	Replacing specific imports with star-import	Listing 2b, Line 1
`inlineField`	Inlining the values of static fields into the code	Listing 2b, Line 8
`partiallyEvaluate`	Partially evaluate constants	Listing 2b, Line 6

(b) The Readability Decreasing Heuristics that are performed while pretty-printing.

RDH	Description	Example
`newline`	Replacing newlines with none or multiple ones	Listing 2b, Line 2-6
`incTab`	Replace a tab indentation with none or multiple ones	Listing 1b, Line 1
`decTab`	Replace a tab outdentation with one or more ones	Listing 1b, Line TODO
`space`	Replacing a single space with multiple ones	Listing 1b, Line 1
`newLineInsteadOfSpace`	Replacing a space with a newline	Listing 1b, Line TODO
`spaceInsteadOfNewline`	Replacing a newline with a space	Listing 1b, Line 1
`incTabInsteadOfDecTab`	Replace a tab outdentation with an indentation	Listing 1b, Line 2
`decTabInsteadOfIncTab`	Replace a tab indentation with an outdentation	Listing 1b, Line 2

```java
import java.util.Random;

public class TimeConverter {
        public static final int MINUTES_PER_HOUR = 60;
        public static final int HOURS_PER_DAY = 24;
        public static final int MINUTES_PER_DAY = MINUTES_PER_HOUR *
        ↪ HOURS_PER_DAY;
        public static final int SEED = 4242;

        public static void main(String[] args) {
                Random random = new Random(SEED);
                int days = random.nextInt(10);
                int minutes = days * MINUTES_PER_DAY;
                System.out.println(days + " days have " + minutes + "
                ↪ minutes");
        }
}
```

(a) An example of a simple and well readable Java class file.

```java
import java.util.*;



public class TimeConverter {
        public static final int f0 = 1440;
        public static void main(String[] args) {
                Random random = new Random(4242);
                int days = random.nextInt(10);
                int minutes = days * f0;
                System.out.println(days + " days have " + minutes + "
                ↪ minutes");
        }
}
```

(b) The same example as in Listing 2a but modified for poorer readability.

Listing 2.: Well readable (Listing 2a) vs. poorly readable (Listing 2b) code.

Table 3.: Available Readability Decreasing Heuristics along with when they are executed (on AST or Code), their configuration type (a array of probabilities or a single one) and whether they are included in the final dataset. See Appendix II for a concrete configuration.

#	Heuristic	AST/Code	Config. Type	In Dataset
1	`newline`	Code	Array	✓
2	`incTab`	Code	Array	✓
3	`decTab`	Code	Array	✓
4	`space`	Code	Array	✓
5	`newLineInsteadOfSpace`	Code	Single	✓
6	`spaceInsteadOfNewline`	Code	Single	✓
7	`incTabInsteadOfDecTab`	Code	Single	✓
8	`decTabInsteadOfIncTab`	Code	Single	✓
9	`renameVariable`	AST	Single	✓
10	`renameField`	AST	Single	✓
11	`renameMethod`	AST	Single	✓
12	`inlineMethod`	AST	Single	
13	`removeComment`	Code	Single	✓
14	`add0`	AST	Single	
15	`insertBraces`	AST	Single	
16	`starImport`	AST	Single	
17	`inlineField`	AST	Single	
18	`partiallyEvaluate`	AST	Single	

The removal of spaces is not supported, as this would cause keywords or identifiers to merge. Tab *indentation* refers to the process of adding a tab ($\backslash t$) while *outdentation* refers to the opposite, namely removing a tab ($\backslash t$). For example:

1. The current tab count is 1 ($\backslash t\langle CODE\rangle$) (see Listing 1a Line 7)
2. In the next line, we perform a tab indentation
3. The current tab count is now 2 ($\backslash t\backslash t\langle CODE\rangle$) (see Listing 1a, Line 8)
4. In the next line, we perform a tab outdentation
5. The current tab count is now 1 ($\backslash t\langle CODE\rangle$) (see Listing 1a, Line 9)

For the final dataset we excluded some of the RDHs as we can see in Table 3. We excluded `inlineMethod` as it increased the length of methods drastically and made the methods too long. While `starImport` might have an impact on the readability of class files it has none on methods as in Java the import statement are not within the methods in Java. As we finally extract methods for our dataset, `starImport` has no impact. We chose to not include `add0`, `insertBraces`, `inlineField` and `partiallyEvaluate` for the reason of a limited survey capacity. For the same reason, we did not investigate the usage of Code2Vec for `renameMethod` either.

The RDH tool works with a configuration file in which one can specify a probability for each heuristic that can be applied. For heuristics of configuration type *Array* (`newline`, `incTab`, `decTab` and `space`), a array of probabilities must be defined for the respective number of replacements. The probabilities of the array must sum up to 1. For heuristics of configuration type *Single* (others) a single probability must be defined (see Table 3). For example, `spaceInsteadOfNewline` can be configured with 0.05 meaning that each space is replaced with a newline ($\backslash n$) with a probability of 5 %. `space` can be configured with $[0.0, 0.7, 0.2, 0.1]$ meaning that each space is replaced with

- no space with a probability of 0 %
- a single space with a probability of 70 % (no change)
- two spaces with a probability of 20 %
- three spaces with a probability of 10 %

We have chosen the probabilities so that the generated code snippets are still realistic in the sense that they could also be written by humans. You can find the configurations in Table 4 and an exemplary file for `none` in appendix II.

The individual methods are then extracted from the class files. As mentioned (see Section 3.3), we require a method comment for all methods. We therefore use `removeComment` after completing the method extraction.

3.5. CONSTRUCTION OF QUESTIONNAIRES

We evaluated the generated data set and the new approach with a survey. To do this, we had to carefully select suitable code snippets from the dataset. An overview of the approach can be found in Figure 2.

The first step was to find realistic configurations for the RDH tool. After an initial data set with the heuristics was created, a pilot study was conducted. Subsequently, heuristics with a low rating were adjusted to be weaker according to the results of the pilot survey. The result consisted of 9 different RDHs, which can be found in Table 4. Together with the original methods this resulted in 10 different configurations.

Table 4.: Chosen configurations and their probabilities for the Readability Decreasing Heuristics.

Configuration	**Probabilities**
none	-
comments_remove	removeComment: 10 %
newline_instead_of_space	newLineInsteadOfSpace: 15 %
newlines_few	removeNewline: 30 % spaceInsteadOfNewline: 5 %
newlines_many	add1Newline: 15 % add2Newlines: 5 %
rename	renameVariable: 30 % renameField: 30 % renameMethod: 30 %
spaces_many	Add1Space: 20 % Add2Spaces: 10 % spaceInsteadOfNewline: 5 %
tabs	remove1IncTab: 20 % add1IncTab: 10 % remove1DecTab: 10 % add1DecTab: 10 % incTabInsteadOfDecTab: 5 % decTabInsteadOfIncTab: 5 %
all7	all probabilites/7

The configurations are based on probabilities for different heuristics. A heuristic is applied with the specified probability to each occurrence of the object to which it refers. For example, when `removeComment` is applied with a probability of 10 % to each comment that occurs within the code snippet. The exact scope of changes is therefore uncertain. It can happen (especially with short methods such as getters and setters) that a method is not changed at all. For example, if a method only has a single comment and we use `removeComment`, the probability that the method will not be changed is 90 %.

In a second step, we applied stratified sampling [35] to distinguish between very simple methods such as getter and setter and more complex methods. In order to be able to compare the original methods with their modified variants, we only carried out the random sampling for the original methods and compared the rdh methods with these in a later step.

