TUM School of Computation, Information and Technology

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation in Informatik

On Machine Learning Assisted Software Maintainability Assessments

Markus Schnappinger

TUM School of Computation, Information and Technology

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

On Machine Learning Assisted Software Maintainability Assessments

Markus Schnappinger

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 29.09.2022 bei der Technischen Universität München eingereicht und durch die School of Computation, Information and Technology am 26.03.2023 angenommen.

Acknowledgments

Pursuing a Ph.D. has helped shaping the person I am today, both professionally and personally. I am grateful for the opportunity to acquire and hone important skills, such as striving for continuous improvement, critical thinking, identifying the big picture, and acknowledging the limitations of both existing and novel ideas. This experience was only possible with the help of the great people around me.

First and foremost, my sincere thanks go to my supervisor Alexander Pretschner. He taught me how to think and reflect, encouraged me to put my arguments forward with a strong stand, and helped to recognize when to focus on details and when to consider the big picture. It is widely known that *'it depends'* [Pretschner, recurring]. Over the years, I learned to respect the existence of different perspectives even more. A notable exception to the rule is our discussion on what makes coffee good coffee, where a universal truth does indeed exist. Furthermore, I have learned to articulate my thoughts concisely and to explain complex matters in simple words. In that sense: Thank you, Alex!

Working on this dissertation would not have been possible without a supportive industrial partner. I had the pleasure of working with inspiring people at itestra GmbH. My gratitude goes to Dr. Markus Pizka for making this project possible and contributing his immense experience; and Dr. Arnaud Fietzke for his valuable remarks and feedback in our regular discussions.

I am grateful to have spent this time in such a supportive research group. Many members started out as colleagues and quickly became friends. I admire your technical expertise, your inspiring ideas, and your ability to effortlessly deep-dive into each other's non-trivial research problems within seconds. That said, I appreciate your patience in listening to my unsolicited advice, enduring unnecessarily detailed reviews, and letting me scribble all over your whiteboards. Most of all, it was a pleasure to share the stable with the best office mate. It will be hard to find a comparable work environment with such great people.

Dissertations are hardly possible without the support and encouragement of your loved ones. I have enjoyed a great deal of this support, first and foremost from my parents and my sister Julia. Now there are actually two of those 'Dr. Schnappinger' — I can only imagine how proud our grandparents are and would be.

Eventually, lots of love goes to that very special person, who makes me feel like I can accomplish everything, who is unconditionally rooting for me despite the occasional overtime or weekend work, and who keeps me down to earth at the same time. Sometimes words fail to express gratitude.

Zusammenfassung

Problemdomäne Die Wartung eines Softwaresystems ist für einen großen Teil der Gesamtkosten dieses Systems verantwortlich. Daher ist die Wartbarkeit eines Systems für seine langfristige Rentabilität von entscheidender Bedeutung. Um die Wartbarkeit steuern und gezielt verbessern zu können, müssen Systemeigentümer sie zunächst *bewerten*. Die meisten gängigen Analysewerkzeuge liefern dafür jedoch nur ungenaue Warnungen oder Metriken, die wiederum vom Anwender interpretiert werden müssen. Bewertungen durch Experten gelten als zuverlässiger, präziser und vertrauenswürdiger als rein toolbasierte Analysen. Sie stellen damit eine leistungsstarke Methode dar, sind aber meist mit erheblichen Aufwänden verbunden und erfordern für die Durchführung erfahrene Experten. In einem industriellen Umfeld ergibt sich aus der Größe der zu analysierenden Systeme der Bedarf nach einem möglichst effizienten und toolgestützten Analyseprozess. Eine Möglichkeit, die Vorteile von schnellen automatisierten Ansätzen und präzisen Expertenbewertungen zu kombinieren, ist der Einsatz von Machine Learning-Techniken zur Klassifizierung der Wartbarkeit, wie Experten sie wahrnehmen würden.

Offene Forschungsfragen Bestehende Arbeiten verwenden zur Klassifizierung der Wartbarkeit anstelle von Expertenbewertungen oftmals problematische Ersatzgrößen. Werden Expertenbewertungen herangezogen, stützen sich diese meist auf die subjektive Meinung eines Einzelnen oder einer sehr kleinen Gruppe. Dies führt zu Verzerrungen und schränkt die Übertragbarkeit der trainierten Modelle ein. Des Weiteren ist nur wenig darüber bekannt, wie Wartbarkeitsanalysen im industriellen Maßstab im Detail durchgeführt werden. Folglich ist unklar, wie genau die entwickelten Modelle in diesem Kontext eingesetzt und die Analyse unterstützen können.

