A Model to Detect Readability Improvements in Incremental Changes

Devjeet Roy
Washington State University
Washington, USA
devjeet.roy@wsu.edu

Sarah Fakhoury
Washington State University
Washington, USA
sarah.fakhoury@wsu.edu

John Lee
Washington State University
Washington, USA
john.k.lee@wsu.edu

Venera Arnaoudova
Washington State University
Washington, USA
venera.arnaoudova@wsu.edu

ABSTRACT

Identifying source code that has poor readability allows developers to focus maintenance efforts on problematic code. Therefore, the effort to develop models that can quantify the readability of a piece of source code has been an area of interest for software engineering researchers for several years. However, recent research questions the usefulness of these readability models in practice. When applying these models to readability improvements that are made in practice, i.e., commits, they are unable to capture these incremental improvements, despite a clear perceived improvement by the developers. This results in a discrepancy between the models we have built to measure readability, and the actual perception of readability in practice.

In this work, we propose a model that is able to detect incremental readability improvements made by developers in practice with an average precision of 79.2% and an average recall of 67% on an unseen test set . We then investigate the metrics that our model associates with developer perceived readability improvements as well as non-readability changes. Finally, we compare our model to existing state-of-the-art readability models, which our model outperforms by at least 23% in terms of precision and 42% in terms of recall.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; • **Computing methodologies → Machine learning**.

KEYWORDS

Source code readability, Machine learning, Code quality.

ACM Reference Format:
Devjeet Roy, Sarah Fakhoury, John Lee, and Venera Arnaoudova. 2020. A Model to Detect Readability Improvements in Incremental Changes. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387904.3389255

1 INTRODUCTION

In the past 40 years, researchers in the software engineering community have been studying various factors that effect program comprehension, including the source code lexicon, comments, and readability [5, 8, 9, 25, 26, 47, 49, 57]. Studies have shown that roughly half of the cost of software maintenance stems from activities related to program comprehension [13], and developers spend the majority of their time understanding source code [48, 50, 53, 58].

Thus, an important aspect of software that developers must be mindful of during development, review, and maintenance activities is source code readability. What is considered as 'readable' source code can vary based on individual, project, or even team preference. Researchers have long since aimed to develop models that can distinguish readable source code from non-readable source code [6, 8, 15, 34, 39, 45], with the goal of creating tools to help developers focus maintenance efforts. These models have been shown to have good performance, with accuracy numbers as high as 85% [45], when classifying a method or file as readable or non-readable.

However, recently, several models for the detection of source code readability have come under question regarding the extent of their usefulness in practice. Research by Pantiuchina et al. [38] has shown that more often than not, in practice, state-of-the-art code quality models are unable to capture quality improvements in the source code. In other words, in the context of incremental changes made to a pre-existing file, models are unable to capture improvements in the source code's cohesion, complexity, coupling, and readability. Recently, we found that there exists a significant discrepancy between the readability models created by researchers and developers' perception of readability improvements in practice, as captured in commit messages [16]. When applying these models to both versions of a file, before and after the change was made, readability improvements can only be captured 40% of the time.

Possible reasons for the poor performance of these models in practice could be attributed to the selection of external developers, types of source code snippets, and classification of snippets on a 5-point Likert scale from 'readable' to 'unreadable'. In fact, multiple state-of-the-art models in software engineering have relied on datasets derived by categorizing the readability of code snippets by developers using such scale, in order to create an oracle from which to derive metrics [8, 15, 45]. Thus, these models are not created to detect the kind of readability improvements made in practice by internal developers, which are often incremental in nature. Instead, they are developed with the intent to rank readability on a universal scale with two extremes.

Results from our previous work [16] highlights a clear need for a readability model that is able to successfully capture readability improvements in practice. We identify several promising metrics that are able to successfully capture these improvements and should be considered by future readability models. In this paper, we expand upon our initial work by proposing a novel readability model that successfully identifies readability improvements made by developers in practice, achieving a precision of 79.2% and a recall of 67%. The proposed approach outperforms existing models by at least 23%.

The main contributions of this work are as follows:

(1) An expanded dataset previously made available by Fakhoury et al. [16]. We increased the dataset by adding 886 new commits from 13 additional Java projects.
(2) Manually validated a total of 2,781 new files for both readability and non-readability improvement commits.
(3) Proposed a novel model for the detection of readability improvements made in practice.
(4) We provide a replication package, including the expanded dataset and files needed to reproduce the results [42]

Paper Structure. The rest of the paper is organized as follows. In Section 2, we present the study definition and design, including discussion of the research questions, data collection procedures, source code analysis tools, and the proposed approach. We answer the research questions in Section 3 and discuss the implications of our work in Section 4. We survey the related work in Section 5 and discuss the threats to validity of our work in Section 6. We conclude the paper in Section 7.

2 STUDY DEFINITION AND DESIGN

The *goal* of this study is to create a model that is able to capture incremental readability improvements in Java source code. The *quality focus* is the performance of the proposed model, as well as the features that contribute to the model. The *perspective* of the study is that of researchers and developers, who are interested in measuring readability improvements during software maintenance tasks. The evaluation is carried out on 2665 commits with readability improvements from 76 *engineered* Java projects collected from GitHub[1].

Figure 1 depicts the overview of our approach. First, we expand upon the initial oracle in our previous work [16] by collecting commits from engineered open source Java projects on GitHub. To construct the oracle—described in Section 2.2.2—we extract commits that contain both readability and non-readability improvements, as denoted by the authors of the commits in their commit messages. This dataset is then manually validated by two annotators and an oracle is created. We separate this oracle into a training and testing set. Next, we use a set of static code analysis tools, such as SourceMeter [27] and PMD [1], to collect metrics on files involved in the commits.

For each readability and non-readability commit in the oracle, metrics are collected on files before and after changes are made. Descriptions of the tools and methods used can be found in Section 2.3. Using these metrics as input to various machine learning algorithms, we perform automatic feature selection and hyperparameter tuning. We use the best performing configuration of these two steps to devise our model; details about the approach are provided in Section 2.4. The model is then evaluated on the test set using a set of evaluation metrics and compared to state-of-the-art readability models; details are provided in Section 2.5.

