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

18th February 2024

Abstract

This work presents an innovative method for generating datasets intended for code readability classification in the context of a master's thesis. We offer a comprehensive overview of code readability, delving into existing classifiers that consider both manually crafted and automatically extracted code readability features. Furthermore, we summarize existing datasets of manually annotated Java code snippets.

The core contribution of this work lies in the introduction of an automatic data generation technique, alongside a dataset produced using this methodology. Our approach relies on the extraction and modification of code snippets sourced from public GitHub repositories. Notably, our dataset significantly surpasses the scale of any previously available dataset designed for readability classification.

To evaluate the newly generated dataset, we conducted a user survey and train a state of the art code readability classification model, both with and without the integration of the new dataset. This analysis aims to assess the quality of our new dataset and the effectiveness of the generation approach employed.

Contents

1 Introduction . . . 1
2 Background and related work . . . 2
2.1 Code Readability . . . 2
2.2 Classical calculation approaches . . . 4
2.3 Deep Learning based approaches . . . 5
2.4 Related work . . . 6
3 Dataset Generation Approach . . . 10
4 Readability Decreasing Heuristics . . . 11
5 Survey . . . 12
6 Readability Classification Model . . . 15
6.1 User study . . . 17
6.2 Comparing models . . . 17
6.3 Research questions . . . 18
7 Evaluation . . . 19
8 Discussion . . . 20
9 Conclusion . . . 20

1 INTRODUCTION

In the realm 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 [19, 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 [6, 9, 20, 4]. 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% [6, 19, 10, 8]. In recent years, deep learning based models are able to achieve an accuracy of up to 85.3% [16, 17]. However, these models do not capture what developers think of readability improvements [11]. This suggests that there is room for improvement in readability classification of source code.

In the following sections we will clarify what source code readability is. We will summarize background knowledge for both classical and deep learning based classification approaches. We will have a look at related work.

In section TODO we will explain our new dataset generation approach in detail. We will SUBSECTIONS

After introducing our new dataset we will evaluate it based on a user study and a state of the art readability classification model in section TODO. Based on the dataset, the user study and the model we will propose our research questions.

We will then answer our research questions based on results of the survey and model performance in section TODO.

We will show possible threads to the dataset generation approach and our evaluation and discuss them in section TODO.

In the last section we summarize our work, draw conclusions from it and propose future work.

2 BACKGROUND AND RELATED WORK

In the following sections you find an overview over background and related work regarding code readability and our dataset generation approach.

2.1 CODE READABILITY

To properly discuss readability, in particular readability of source code we first define this term. There are various differing definitions for code readability:

klare1964measurement defines readability in general as “the ease of understanding or comprehension due to the style of writing.”

Buse and Weimer states regarding code readability: "We define readability as a human judgment of how easy a text is to understand."

tashtoush2013impact combines multiple other aspects from various definitions to their definition of code readability:

1. Ratio between lines of code and number of commented lines
2. Writing to people not to computers

3. Making a code locally understandable without searching for declarations and definitions
4. Average number of right answers to a series of questions about a program in a given length of time

However, this results in a long and complex definition.

Scalabrino et al. says, that it is a subjective concept that is influenced by a number of factors, including the complexity of code, the usage of design concepts, the formatting of code, source code lexicon, and the visual aspects of the code.

Recent definitions of code readability are shorter, trying to focus on the key aspects. **oliveira2020evaluating** defines readability as "what makes a program easier or harder to read and apprehend by developers".

Also **mi2021effectiveness** summarizes code readability as "a human judgment of how easy a piece of source code is to understand". This is again close to the definition of Buse and Weimer.

However, there are various related terms to readability: Understandability, usability, reusability, complexity, and maintainability **tashtoush2013impact**.

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 [6, 5].

Previous definitions are close to understandability.

Scalabrino et al. defines aspects of understandability: "Complexity, usage of design concepts, formatting, source code lexicon, and visual aspects (e.g., syntax highlighting) have been widely recognized as elements that impact program understanding" [**martin2009clean**, **wilson2007beautiful**, **beck2007implementation**].

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

Based on Posnett et al., Scalabrino et al. says 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."

Readability is a human judgment of how easy a program source code is to read and comprehend [**buse2008evaluating**, **sedano2016code**]. It is concerned with the syntactic aspects of code, such as the use of meaningful variable names, consistent formatting, and clear commenting.

Understandability is the ability to grasp the meaning of a program source code and how it works [**oliveira2020evaluating**, 19]. It is concerned with the semantic aspects of code, such as the underlying logic and the use of design patterns.

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 [22].

Comparing the definitions of code readability in literature one can see, that there are some common aspects in most definitions. Those are: "ease/complexity of understanding/comprehension/apprehension", "human judgement" as well as the differentiation to understandability. Based on this, we come up with the following definition: Code Readability is a human judgment of the effort it takes to read and understand code.

Now that we have a grasp of what code readability refers to, let's take a brief look at the question, why code readability is important. In the domain of software development, the importance of code readability cannot be emphasized enough. Alongside understandability, it forms the basis for effective collaboration, comprehension, and maintenance of software systems [19, 1]. It is a critical aspect of software quality, significantly influencing the maintainability, reusability, portability, and reliability of the source code [**alawad2019empirical**, **sedano2016code**]. Poorly readable code increases the risk of introducing bugs [**mannan2018towards**, 21] and can lead to higher costs during subsequent software maintenance and development [**johnson2019empirical**]. On the other hand, readable code allows developers to identify and rectify bugs more easily [**mi2023graph**]. Recent studies indicate that developers spend nearly 58% of their time reading and comprehending source code [**tashtoush2013impact**, **sedano2016code**, **xia2017measuring**, 6, 9, 20, 4]. 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% [6, 19, 10, 8]. In recent years, deep learning based models are able to achieve an accuracy of up to 85.3% [16, 17]. However, these models do not capture what developers think of readability improvements [11]. This suggests that there is room for improvement in readability classification of source code.