Lösungsansatz Um diese Lücken zu schließen, integrieren wir Expertenbewertungen und maschinell erlernte Klassifizierungen in ein gemeinsames Framework. In einem ersten Schritt analysieren wir die aktuelle Praxis der Wartbarkeitsanalyse und bestimmen, welche Aspekte des Prozesses automatisiert werden können. Wir konzentrieren uns dabei auf die Wartbarkeit von Java-Klassen und erstellen einen öffentlich zugänglichen Datensatz, der den Konsens mehrerer Experten bezüglich der Wartbarkeit des Quellcodes beinhaltet. Auf der Grundlage dieses Datensatzes vergleichen wir in einem zweiten Schritt verschiedene Klassifizierungsalgorithmen und untersuchen unterschiedliche Möglichkeiten den Quellcode als Input für diese Modelle darzustellen. Schließlich werden die leistungsstärksten Modelle in ein prototypisches Analysewerkzeug konsolidiert und bei einem Industriepartner exemplarisch eingesetzt.

Beitrag Wir stellen fest, dass nur wenige statische Code-Metriken wertvolle Indikatoren für die Wartbarkeit sind. Mit Hilfe ausgewählter Metriken wie der Größe eines Programms oder seiner kognitiven Komplexität können die entwickelten Klassifizierungsmodelle die Wartbarkeit einer Java-Klasse auf einem menschlichen Leistungsniveau bestimmen. Sowohl die experimentellen Ergebnisse als auch die Erkenntnisse aus der Anwendung in der Praxis bestätigen, dass die entwickelten Modelle schwer zu wartenden Quellcode mit hoher Präzision identifizieren können. Zusammenfassend legt diese Doktorarbeit dar, wie das darin vorgestellte Framework die Effizienz und Effektivität von Wartbarkeitsanalysen durch den Einsatz von Machine Learning erhöhen kann.

Abstract

Problem Domain The maintenance of a software system is responsible for a large share of the total cost of the system. Hence, the maintainability of a system is essential for its long-term viability. To be able to control and improve the maintainability, system owners need to first *assess* it. However, most analysis tools provide only imprecise warnings or metrics for this purpose, which in turn must be interpreted by the user. Assessments by experts are considered more reliable, precise, and trustworthy than purely tool-based assessments. Though a powerful technique, they involve enormous effort and require experienced experts to perform the analysis. In an industrial environment, the size of the systems to be analyzed asks for an efficient and tool-supported process. One way to combine the advantages of fast automated approaches and precise expert assessments is to use Machine Learning techniques to classify maintainability as experts would perceive it.

Research Gap Existing works often utilize problematic surrogate measurements instead of expert judgment to classify maintainability. If expert evaluations are used, they are based on the subjective opinion of an individual or a very small group of experts. This introduces bias and limits the transferability of the trained models. Furthermore, little is known about how industry-scale maintainability assessments are performed in detail. Consequently, it remains speculative how exactly the developed classification models can be used in this context and how they can facilitate the assessment process.

Solution To close these gaps, we integrate expert-based assessments and machine-learned classifications into a combined framework. As a first step, we analyze the current practice of maintainability assessments and stipulate which aspects of the analysis can be automated. We focus on the maintainability of Java classes and contribute a publicly available dataset that includes the consensus of multiple experts regarding the maintainability of source code. Based on this dataset, we compare several classification algorithms and investigate different ways to represent source code as input to these models. Eventually, the best-performing models are consolidated into a prototypical analysis tool and exemplarily applied at an industry partner.

Contribution We find that only few static code metrics are valuable indicators of maintainability. Using selected metrics such as the size of a program or its cognitive complexity, the developed classification models can determine the maintainability of a Java class with human-level performance. Both experimental results and findings from the application in practice confirm that the developed models can identify hard-to-maintain code with high precision. In summary, this thesis illustrates how the proposed framework can increase the efficiency and effectiveness of maintainability assessments through Machine Learning.

Contents

Part I.

Introduction and Background

1. Introduction

Some contents of this chapter have previously appeared in other publications, which are co-authored by the author of this thesis [183, 185, 187, 188, 189, 190].

This chapter introduces the topic of maintainability assessments and motivates using machine-learned classifications to support them. It describes the challenges of contemporary assessments, elaborates on current approaches to automate them, and illustrates how machine-learned classifications can improve the state of the art. Eventually, gaps in the state of research are briefly summarized before we discuss the contributions of this thesis.

1.1. Motivation

1.1.1. Importance of Maintainability

Maintaining a software system is responsible for the majority of its total costs [20, 21, 128, 164]. The ease to perform maintenance-related activities is referred to as the maintainability of a system. Typical activities include the comprehension of the existing code base [46, 163, 215] as well as the actual modification of the source code [46]. Therefore, in addition to design and architecture decisions, properties such as the readability and comprehensibility of the code also have a major influence on the maintenance effort [46, 215]. Virtually every software system is subject to maintenance. In fact, software is known to age and decay over time [54, 157]. Hence, different types of maintenance become necessary due to changing ecosystems, new user requirements, or because faulty behavior became evident [31, 206]. As the code base grows, quality violations are propagated further and may result in enormous future costs [54]. While there might not be any immediate negative impact, the malign long-term effects are exponentially increasing. Hence, the relationship between the maintainability of a system and its economic sustainability is explicit [164].