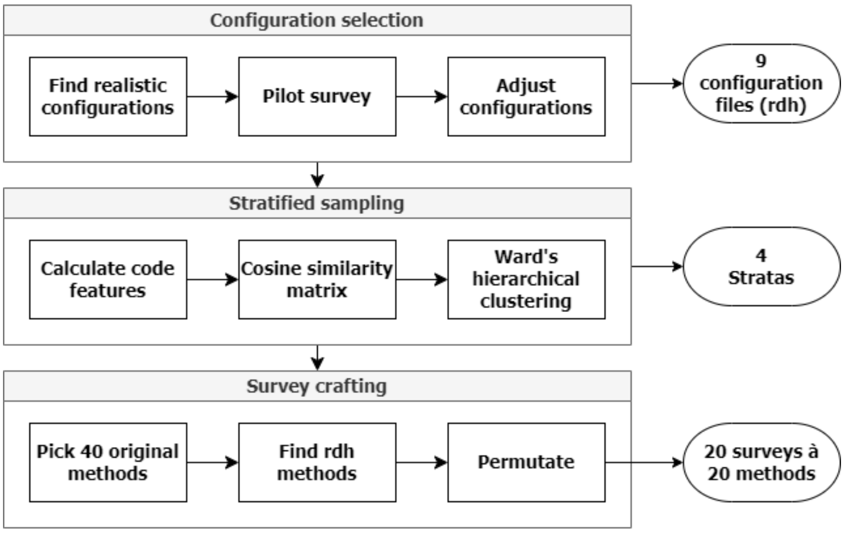


Figure 2.: Steps performed to craft questionnaires from the mined-and-modified dataset.

Thus, we first calculated features for the original code snippets. This was done using the tool of Scalabrino et al. [30]. A 110-dimensional feature vector was calculated for each original code snippet. Next we computed the cosine similarity metrix between all feature vectors using scikit[10]. Finally, using the fastcluster implementation [24] of Ward's hierarchical clustering we were able to cluster the methods into an arbitrary amount of clusters.

By comparing the merge distances in each step (see Figure 3), we found that a cluster size of 4 makes the most sense: the merge distance of 5 to 4 is small, so we should still perform this merge, but the merge distance of 4 to 3 is large, so it is better not to perform this merge. Also, 4 is the size with the last possibility for a small merge distance. Each of the clusters is one stratum of our stratified sampling. We manually assigned a name to each of the 4 strata (see Table 5).

In a third step, we crafted the questionnaires from the strata. We decided to provide all 10 previously mentioned configurations for each original method, as we want to compare the original methods with their rdh variants. We have a survey capacity of 400 code snippets (see Section 4.1). Therefore, the capacity for each rdh variant is $400/10 = 40$ code snippets. We start by selecting 40 original code snippets and then add all their rdh variants. We opted for a random sample

[10] `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html`, accessed: 2024-02-20

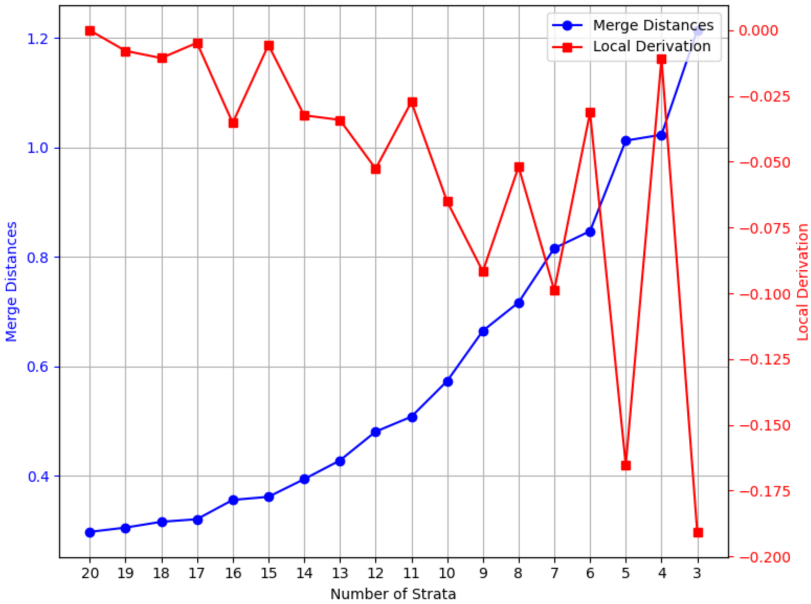


Figure 3.: Merge distances and local derivation for number of strata.

within the strata. However, we distributed the 40 snippets across the strata as shown in Table 5.

This decision was made due to the relatively high frequency of methods that do not differ from their original methods (see Figure 4). Another reason for this decision is that particularly simple methods are rather uninteresting for the classification of readability, as they are often generated (e.g. by IDEs) and usually follow a straightforward pattern.

After selecting the 40 original methods, we next selected all $9 * 40$ rdh variants that belong to the original methods. This was mostly done automatically based on the names of the original methods and the names of the rdh variant methods. However, if the method was renamed at an earlier stage due to the method renaming heuristic, the new method did no longer match the original method, in which case we had to match them manually.

Once we collected all 400 methods, we distributed them across the 20 questionnaires, each with 20 methods. In order not to manipulate the raters, we decided that a variant of each method could only appear once in each questionnaire. For example, if the original method is in questionnaire 1, the removeComment variant (or another variant of the same method) must not be included in the same questionnaire.

Table 5.: Computed strata, manually assigned names based on the methods within and distribution for survey creation.

Stratum	Method Type	Percentage	Count
Stratum 0	Simple methods	10 %	4
Stratum 1	Complex methods	40 %	16
Stratum 2	Magic number methods	10 %	4
Stratum 3	Medium complex methods	40 %	16
Total		**100 %**	**40**

For this purpose, we created four permutation matrices with 10 snippets each. The number 10 was chosen because it is possible to distribute 10 snippets, each with 10 variants, across at least 10 survey questionnaires without violating our condition. By combining two 10-permutation matrices, we were able to create 10 survey questionnaires with 20 code snippets each. An implication of this approach is that each questionnaire contains each variant kind exactly twice. By doing this twice, we obtain the desired distribution of 20 questionnaires with 20 methods each. Our condition applies: There is only one variant of the same method in each questionnaire.

Finally, the methods of each questionnaire were randomly shuffled within itself. This was done to minimize the impact of the position of a snippet or variant within a survey on the rating.

3.6. READABILITY CLASSIFICATION MODEL

Next we investigate whether it is possible to score a higher accuracy as the towards model in classifying code readability with the mined-and-modified dataset.

Therefore we created our own implementation of the towards model [22] of Mi et al. using Keras[11]. The model consists of three layers. A code representation layer, a feature extraction layer and a code readability classification layer. You can find an overview of the model architecture in Figure 5.

The input for the model is a labeled dataset consisting of code snippets and whether they are badly or well readable. In the code representation layer three different code representations are generated from each code snippet: A visual, a semantic and a structural representation.

[11] `https://keras.io/about/`, accessed: 2024-02-20

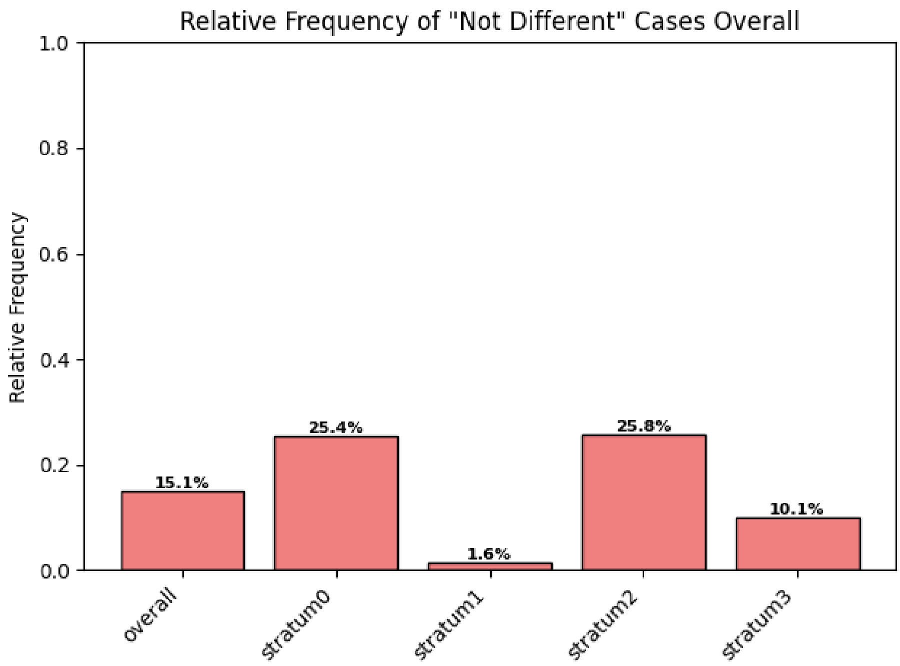


Figure 4.: Frequency of the case that a rdh-method is not different from its original method.