[1]https://github.com

2.1 Research Questions

(1) **RQ1: *Can we use machine learning to capture readability improvements made in practice?***
Motivation: Recent research has shown that existing state-of-the-art readability models are unable to capture readability improvements made in practice when applied to incremental software changes. Furthermore, there seems to exist a discrepancy between existing models and developers' perception of readability. Creating a model that is able to detect incremental improvements in source code readability is the first step towards advancing tools that are designed to help developers focus maintenance efforts.
Approach: We expand upon the intial dataset introduced in our previous work [16] and use seven different static code analysis tools to extract metrics for our model. We then use a combination of automatic feature selection techniques to create our feature set. We explore various machine learning algorithms to create a model that can successfully identify incremental readability improvements made in practice.
(2) **RQ2: *What features align with developers' perception of readability improvements in practice?***
Motivation: By investigating the features our model relies on for accurate detection of readability improvements, we can gain a deeper understanding into which metrics are most strongly aligned with developers' perception of readability in practice.
Approach: We perform a qualitative analysis on the set of top features selected by our model. We explore which of these features can help improve readability in practice and compare them to the readability recommendations that currently exist.
(3) **RQ3: *How does the proposed model perform when compared to existing state-of-the-art readability models?***
Motivation: By investigating how and why our model performs against existing readability models, we can gain a deeper understanding about how certain features contribute to readability improvements in different development contexts.
Approach: We select three state-of-the-art readability models and run them on our oracle. We then compare their precision and recall with that of our proposed model and perform qualitative analysis of our findings.

2.2 Data Collection

2.2.1 Subject Systems. Our dataset consists of 5104 data points from 2665 commits in 76 projects. Each data point in the dataset represents metrics collected for a file at a specific revision in its git repository, and is labelled as either a readability improvement

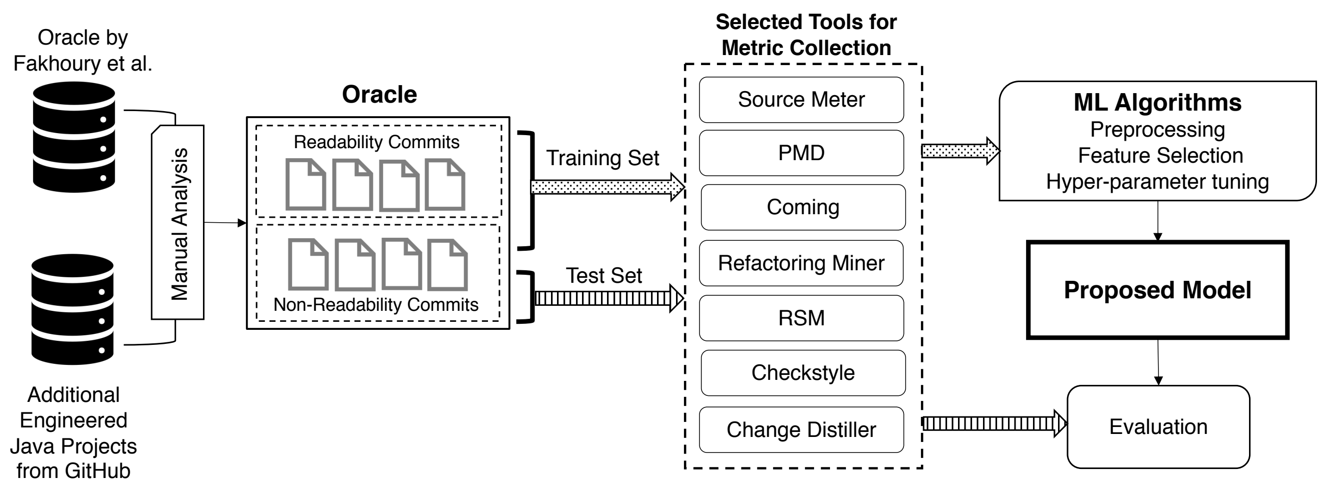

Figure 1: Overview of the approach.

Oracle		# Projects	# Commits	# Files
Original [16]	Readability	63	548	2323
	Non-Readability		1231*	2275*
	Total	63	1779	4598
Expanded	Readability	76	653	2449
	Non-Readability		2008	2661
	Total	76	2661	5110

Table 1: Project, commit and file statistics for the Oracle used in our initial work and the expanded version used in this paper. *The expanded dataset contributes 2661 validated non-readability files.

or not. Our dataset is a combination of the oracle we constructed in [16] and data from new Java projects we identified on GitHub. Table 1 outlines how our oracle compares to the original in terms of the number of projects and the number of readability and non-readability commits and files. We added 13 new projects and 882 commits. Given that non-readability data points from the original dataset were not validated, the expanded dataset contributes 2,661 validated non-readability files. This is a total of 2,781 new data points.

2.2.2 Oracle Creation. We follow a similar oracle construction methodology as in [16] and expand upon our dataset by adding new data points from engineered Java projects. An engineered project is a software project that leverages sound principles of software engineering across various aspects such as soure code, documentation and testing [37]. In the original dataset, we only validate commits belonging to readability improvements. Two annotators, the authors of this paper, manually validate all data points in our oracle, including the commits that do not contain readability improvements. If the annotators had a doubt and could not decide, the data point was discarded. The number of discarded data points were less than 5%. The procedure to create the oracle is outlined below:

(1) Identify suitable Java projects using Reaper [37], a tool that calculates a score for GitHub repositories to determine whether they are engineered projects or not.
(2) Identify candidate readability improvement commits in a project by using a keyword match on the commit message using keywords: 'readable', 'readability', 'easier to read', 'comprehension', 'comprehensible', 'understand', 'understanding', and 'clean up'. We manually validate and exclude commits that contain the keywords but do not explicitly reflect readability improvements of the source code. For example, commit messages stating the improvement of the readability of UI elements for user-facing applications.
(3) Commits containing readability improvements might also contain other types of changes, such as bug fixes. To ensure that only files that contain readability improvements are included in the oracle, we manually untangle commits by excluding the files that contain changes not related to readability.
(4) Randomly sample data points that do not contain readability improvements from the remaining commits for each project. We manually validate each commit, to make sure commits do not contain readability improvements. For example, an ambiguous commit that contains added functionality but also replaces pre-existing magic numbers with a constant.

2.3 Source Code Analysis Tools

In order to create a model that can distinguish between readability and non-readability changes, we must identify features that can capture the differences in both categories. We use seven different source code analysis tools that collect a wide variety of source code metrics at a file level. By running these tools on versions of the source code before and after a readability or non-readability change was made, we can capture the differences in metrics caused by the change. These differences then become the defining features for our model. We use all five of the tools originally explored in [16], and add two new tools: RSM and Coming. We describe the tools and the types of metrics they collect in the following paragraphs.

2.3.1 *SourceMeter.*

SourceMeter [27] is a static analysis tool that computes a large variety of source code quality metrics. These metrics are grouped into 6 categories: cohesion, complexity, coupling, documentation, inheritance, and size. Complexity metrics, for example, include Halstead Effort (HE), McCabe's Cyclomatic Complexity (McCC), and Weighted Methods per Class (WMC).