In order to control these costs, system owners need to assess the maintainability of their systems regularly [164]. Ideally, maintainability assessments identify future maintainability problems with high precision, provide accurate cost estimates, and report the results fast and automatically.

1.1.2. Existing Approaches to Assess Maintainability

Measuring and controlling quality is a primary concern in software engineering. While testing is the de-facto standard to evaluate the functional correctness of a system, internal attributes such as maintainability are harder to assess.

Awareness of maintainability and the need for its assessment are not recent. Early examples of quantifying the maintainability of a system are software metrics. In 1977, Halstead introduced several formulas to calculate the comprehensibility of a program, its difficulty, and the effort needed to implement it [10, 75]. Another popular example is the Maintainability Index introduced by Oman et al. [153] in 1994. It combines Halstead's volume metric, the cyclomatic complexity [140], and the number of lines of code in the system. In addition to these aggregated values, there exists a multitude of atomic code metrics [57]. These include, e.g., the size of a program, the number of variables, the length of the longest method, and many more.

Most of these metrics can be computed automatically and without executing a program. Thus, they are easy to extract and there exists a multitude of static analysis tools providing these measurements. These include, among others, Continuous Quality Assessment Toolkit (ConQAT) [44], Designite [194], PMD [169], SD Metrics [220], Sonarqube [28], Sourcemeter [131], and Teamscale [84].

However, questions have been raised about the applicability of metrics in maintainability assessments. Static metrics can only describe the structure of software, not the quality of the behavior [214]. A majority of currently used static metrics are indeed not empirically validated to correlate with maintenance effort [147, 150, 199]. For instance, several studies examined the influence of inheritance on maintainability. They could not verify a relevant connection between the depth of the inheritance tree and maintainability [29, 41, 42, 168]. As early as 1983, Basili et al. [13] pointed out that no investigated metric provides additional information compared to the size of a program.

Lately, static analysis tools, in general, have received criticism. As metrics are easy to extract from source code, tools tend to provide too many measurements, thus overwhelming the users [94]. For effective metric-aided assessments, clear strategies need to be in place which metrics to consider and how to interpret them [17]. To lower the bar for the user, some tools consolidate measurements into warnings. Such warnings are raised, e.g., if certain metric thresholds are violated. However, several researchers caution against the use of fixed thresholds [15, 152]. Studies revealed that analysis tools tend to provide too many false-positive warnings [94, 113, 213, 226]. Unfortunately, configuring a tool to personal or project-specific needs requires enormous effort [213]. Furthermore, users often criticize the incomprehensible presentation and lack of justification of warnings [94].

In summary, the output of static analysis tools does often not match the expectation and intuition of the users. This decreases the motivation to apply such tools at all [94]. Interestingly, the calculation of metrics differs between tools and they often provide different values for the same software metrics [18]. Thus, Bertrand et al. [18] advocate basing all maintainability analysis activities only on the structure of the code.

A different line of research aims to detect code smells, i.e. common anti-patterns related to the maintainability of code. Prominent examples are *god class*, *feature envy*, *long method*, and *duplicated code*. While widely accepted as indicators for potential maintenance issues, it is important to treat them as hints and not maintainability issues themselves [15].

In the literature, the importance of code smells for maintainability prediction is controversially debated. Some studies showed no significant impact of most smells on maintenance activities [1, 74, 151, 200], while others report that smells led to decreased maintainability [47] or increased error-proneness [105, 106, 154].

1.1.3. Expert-based Assessments

Manual code reviews are a well-studied and accepted method to evaluate and control the quality of software [16, 141]. When predicting maintenance effort, estimations by experts are often more accurate than formalized estimation methods [111] and code smells [222, 223].

As there is no formalized definition of maintainability [90, 91] and the existing standards are deliberately vague [99], the outcome of an assessment depends on the subjective opinion of the analyst. A study by Tokmak et al. [210] confirmed experts and novices tend to evaluate software quality differently. However, the experience of an expert is not an indicator for more precise estimates of the maintenance effort [96].

Expert-based maintainability assessments are often referred to as Software Health Checks (HCs) [164]. They are shown to be effective and can lead to improvements that mitigate future risks and costs [139, 164, 204]. Yet an effective method, manual assessments are time-consuming and tedious [95]. During an HC, the experts collect tangible facts and manifestations of subpar maintainability before concluding about the state of the product [139, 164, 188, 204]. These facts are deduced from the underlying quality model [164], e.g. the activity-based maintainability model proposed in [45]. Relying on this kind of evidence makes the HCs less subjective [139] and their conclusions more persuasive [204].

1.1.4. Using Machine Learning to Assist Maintainability Assessments

Machine Learning (ML) has emerged as a key instrument to support various tasks in other disciplines of software engineering: It can assist in fault localization [24, 93, 219] and defect prediction [33, 191, 196, 217], or can help to summarize source code [5, 6].