For the visual representation the syntax of the code is highlighted. Therefore Mi et al. assigned each type of syntactic element a color (see Table 6). Instead of highlighting the words in the respective color, as done by an IDE, the words are replaced by color blocks instead (see Figure 6b). Mi et al. used Eclipse[12] to highlight the code snippets and then took screenshots to obtain a RGB Matrix [22].

For the semantic representation we split the code into tokens (e.g., keywords and operators) and use BERT [**devlin2018bert**] to embed each token as a vector [22].

For the structural representation we split the code into characters and convert each into its corresponding ASCII value to obtain a ASCII matrix [22].

The model itself takes the three representations as input. We perform feature extraction on the RGB matrix and the ASCII matrix using a CNN for each. Each of the CNNs consists of multiple convolution and max pooling layers and a single flatten layer [22].

[12] `https://www.eclipse.org/`, accessed: 2024-03-02

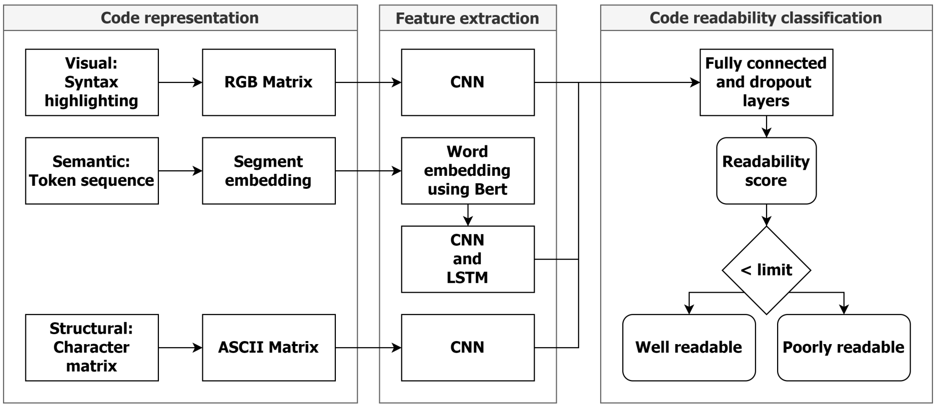


Figure 5.: The architecture of the towards model [22] of Mi et al.

Table 6.: The color encoding used by the visual component of the towards model [22].

Element	Color	Hex Code
Comment		#006200
Keyword		#fa0200
Identifier		#01ffff
Literal		#01ffff
Punctuation		#fefa01
Operator		#fefa01
Generics		#fefa01
Whitespace		#ffffff

On the token embedding the model performs feature extraction using a BERT embedding layer, convolution layers, a max pooling layer and a BiLSTM [22].

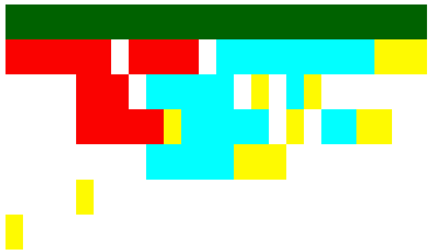
After extracting the feature from the three individual representations the output is merged and used as a input for the final step: code readability classification. In this step the model consists of multiple fully-connected layers and a dropout layer. The output is a single value, namely the readability score. If the score is above a certain limit, we classify the input as well readable, otherwise it is poorly readable [22].

We implemented this model as described by Mi et al. [22] with a few adjustments: In contrast to the publicly available code of Mi et al.[13], our model includes (batch) encoders required for the model to be trained on new data and to perform the

[13] `https://github.com/swy0601/Readability-Features`, accessed: 2024-02-20

```java
// A method for counting
public void getNumber(){
        int count = 0;
        while(count < 10){
                count++;
        }
}
```

(a) An exemplary Java code snippet.



(b) The visual encoding of the code snippet in Figure 6a.

Figure 6.: A code snippet and its visual encoding.

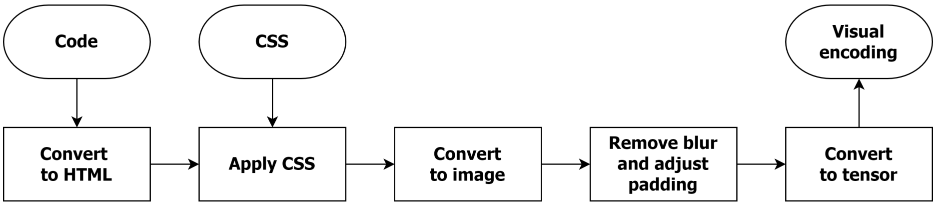


Figure 7.: The steps to automatically, visually encode code.

prediction task for new code snippets. In addition, our model supports fine-tuning by freezing certain layers as well as storing intermediate results, such as the encoded dataset. During evaluation, the model returns the evaluation statistics in form of a JSON file.

A larger adjustment was made when it comes to image encoding. To automate the generation of visual encodings we propose a different approach leading to a similar result. You can find an overview of our approach in Figure 7.

In a first step we use Imgkit[14] to convert the code to HTML. Thereby, a HTML class is assigned to each type of syntactic element. Next, we apply syntax highlighting using a CSS style sheet (see Appendix IV). In a third step we use pygments[15] to convert the HTML with the applied CSS to an image. We use pillow[16] to remove blur and adjust the padding of the image. Finally, the image is loaded using opencv-python[17] which allows us to convert the image to an RGB tensor that is suitable as a model input.

[14] `https://pypi.org/project/imgkit/`, accessed: 2024-03-02
[15] `https://pygments.org/`, accessed: 2024-03-02
[16] `https://pypi.org/project/pillow/`, accessed: 2024-03-02
[17] `https://pypi.org/project/opencv-python/`, accessed: 2024-03-02

```
1  [CLS] / * *
2  * This method determines the sign of a given number and prints a
   ↪  corresponding message .
3  *
4  * @ para m number The input number to be checked .
5  * /
6  public static void print S ign ( in t number ) {
7  if ( number > 0 ) {
8  System . out . print ln ( " Number is positive " ) ;
9  } else if ( number < 0 ) {
10 System . out . print ln ( " Number is negative " ) ;
11 } else {
12 System . out . print [SEP]
```

Listing 3.: A Java method that was encoded and decoded using Bert-base-cased with a limit of 100. Space characters separate the tokens. Newlines are preserved for readability.

During implementation, we encountered the following potential problem with the model: The token length for the BERT encoding (BERT-base-cased[18]) used in the model is 100. What implications does this have? To answer this question, we first take a look at what a token comprises. In addition to special tokens that mark the beginning *[CLS]* and the end *[SEP]* of the input, each word represent a token. Furthermore, each special character (such as a slash (/), parentheses ((,), {, }), a comma (,), a semicolon (;), arithmetic signs (=, <, >) and many more) is also represented by its own token. Java identifiers are split into several tokens according to the convention of upper and lower case. If a identifier is not present in the model's vocabulary, the tokenizer splits it further into sub-identifiers or characters that are in the vocabulary. For example in Listing 3, Line 6, the word "int" is split into the tokens "in" and "t" as "int" is not part of the vocabulary of BERT-base-cased.

Consider the method from Listing 1a. With a token limit of 100, the last encoded token is the last closing parenthesis in line 9. Everything from line 10 onwards is not encoded, which means that the information is lost for the semantic part of the model. To put it in other words: The model of Mi et al. only considers the first few lines of code snippets in its semantic component.