2.2 CLASSICAL CALCULATION APPROACHES

A first estimation for source code readability was the percentage of comment lines over total code lines [1]. In the last years, researchers have proposed several more complex metrics and models for assessing code readability [6, 19, 10, 21].

```java
/**
* Logs the output of the specified process.
*
* @param p the process
* @throws IOException if an I/O problem occurs
*/
private static void logProcessOutput(Process p) throws IOException
{
        try (BufferedReader input = new BufferedReader(new
        ↪ InputStreamReader(p.getInputStream())))
        {
                StrBuilder builder = new StrBuilder();
                String line;
                while ((line = input.readLine()) != null)
                {
                        builder.appendln(line);
                }
                logger.info(builder.toString());
        }
}
```

Listing 1: An example for well readable code of the highly rated Cassandra GitHub repository

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 [21].

2.3 DEEP LEARNING BASED APPROACHES

More recent models use Deep Learning approaches in order to generate the features automatically. Those models have proven to be more accurate, achieving an accuracy of up to 85.3% [16, 17].

All the mentioned models were trained on the data of Buse, Dorn and Scalabrino consisting of in total 660 code snippets. The data was generated with surveys. They therefore asked developers several questions, including the question, how well readable the proposed source code is [6, 10, 21].

Fakhoury et al. showed based on readability improving commit analysis that these models do not capture what developers think of readability improvements. They therefore analyzed 548 GitHubhttps://github.com/2023-07-25 commits manually. They suggest considering other metrics such as incoming method calls or method name fitting [11].

```
private
        static
void
debug( Process
v1
)       throws IOException
{
        // Doo debug
        try (BufferedReader   b
        = new
        BufferedReader(
        new InputStreamReader(
        v1.getInputStream()
        )
        )
        )
        {
                StrBuilder b2=new StrBuilder();String v2;while
                ↪   (null!=(v2=input.readLine())){b2.appendln(v2);}
                ↪   // Doo stuff
                m.info(  builder.toString()
                );
        }
}
```

Listing 2: The same example as in listing 1 but modified to be poorly readable

2.4 RELATED WORK

Loriot et al. created a model that is able to fix Checkstyle[1] 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 [15]. A similar idea has been explored by Yasunaga and Liang [23]. 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[2] 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. As the projects have thousands of forks and stars and are widely used among software developers, they can be assumed to be of high quality [2].

[1]`https://checkstyle.org/`, accessed: 2023-07-25
[2]`https://github.com/`, accessed: 2023-07-25

segedinac2024assessing introduces a novel approach for code readability classification using eye-tracking data from 90 undergraduate students assessing Python code snippets.

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 [**ribeiro2018attributes**].

mi2018inception introduces a novel approach, IncepCRM, for Code Readability Classification using deep learning, specifically based on the Inception architecture. Unlike traditional methods that rely on handcrafted features and various machine learning algorithms, IncepCRM automatically learns multi-scale features from source code with minimal manual intervention. The study empirically validates IncepCRM's performance with human annotator information as auxiliary input, demonstrating its superiority in accuracy compared to previous models across three publicly available datasets. The results confirm the feasibility and effectiveness of deep learning for code readability classification.

Another study addresses concerns regarding the classification of code script readability by proposing a novel approach using Generative Adversarial Networks (GANs). Unlike previous studies relying on labor-intensive manual feature engineering, the proposed method involves encoding source codes into integer matrices with multiple granularities and utilizing an EGAN (Enhanced GAN) for code readability classification. The EGAN, comprising three GANs with identical architectures trained on differently preprocessed data, outperforms five state-of-the-art code readability models, achieving a quality increase ranging from 2% to 15%. This approach demonstrates the effectiveness of GANs in code readability classification tasks, providing a substantial improvement without the need for manual interface design [**sharma2020egan**].

The paper of **choi2020metric** introduces 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 75.74%. 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. Experimental results indicate that methods authored using the proposed approach exhibit higher readability compared to those without it. The study concludes by highlighting the potential future contribution of Instant R. Gauge as an open-source project to assist developers in enhancing their source code readability.

mi2023graph's study addresses the importance of code readability in software development and introduces a novel 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. Evaluation on a Java dataset demonstrates superior performance with 72.5% and 88% accuracy in three-class and two-class readability classification, respectively. The graph-based approach outperforms existing models, indicating its effectiveness in capturing code readability-related features. Future work includes overcoming dataset limitations, exploring heterogeneous graph methods, and improving model performance.

mi2022enhanced's paper addresses the challenge of multi-class code readability classification, emphasizing the importance of code readability in software maintenance. Due to a scarcity of labeled data, most prior research focused on binary classification. The authors propose an enhanced data augmentation approach, incorporating domain-specific data transformation and Generative Adversarial Networks (GANs). Experimental results demonstrate a significant improvement of 6.3%, reaching a state-of-the-art multi-class code readability classification accuracy of 68.0% compared to using only the original data. The proposed method is proven effective, and future work aims to further enhance classifier performance by exploring alternative code representation methods, increasing the number of readability levels, and addressing the challenge of limited dataset size through additional labeling efforts.

vitale2023using's paper introduces a novel approach to automatically identify and suggest readability-improving actions for code snippets. The authors develop a methodology to identify readability-improving commits, creating a dataset of 122k commits from GitHub's revision history. They train the T5 model to emulate developers' actions in improving code readability, achieving a prediction accuracy between 21% and 28%. The empirical evaluation shows that 82-90.8% of the dataset commits aim to improve readability, and the model successfully mimics developers in 21% of cases. The dataset and scripts are released for further research. Future work includes refining the methodology, testing additional Language Model Models (LLMs), and extending the approach to other programming languages. The goal is to create a comprehensive tool for readability improvement.