Considering maintainability, machine learning was applied by several studies to predict the number of code changes as a surrogate for maintainability [3, 92, 103, 121, 126, 137, 138, 173, 209, 212, 227, 229]. In 1993, Li and Henry were the first to count the number of changed code lines and utilize this measurement as a proxy for maintenance effort [126]. Later on, their dataset of 110 code files and the respective changes was used by several related works. An overview of 27 studies is provided by Kaur and Kaur in [103]. Zhang et al. used a similar methodology to create their own dataset and also incorporated the number of commits in addition to the number of changed lines [227].

However, we do not consider the number of revised lines an adequate surrogate for maintenance efforts. First, this approach assumes that all changed lines require identical implementation effort and cognitive load. Second, it assumes that change takes place at the same spot as underlying maintainability issues. Consider code that has historically grown over decades and is no longer comprehensible and maintainable. This might result in all adaptions being performed in the surrounding classes, but not in the class itself due to its black box nature. Third, the number of changed lines does not consider the trigger for the change. In perfective maintenance [31], modifications originate from new requirements. The number of changed lines does not differentiate between efforts that stem from bad design and efforts introduced by the complexity of a requirement. Fourth, it neglects all maintenance activities except the actual code modification. However, understanding the code [46, 163, 215] and impact analysis [46] are essential parts of maintenance, too.

There are several studies using ML to predict the occurrence of code smells [1, 7, 47, 50, 59, 102, 105, 106, 107, 142, 145, 154, 156, 160, 161, 192]. However, even if smells were found with high accuracy, these approaches only show the occurrence of indicators for potentially problematic maintainability. Still, manual intervention is necessary to reach an informed opinion on the maintainability of a system.

Only few works exist that aim to predict expert maintainability judgment. Here, creating manually-labeled datasets is a major challenge [82, 228]. To develop reliable predictors, reliable maintainability ratings have to be collected. However, the accuracy of an analyst is unknown a-priori [95]. To mitigate this, Jørgensen et al. propose using the aggregation of votes by several experts [95]. However, this increases the effort needed to create the dataset and consequently decreases the number of data points that can be analyzed in a given time.

On the granularity of methods, Hegedűs et al. performed several experiments [81, 82] on a dataset labeled by one expert per data point. Still, their classification algorithm could only slightly improve against the baseline [81], and the subjective bias of their label was recognized as a limitation [82]. Other studies also rely on single experts [79, 80, 165], use unsupervised learning [228] to assign labels, or refer to students as analysts [79, 80, 82]. All investigated studies did not share their created datasets with the scientific community, hence limiting the reproducibility of their results.

1.1.5. Goal and Separation from Related Work

The goal of this work is to support expert-based maintainability assessments, which are currently considered effective but tedious and time-consuming. We hypothesize their efficiency can be improved by appropriate tool support. Here, the state-of-practice of industrial expert assessments is to be considered for the tool to integrate well with the established process. In addition, the maintainability issues identified by the tool should reflect the experts' intuition of maintainability. This facilitates the collection of relevant evidence to be interpreted by the expert.

There is a growing body of literature that recognizes the potential of ML to perform tasks, that have traditionally involved human experience. Thus, this thesis proposes to use ML techniques to learn this expert judgment.

Until now, related works on using ML to assist maintainability assessments have either

- used metric-based heuristics to identify maintainability problems [10, 35, 37, 75, 144, 153],
- focused on the occurrence of code smells [1, 7, 47, 50, 59, 102, 105, 106, 107, 142, 145, 154, 156, 160, 161, 192],
- predicted the number of revised code lines as a proxy measurement for maintenance effort [3, 92, 103, 121, 126, 137, 138, 173, 209, 212, 227, 229], or
- referred to expert-labeled data, but lack generalizability due to subjective bias and small, non-public datasets [79, 80, 81, 82, 165, 228].

However, literature has challenged the relationship between maintainability and quality metrics [29, 41, 42, 152, 168, 199] as well as between maintainability and code smells [74, 151, 200, 222, 223]. Utilizing them as a surrogate for maintainability bears risks and questionable value. The use of changed code lines also suffers from several assumptions and neglects all maintenance-related activities except code modification. As expert judgment arguably yields the highest value to assist expert-based assessment, we aim to predict it directly without proxy measurements. Eventually, to the best of our knowledge, no related study has demonstrated how their approach can be applied in practice to facilitate maintainability assessments.

The following sections will describe the research gaps in more detail, elaborate on our proposed solution, and highlight the key contributions of this thesis.

