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

2nd November 2023

Contents

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 [18, 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, 19, 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, 18, 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 BACKGROUND AND RELATED WORK

2.1 READABILITY

To improve readability classification, we need to capture what readability is. We define readability as a subjective impression of the difficulty of code while trying to understand it [18, 6]. Readability of code is a perceived barrier that needs to be overcome before it is possible to work with the code. The more readable code is, the lower the barrier [18]. To give an example for high vs low readability, consider the code of listing 1 from GitHub[1] and compare it to the code with the same functionality of listing 2. You will notice that the first piece of code is more readable than the second one.

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].

There is another related term: understandability. Readability is the syntactic aspect of processing code, while understandability is the semantic aspect [18]. 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 [21].

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, 18, 10, 20]. 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 [20].

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, 20].

[1] `https://github.com/apache/cassandra/blob/trunk/src/java/org/apache/cassandra/utils/HeapUtils.java`, accessed: 2023-25-07

```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

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 GitHub[2] commits manually. They suggest considering other metrics such as incoming method calls or method name fitting [11].

2.4 RELATED WORK

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 [15]. A similar idea has been explored by Yasunaga and Liang [22]. 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

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

```
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

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].

3 DATASET GENERATION APPROACH

4 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 1. We will focus on generating

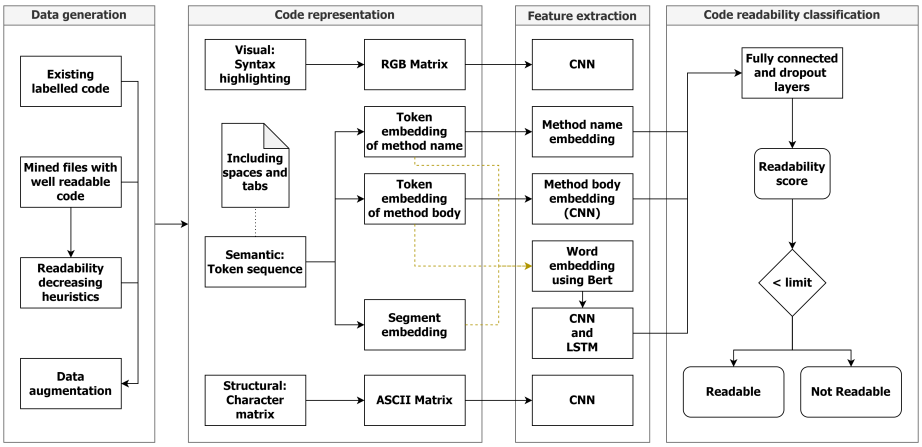

Figure 1: Overview of the planned approach.

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.

4.1 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[4]. 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, 20], 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.

4.2 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].

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

4.3 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 4.1).

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 4.1).

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 4.2) 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 4.2).

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 4.2).

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 4.2).

5 EVALUATION

TODO

6 CONCLUSION

TODO

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] 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.

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

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

[21] 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.

[22] 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.