mi2022rank's paper introduces a novel approach to code readability assessment by framing it as a learning-to-rank task. The proposed model employs siamese neural networks to rank code pairs based on their readability. The evaluation on three publicly available datasets demonstrates promising results, with an accuracy of 83.5%, precision of 86.1%, recall of 81.6%, F-measure of 83.6%, and AUC of 83.4%. The research suggests that the approach effectively ranks code readability by comparing code pairs. However, the study acknowledges areas for improvement, such as addressing ties in readability levels, experimenting with different source code representations, exploring more sophisticated neural network architectures, and considering practical applications, such as detecting readability changes in version control systems to aid developers in monitoring code changes.

mi2023makes' paper aims 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. The linear regression algorithm, based on the back-door criterion, is then employed to determine the causal effect of various features on code readability. 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. The proposed approach provides developers with insights into code readability patterns.

oliveira2020evaluating's study addresses the challenge of acquiring large datasets with manual labels for training deep learning models in code readability classification. To overcome this limitation, the authors propose data augmentation approaches to artificially increase the training set size, reducing the risk of overfitting and aiming to improve classification accuracy. They create transformed versions of code snippets by manipulating aspects such as comments, indentations, and names of classes/methods/variables, based on domain-specific knowledge. Additionally, they explore the use of Auxiliary Classifier GANs to generate synthetic data. Experimental results show a significant improvement in the classification performance of deep neural networks when trained on the augmented corpus, achieving a state-of-the-art accuracy of 87.38%. The findings highlight the effectiveness of data augmentation in code readability classification. Future work involves extending the proposed approach systematically, exploring more ground-truth data, experimenting with other classifiers, and aiming for better performance with larger datasets and more complex deep learning models.

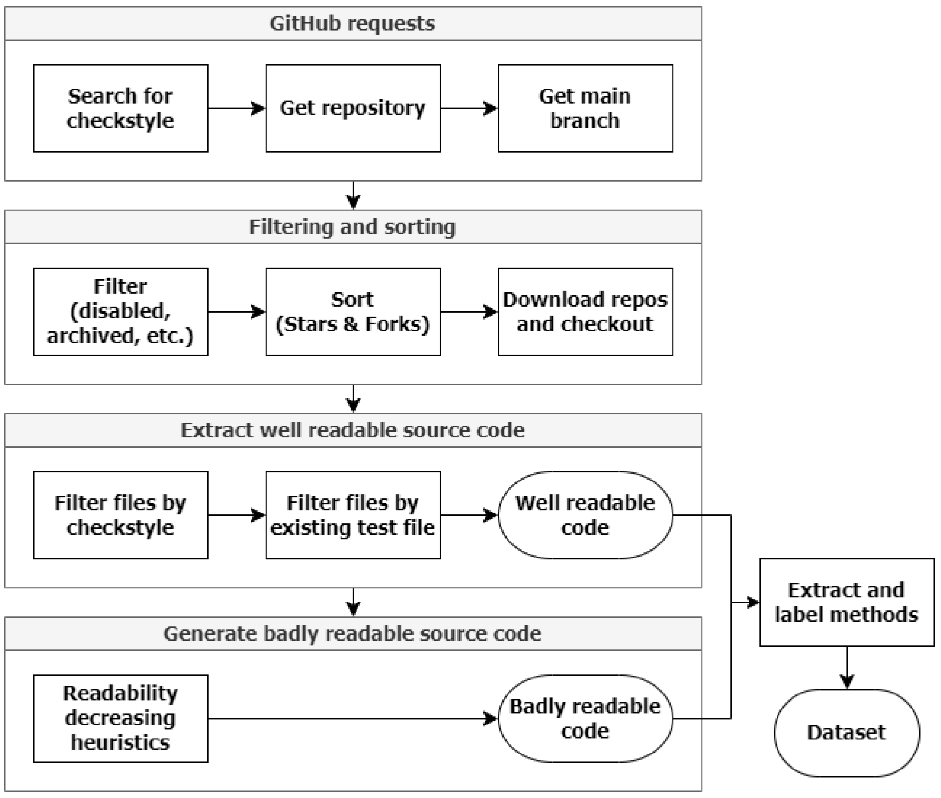
GitHub requests
Search for checkstyle
Get repository
Get main branch
Filtering and sorting
Filter (disabled, archived, etc.)
Sort (Stars & Forks)
Download repos and checkout
Extract well readable source code
Filter files by checkstyle
Filter files by existing test file
Well readable code
Generate badly readable source code
Readability decreasing heuristics
Badly readable code
Extract and label methods
Dataset

Figure 1: Overview of the used dataset generation approach.

3 DATASET GENERATION APPROACH

Other than previous datasets for readability classification, our dataset was generated using an automated approach. This approach is novel to the best of our knowledge. You can find a visualization in Figure 1.

The approach is divided into four parts. First, the GitHub REST API ([3]) is queried for repositories that use checkstyle. The repository information (containing the URL) are stored together with their main branch.

Next, certain repositories are filtered out by the following criteria:

- TODO

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.

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

In a third step we run checkstyle ([4]) against the project own checkstyle configuration to get all java class files, that pass the own checkstyle test. 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 code snippets which we assume to be well readable.

The fourth and final step is to generate badly readable code from the well readable one. Therefore we use the proposed Readability Decreasing Heuristics (RDH). Afterwards we again extract all methods with a comment at the beginning of the method. Initially we planned to not require comments for the badly readable dataset part. However, it turns out that in this case all well-readable methods have a comment while most of the badly-readble don't. This lead to shortcut learning, whether a method has a comment or not instead of learning the readability of the code snippet.