2.3.2 *Checkstyle.*

Checkstyle [7] is a static analysis tool which checks source code adherence to configurable rules. We use the standalone version of Checkstyle, along with two of the configuration files provided by Checkstyle, sunchecks.xml and googlechecks.xml, modified to add a warning for magic numbers. For each file in a commit, we run the tool on the version of the file before and after the change. We then compute the difference in the number of warnings between the before and after versions of a file.

2.3.3 *RSM.*

Resource Standard Metrics (RSM) [2] is a fast and lightweight command line tool for gathering source code metrics. RSM can calculate several metrics pertaining to file diff information, such as new comment lines, removed logical lines of code and added non-logical lines of code. We use RSM metrics to capture important changes that are often made by developers during readability improvements, for example adding and updating documentation or formatting changes.

2.3.4 *Coming.*

Coming [30] is a tool that mines code change patterns on git commits. Coming computes fine-grained code changes between two consecutive revisions, and can be used to analyze those changes to determine if they correspond to an instance of a change pattern. In this study, Coming is used for its ability to detect fine grained changed made in files at each commit in our dataset.

2.3.5 *PMD.*

PMD [1] is a cross-language static source code analyzer. It detects common poor programming practices and issues related to coding style, design, documentation, and performance. We use the warnings generated by PMD on a file before and after a commit to help identify differences between readability and non-readability improvements.

2.3.6 *ChangeDistiller.*

ChangeDistiller [17] as a tool that extracts and categorizes statement level changes in Java source code. ChangeDistiller uses the abstract syntax tree (AST) of the source code to extract fine grained changes using the change distiller algorithm [18]. Statement level source code changes are classified according to 41 different change types. For each file, we compute the total number of changes belonging to each of the 41 types.

2.3.7 *RefactoringMiner.*

RefactoringMiner detects refactorings across the history of Java projects, using the RMiner technique as proposed by Tsantalis et al. [51]. It supports 21 refactoring types, such as Extract Method, Move Method, Replace Variable with Method, and Parameterize Variable. RefactoringMiner has 98% precision and 87% recall.

2.4 Approach

2.4.1 *Model Building Process.*

Before we begin the model building process, we randomly extracted 10% of the data into a holdout or test set. Only the training set was used for the entire model building process, including all aspects of feature selection, model selection and hyperparameter optimization. The train/test split was conducted so that data from the same project could only exist in one of the two sets. This was done to prevent information leakage from the test to the train set, enabling test set performance to serve as an estimator of model's ability to generalize.

Model creation was performed incrementally using the training set, employing 10-fold nested cross validation [52]. During the cross validation phase, each combination of imputation technique, feature selection technique and model selection was performed for each fold. Once the model was built and trained on the training set, it was evaluated using performance metrics described below. The focus here was to identify and use the best, i.e., unbiased, model building *process* rather than building the best model. This model building process consists of several steps: feature scaling, imputation of missing values, feature pruning, feature selection, model selection, and hyperparameter optimization algorithm selection. Performing this process for each fold is critical in reducing selection bias as can happen when these steps are performed on the entire training and validation sets [3]. Once the best model building procedure was identified, we used the same methodology on the entire training set to build the model, and then evaluated its generalization ability on the test set that was set aside at the beginning of the process.

2.4.2 *Initial Feature Pruning.*

We performed an exploratory investigation of the features in our data to prune irrelevant features, such as line and column numbers. We also analyzed the number of missing datapoints that had missing data for every given feature. We found that that both ChangeDistiller and RefactoringMiner were missing for more than 60% of the columns, and they were thereby excluded. However, the removal of these two tools represented a significant loss of information for our model to learn from; namely the nature of changes and refactorings made. Hence, we looked for tools to replace them. In order to replace change information provided by ChangeDistiller, we use Coming, which provides change information at a finer granularity than ChangeDistiller. However, we were not able to find a suitable replacement for RefactoringMiner, and as a result, our model does not take into account any refactoring changes.

2.4.3 *Missing Value Imputation.*

Our data has several features which contain a significant amount of missing values (up to 863 out of 5100), due to tool failure. Therefore, missing value imputation was a major consideration in the model design process. We used four different types of single imputation for the purposes of building this model. In single imputation, missing values are filled using an estimate of what the value should be based. This can be done by simple using a summary statistic such as the mean of the non-missing values or using regression to predict the missing value based on other features. We tried using mean, median, and constant value imputation, as well as imputation by chained equations [55]. For constant value imputation, we simply replaced all missing values with zeros. In addition, when evaluating tree based models, we also provided the models with the data without any imputation due to their inherent robustness to and handling of missing values.

2.4.4 Automatic Feature Selection. Due to the large number of features from the static analysis tools, we performed automatic feature selection before providing training examples to all the models we evaluated. We perform this in two passes. In the first pass, we remove features with 0 variance. This is done before the cross-validation phase, to remove features that are clearly redundant. Next, during the cross validation, we apply a random forest based feature elimination algorithm known as the Boruta algorithm [43]. The Boruta algorithm is a wrapper method that uses random forests to perform feature selection. It offers several benefits over Recursive Feature Elimination (RFE) [19], a widely used feature selection technique. Unlike RFE, which strives to create the minimum subset of relevant features, Boruta captures *all-relevant* features; features that could potentially be pruned by RFE due to their low importance are selected by Boruta if they are relevant to making a prediction in a relatively number of instances. Although this might result in a slightly larger set of features than RFE, it also holds the potential to improve model performance, as we found to be the case during our evaluation. Secondly, it does not always remove correlated features, which is beneficial for model performance in some scenarios. For example, difference in the number of logical lines of code and lines of code changed by a commit might be highly correlated, but the presence of both features is important in the context of classifying readability improving commits; removing one removes important information regarding the number of formatting changes that did not affect the functionality of the source code.

2.4.5 Model Selection. We started the model selection process by evaluating several classical algorithms, including Logistic Regression, Support Vector Machines, Naive Bayes, k-Nearest Neighbors and Decision Trees. Linear models performed very poorly on our dataset, which is to be expected considering the high degree of multicollinearity of the features. For example, RSM reports effective, logical and total lines of code, which are highly correlated. Similarly, Naive Bayes does not perform any better due to its assumption of conditional independence. k-Nearest Neighbors performed marginally better. The models that did perform well were tree based methods, including ensemble methods. Tree based methods have been used specially in the domain of fault prediction [22, 33, 54, 60]. The main benefits they offer in ours case is that they are not highly affected by multi-collinearity, they are robust to noise, perform implicit feature selection, and are easier to interpret than other models listed above. We evaluated 5 tree based methods, including Random Forests, Gradient Boosted Trees, AdaBoost, XGBoost [10] and LightGBM [23]. Out of these, we selected XGBoost based on the quality and consistency of its performance.