The visual and structural encoders have similar limitations, but to a much smaller extent. The structural encoder encodes the first 50 lines of each code snippet and the visual encoder encodes the first 43 lines. While the constraints for these two encoders seem to be long enough to fully capture most code snippets, the semantic encoder seems to be too limited to do so.

[18] `https://huggingface.co/google-bert/bert-base-cased`, accessed: 2024-02-20

Although we want to note these limitations, we will keep them to allow a fair comparison of the datasets.

Our code is publicly available on GitHub[19].

4. EVALUATION

We tested the mined-and-modified dataset in two ways. On the one hand, we conducted a user study. On the other hand, we evaluated the impact of using the dataset for the towards model. In detail, we answer the following questions with both experiments:

We are conducting a user study to determine the quality of the new data set. In detail the aim of the user study is to answer the following key questions:

1. Does the well-readable-assumption (Assumption 1) hold?
2. Does the poorly-readable-assumption (Assumption 2) hold?

Our assumptions are as follows:

Assumption 1 **(well-readable-assumption)** The selected repositories contain mostly well readable code.

Assumption 2 **(poorly-readable-assumption)** After applying Readability Decreasing Heuristics, the code is poorly readable.

Therefore, we come up with the following research questions:

Research Question 1: ***(mined-well)*** *Can automatically mined code be assumed to be well readable?*

In our new approach for generating training data, we assume that the code from repositories is well readable under certain conditions (Assumption 1). We want to check whether that holds. To answer this question we will use the results of the user study.

Research Question 2: ***(modify-poor)*** *Can poorly readable code be generated from well readable code?*

It is not sufficient to have only well readable code for training a classifier. We also need poorly readable code. Therefore, we will try to generate such code from the well readable code. We will investigate whether this is possible in

[19] `https://github.com/LuKr02011`, accessed: TODO

principle, and whether the proposed RDH tool (see subsection 3.4) is able to achieve this.

As the name already suggests, the applied transformations on the source code are only heuristics. To answer, whether the generated code is poorly readable (Assumption 2) we will utilize the results of the user study.

Research Question 3: ***(new-data)*** *To what extent can the new data improve existing code readability classification models?*

It was shown that Deep Learning models get better the more training data is available [12]. This holds under the assumption that the quality of the data is the same or at least similar. We want to check if the quality of our new data is sufficient for improving the deep learning-based readability classifier of Mi et al. [22]. Therefore we will train their proposed model with combinations of the merged and the mined-and-modified dataset and compare the evaluation statistics.

4.1. SURVEY

The results of our survey are divided into two parts: The results of the pilot survey (see Section 4.1), which were used to improve the main survey pre-launch, and the results of the main survey (see Section 4.1), which were used to answer our research questions (RQ1 and RQ2) and to craft our dataset.

PILOT SURVEY

1. Experimental setup: We manually sampled 20 code snippets across all strata but mainly from statum 1, due to reasons mentioned in subsection 3.5. From January 6 to 14, 2024, ten people took part in the survey. Eight of them were students and two of them worked in industry. All of them had knowledge of computer science. They were were not paid. Additionally to rating 20 code snippets the participants were also asked to answer additional questions to provide feedback about the survey:

1. *Short answer*: How long did it take you to complete the survey?
2. *Single choice (1 (very unclear) to 5 (very clear)):* How clear was your task?
3. *Long answer:* What problems were with the task? If there were none, leave blank.
4. *Long answer:* What problems were there with the survey tool? If there were none, leave blank.

5. *Long answer:* What improvements would you make to the survey? If none, leave blank.
6. *Long answer:* you have any other feedback? If none, leave blank.

The participants answers can be found in appendix I.

The feedback of the pilot survey was used in the following ways to prepare the Prolific study:

- To adjust the survey texts and questions
- To estimate how long completion of one questionnaire will take
- To adjust the RDH settings
- To discover problems with the survey tool
- To discover fundamental problems with the dataset

2. Threats: The results do not generalize. We did not sample the Java snippets for rating in a specific, (semi-) automated way, so there is a selection bias. The participants came from a private environment. This leads to a possible selection bias. We adjusted the surveys instructions, explanations and questions afterwards, which might influence the ratings of the participants. However, we did not use the results from this survey to evaluate our dataset or the generation approach, as the intention of the survey was rather to prepare for the main survey.

3. Results: The results of the pilot survey were analyzed with regard to three points: the time it took to complete the survey, the feedback from the participants and the ratings of the selected code snippets.

Completion Time: In Figure 8 we find the time it took the participants to complete the pilot survey and thus to rate 20 code snippets according to their readability. The fastest participant completed the survey in 7 minutes and 35 seconds, while the slowest participant took 18 minutes. Both the average value and the mean value are around 12 minutes. The boxplot (see Figure 8) shows that the times are close together and there are no outliers in the time taken. We suspect that the participants in the pilot survey put more effort into completing the survey as we know them personally. Other participants may not make as much effort, so we set the time estimation for a questionnaire below average at 10 minutes.

Participant Feedback: All feedback of the participants regarding the pilot survey is listed in Appendix I. Most of the problems that occurred were due to the survey tool (e.g.: "I also felt that I should use the drop-down menu at the top

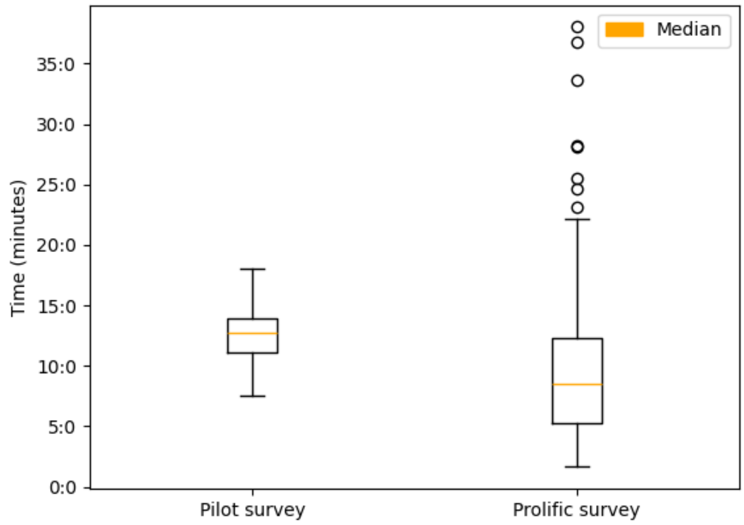


Figure 8.: Time required to complete a questionnaire.

left."). Some of the feedback was regarding fully qualified class names, such as "Java.io.InputStream". We found that the pretty printer of the RDH tool specified each imported method or class with its fully specified classifier. For example, instead of "InputStream", "Java.io.InputStream" was written in the code snippets of the RDHs. This gave the participants the feeling that the code was not written by a human and drastically reduced readability. Therefore we adapted the RDH tool to print the shorter name.

Snippet Ratings: The evaluation of the last places of Table 7, such as *1.2* for *stratum 1 - all*, suggests that these code snippets were particularly poorly readable. The assumption arose that they could be so poorly legible that it can no longer be assumed that they were written by a human. We adjusted the RDH configurations. The rating for the last places (see Table 7), such as *1.2* for *Stratum 1 - all*, suggest that these code snippets were particularly poorly readable. Due to this, we re-examined the configurations of all RDHs and found that some of them are configured too strongly. This not only impairs their readability, but also makes them look as if they were not written by human hands. Thus, we took another close look at the RDH configurations and found that some of them are over-configured. This not only affects their readability, but also makes them look as if they were not written by human hands. We have therefore reduced the probabilities for these RDHs.

Table 7.: Mean score ratings for the pilot survey.