4 READABILITY DECREASING HEURISTICS

We already mentioned, that Readability Decreasing Heuristics (RDH) are used to generated badly readable code from well readable one. In this section we will cover, what is meant by RDH. The RDH is a set of code manipulation heuristics that are performed on the abstract syntax tree of java files using the spoon library ([5]) [18].

The RDH tool converts the java code of each well-readable java class file into an abstract syntax tree. In the end the abstract syntax tree (AST) is parsed back to java code using an pretty printer. If nothing else is done, this results in the "none" rdh. Note that the code that is delivered by the tool in this way is slightly different from the original input code, as the styling and formatting of the original code is overwritten by the default formatting of the java pretty printer.

In between of the two steps and while pretty printing various code modifications can be applied (see Figure 2). Renaming is done while the code is in it's AST representation to make sure that the declaration and usages of variables, fields and methods are all renamed to the same new name. The other rdhs are applied while transforming the AST back to source code. Those include adding additional spaces, adding or removing newlines and changing the indentation of code in term of tabs.

[4]`https://checkstyle.sourceforge.io/`, accessed: 2024-02-15
[5]`https://spoon.gforge.inria.fr/`, accessed: 2024-15-02

After this the individual methods are extracted from the class files. As already mentioned, we require for all methods to have a method comment. Therefore, we apply the remove-comment RDH after method extraction is completed.

The RDH works with a configuration file where one can specify a probability for each heuristic that can be applied. We chose the probabilities so that the generated code snippets are still realistic in a sense that they might be written as well by humans.

5 SURVEY

We evaluate the dataset generated with the new approach with a survey. Therefore we had to carefully sample suitable code snippets from the dataset. You can find an overview over the approach in Figure 3.

The first step was to find realistic configurations for the readability decreasing tool. After creating an initial dataset using the heuristics a pilot survey was conducted. Afterwards the heuristics that seemed to be too strong were weekend and adjusted according to the pilot survey result. The result of this were 9 different rdhs, which can be found in Table 1.

Together with the original methods this resulted in 10 different configurations.

The configurations are based on probabilities for different heuristics. A heuristic is applied with the given probability one each occurrence of the object to apply it to. For example, comment_remove is applied with a probability of 10% on each comment that occurs within the code snippet. Thus, the exact amount of changes for one method is uncertain. It might even happen (especially for short methods, such as getters and setters) that a method is not changed at all. For example, if a method has only a single comment and we apply comment_remove, it is very likely, that a method is not changed at all.

In a second step we applied stratified sampling to those to be able to distinguish between very simple methods such as getters and setters and more complex methods. In order to be able to compare original methods with their modified variants, we performed the sampling only on the original methods and matched the rdh methods to those in a later step.

Thus we first had to calculate features for the original code snippets. This was done using TODOs tool. Therefore a TODO-dimensional features vector was calculated for each original code snippet. Next we calculate the cosine similarity metrix between all feature vectors and finally, using Ward's hierarchical clustering we were able to cluster the methods into an arbitrary amount of clusters.

Configuration	RDH probabilities
none	-
comments_remove	removeComment: 1.0
newline_instead_of_space	newLineInsteadOfSpace: 0.15
newlines_few	removeNewline: 0.3 spaceInsteadOfNewline: 0.05
newlines_many	add1Newline: 0.15 add2Newlines: 0.05
rename	renameVariable: 0.3 renameField: 0.3 renameMethod: 0.3
spaces_many	Add1Space: 0.2 Add2Spaces: 0.1 spaceInsteadOfNewline: 0.05
tabs	remove1IncTab: 0.2 add1IncTab: 0.1 remove1DecTab: 0.1 add1DecTab: 0.1 incTabInsteadOfDecTab: 0.05 decTabInsteadOfIncTab: 0.05
all7	all probabilites/7

Table 1: Chosen configurations and their probabilities for the RDH tool

By comparing the merge distances in each step (see Figure 4) we evaluated that a cluster size of 4 makes most sense: The merge distance from 5 to 4 is small, so we should still do this merge, but the merge distance from 4 to 3 is large, so one should rather not do this merge. Additionally, 4 is the size with the last possibility for a small merge distance. We assigned each cluster/strata manually a name, which can be found in Figure 2.

In the third step we crafted the surveys from the stratas. We decided to provide for each original method all 10 configurations mentioned earlier as we want to compare the original methods to their rdh-variants. As we have a capacity of rating 400 code snippets, we require to sample 40 original code snippets (and than add all their rdh-variants).

We chose to sample randomly from within the stratas. However, we distributed the 40 snippets among the stratas as can be seen in table 3.

This decision was made due to the relative high frequency of methods not being different from their original methods (see Figure 5). Another factor for this

Stratum	**Method Type**
Stratum 0	Simple methods
Stratum 1	Complex methods
Stratum 2	Magic number methods
Stratum 3	Medium complex methods

Table 2: The created stratas and a manually assigned name based on the methods within.

Stratum	**Percentage**	**Count**
Stratum 0	10%	4
Stratum 1	40%	16
Stratum 2	10%	4
Stratum 3	40%	16
Total	**100%**	**40**

Table 3: Distribution of Sampled Methods

decision is, that especially simple methods are rather uninteresting for readability classification, as they are often generated and mostly follow a straight forward pattern.

After sampling the 40 original methods, we next sampled all 9*40 rdh-variants that belong to the original methods. This was done mostly automatically using the sample name of the original methods and the sample names of the rdh-variant-methods. However, if the method was renamed earlier due to the rename-method-heuristic, the new method name did not match the one of the original method anymore and therefore we had to match those manually.