2.4.6 Parameter Optimization. The parameter optimization procedure we use is 10 fold cross validated grid search. For XGBoost, we optimize the following hyperparameters: maximum depth, number of estimators and minimum weight for a new child node.

2.5 Evaluation

We evaluate our models using the following metrics:

2.5.1 Precision. Precision is defined as the ratio of number of true positives (TP) to the sum of the TP and the false positives (FP). Precision can vary from 0 to 1, the latter indicating a perfect precision.

2.5.2 Recall. Recall is calculated as the ratio of the number of true positives to the sum of the true positives and the false negatives (FN). The higher the recall the better, with 1 indicating a perfect recall.

2.5.3 F-measure. F-measure or F_1 score is the harmonic mean of precision and recall. As precision and recall are inversely related, F_1 score allows to combine both metrics in one score.

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

2.5.4 Area Under Curve (AUC)-Receiving Operator Characteristic (ROC). ROC is a plot of the TP rate against the FP rate at various discrimination thresholds. The area under ROC is close to 1 when the classifier performs better and close to 0.5 when the classification model is poor and behaves like a random classifier.

2.5.5 Matthews Correlation Coefficient (MCC). MCC is a measure used in machine learning to assess the quality of a binary classifier especially when the classes are unbalanced [31].

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(FN + TN)(FP + TN)(TP + FN)}}$$

Values range from -1 to 1. Zero means that the approach performs like a random classifier. Other correlation values are interpreted as follows: $MCC < 0.2$: low, $0.2 \leq MCC < 0.4$: fair, $0.4 \leq MCC < 0.6$: moderate, $0.6 \leq MCC < 0.8$: strong, and $MCC \geq 0.8$: very strong [12].

2.5.6 Comparison to state-of-the-art models. Building on our work in [16], we compare our model to three state-of-the-art readability models which assess different aspects of the source code. The first model we used is proposed by Scalabrino et al. [45] and uses metrics that measure the quality of the source code lexicon as a proxy of readability. The second model we consider is proposed by Dorn et al. as a generalizable model for source code readability [15]. This model relies on features like visual, spatial, alignment, and linguistic aspects of the source code. We use the implementation provided by Scalabrino et al. in their paper comparing state-of-the-art readability models [45]. The last model we consider is also implemented by Scalabrino et al. [45], but is a combination of multiple state-of-the-art readability models. We refer to this model as the *Combined model.* It combines the first two models as well as Buse & Weimer's model [8] and Posnett's model [39]. This model is shown to have the highest accuracy scores when evaluated against all the individual models on the same dataset.

The model we propose in this paper classifies the changes introduced at a file in a commit as either a readability improvement or not. However, the state-of-the-art models we consider measure the readability of a file on a scale. To make valid comparisons with our model, we measure the readability scores of these models on the before and after snapshots of a file for a given commit, and mark an

Metric	CV(%)	Test(%)
Precision	69.4	79.2
Recall	59.3	67
MCC	0.41	0.39
F-1	63.9	72.6
ROC-AUC	70	70

Table 2: Precision, Recall and F-1 score of proposed model for 10 fold cross validation (CV) and the test set.

Metric	Non-Readability	Readability
Precision	72.8	69.4
Recall	80.6	59.3
F1	76.5	63.9

Table 3: Precision, Recall and F-1 score of proposed model for 10 fold cross validation.

	P. Non-Readability	P. Readability	Total
Non-Readability	141	52	193
Readability	98	199	297
Total	239	251	490

Table 4: Confusion matrix for the proposed model.

increase in the score as a readability improvement. Scores that stay the same or decrease are not considered readability improvements. This allows us to compare these models to our approach.

2.5.7 Shapley Additive Explanations (SHAP). In order to investigate the impact of different source code metrics on the model's predictions, we employ the use of SHapley Additive exPlanations [28], a unified framework used to identify the role of individual features in a model's predictions using a game theoretic approach. For a given input, SHAP is able to identify whether a given feature raises or decreases the model's output, and the weight of the feature for making that decision. For tree based models, SHAP offers an exact algorithm to determine the contribution of each feature in a decision made by the model. We utilize it to identify features that are aligned with developer's perceptions of incremental readability improvements.

3 RESULTS

3.1 RQ$_1$: *Can we use machine learning to capture readability improvements made in practice?*

Table 2 reports the performance of the proposed model for the 10 fold cross validation and the hold out test set. The model achieves an average precision and recall of 69.4% and 59.3% respectively. This results in an average F-1 score of 63.9% and MCC of 0.41 (moderate). On the test set, the model achieves better performance, at 79.2% precision, 67% recall, 0.39 MCC (fair), 72.6% F-1 and 70% AUC-ROC. The model performs better on the test set, which could be due to several factors. For example, the model training on a larger dataset, as well as the nature of the projects that were randomly selected for the test set.

Table 4 shows the confusion matrix of our model's predictions for the test set. 297 (60.1%) out of the 490 examples in our test set were readability improvements. Out of these, our model was able to classify 199 correctly, while classifying the remaining 98 examples incorrectly. Hence, our model is able to identify readability commits with a reasonable degree of recall (67%). However, it is conservative in its approach, falsely classifying 98 examples.

We manually investigated a sample of training examples to identify the nature of the mistakes made by the model. We observed that for some files, the readability commit affected only a single line. Since many of the tools we use to collect metrics measure at a file level, this would result in most of the these metrics remaining the same before and after the readability commit, and as a result, most of the features of our model would be zero. We only expected to find metrics reported from coming and RSM for these cases. Coming detects changes at sub-statement granularity, whereas RSM measures changes not just in lines of code, but also effective lines of code, comment lines of code and logical lines of code.

This was especially the case for certain examples where only formatting changes were made to a very small number of lines of code (less than 5). A similar issue lies in changes that affect only comments, as most tools do not collect exhaustive metrics for source code comments; the only metrics that picked up changes in comments were reported by RSM. Coming supports detection of changes in comments according to its documentation, but it didn't detect any for our dataset despite their presence.

Another observation we made during our manual investigation was that often, non-readability changes introduce changes in a source code that result in features changing in a manner similar to that for readability changes. For example, in one case, we observed that a method was introduced into a class to add functionality. In order to add this method, source code from another method had to be refactored in a way that improves readability. As a result, the signal for both the features associated with readability improvements and non-readability changes were strong.

RQ$_1$ Summary: Our readability model is able to identify developer perceived readability improvements in source code with 79.2% precision and 67% recall on the test set.