1.2. Problem Statement and Research Gap

This thesis proposes the use of Machine Learning (ML) to assist software maintainability assessments. Expert-based maintainability assessments, often referred to as Software Health Checks, are effective [139, 164, 204] but time-consuming [95]. The goal of this thesis is to help automate this task. However, existing automation approaches focus on providing software metrics, identifying code smells, or predicting the number of changed lines. In contrast, we aim to capture the experience of human experts and predict their maintainability rating directly. Eventually, we provide a framework to use ML-based maintainability ratings and integrate the results into an established assessment process.

A detailed discussion of related work is presented in Chapter 2. In the context of this work, the following research gaps can be identified:

- **Gap 1 - State of Practice in Maintainability Assessments** One effective way to evaluate the maintainability of a software system is an expert-based assessment [95, 111]. They are shown to be effective [139, 164, 204], but are also expensive and time-consuming [95]. This poses a barrier to regularly performed assessments and efficient quality control. However, the state of practice in expert-based maintainability assessments is mostly unreported. It is important to understand how quality audits are performed in order to support them. Particularly, the role of tools and the extent to which they are already applied and relied on is unclear. There is a need for research to model the HC procedure in detail and investigate the performed activities.

  Outlook: In cooperation with itestra GmbH, we participated in two industrial HCs. We consolidate our experiences and the experience of the industrial partner into a structured framework and report our lessons learned.

- **Gap 2 - Identify Improvement Potential** We hypothesize ML can identify code with problematic maintainability and thus facilitate expert-based maintainability assessments. Related studies [79, 80, 81, 82, 165, 228] focus on the prediction experiments and the observed performance in an isolated, experimental setting. The context of the research is only briefly discussed. It remains unclear which parts of the assessment process can be supported by ML models, and which tasks shall remain in the hands of human experts.

  Outlook: We perform initial experiments to i) examine the applicability of the approach and ii) limit the focus of further experiments with respect to the context. We find machine-learned classifications are a valuable asset to generate quick overviews and guide the selection of code files to manually analyze in depth. Despite the good performance observed in our experiments, long-term strategic recommendations should not be derived solely based on ML classifications. They can aid the expert-based assessment but should not aim to replace them.

- **Gap 3 - Software Maintainability Dataset** The choice of the used dataset is crucial for the success of ML applications. The provided label has to precisely represent the task to be solved by the ML algorithm. Existing works on using ML to assist in maintainability assessments often fall back to surrogates for maintainability. Some studies refer to the number of changed lines of code [3, 92, 103, 121, 126, 137, 138, 173, 209, 212, 227, 229]. However, this label does not consider several relevant maintenance activities such as code comprehension. Furthermore, it assumes equal efforts for all code changes and suffers from several more assumptions, as explained in Section 1.1.4. Only few studies exist that use manually labeled datasets [79, 80, 81, 82, 165, 228]. However, they rely on the subjective opinion of a single expert or a small group, and contain only few data points. This is problematic, as it threatens the generalizability of the results. To date, no dataset reflects human opinions without bias, is large in size, and shares the data with the scientific community.

  Outlook: We close this gap by creating a manually-labeled dataset containing more than 500 Java classes. It provides the consensus rating of at least three experts per code file.

- **Gap 4 - Metric-based Classification** ML has emerged as a useful technique to automate tasks that traditionally require human expertise. To predict the maintainability of source code, software metrics are often used as input. However, many studies [79, 80, 103, 121, 126, 165, 209, 212, 228, 229] commit early to using a specific metric suite. This excludes potentially beneficial information sources. Furthermore, due to the available data, the majority of studies apply regression or binary classification. This is problematic as it does not correspond to the way how maintainability is perceived by human analysts. Therefore, the reported performances are hard to interpret. There is a need to investigate ordinal classification approaches, evaluate a broad variety of metrics as input, and put the results into context by comparing them with human performance.

  Outlook: We contribute several ordinal classification experiments and find the best performing algorithm to reach the same level of prediction accuracy towards the consensus of several experts as a single average expert.

- **Gap 5 - AST-based Classification** Source code can be represented in different formats. So far, often static measurements are used as input for ML. However, in other disciplines, embeddings of the Abstract Syntax Tree (AST) revealed promising results [5, 6, 26, 76, 109, 125, 167, 192, 218]. Evaluating the use of AST embeddings to predict the maintainability of code is a natural progression of this development and should be investigated.

  Outlook: We adopt code2seq, *a state-of-the-art framework for AST-based ML, to predict the maintainability of a code file on an ordinal scale. Although the results can be improved by combining both AST and metrics as input, code2seq could not reach the performance of purely metric-based models.*

- **Gap 6 - Application in Practice** Typically, machine learning experiments divide the available data into subsets for training and testing. While the observed results offer interesting insights, it remains unclear how the results transfer to practice. To validate the usefulness of the proposed ML-assisted maintainability assessments, the classification models need to be evaluated in a real-world setting. As a prerequisite, it is necessary to implement tool support to make the classifications accessible and usable.

  Outlook: We developed a prototypical tool, which integrates the data preprocessing, classification models, and a web-based frontend. We demonstrate how the tool can be integrated into the existing HC process and apply it to three industrial projects at itestra GmbH.