After gathering all 400 methods we than had to distribute those among the 20 survey sheets, 20 snippets each. To not manipulate the raters we decided, that a variant of each method can only be once in each survey sheet. For example, if the original method is in survey sheet 1, the comment_remove-variant (or any other variant) must not be in the same sheet.

To achieve this, we created four permutation matrices with 10 snippets each. 10 was chosen, as it is possible to distribute 10 snippets with 10 variants each among at least 10 survey sheets while not violating our condition. By combining two 10-permutation-matrices we could create 10 survey sheets with 20 code snippets each. One implication of this approach is, that each survey sheet has each variant exactly twice. By doing this twice, we ended up with a desired distribution of 20 survey sheets à 20 methods. Additionally, our condition holds: In each survey sheet there is only one variant of the same method.

Finally the methods for each survey sheet where randomly shuffled within each survey sheet so that the positions of the variants within each sheet are randomly distributed. This was done to minimize any effect of the position of a snippet/variant within a survey on the rating.

6 READABILITY CLASSIFICATION MODEL

We will investigate whether it is possible to score a higher accuracy as current models in classifying code readability for Java using Deep Learning. Therefore, we will train the model from Mi et al. [17] with more data. We will consider augmenting the model with a method name classifier and incorporating semantic encoding for tabs and spaces. The training data will be generated in a novel way for classification of readability, inspired by Loriot et al. [15]. The method name classifier is similar to Code2Vec [3]. The combination of all components is novel to the best of our knowledge. You can find a visualization of the planned modifications of Mi et al.'s model in figure 6. We will focus on generating training data, as the approach will be usable for further research in the field of source code readability.

Deep Learning based models perform better the more training data they get [12]. Therefore, one approach in order 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. We present another approach for gathering training data.

In a first step, GitHub repositories with known high code quality are downloaded and labeled as highly readable. We select repositories using a similar approach as Allamanis et al. [2] and then assume that they contain only well readable code. In a second step, the code is manipulated so that it is subsequently less readable. This approach is similar to the approach of Loriot et al. [15]. After both steps, we have a new, automatically generated training dataset for source code readability classification.

This brings up the question, how to manipulate code so that it is less readable afterwards. We therefore introduce a tool called Readability Decreasing Heuristics. As the name suggests this is a collection of heuristics that, when applied to source code, lower the readability of it. For example such a heuristic is to replace spaces with newlines. Another example is to increase the indentation of a code block by a tab or multiple spaces. Moreover, with most changes it is also possible to do exactly the opposite (replacing newlines with spaces, decreasing indentation), which in most cases also decreases the readability of source code.

Code snippets in Java are syntactically the same, before and after applying Readability Decreasing Heuristics. Complexity did not change either. However, if various modifications are applied many times, those changes are capable of lowering the readability of source code, as the comparison of listing 1 and listing 2 suggests.

Note that we assume two things for the data generation approach:

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

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

In recent years it was shown that Deep Learning models can be further improved by modifying the structure of the architecture or by introducing new components, parts or layers to existing architectures. We suggest two improvements for the model of Mi et al. [17]. Firstly, we want to embed spaces and tabs as semantic tokens. Secondly, adding a method name fitting classifier as a component of the overall model could be an improvement. If there is time left, we will try to surpass the performance of recent source code readability classifiers with those improvements to data generation and the model.

We will evaluate our suggestions with two methods. Firstly, we conclude a user study. Secondly, we compare code readability models with each other.

6.1 USER STUDY

The goal 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?

We will achieve this by showing programmers code snippets that were generated with the presented approach. Therefore, human annotators give each code snippet a rating of its readability. The annotators are selected by prolific[6]. Particular attention is paid to a high proportion of people from industry. The readability rating is based on a five-point Likert scale [13] ranging from one (i.e., very unreadable) to five (i.e., very readable). We apply the same rating as done previously [6, 10, 21], but, other than before, we will not use the rating for labeling the training data. Instead, we will only use the ratings to validate a few randomly selected code snippets out of many that are automatically labeled.

6.2 COMPARING MODELS

Besides the user study we will evaluate our suggestions by comparing machine learning models against each other. The comparisons are based on common metrics such as accuracy, F1-score and MCC [7]. One can distinguish further between the following variants of comparing models:

In one variant we compare models that have the same architecture (same layers, same weight initialization, same components, etc.) while they differ in the data they are trained on. For example, we can train a model with the old and new datasets, separately and combined. If done for multiple model architectures we can evaluate how the differences in training data influence the model performance.

Another variant would be to compare models with different architecture but the same training data. In this way, we can evaluate newly introduced components by measuring and comparing the performance of such models.

A third comparison variant is created by combining the first two. Both of them lead to many options in what to compare, especially if only small changes to training data or model architecture are done. To find out, if our suggestions lead to a better model overall, we will compare our newly created model with all changes at once to the state-of-the-art model of Mi et al. [17].

[6] `https://www.prolific.com/`, accessed: 2023-09-30

6.3 RESEARCH QUESTIONS

We come up with the following research questions:

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

In our new approach for generating training data, we assume that the code from repositories is 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 (section 6.1).

Research Question 2: ***(generate-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 principle, and we will propose an automated approach for archiving this: Readability Decreasing Heuristics.

As the name already suggests, the applied transformations on the source code are only heuristics. To answer, whether the generated code is badly readable (assumption 2) we will utilize the results of the user study (section 6.1).

Research Question 3: ***(best-heuristics)*** *Which heuristics are best to generate poorly readable code from well readable code?*

We want to compare the modifications of the proposed heuristics for generating poorly readable code to each other. Therefore we will train the same classifier model with badly readable code generated by different Readability Decreasing Heuristics. We will then evaluate the model variations against each other (section 6.2) to answer the research question.