Stratum	RDH	Score
Stratum 3	methods	4.6
Stratum 0	tabs_few	4.3
Stratum 2	tabs_few	3.8
Stratum 1	methods	3.7
Stratum 2	methods	3.7
Stratum 3	newlines_many	3.3
Stratum 1	comments_remove	3.1
Stratum 0	spaces_few	3.0
Stratum 1	all_weak_3	3.0
Stratum 1	newlines_many	2.9
Stratum 1	spaces_few	2.6
Stratum 1	misc	2.4
Stratum 2	newlines_few	2.4
Stratum 1	tabs_few	2.2
Stratum 1	tabs_many	2.2
Stratum 1	spaces_many	2.1
Stratum 1	newlines_few	1.7
Stratum 3	tabs_many	1.7
Stratum 1	all_weak	1.3
Stratum 1	all	1.2

We adjusted the RDH configurations and the survey tool according to the feedback. Then the Prolific study was launched.

PROLIFIC SURVEY

In this section we summarize the results of the main study conducted via Prolific.

1. Experimental setup: The survey was conducted using Tien Duc Nguyen's Code Annotation Tool (see Figure 9) along with the platform Prolific[20] for the recruitment and payment of participants. The survey was conducted between 31 January and 7 February 2024. A total of 221 participants took part. Each of the 20 questionnaires was answered by 11 participants (similar to the survey of Scalabrino et al. [30]). In one survey, one more participant was assigned by mistake. We estimated the time to complete one questionnaire at 10 minutes (see subsubsection 4.1). Prolific set the maximum time allowed at 44 minutes.

[20] `https://app.prolific.com/`, accessed: 2024-02-21

Figure 9.: Tool for rating a code snippet from the perspective of a survey participant.

Participants who took longer received a time-out. This results in a margin of error of 29.55% at a confidence of 95% for an individual snippet. However, we aggregate over strata and multiple snippets later to reduce the margin of error. Each questionnaire consists of 20 code snippets. Consequently, 400 different code snippets are rated in total. The questionnaires were configured in a way that each participant could only take part in one of the questionnaires. You can find the texts for the survey in Appendix III. The questionnaires were crafted as described in Section 3.5.

The target population consists of Java programmers selected by Prolific. They may be students or work in industry. They can come from any country. Overall, there were no requirements other than familiarity with Java.

The internal research questions are as follows:

- Does the well-readable-assumption (Assumption 1) hold?

- Does the poorly-readable-assumption (Assumption 2) hold?

The results for these questions are equally important, and thus none of them is prioritized over the other. To answer them, the assumptions are considered as hypotheses along with the following associated null hypotheses:

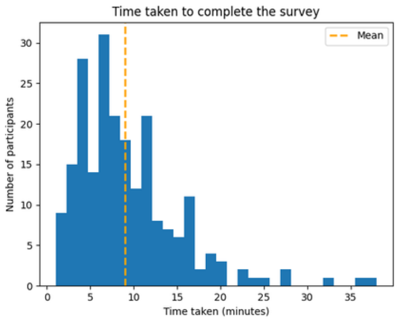
- For Assumption 1: The mined code (original) is on average not better readable than the code from previous studies.
- For Assumption 2: The readability of code does not significantly deteriorate compared to the original code snippet.

The survey neither contained demographic questions nor filter questions. Besides the readability questions, each user was asked the following dependent question: "How would you describe your familiarity with Java?". The user could answer within a five point Likert scale: expert (5), advanced (4), intermediate (3), beginner (2), novice (1).

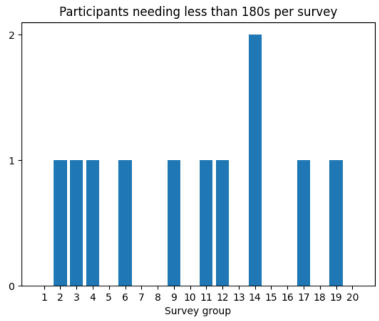
2. Threats: We identified the following threats:

- **Ill-defined Target Population:** Ensuring a well-defined target population is critical to the survey's quality. To mitigate this threat, we define our target population.
- **Sampling Method:** Stratified Sampling is well-defined and proven in practice. The approach ensures that our sample represents all parts of the population under investigation. This is improving the survey's external validity.
- **Insufficient Responses for Drawing Conclusions:** To prevent drawing conclusions from an insufficient number of responses, we scale our survey to an appropriate size. This guarantees that we collect a substantial volume of responses, allowing for robust statistical analysis.
- **Piece-work Effect:** Survey participants are paid for taking part and completing a questionnaire. However, they receive the same amount of money regardless of their speed. Therefore, they receive more pay per minute if they hurry. This could have an impact on the accuracy with which they scored the code snippets. A comparison between the time required by a participant for a pilot questionnaire and a Prolific questionnaire (see Figure 8) supports this suggestion. Especially the ratings of participants requiring less than 3 minutes (see Figure 10b) to complete a questionnaire could have a negative impact on validity.

3. Results: An overview of the time required by the participants can be found in the Figure 8 and Figure 10a. The fastest participant completed the survey in



(a) Time required by participants to complete the survey.



(b) Participants per questionnaire requiring less than 3 minutes.

Figure 10.: Time analysis of participants completing the Prolific survey.

1 minute and 39 seconds, while the slowest participant took about 38 minutes. The average time is 9 minutes and 45 seconds. The median time is 8 minutes and 30 seconds. The boxplot (see Figure 8) shows that the times are not as close together as for the pilot survey. There are a couple of outliers.

The participants' familiarity with Java is shown in Figure 11. According to their own estimation, 44.8 % of the participants are experts with Java. Another 44.8 % of the participants are either advanced or intermediate. This high level of familiarity with Java suggests that the quality of the evaluations received is also high.

The ratings for each RDH for all strata combined can be found in Figure 12a and Figure 12b. In line with our expectation, we see that the ratings for `none` and `methods` are almost the same. There is no significant difference between the two, as a Mann-Whitney U test confirmed. The probability that the difference between `none` and `methods` is due to random variation is 92 %. Therefore, we are sure that other differences are actually caused by the RDHs and not by the pretty-printer.

Figure 12a shows that the mean value of our original methods is 3.68. The mean score of all rated code snippets in the merged dataset is 3.45. The difference of 0.23 is significant.

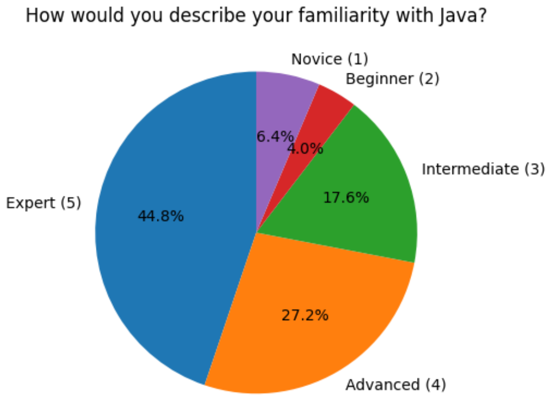


Figure 11.: Familiarity of Prolific survey participants with Java.

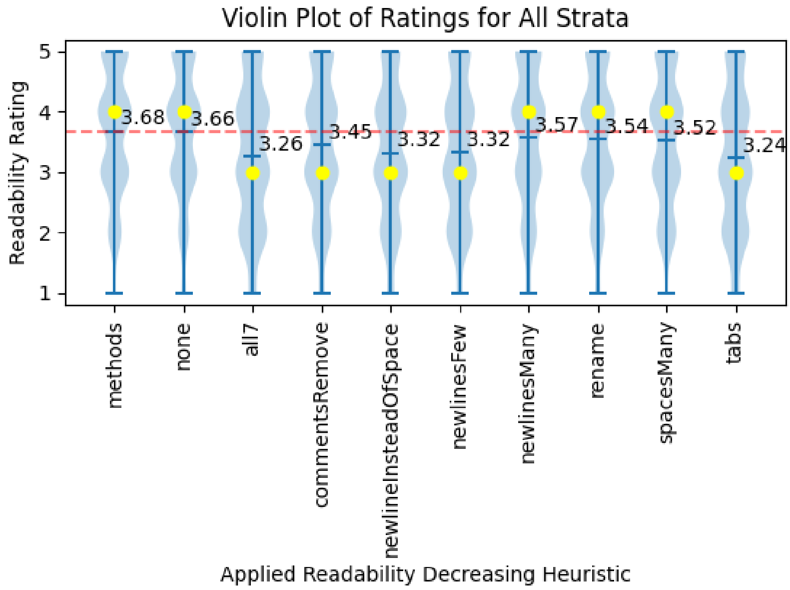
Summary (RQ1 - mined-well):

The mean score for the `original` methods of the mined-and-modified dataset (3.68) is significantly larger than the mean score for all ratings in the merged dataset (3.45). Therefore we reject the null hypothesis and conclude that well readable assumption (Assumption 1) holds.