1.3. Achievements of This Dissertation

This thesis proposes machine learning assisted maintainability assessments. An overview of the framework is provided in Figure 1.1. In summary, it introduces ML classifiers for

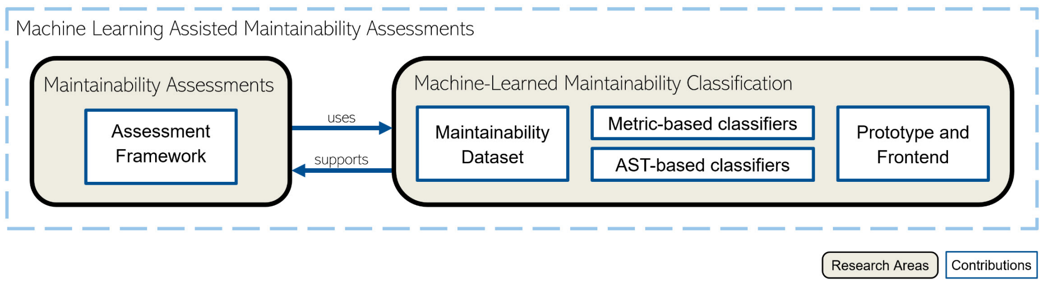


Figure 1.1.: Overview of ML-assisted maintainability assessments and the contributions made by this thesis

predicting the maintainability of source files and shows their integration into industrial assessment processes. The developed classifiers combine the benefits of fast analysis tools with the accuracy of maintainability ratings by human analysts. As a result, they can highlight potentially problematic code files with high precision. This guides the manual analysis and contributes to a more efficient expert-based assessment.

The solution touches upon two research areas. First, expert-based maintainability assessments are researched. In order to facilitate them, one has to understand the state of practice first. Here, we contribute a structured assessment framework that models HCs at itestra GmbH. Second, we investigate machine-learned maintainability classification. The goal is to develop models that predict the maintainability of code files as it has been judged by human experts. Therefore, we contribute a manually-labeled maintainability dataset and evaluate two types of classifiers on it. Eventually, the best-performing models are deployed as a prototype. This prototype combines the trained classifiers, encapsulates the data preprocessing, and provides a web-based frontend. This way, it can be easily integrated into the existing assessment process.

In detail, this thesis makes the following contributions, which are visualized as white boxes in Figure 1.1:

- To fill **Gap 1**, we contribute an assessment framework (cf. Chapter 3). To support maintainability assessments, it is a primary concern to understand the current practice. Our model sheds light on the industrial practice of HCs at itestra GmbH and highlights the importance of hypotheses. These hypotheses represent the analysts' current understanding of the system's main problems. Tangible evidence of quality violations is crucial to i) iteratively refine this understanding, ii) convince stakeholders of quality problems, and iii) guide which parts of the systems to analyze in depth. Tools can help in collecting this evidence, which is then interpreted by the experts.

- To fill **Gap 2**, we contribute several insights from initial experiments (cf. Chapter 4). Here, we used an early version of a manually-labeled dataset, a threefold maintainability rating, and a variety of static code metrics as input. In 10-fold cross-validation, our models achieved an F-Score of 0.80 and an accuracy of 0.81. We conclude that this performance is adequate to create overviews and to indicate code that experts should analyze in more detail. However, it is not desirable to make long-term strategic decisions based on the classification results.

- To fill **Gap 3**, we contribute a software maintainability dataset (cf. Chapter 5). We selected a representative sample from nine software projects and surveyed 70 professionals affiliated with 17 institutions. Every code file was evaluated for its maintainability, understandability, readability, complexity, and modularity by at least three participants. The resulting dataset contains the consensus of the experts for a total of 519 code files. 304 of them are sampled from non-commercial projects and made publicly available in [184]. Interestingly, our analysis of the submitted ratings revealed frequent disagreement between experts.

- To fill **Gap 4**, we implemented and compared several ML approaches based on static code metrics (cf. Chapter 6). Starting with a corpus of 132 metrics provided by seven different analysis tools, we investigated various data preparation and feature selection techniques. The features with the highest predictive power include, in descending order, the size of the longest method within a class, the length of a class, its cognitive complexity, the total amount of lines within methods, and the average length of a method. Based on these metrics, we trained both novel meta-classifiers as well as established ML algorithms. A Random Forest classifier achieved the best performance with a Matthews Correlation Coefficient (mcc) of 0.525. To put the results into context, we establish a human-level baseline. Due to the observed dissent between experts (cf. Gap 3), most experts are not congruent with the eventual consensus in several cases. We find the performance of an average study participant to be at an mcc of 0.529, which is only 0.004 higher than the best classifier.