Research Question 4: ***(new-data)*** *To what extent can the new data improve existing readability 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. [17]. Therefore we will train their proposed model with and without the new data and then evaluate the models against each other (section 6.2).

Research Question 5: ***(embedding-spaces)*** *Optional: To what extend does the embedding of spaces and tabs in semantic code representations improve readability classification?*

The state-of-the-art model of Mi et al. [17] does consider spaces and tabs only in its visual component. We want to investigate if it can improve the quality of a Deep Learning based model if spaces and tabs are encoded as semantic tokens. We also want to investigate if this makes the visual component superfluous. We will evaluate the proposed improvement as described earlier (section 6.2).

Research Question 6: ***(name-classifier)*** *Optional: To what extend does the usage of a method name classifier improve readability classification?*

Correct naming of identifiers is crucial for ensuring readability of software programs. It is of outstanding importance for readability of code that the name of methods fit the method bodies [14]. We want to introduce a new component to the model of Mi et al. [17] that is built similar to Code2Vec [3]. We want to investigate if the newly introduced component improves the quality of the resulting model. We will evaluate the proposed improvement as previously described (section 6.2).

7 EVALUATION

The readability ratings of code snippets mined from Github are not very accurate. Therefore the well readable assumption TODO only holds for certain clusters of code snippets: TODO. For clusters that can be labeled with X or Y this assumption does not hold. Therefore we labeled the mined code depending on the cluster they where grouped in as you can see in Table TODO. While the rating does not hold for each and every snippet within such a cluster it is a good estimation on average. This should be sufficient to train a readability classifier.

The badly readable assumption holds. Especially the heuristics X Y and Z decrease the readability by a significant extend. We estimate the readability decrease for a certain probability of a certain type as can be seen in table TODO. We therefore calculate the readability of a new snippet by taking the original readability score and decreasing the readability percent depending on the probability of such a refactoring beeing applied. In Table TODO you can see how can to which extend we combined certain manually selected probabilities and how we then calculated the new readability score.

When we compare the model of TOWARDS trained with the old dataset we can reproduce the results described in their paper. Once we add our own dataset we achieve an accuracy improvement of TODO. You can find a detailed comparison in Figure TODO.

When we scale up the architecture by increasing XY we can achieve an even higher accuracy of TODO when using the new training data. However, with the old dataset only the results are worse. This suggests that the new fits the new larger dataset better while the architecture of TOWARDS was built for small datasets.

By combining the results of the study and model evaluation we can answer the third research question: The rating of our automatic dataset generation approach is not as accurate as letting multiple users rate a certain code snippet. However, being able to fully automate the dataset generation approach and the vast amount of data generated that way makes up for this. The larger dataset improves the performance, especially of our new adjusted model by a significant extent of TODO. Therefore we conclude, that our new dataset and it's generation approach is suitable for training deep learning based code readability classifiers.

8 DISCUSSION

The biggest thread to our approach is the reliance on heuristics for the dataset generation. We can not show, that the labeled code snippets of our dataset actually fit the score we assigned them. This would require an manual evaluation of X code snippets by human annotators and is therefore not feasible. We can reduce the extent of this with our model results.

TODO: Copy from study paper

TODO: Copy from towards and newer model

9 CONCLUSION

TODO: Add conclusion

The new dataset has another advantage that is not yet utilized in this work: For the first time there is a dataset with one well readable and a second one less readable code snippet that is functionally equivalent. This could be used to train a transformer on source code readability improvement. Such a transformer could take code as input and improve it's readability. Such a tool would probably be of high usability among programmers.

A current restriction of the dataset is that it only works for java code. Another proposal for future work is therefore to overcome this restriction by extending the tool for other languages. This is not trivial as one has to adjust the readability decreasing heuristics to work with a different language. Furthermore a general tool that works for all languages will be very hard, if possible at all.

In order to further improve the readability estimations for both, the well and the badly readable code, one could conduct more s. By separating the code snippets into more fine granular clusters and by getting a more accurate average score by asking more persons about readability of code within the same cluster one can increase the accuracy of label estimation. Such an improvement can then again improve the model predictions as it is learning from this data.

As XY suggested another useful representation for code readability studies is the syntax tree representation of code. One could improve the performance of this model by adding another representation encoding extractor for java code that automatically extracts the abstract syntax tree of code.

A crucial aspect of code readability is naming. For the scope of methods, the most crucial part are methods names. Therefore one could improve this tool by adding a component that explicitly considers how well a method name fits its body.

Further research could also be to come up with another encoding that represents code in a different way.

Another way to improve existing code readability classifiers could be to come up with a different structure for some layers or entire different models.

The heuristics described in this work is only a part of the possible heuristics one could come up with. One could come up with more heuristics and evaluate them using user studies in order to further improve the variety of badly readable code. This might increase the number of internal features the model could learn which might again increase the tools accuracy.