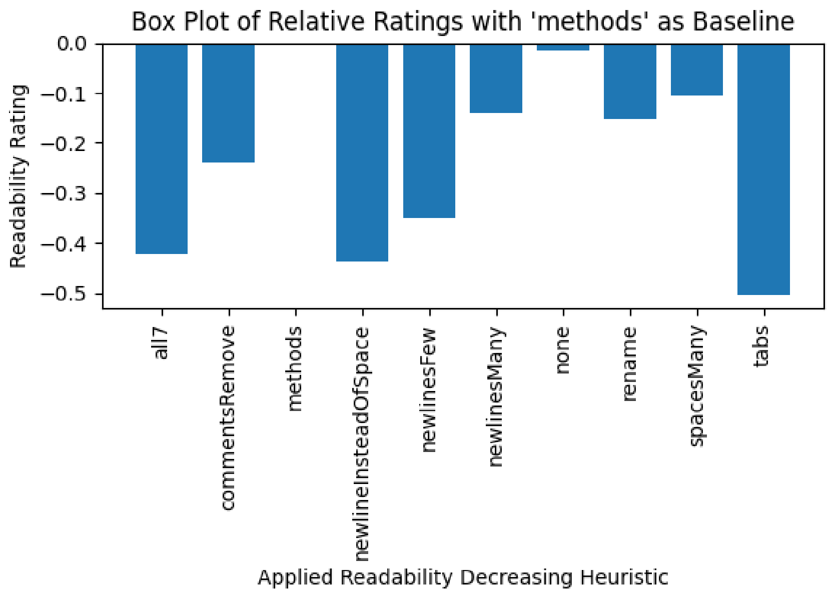
We analyzed whether the difference in ratings between the different RDHs is statistically significant. To do this, we used the Mann-Whitney U test to compare the ratings for all snippets for an RDH with the corresponding `none` snippets. The results can be found in Table 8. If we compare the RDHs with the `none` methods, we can be sure that the scores of all methods except newlines many and rename are indeed statistically different from the scores of `none`.

If we consider binary readability classification and split the data into two classes (poorly readable: 1,2; well readable: 3-5) all but `none` and `rename` are statistically different from each other. This includes `newlines many` (TODO: Add p Values) for which we could not confirm statistical difference without binary classification.

Besides `none`, this leaves only `rename` where we can not confirm statistical significance. Overall, we showed that the heuristics actually reduce the readability of source code.



(a) Absolute survey ratings for each rdh and all strata.



(b) Relative survey ratings for each rdh and all strata compared to original.

Figure 12.: Survey ratings for each rdh and all strata.

Table 8.: Mann-Whitney U test results of each rdh against `none`. When p is smaller than 5.00×10^{-2} we conclude that the difference is significant.

Comparison	p
None - Methods	9.22×10^{-1}
None - Newlines Few	5.23×10^{-6}
None - Spaces Many	4.07×10^{-2}
None - Newlines Many	3.00×10^{-1}
None - Comments Remove	3.64×10^{-3}
None - Rename	9.90×10^{-2}
None - Newline Instead Of Space	4.57×10^{-6}
None - Tabs	3.06×10^{-8}
None - All7	1.80×10^{-7}

Summary (RQ2 - modify-poor):

All of the 7 heuristics but `rename` decrease readability by a significant extend compared to `none`. We estimate the readability decrease for a certain probability of a certain type as can be seen in Figure 12b. We reject the null hypothesis and conclude that the poorly readable assumption (Assumption 2) holds

4.2. MODEL TRAINING RESULTS

We aim to investigate the following things:

1. **Model evaluation:** To confirm that our implementation of the model scores similar accuracy as the original one of Mi et al. [22], we trained and evaluated the model on the merged dataset (merged-merged).
2. **Internal evaluation:** To investigate how effectively the model captures the readability aspects of the mined-and-modified dataset, we train and evaluate it on this dataset (mam-mam). By doing so, we examine also how effectively the model captures the differences between the `original` and `all7` methods which are modified with our RDHs.
3. **Cross evaluation:** To assess how effective the mined-and-modified dataset is for predicting readability, we train the model on this dataset and then evaluate its performance on the merged dataset (mam-merged). For advanced insights, we also train the model on the merged dataset and

Table 9.: Performance of different dataset configurations for the same model. *mam* stands for the mined-and-modified dataset.

Train	Eval	Acc	Prec	Rec	AUC	F1	MCC
merged	merged	84.7 %	87.7 %	82.3 %	85.0 %	83.7 %	70.4 %
mam	mam	91.8 %	92.3 %	91.3 %	91.8 %	91.7 %	83.6 %
mam	merged	61.9 %	63.6 %	63.6 %	63.6 %	63.6 %	23.6 %
merged	mam	53.8 %	52.6 %	77.8 %	65.2 %	62.8 %	08.7 %
mam-merged	merged	80.4 %	84.0 %	73.8 %	78.9 %	77.2 %	60.0 %

evaluate it on the merged dataset (merged-merged) and on the mined-and-modified dataset (merged-mam).

4. **Fine-tuning:** To assess what accuracy we can score in predicting readability, we investigate training on the mined-and-modified dataset and fine-tuning and evaluating on the merged dataset (mam-merged-merged).

We therefore used different combinations of training and evaluation datasets with the towards mode. The results can be found in Table 9. We used 10-fold cross-validation for evaluation.

Implementation evaluation. When we train and evaluate the model on the merged dataset, we obtain an accuracy of 84.7 %. This is very similar to the results of Mi et al. (84.7 % vs 85.3 %). The deviation of 0.6 % accuracy might be due to randomness of the splits for 10-fold cross-validation. We can confirm the results of the paper [22].

Internal evaluation. When we train the model on the mined-and-modified dataset and evaluate it, we obtain an average accuracy of 91.8 %. The towards model architecture is well suited to learn the structure of the mined-and-modified dataset. In particular it learns the differences between `original` and `all7` methods and thus it learns how to predict if a code snippet has been modified using RDHs.

Cross evaluation. When we train the model on the mined-and-modified dataset and evaluate it on the merged dataset (mam-merged), we get an accuracy of 61.9 %. This is 22.8 % worse than the accuracy we get when we train and evaluate the model on the merged dataset (merged-merged). When we train the model on the merged dataset and evaluate it on the mined-and-modified one (merged-mam), we get an accuracy of 53.8 %, which is close to the approximate accuracy of 50.0 % of a random classifier. If the scores for mam-merged, merged-merged and merged-mam would be similar, we would conclude that both datasets, the merged and the mined-and-modified one, address readability in general. Given, that this is not the case and knowing for both datasets that they at least address some aspects of readability we conclude that we address different aspects of readability.

Fine-tuning. We tried to fine-tune the merged dataset by freezing different layers of the model trained with the mined-and-modified dataset. During evaluation we achieved the best results when freezing the input layers as well as the first convolution and pooling layer of all encoders. However, when evaluated on the merged dataset, the performance is still worse than the merged-merged variant. One explanation for this could be that the model is too small to be effective with the larger amount of data. Introducing more or bigger layers so that the model can store more features internally could lead to an improvement. However, this is not part of this work, in which we mainly focus on a new dataset (generation approach).