3.2 RQ$_2$: *What features align with developers' perception of readability improvements in practice?*

A visualization of the SHAP values for our top features is shown in Figure 2. In the figure, for each feature, the SHAP values are plotted on the x-axis, while feature values are represented on the y axis, as a scatterplot. Features on the y-axis are sorted in descending order of their overall importance to the model. Each point is shaded based on the gradient shown; when feature values are high, their color is closer to red, whereas when feature values are low, their color is closer to blue. The SHAP values shown on the x-axis indicate both the magnitude and the direction of the impact of a given point. Combining these two pieces of information provides us a good picture of the impact of a feature on the model's predictions.

For example, for the feature UPD_Class_Package, if we consider the rightmost point that is colored red, it means that a high value of UPD_Class_Package results in a large positive impact on the model's output.

We observe from Figure 2 that when the number of effective lines of code that change (Dif_Av.eLOC% and Dif_eLOC), change in comment density (CD) and change in comment lines of code (Dif_Comment) are high, the model tends to classify the change as a readability improvement. This falls in line with our intuition; in readability changes, developers tend to modify existing lines of code rather than introducing new lines of code, introduce new comments (thereby increasing comment density) and update existing comments. On the other hand, we observe that high values of the number of effective lines, logical lines and comment lines of code that remain unchanged (Equ_eLOC, Equ_lLOC and Equ_Comment) drive the model towards classifying a change as a non-readability change. In addition, an increase in the total lines of code (TLOC) is associated with non-readability change prediction by the proposed model. This indicates that changes in which existing lines of code are not changed, but instead either new lines are added or existing lines are completely removed tend to be identified by the proposed model as non-readability changes. We investigated some of these non-readability changes to find that this was indeed the case, especially for changes that introduced new functionality.

We note that the several lines of code metrics discussed above are highly correlated, yet highly important to the model. Although their inclusion would give the impression of redundancy, we hypothesize that they help the model detect large formatting changes in the source code. For example, for a change that only consists of a large number of formatting changes, the number of logical lines of code remains the same, but the total number of lines of code is likely to change. The model, however, is unaware of the nature of the formatting change, i.e. whether the change had a positive or negative impact on readability. We manually inspected several examples in our dataset to find this to be the case, however, further investigation on a larger data set is needed to establish conclusively whether this holds true in general.

Several software complexity metrics, such as Halstead Effort (HEFF), Halstead Program Length (HPL), McCabe's Cyclomatic Complexity (McCC) and Halstead's Number of Delivered Bugs (HNDB), drive the model to classify a change as a non-readability change. Although readability and complexity are separate dimensions of software quality, we observe that for our dataset, developer perceived readability and several complexity metrics are correlated. This aligns with the observation we made earlier regarding how lines of code metrics change for non-readability changes; they tend to introduce new lines of code that can potentially have a negative impact on source code complexity.

RQ$_2$ Summary: Developer perceived readability improvements are characterized by greater changes to existing lines of code, while non-readability changes tend to add new lines of code and leave existing lines untouched. Non-readability changes are also associated with increased values of several software quality metrics such as Halstead Effort, Halstead Number of Delivered Bugs and McCabe's Cyclomatic Complexity.

	Dorn	Scalabrino	Combined
Precision	42.6	41.34	44.05
Recall	40.25	40.00	36.71
F-1	39.70	38.89	38.19
MCC	0.00	-0.01	0.03
ROC-AUC	50.55	49.58	51.85

Table 5: Mean 10 fold cross validation metrics for three state of the art readability models.

	Dorn	Scalabrino	Combined
Precision	60.00	64.51	62.22
Recall	41.14	47.14	37.71
F-1	49.00	54.47	46.96
MCC	-0.01	0.07	0.02
ROC-AUC	49.46	53.62	51.24

Table 6: Test set metrics for three state of the art readability models.

3.3 RQ$_3$: *How does the proposed model perform when compared to existing state-of-the-art readability models?*

In order to answer this research question, we use three state of the art readability models: Dorn's model, Scalabrino's model and the combined model, and compare their performance on our test set to the proposed model. For each model, we calculate the readability scores for a file before and after a commit and mark the model as having identified the commit as a readability improvement if the score increases after the commit is made. We then calculate the same metrics as we did for the proposed model on the predictions of the readability models. We do this for our test set, as well as the 10 combinations of train and validation sets that were generated during cross validation phase of training the proposed model.

Figures 5 and 6 show the evaluation metrics for the readability models for the 10 fold cross-validation and the test set respectively. Overall, we observe that state of the art readability models perform better in the test set as compared to the cross validation, similar to the proposed model. The Combined model obtains the best precision during cross validation at 44.05%, followed by Dorn's model and Scalabrino's model at 42.6% and 41.34% respectively. For recall, Dorn's model performs the best at 40.25%, followed by Scalabrino's model at 40.00% and the Combined model at 36.71%. For the test set, Scalabrino's model attains the best precision at 64.51%, followed by the Combined model at 62.22% and Dorn's model at 60%. Scalabrino's model also performs best in terms of recall, at 47.14% followed by Dorn's model and the combined model at 41.14% and 37.71% respectively. The proposed model performs better than all three state of the art models with a precision of 69.4% and recall of 59.3% for the cross validation and a precision of 79.2% and recall of 67% for the test set. We note that Mathew's Correlation Coefficient for all three state of the art models are close to 0 (low) for both the cross validation and the test sets, whereas for the proposed model,