- To fill **Gap 5**, we adopt the *code2seq* framework to predict the maintainability of code (cf. Chapter 7). This framework is designed to embed the paths in the AST of a method and predict attributes such as the name of a method [5]. We contribute an implementation to consider the AST of Java classes and predict the ordinal maintainability label from our dataset. Furthermore, it complements the AST embeddings with code metrics. While adding metrics can increase the prediction performance, it could not improve against the metric-based classifiers.

- To fill **Gap 6**, we provide a prototypical tool to be used in maintainability assessments (cf. Chapter 8). It integrates the necessary data preprocessing, pretrained prediction models as well as a frontend. We applied it to three industrial projects at itestra GmbH. Our initial evaluation showed both metric-based and AST-based classifiers can identify problematic code files with high precision. Still, the experts expressed a preference for the metric-based classifier, which was superior in our previous experiments, too. In summary, the predicted maintainability ratings enabled rapid overviews and facilitated the selection of code files to inspect manually.

Parts of this dissertation have previously appeared in peer-reviewed publications co-authored by the author of this thesis [183, 185, 187, 188, 189, 190]. This applies in particular to the following contributions:

- Chapter 3 addresses **Gap 1** and is based on [188]:
  Markus Schnappinger, Mohd Hafeez Osman, Alexander Pretschner, Markus Pizka, and Arnaud Fietzke: *Software quality assessment in practice: a hypothesis-driven framework*, In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '18). Association for Computing Machinery, New York, NY, USA, Article 40, 1–6. October 2018
- Chapter 4 addresses **Gap 2** and is based on [187]:
  Markus Schnappinger, Mohd Hafeez Osman, Alexander Pretschner, and Arnaud Fietzke: *Learning a Classifier for Prediction of Maintainability Based on Static Analysis Tools*, 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 2019, pp. 243-248, May 2019
- Chapter 5 addresses **Gap 3** and is based on [183]:
  Markus Schnappinger, Arnaud Fietzke and Alexander Pretschner: *Defining a Software Maintainability Dataset: Collecting, Aggregating and Analysing Expert Evaluations of Software Maintainability*, 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 278-289, October 2020
- Chapter 6 addresses **Gap 4** and is based on [185]:
  Markus Schnappinger, Arnaud Fietzke, and Alexander Pretschner: *Human-level Ordinal Maintainability Prediction Based on Static Code Metrics*, In Evaluation and Assessment in Software Engineering (EASE 2021). Association for Computing Machinery, New York, NY, USA, 160–169. June 2021

1.4. Limitations

The limitations of this thesis and threats to the validity of the results will be discussed in detail in Part V, Section 10.2. Collectively, the limitations of this thesis arise from the industrial perspective and the used dataset:

The goal of this thesis is to support maintainability assessments. To foster the applicability of our approach, we rely on an industrial partner to provide the perspective of practitioners. Based on almost 20 years of experience in analyzing and maintaining systems, itestra GmbH can provide an informed opinion on the problems studied in this thesis. Still, the perspective is limited to only one industrial partner and thus potentially biased. Their processes and viewpoints might not translate to other companies.

The biggest threat to validity for all ML applications stems from the used data. In this thesis, we create a manually-labeled dataset and utilize it for all further ML experiments. The dataset considers only Java code files and was labeled solely based on the source code of that file. This means neither the architecture, data model, afferent coupling, nor inter-class cloning was respected by the analysts. Thus, the created labels do not reflect these inter-class characteristics. This is a trade-off. Displaying every code file in its broader context may result in more holistic maintainability ratings. However, it does also increase the labeling effort per data point. As a consequence, fewer data points could have been labeled within the same time. Furthermore, the motivation of the voluntary participants might decrease due to the increased complexity of the labeling task. In summary, respecting more context and inter-class attributes during the labeling decreases the number of labeled data points. This is problematic, as small datasets jeopardize the ability to learn useful information from the available data. Thus, this study focuses on intra-class maintainability characteristics.

1.5. Outline

After the introduction, related work and relevant concepts are explained in Chapter 2. The remainder of this thesis follows the structure of the research gaps as defined in Section 1.2:

Part II elaborated on expert-based software maintainability audits. It contains Chapter 3, which closes the first research gap. Part III focuses on machine-learned maintainability classification. It consists of Chapters 4 to 7, which describe the solutions for the research gaps 2 to 5, resp. To close gap 6, the developed tool and its application to three real-life projects in described in Part IV. Eventually, in Part V, the contributions of this thesis and their limitations are discussed. An outlook on future work concludes the dissertation.

2. Background and Related Work

Some contents of this chapter have previously appeared in other publications, which are co-authored by the author of this thesis [183, 185, 187, 188, 189, 190].

This chapter provides a general overview of the background of this thesis and related works. It describes the fundamentals of software maintainability and its assessment. In addition, it elaborates on the usage of machine learning to predict maintainability and discusses the shortcomings of related studies. Eventually, it introduces selected machine learning concepts that will become relevant during this dissertation.