> Summary (RQ3 - new-data):
>
> When trained and evaluated on the mined-and-modified dataset we achieve an accuracy of 91.8 %. When evaluated on the merged dataset, the model trained with the mined-and-modified dataset achieves an accuracy of 61.9 %. When we train the model with merged dataset 84.7 % is achieved. We concluded that the mined-and-modified dataset captures different aspects of readability. We did not outperform the model trained only on the merged dataset.

We also trained the model with a random sample of 210 data points to gain insight into what a change in training size might do. As we can see, the model has similar metrics to the all-all model. If we now compare the accuracy of the new-new compared to the merged-merged model, we see that with a larger dataset an improvement of about 8 % is possible. Similar results are suggested by previous research on data augmentation, where an accuracy of 87.3 % was achieved Mi et al. This emphasizes the importance of finding new ways of generating data for readability classification.

5. DISCUSSION

The main drawback of our approach is that we rely on estimations to create the mined-and-modified dataset. The score labels of our code snippets are rough estimations and not exact values. Accurate ratings would require manual review of 69k code snippets by human annotators and is therefore not feasible. However, for two class classification this does not matter.

A thread related to the study could be the sampling approach used. While we argue that we avoided spending resources on labeling rdh data that is likely not different from the original methods or rather uninteresting, this might also introduce statistical errors to our survey results.

When comparing the model performance trained on the new and merged data set, it should be noted that the merged data set is small. Consequently, comparisons between classifiers trained on the merged dataset may be unreliable [22].

6. CONCLUSIONS

Recent research in the field of code readability classification has mainly focused on various deep learning model architectures to further improve accuracy. Little attention is paid to the fact that only 421 labeled code snippets are available to train these models. We introduced a novel approach to generate data, with which we created a dataset of 69k code snippets. Although our results show that the dataset does not have the same quality as previous data, it still captures the readability of code and could accordingly help to improve code classification in future research.

The new approach for generating data has an advantage that is not yet used in this work: For the first time, it is possible to generate a dataset with one well readable and a second, less readable and functionally equivalent code snippet. This could be used to train various models, including transformers. Such a transformer could take the code as input and improve its readability. We suspect that such a tool could be of great benefit to programmers.

A current limitation of the mined-and-modified dataset is that it only works for Java code. A suggestion for future work is to overcome this limitation by extending the tool for other programming languages. This is not trivial, as one has to adapt the Readability Decreasing Heuristics to work with another language. Furthermore, a general tool that works for all languages will be difficult if not impossible.

As Mi et al. suggested, another useful representation for code readability studies could be the syntax tree representation of code [17]. One could try to improve the performance of the towards model by adding another representation encoding extractor for Java code that automatically extracts the abstract syntax tree of the code.

An important aspect of the readability of code is the naming. For the scope of methods, the method names are the most important part. Therefore, the towards model could be improved by adding a component that explicitly takes into account how well a method name matches its body. This component might be similar to Code2vec [3].

Further research could also consist of finding and evaluating other encodings that represents the code in a different way.

Another way to improve existing code readability classifiers, such as the towards model, could be to develop a different structure for some layers of the model. We suggest increasing the size and depth of the layers so that the mined-and-modified dataset can be made useful. Alternatively, a completely different model architecture could be developed.

The heuristics described in this work are only part of the possible heuristics that could be developed. Additional heuristics could further improve the diversity of poorly readable code. This could increase the number of internal features that a model can learn, which in turn could increase the accuracy of the model.

In summary, there are many opportunities to further investigate and thus most likely improve the classification of code readability. Our new data set and the generation approach could be useful here.

Bibliography

[1] Krishan K Aggarwal, Yogesh Singh and Jitender Kumar Chhabra. 'An integrated measure of software maintainability'. In: *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No. 02CH37318)*. IEEE. 2002, pp. 235–241.

[2] Miltiadis Allamanis, Hao Peng and Charles Sutton. 'A Convolutional Attention Network for Extreme Summarization of Source Code'. In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 2091–2100. URL: `http://proceedings.mlr.press/v48/allamanis16.html`.

[3] Uri Alon et al. 'code2vec: Learning distributed representations of code'. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[4] Kent Beck. *Implementation patterns*. Pearson Education, 2007.

[5] Barry Boehm and Victor R Basili. 'Defect reduction top 10 list'. In: *Computer* 34.1 (2001), pp. 135–137.

[6] Frederick Brooks and H Kugler. *No silver bullet*. April, 1987.

[7] Raymond PL Buse and Westley R Weimer. 'Learning a metric for code readability'. In: *IEEE Transactions on software engineering* 36.4 (2009), pp. 546–558.

[8] Sangchul Choi, Sooyong Park et al. 'Metric and tool support for instant feedback of source code readability'. In: *Tehnički vjesnik* 27.1 (2020), pp. 221–228.

[9] Lionel E Deimel Jr. 'The uses of program reading'. In: *ACM SIGCSE Bulletin* 17.2 (1985), pp. 5–14.

[10] Jonathan Dorn. 'A General Software Readability Model'. In: 2012.

[11] Sarah Fakhoury et al. 'Improving source code readability: Theory and practice'. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 2–12.

[12] Joel Hestness et al. 'Deep learning scaling is predictable, empirically'. In: *ArXiv preprint* abs/1712.00409 (2017). URL: `https://arxiv.org/abs/1712.00409`.

[13] Rensis Likert. 'A technique for the measurement of attitudes.' In: *Archives of psychology* (1932).

[14] Benjamin Loriot, Fernanda Madeiral and Martin Monperrus. 'Styler: learning formatting conventions to repair Checkstyle violations'. In: *Empirical Software Engineering* 27.6 (2022), p. 149.

[15] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[16] Qing Mi. 'Rank Learning-Based Code Readability Assessment with Siamese Neural Networks'. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–2.

[17] Qing Mi et al. 'A graph-based code representation method to improve code readability classification'. In: *Empirical Software Engineering* 28.4 (2023), p. 87.

[18] Qing Mi et al. 'An enhanced data augmentation approach to support multi-class code readability classification'. In: *International conference on software engineering and knowledge engineering*. 2022.

[19] Qing Mi et al. 'An inception architecture-based model for improving code readability classification'. In: *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. 2018, pp. 139–144.

[20] Qing Mi et al. 'Improving code readability classification using convolutional neural networks'. In: *Information and Software Technology* 104 (2018), pp. 60–71.

[21] Qing Mi et al. 'The effectiveness of data augmentation in code readability classification'. In: *Information and Software Technology* 129 (2021), p. 106378.

[22] Qing Mi et al. 'Towards using visual, semantic and structural features to improve code readability classification'. In: *Journal of Systems and Software* 193 (2022), p. 111454.

[23] Qing Mi et al. 'What makes a readable code? A causal analysis method'. In: *Software: Practice and Experience* 53.6 (2023), pp. 1391–1409.

[24] Daniel Müllner. 'fastcluster: Fast hierarchical, agglomerative clustering routines for R and Python'. In: *Journal of Statistical Software* 53 (2013), pp. 1–18.

[25] Delano Oliveira et al. 'Evaluating code readability and legibility: An examination of human-centric studies'. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2020, pp. 348–359.

[26] Renaud Pawlak et al. 'Spoon: A library for implementing analyses and transformations of java source code'. In: *Software: Practice and Experience* 46.9 (2016), pp. 1155–1179.

[27] Daryl Posnett, Abram Hindle and Premkumar Devanbu. 'A simpler model of software readability'. In: *Proceedings of the 8th working conference on mining software repositories*. 2011, pp. 73–82.

[28] Talita Vieira Ribeiro and Guilherme Horta Travassos. 'Attributes influencing the reading and comprehension of source code–discussing contradictory evidence'. In: *CLEI Electronic Journal* 21.1 (2018), pp. 5–1.

[29] Spencer Rugaber. 'The use of domain knowledge in program understanding'. In: *Annals of Software Engineering* 9.1-4 (2000), pp. 143–192.

[30] Simone Scalabrino et al. 'A comprehensive model for code readability'. In: *Journal of Software: Evolution and Process* 30.6 (2018), e1958.