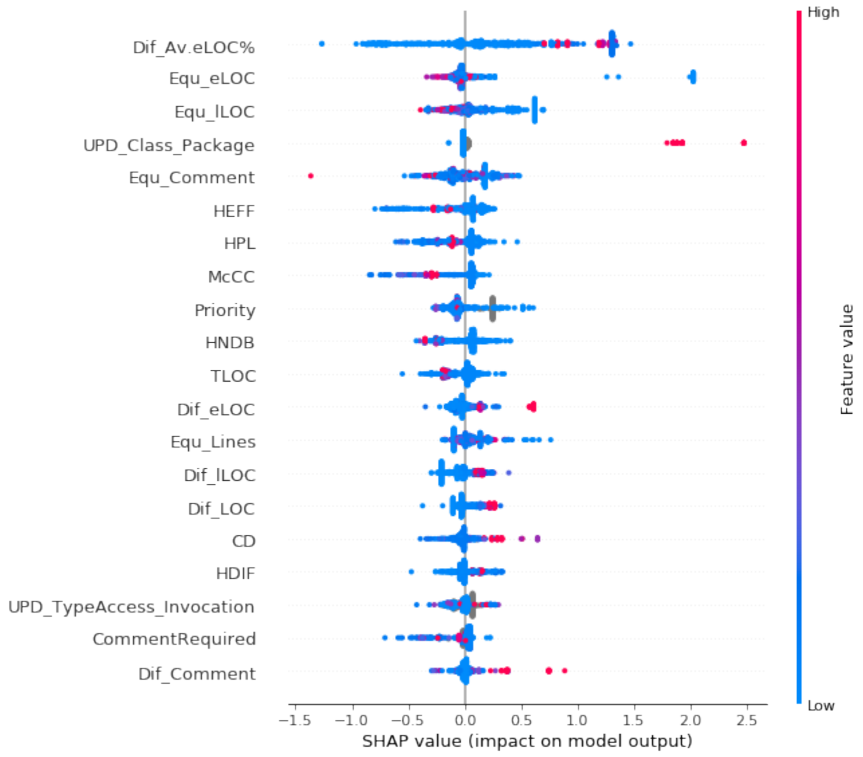

Figure 2: Visualization of top feature impact on the proposed model

it is 0.41 (moderate) and 0.39 (fair) respectively. Mathew's Correlation Coefficient has been shown to be a more accurate assessment of model performance in binary classification settings, especially when classes are imbalanced [11]. This is the case for our test set, where 60.6% of the examples are readability improvements.

We investigated some of the examples that were incorrectly classified by the state of the art readability models. We observed that for readability improvements where only comments were changed, these models were not able to detect an improvement in readability. This issue also affects the performance of the proposed model, but it is able to somewhat mitigate the effects by the use of the various lines of code metrics reported by RSM.

We find that several of the Checkstyle and PMD coding violations considered by our model play a significant role in correctly classifying files that were incorrectly scored by the 3 readability models. One such style violation is comment required, reported by PMD, and is activated when comments are missing from specific code elements. This violation has a strong negative correlation with readability improvements, i.e. there are several examples where this warning exists for the 'before' version of a file but disappears in the 'after' version, indicating that the developer added a comment to document a source code element. The proposed model is able to detect this decrease in rule violation, whereas the readability models are unable to detect this change. Another benefit of our model's usage of Checkstyle and PMD is that these are customizable tools that professional developers use, and the information obtained from them are highly relevant in terms of identifying developer perceived improvements in readability.

Lastly, we note that these state of the art readability models are not designed to be used in the context our proposed model targets: identifying *incremental* improvements in developer perceived readability in practice. These tools were developed to assign an overall readability score to a file, and perform well when discerning differences in readability between two files with very different readability.

RQ$_3$ Summary: Our proposed model is able to significantly outperform three state of the art readability models, Scalabrino, Dorn, and the combined model, in the context of incremental readability improvements. For the test set, we obtain a precision of 79.2% and recall of 67%, which is significantly higher than the compared models, the best of which achieves 64.51% precision and 47.14% recall.

4 DISCUSSION

In this section we discuss the predictions made by the proposed model and the state of the art readability models and the implications of this work.

Figure 3 shows an example of a readability improving change that was misclassified by all 3 state of the art models, but was correctly classified by the proposed model. The author of this commit explicitly state that the method invocation *Condition.column* is replaced by *column* to improve readability. The three state of the art models we consider in this paper all assigned the same readability

```
assertEquals(new Select().from(CacheableModel.class).
        where(Condition.column(CacheableModel$Table.ID).is(id))
        where(column(CacheableModel$Table.ID).is(id))
        .querySingle(), cacheableModel);
```

Figure 3: A commit that improves readability— misclassified by state of the art readability models but correctly classified by proposed model

score to the before and after versions of the snippet. The proposed model predicts this change to be a readability improvement with a probability of 75.8%. When we investigated the features that contributing to this prediction, we found that the Coming feature UPD_Type_Invocation and several lines of code metrics to be the most significant contributors. UPD_Type_Invocation represents a change in which the type of a method invocation has been changed, as in the case of this commit. The proposed model combines this information with the fact that the effective, logical and total lines of code had insignificant changes, to make the correct prediction.

```
-       inflater.inflate(R.menu.call_log_options_new, menu);
+       inflater.inflate(R.menu.call_log_options, menu);
```

Figure 4: A readability improvement incorrectly identified by the proposed model

Figure 4 shows an example of a readability improvement incorrectly classified as a non-readability change by the proposed model. The change shown in the figure is the only change in the 184 lines of code in the file. We noticed this to be a trend with the proposed model; it has difficulty classifying changes which affect a very small portion of the entire source code. We hypothesize that this is due to our use of several file level metrics; small changes in the files result in small changes in these metrics which the proposed model is not able to effectively utilize to discern the type of change. Moreover, the fine grained change information provided by Coming are not able to aid the model in these cases. Lastly, the proposed model does not consider refactorings; in the example presented in Figure 4, a rename refactoring takes place. However, no metric used for the proposed model is able to capture the nature of this change. This is one area of improvement that will be addressed in future work.

Another opportunity for improvement of the proposed model is the incorporation of metrics tools that perform dynamic analysis, such as SonarQube. Warnings detected by SonarQube could be potentially helpful as they align with qualitative observations of readability improvements made by developers.

For example, removal of redundant strings, replacing commented out source code, and removal of unused variables. We ran SonarQube on the expanded dataset and were only able to gather metrics for 1000 data points, which would is would be too few for the development of a machine learning model. SonarQube require compilation and building of projects in order to successfully analyze files, all of the failed instances are the result of non-passing builds at a commit's SHA. Future work could involve manual intervention to fix failing builds in order to benefit from these dynamic analysis metrics.

5 RELATED WORK

5.1 Source Code Readability Models

Buse and Weimer [8] conduct a study investigating code readability metrics and find that structural metrics such as the number of branching and control statements, line length, the number of assignments, and the number of spaces negatively affect readability. They also show that metrics such as the number of blank lines, the number of comments, and adherence to proper indentation practices positively impact readability.

Posnett et al. [39] show that metrics such as McCabe's Cyclomatic Complexity [32], nesting depth, the number of arguments, Halstead's complexity measures [21], and the overall number of lines of code impact code readability. An empirical evaluation conducted on the same dataset used by Buse and Weimer [8] indicates that the model by Posnett et al. is more accurate than the one by Buse and Weimer.

Scalabrino et al. [45] propose and evaluate a set of features based entirely on source code lexicon analysis (e.g., consistency between source code and comments, specificity of the identifiers, textual coherence, comments readability). The model was evaluated on the two datasets previously introduced by Buse and Weimer [8] and Dorn [15] and on a new dataset, composed by 200 Java snippets, manually evaluated by nine developers. The results indicate that combining the features (i.e., structural and textual) improves the accuracy of code readability models.