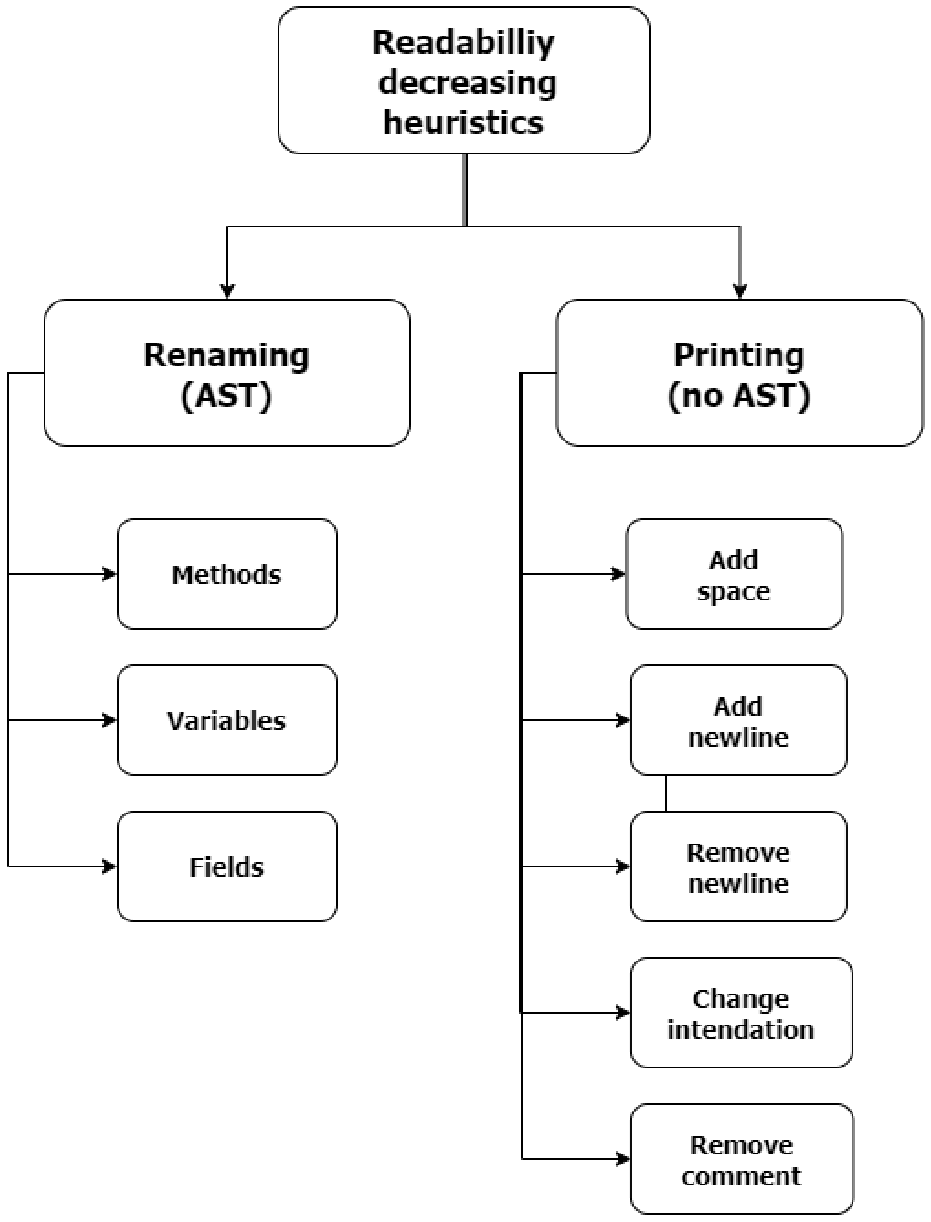

Figure 2: Overview of the used dataset generation approach.

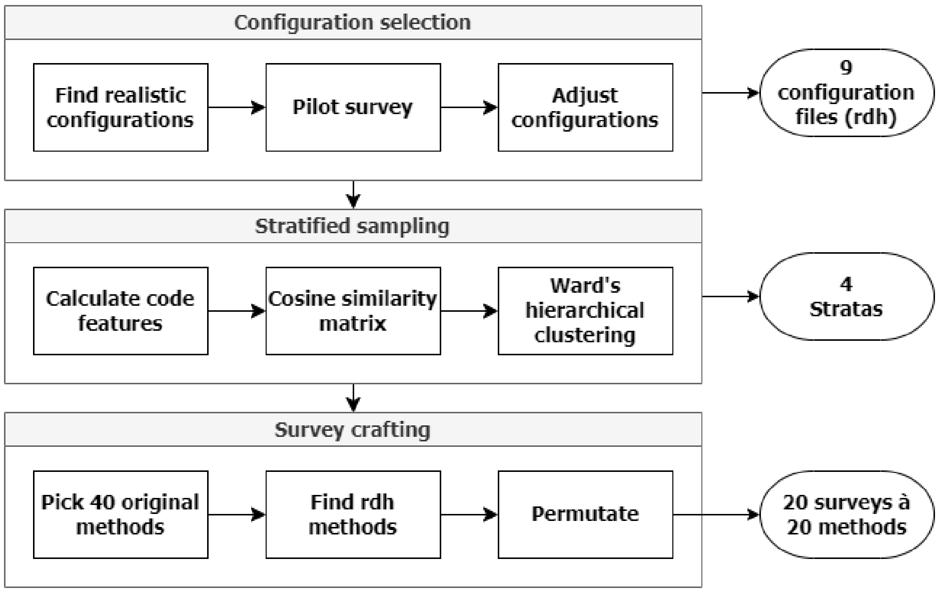

Figure 3: Overview over the steps performed to craft survey sheets from the dataset.