[31] Simone Scalabrino et al. 'Automatically assessing code understandability: How far are we?' In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 417–427.

[32] Milan Segedinac et al. 'Assessing code readability in Python programming courses using eye-tracking'. In: *Computer Applications in Engineering Education* 32.1 (2024), e22685.

[33] Shashank Sharma and Sumit Srivastava. 'EGAN: An Effective Code Readability Classification using Ensemble Generative Adversarial Networks'. In: *2020 International Conference on Computation, Automation and Knowledge Management (ICCAKM)*. IEEE. 2020, pp. 312–316.

[34] Yahya Tashtoush et al. 'Impact of programming features on code readability'. In: (2013).

[35] Steven K Thompson. *Sampling*. Vol. 755. John Wiley & Sons, 2012.

[36] Antonio Vitale et al. 'Using Deep Learning to Automatically Improve Code Readability'. In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2023, pp. 573–584.

[37] Greg Wilson and Andy Oram. *Beautiful code: Leading programmers explain how they think*. " O'Reilly Media, Inc.", 2007.

[38] Michihiro Yasunaga and Percy Liang. 'Graph-based, Self-Supervised Program Repair from Diagnostic Feedback'. In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 10799–10808. URL: `http://proceedings.mlr.press/v119/yasunaga20a.html`.

I. PILOT SURVEY FEEDBACK

How clear was your task?

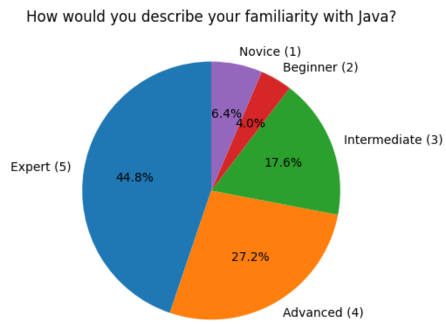


Figure 13.: TODO: Replace

What problems were with the task? If there were none, leave blank.

- Did at first not know where to rate the code.
- I was confused about the textfield for the comments because I only remembered that we should rate the code snippets, not that we have to make comments. Since I was not able to navigate back to the task description, I did not know what to do with them.
- Für einen Anfänger mit sehr wenig Java Erfahrung ist meiner Meinung nach der Code zu kompliziert.
- In the first place, I didn't really understand what readability meant. But after slide 3 or 4, I understood what this was about.
- I found it difficult to categorize the first examples because you don't know what's still to come. For example, what the least readable code is.

What problems were there with the survey tool? If there were none, leave blank.

- Mobile is not easy to use because of the scrolling needed to complete the survey.
- First, I needed to figure out how this tool works and that the rating is done with the stars below. I thought I should write my rating as a comment in

the comment field below. After number 20, I didn't know whether I could close the survey or not.

- I also thought that I should use the drop-down menu on the upper left.
- It is sometimes necessary to swipe horizontally to see all of the code, which is a bit inconvenient.
- Für einen Anfänger ist das Tool meiner Meinung nach nicht geeignet. Der Code ist zu verschachtelt und teilweise unverständlich.
- After finishing the task, at least a message should be shown.
- I didn't understand what the button at the top left meant, where you could select the programming language. There were too many fonts to choose. I also wasn't sure whether to write a comment or not. It wasn't described at the beginning.

What improvements would you make to the survey? If none, leave blank.

- Maybe one sentence that one should use the stars for the rating, then it would be clear. Also, the submit note after the last question could contain that one can close the survey now.
- I suggest making the task description accessible during the rating.
- Maybe the option to leave the survey when clicking to submit.
- Mehr Hilfestellung zum Lesen des Codes. Mehr Beschreibung oder ein zusätzliches Cheat Sheet mit Bedeutungen von Befehlen.
- I think it's a good idea to ask the participant at the beginning to explain what readability means for him.
- I would leave out the buttons described above. I was missing a scrollbar at the bottom of the code-window. A conclusion page with a message like "Thank you for your participation", "You're Done!" or other further information was missing, too.

Do you have any other feedback? If none, leave blank.

- There were drop downs for the programming language, but choosing another language did not change anything. It was a bit confusing that (almost?) all code snippets had very long imports within the code, which made them poorly readable.
- I spent the most time understanding methods with complete Java import names. (org.foo.bar.ClassName).

- GOOD LUCK

II. READABILITY DECREASING HEURISTICS CONFIGURATION FILE

```yaml
newline:
- 0.0  # Probability for no newline
- 1.0  # Probability for one newline
incTab:
- 0.0  # Probability for no tab
- 1.0  # Probability for one tab
decTab:
- 0.0  # Probability for no tab
- 1.0  # Probability for one tab
space:
- 0.0  # Probability for no space; Must be 0.0
- 1.0  # Probability for one space
newLineInsteadOfSpace: 0
spaceInsteadOfNewline: 0
incTabInsteadOfDecTab: 0
decTabInsteadOfIncTab: 0
renameVariable: 0
renameField: 0
renameMethod: 0
inlineMethod: 0
removeComment: 0
add0: 0
insertBraces: 0
starImport: 0
inlineField: 0
partiallyEvaluate: 0
```

III. PROLIFIC SURVEY TEXTS

On Prolific:

Readability of Java Code

We study the readability of Java source code. Therefore, please read Java methods and rate their readability on a scale from 1 (very unreadable) to 5 (very readable).

At the top of the tool:

Readability of Java Code

Read the Java methods and rate their readability on a scale from 1 (very unreadable) to 5 (very readable) using the stars below the code box. To navigate between methods, use the arrows above or below the code box. Make sure to rate each snippet.

Introduction page 1:

This study aims to investigate the readability of Java source code. In this survey, we will show you 20 Java methods. Please read the methods thoroughly and rate how readable you think they are. Before we begin, please answer the following question:

How would you describe your familiarity with Java?

1. Expert
2. Advanced
3. Intermediate
4. Beginner
5. Novice

Introduction Page 2:

Below is an example of the interface for displaying and rating the code. Use the stars below the code box for your rating. Please rate the readability on a scale from 1 (very unreadable) to 5 (very readable). At the top left, you can adjust the syntax highlighting and theme (dark/light) according to your preferences (optional). Comments are not available during this survey.

[EXAMPLE]

Introduction Page 3:

This survey should take about 10 minutes to complete. Now you are ready to go!

IV. TOWARDS MODEL - VISUAL ENCODING COLORS

The following CSS was used to generate the background colors for the visual encoding. You can find an overview over all tokens on the pygments homepage[21].

```css
/* Comment Styles */
.c, .ch, .cm, .cp, .cpf, .c1, .cs {
        background-color: #006200;
        color: #006200;
}
/* Keyword Styles */
.k, .kc, .kd, .kn, .kp, .kr, .kt {
        background-color: #fa0200;
        color: #fa0200;
}
/* Parentheses, Semicolon, Braces Styles */
.p, .o, .ow {
        background-color: #fefa01;
        color: #fefa01;
}
/* Whitespace Styles */
.w {
        background-color: #fff;
        color: #fff;
}
/* Names/Identifiers Styles */
.n, .na, .nb, .nc, .no, .nd, .ni, .ne, .nf, .nl, .nn, .nt, .nv {
        background-color: #01ffff;
        color: #01ffff;
}
/* Literals Styles */
.m, .mb, .mf, .mh, .mi, .mo, .s, .sa, .sb, .sc, .dl, .sd, .s2, .se, .sh, .si, .sx, .sr, .s1, .ss, .b, .bp, .f, .fm, .v, .vc, .vg, .vi, .vm, .i, .il {
        background-color: #01ffff;
        color: #01ffff;
}
/* Error Styles */
.err {
        background-color: #fff;
        color: #fff;
}
/* Generics Styles */
.g, .gd, .ge, .ges, .gr, .gh, .gi, .go, .gp, .gs, .gu, .gt {
        background-color: #fefa01;
        color: #fefa01;
}
```

[21] https://pygments.org/docs/tokens/, accessed: 2024-03-02