Borstler et al. [6] propose a simple readability measure for software, SRES, which is based on metrics for word and sentence length. SRES was shown to correlate well to textbook examples. Mi et al. [34] propose the use of CNNs to improve the classification of source code readability. They are able to improve upon existing models by up to 17%. However, their model does not target readability improvements made in practice.

5.2 Code Quality Metrics in Practice

Code quality metrics are at the core of many approaches supporting software development and maintenance tasks. They have been used to automatically detect code smells [24, 36], to recommend refactorings [35, 40], and to predict the code fault- and change-proneness [20, 29, 61]. Some of these applications assume that a strong link between code quality as assessed by metrics and as perceived by developers exists.

Scalabrino et al. [44] perform an extensive evaluation of 121 existing as well as new code-related [8, 15, 39, 46], documentation-related ([46] and 2 newly introduced), and developer-related (3 newly introduced) metrics. They try to (i) correlate each metric with understandability and (ii) build models combining metrics to assess understandability. To do this, they use 444 human evaluations from 63 developers and obtain a bold negative result: none of the 121 experimented metrics is able to capture code understandability, not even the ones assumed to assess quality attributes apparently related, such as code readability and complexity.

Indeed, code smell detectors and refactoring recommenders should be able to identify design flaws/recommend refactorings that align with developer's perception in practice. While such an assumption seems reasonable, there is limited empirical evidence supporting it.

Pantiuchina et al. [38] aim at bridging this gap by empirically investigating whether quality metrics are able to capture code quality improvement as perceived by developers. While previous studies [4], [14], [41] surveyed developers to investigate whether metrics align with their perception of code quality, they mine commits in which developers clearly state in the commit message their aim of improving one of four quality attributes: cohesion, coupling, code readability, and code complexity. They use state-of-the-art metrics to measure the changes relative to the specific quality attribute it targets. To measure code readability, the authors exploit two state-of-the-art metrics.

The first one was presented by Buse and Weimer et al. [8] and the second metric is the one proposed by Scalabrino et al. [46]. Code readability is the quality attribute for which the authors observed the less perceivable changes in the metrics' values. This holds for both metrics they employed, despite the metrics use totally different features when assessing code readability. The two metrics report only 28% [8] and 38% [46] of the modified classes as improving their readability after the changes implemented in the commits.

In our initial work [16], we demonstrate that state-of-the-art readability models are able to capture readability improvements as explicitly tagged by developers in commits messages. We underscore the need for a readability model that is able to capture changes in incremental improvements.

In this study, we expand upon our preliminary work by extending the dataset and proposing a model that is able to identify readability improvements in practice.

6 THREATS TO VALIDITY

This section discusses the threats to the validity of our study, as they pertain to aspects of conclusion, internal, external, construct, and reliability [56, 59].

Threats to *conclusion validity* concern the relation between the treatment and the outcome. We report results using appropriate diagnostics for the performance of the ML algorithms, such as ROC and MCC and when discussing findings we keep into account acceptable ranges for ROC and MCC (i.e., ROC ≥ 0.5 and MCC $>$ 0).

Threats to *internal validity* concern external factors that could affect the variables and the relations being investigated. The biggest threat to internal validity is the experience of the developers who wrote the code and commit messages used in our dataset. Although we evaluated the commit messages manually, we can not be certain that the developers have sufficient understanding about what makes readable or unreadable source code, or that their perceptions of readability are generalizable to the entire community of open source developers. To mitigate this threat we control the quality of the repositories used in our dataset by only using engineered projects.

Threats to *construct validity* concern the relation between theory and observation. One of the major threats to construct validity in this work pertains to the creation of the oracle. Misclassifying commits in which developers state readability improvements is possible. To mitigate this threat, two annotators, authors of this paper, went through the set of readability and non-readability commits to ensure that developers' changes can be classified as such. In case of a doubt, the commit was excluded from the dataset. Another threat to construct validity are the metrics considered in the paper. We select a variety of static analysis tools to generate metrics on commit data before and after changes are implemented. These metrics are then fed as features into our model. The model depends on the accuracy of these tools. Also, different tools could lead to different results.

Threats to *external validity* concern the generalizability of the findings outside the experimental settings. Potential threats to external validity in this study include the selection of sampled open source applications, which may not be representative of the studied population. To mitigate this threat we only sampled engineered open source Java projects from GitHub as identified by Reaper [37]. However, it is possible that a different dataset leads to different conclusions.

Threats to *reliability validity* concern the ability to replicate a study with the same data and to obtain the same results. We use open-source software projects whose source code is available. Moreover, we provide all the necessary details to replicate the model and analysis in our online replication package [42]

7 CONCLUSION

We presented a machine learning based model for detecting incremental improvements in developer perceived source code readability in practice. Our proposed model is able to correctly classify readability commits with a precision of 79.2% and recall of 67% in our test set consisting of data points from commits never seen before by the model.

We then identified several source code metrics that are used by our model to achieve its performance. Several of these metrics were directly related to formatting changes, for which no direct metric exists, but our model is able to leverage information about lines of code and PMD and Checkstyle warnings to mitigate the lack of this information. We then compared the performance of our model to three state of the art readability models and showed that the proposed model performs significantly better than these models at identifying readability improvements.

The work we presented is a first step towards building a model that can assign a readability score to indicate the change in readability for each file in a commit. Such a model could then be used throughout the software development life cycle as a measure for software quality. It could be integrated in during into code review tools to ease the review process for reviewers.

REFERENCES

[1] [n. d.]. PMD. https://pmd.github.io/latest/index.html, last accessed on January 2020.
[2] [n. d.]. Resource Standard Metrics (RSM). https://msquaredtechnologies.com/Resource-Standard-Metrics.html, last accessed on January 2020.
[3] Christophe Ambroise and Geoffrey J McLachlan. 2002. Selection bias in gene extraction on the basis of microarray gene-expression data. *Proceedings of the national academy of sciences* 99, 10 (2002), 6562–6566.
[4] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2013. An empirical study on the developers' perception of software coupling. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 692–701.
[5] David Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. 2009. To CamelCase or Under_score. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. 158–167.
[6] Jürgen Börstler, Michael E Caspersen, and Marie Nordström. 2016. Beauty and the Beast: on the readability of object-oriented example programs. *Software quality journal* 24, 2 (2016), 231–246.

[7] Oliver Burn. 2005. Checkstyle homepage. *URL http://checkstyle. sourceforge. net/. last accessed in March* 14 (2005).

[8] Raymond P.L. Buse and Westley R. Weimer. 2010. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering (TSE)* 36, 4 (July 2010), 546–558.