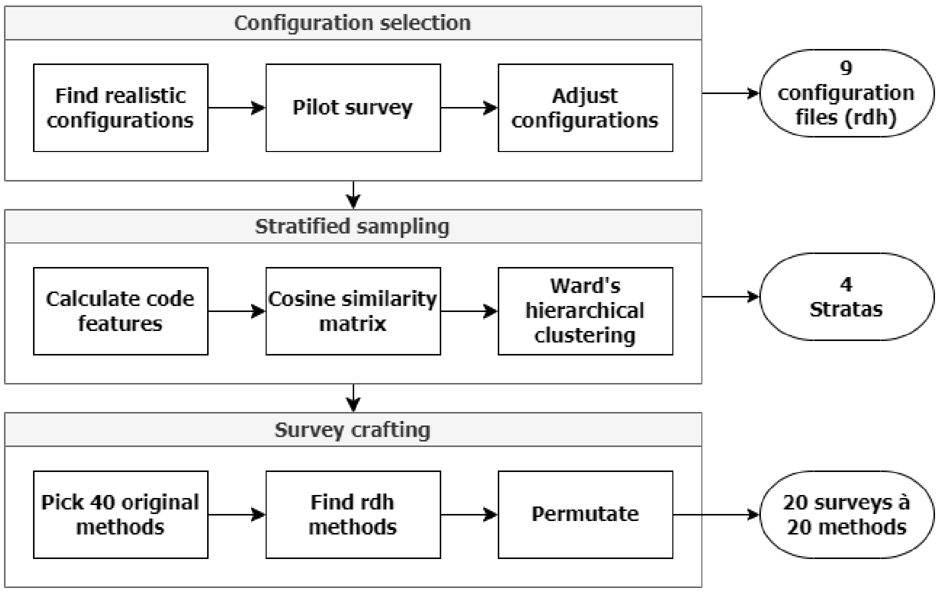

Figure 4: TODO: Replace. The merge difference and the local derivative for amount of clusters between 2 and 10.

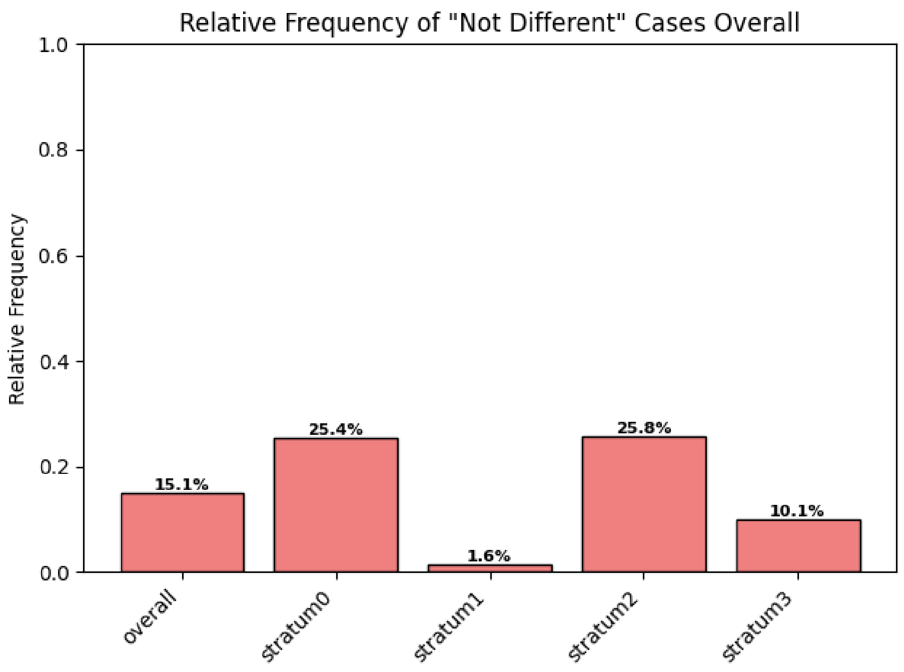

Figure 5: Relative frequency of the case that a rdh-method is not different from it's original method.

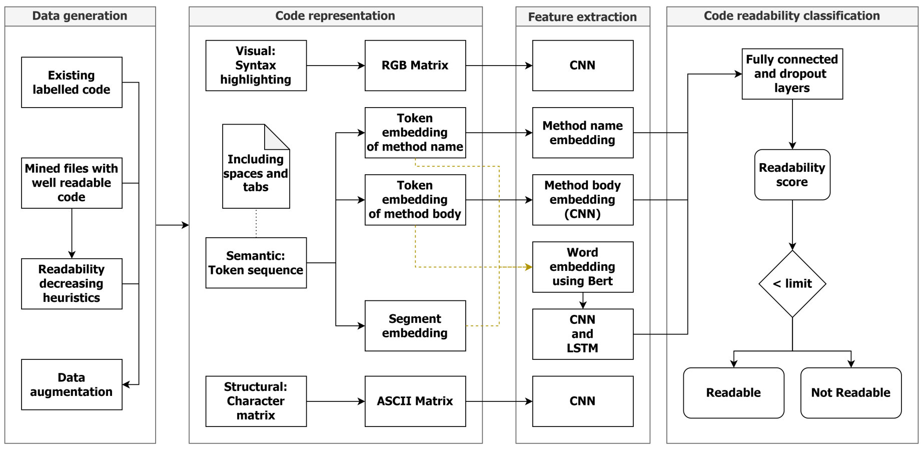
Data generation
Existing labelled code
Mined files with well readable code
Readability decreasing heuristics
Data augmentation
Code representation
Visual: Syntax highlighting
RGB Matrix
Including spaces and tabs
Token embedding of method name
Token embedding of method body
Semantic: Token sequence
Segment embedding
Structural: Character matrix
ASCII Matrix
Feature extraction
CNN
Method name embedding
Method body embedding (CNN)
Word embedding using Bert
CNN and LSTM
CNN
Code readability classification
Fully connected and dropout layers
Readability score
< limit
Readable
Not Readable

Figure 6: Overview of the planned approach.

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: *International conference on machine learning*. PMLR. 2016, pp. 2091–2100.

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

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

[6] 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.

[7] Davide Chicco and Giuseppe Jurman. ‘The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation’. In: *BMC genomics* 21.1 (2020), pp. 1–13.

[8] Ermira Daka et al. ‘Modeling readability to improve unit tests’. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 107–118.

[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 arXiv:1712.00409* (2017).

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

[14] Kui Liu et al. ‘Learning to spot and refactor inconsistent method names’. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 1–12.

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

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] Michihiro Yasunaga and Percy Liang. ‘Graph-based, self-supervised program repair from diagnostic feedback’. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 10799–10808.