[9] B. D. Chaudhary and H. V. Sahasrabuddhe. 1980. Meaningfulness As a Factor of Program Complexity. In *Proceedings of the ACM Annual Conference*. 457–466. http://dl.acm.org/citation.cfm?id=810001&dl=ACM&coll=DL&CFID=953873442&CFTOKEN=18528998.

[10] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[11] Davide Chicco and Giuseppe Jurman. 2020. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC genomics* 21, 1 (2020), 6.

[12] Jacob Cohen. 1988. *Statistical power analysis for the behavioral sciences* (2nd edition ed.). Lawrence Earlbaum Associates.

[13] T. A. Corbi. 1989. Program understanding: Challenge for the 1990s. *IBM Systems Journal* 28, 2 (1989), 294–306. http://dl.acm.org/citation.cfm?id=97124.

[14] Steve Counsell, Stephen Swift, Allan Tucker, and Emilia Mendes. 2006. Object-oriented cohesion subjectivity amongst experienced and novice developers: an empirical study. *ACM SIGSOFT Software Engineering Notes* 31, 5 (Sept. 2006), 1–10.

[15] Jonathan Dorn. 2012. *A general software readability model*. Master's thesis. University of Virginia.

[16] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. 2019. Improving source code readability: theory and practice. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2–12.

[17] Beat Fluri and Harald C. Gall. 2006. Classifying Change Types for Qualifying Change Couplings. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. 35–45.

[18] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering* 33, 11 (2007).

[19] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. 2002. Gene selection for cancer classification using support vector machines. *Machine learning* 46, 1-3 (2002), 389–422.

[20] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. 2005. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Transactions on Software Engineering (TSE)* 31, 10 (October 2005), 897–910.

[21] Maurice H Halstead. 1977. Elements of software science. (1977).

[22] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. 2010. Revisiting Common Bug Prediction Findings Using Effort-Aware Models. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. 1–10.

[23] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in neural information processing systems*. 3146–3154.

[24] Michele Lanza and Radu Marinescu. 2007. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.

[25] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *Proceedings of International Conference on Program Comprehension (ICPC)*. 3–12.

[26] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2007. Effective Identifier Names for Comprehension and Memory. *Innovations in Systems and Software Engineering* 3, 4 (December 2007), 303–318.

[27] FrontEndART Software Ltd. [n. d.]. SourceMeter. https://www.sourcemeter.com/, last accessed on March 15, 2019.

[28] Scott M Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 4765–4774. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

[29] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. 2008. Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems. *IEEE Transactions on Software Engineering (TSE)* 34, 2 (2008), 287–30.

[30] Matias Martinez and Martin Monperrus. 2019. Coming: a tool for mining change pattern instances from git commits. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 79–82.

[31] B. W Matthews. 1975. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA* 2, 405 (1975), 442–451.

[32] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering (TSE)* SE-2, 4 (1976), 308–320.

[33] Thilo Mende and Rainer Koschke. 2009. Revisiting the evaluation of defect prediction models. In *Proceedings of the International Conference on Predictor Models in Software Engineering (PROMISE)*. 7:1–7:10.

[34] Qing Mi, Jacky Keung, Yan Xiao, Solomon Mensah, and Yujin Gao. 2018. Improving code readability classification using convolutional neural networks. *Information and Software Technology* 104 (2018), 60–71.

[35] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. 2016. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering* 21, 6 (2016), 2503–2545.

[36] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering (TSE)* 36, 1 (January-February 2010), 20–36.

[37] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.

[38] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. 2018. Improving Code: The (Mis) perception of Quality Metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 80–91.

[39] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. 2011. A Simpler Model of Software Readability. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. 73–82.

[40] Kata Praditwong, Mark Harman, and Xin Yao. 2011. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering* 37, 2 (2011), 264–282.

[41] Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2011. Using Structural and Textual Information to Capture Feature Coupling in Object-oriented Software. *Empirical Software Engineering (EMSE)* 16, 6 (December 2011), 773–811.

[42] Devjeet Roy. 2020. Online Replication Package. https://github.com/devjeetr/a-model-to-detect-readability-improvements-in-incremental-changes

[43] Witold R Rudnicki, Marcin Kierczak, Jacek Koronacki, and Jan Komorowski. 2006. A statistical method for determining importance of variables in an information system. In *International Conference on Rough Sets and Current Trends in Computing*. Springer, 557–566.

[44] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2019. Automatically Assessing Code Understandability. *IEEE Transactions on Software Engineering* (2019), 1––1. https://doi.org/10.1109/tse.2019.2901468

[45] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *Journal of Software: Evolution and Process* 30, 6 (2018), e1958.

[46] Simone Scalabrino, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2016. Improving code readability models with textual features. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. 1–10.

[47] Ben Shneiderman and Richard Mayer. 1975. *Towards a cognitive model of progammer behavior.* Technical Report 37. Indiana University.

[48] Thomas A. Standish. 1984. An Essay on Software Reuse. *IEEE Transactions on Software Engineering (TSE)* 10, 5 (September 1984), 494–497.

[49] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. 1996. The Effects of Comments and Identifier Names on Program Comprehensibility: An Experiential Study. *Journal of Program Languages* 4, 3 (1996), 143–167.

[50] Rebecca Tiarks. 2011. What Maintenance Programmers Really Do: An Observational Study. In *Proceedings of the Workshop Software Reengineering (WSR)*. 36–37.

[51] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. (2018), 483–494.

[52] Sudhir Varma and Richard Simon. 2006. Bias in error estimation when using cross-validation for model selection. *BMC bioinformatics* 7, 1 (2006), 91.

[53] Anneliese von Mayrhauser, A. Marie Vans, and Adele E. Howe. 1997. Program Understanding Behaviour During Enhancement of Large-scale Software. *Journal of Software Maintenance: Research and Practice* 9, 5 (September 1997), 299–327.

[54] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. 2010. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering (EMSE)* 15, 3 (June 2010), 277–295.

[55] Ian R White, Patrick Royston, and Angela M Wood. 2011. Multiple imputation using chained equations: issues and guidance for practice. *Statistics in medicine* 30, 4 (2011), 377–399.

[56] Claes Wohlin, Per Runeson, Höst Martin, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2000. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers.

[57] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. 1981. The Effect of Modularization and Comments on Program Comprehension. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 215–223.

[58] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering (TSE)* 44, 10 (Oct 2018), 951–976. https://doi.org/10.1109/TSE.2017.2734091

[59] Robert K. Yin. 1994. *Case Study Research: Design and Methods* (2nd ed.). Sage Publications.

[60] Yuming Zhou and Hareton Leung. 2006. Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults. *IEEE Transactions on Software Engineering (TSE)* 32, 10 (October 2006), 771–789.

[61] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 531–540.