2.1. Software Quality and its Assessment

2.1.1. Quality Models

This thesis aims to support the assessment of software quality. In particular, it proposes a framework to assist the assessment of maintainability as one instance of software quality. The concept of quality depends primarily on the viewpoint taken [64]. Examples are the user perspective, which is concerned with the suitability for the user's purpose, the manufacturing view focusing on the techniques used to create a product, and the value-oriented view or the product view [64]. The latter deals with the quality characteristics inherent in the product itself. The perspective taken in this thesis corresponds to this product view. Part II is concerned with how software quality audits are performed. It focuses primarily on the internal characteristics of a software system and its source code.

Over the last decades, several software quality models have been introduced. These include, among others, meta-models for quality [11, 216], Boehm's model [21], the FURPS model [68], and the ISO/IEC standards 9126 [90] and 25010 [91]. There also exist several models tailored to specific application domains or companies [143].

Subdimensions of Quality

There is a consensus among scientists and practitioners, that software quality is an inherently hierarchical concept [20]. As such, it consists of several subcharacteristics.

In 2001, the International Organisation for Standardization introduced the standard ISO/IEC 9126 [90]. It classifies software product quality as the aggregation of its functionality, reliability, usability, efficiency, maintainability, and portability [90]. It was later on

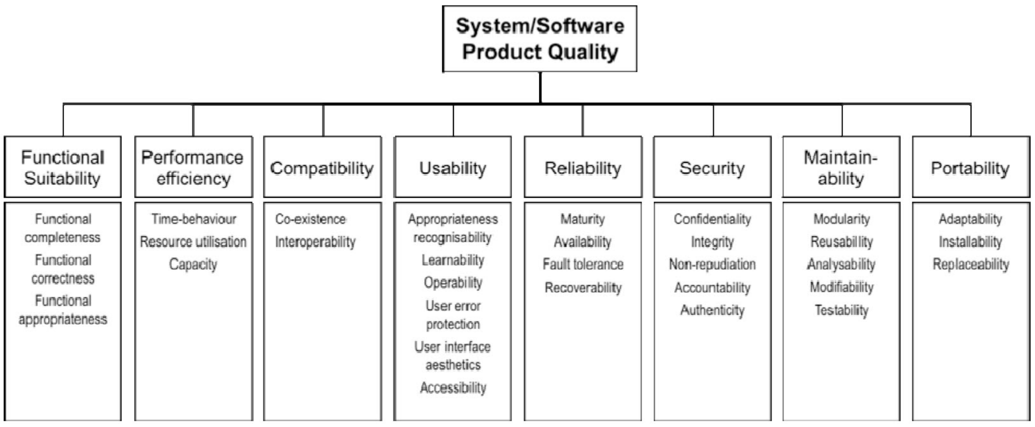


Figure 2.1.: Decomposition of software quality into several subcharacteristics (as appeared as *Figure 4* in ISO/IEC 25010 [91])

refined into ISO/IEC 25010 [91], which also touches upon security and comparability. The decomposition of software product quality according to [91] is visualized in Figure 2.1.

Despite this decomposition, both standards are intentionally vague. On the one hand, this allows for tailoring and universal applicability [11, 99]. On the other hand, it provides little guidance on how to measure or aggregate the subdimensions [11]. This makes the concrete application more difficult.

To reason about software quality, it is best assessed bottom-up [21, 53]. Boehm et al. [20] and Dromey [53] agree that tangible facts should build the bottom of the hierarchy. They need to be collected and interpreted to reason about the intangible, abstract quality attributes. Bakota et al. [11] proposed using a probabilistic model to aggregate the subdimensions. This accounts for uncertainty, which is introduced by the subjective interpretation of the underlying facts. The correlations between quality characteristics as perceived by experts and system properties have also been investigated by Correia et al. [39]. Here, the median of three experts was used to describe the quality of a system. In contrast, Jung et al. [99] referred to end-users of a system to judge its quality. They examined the correlations between different aspects of external software quality. However, the examined quality characteristics have not been related to the overall perceived quality. Interestingly, quality is subject to many influences. For instance, Naggapan et al. showed correlations between the error-proneness of systems and the organizational structures and processes within an institution [148].

One of the key quality characteristics contributing to holistic software quality is the system's *maintainability*. As shown in Figure 2.1, maintainability can be subdivided into several subdimensions as well [21, 91]. The ISO/IEC 25010 [91] decomposes it into modularity, reusability, analyzability, modifiability, and testability. Subsequently, these attributes can be further refined. For instance, analyzability consists of structuredness, self-descriptiveness, conciseness, and legibility [25]. Please note that many authors use different names to describe identical concepts, for example, legibility and readability.

2.1.2. Software Maintainability

Software maintenance is accountable for the largest share of the overall costs of a software system [20, 21, 128, 164]. Hence, the maintainability of a system is a dominant feature for its long-term viability [164]. Typically, research distinguishes four types of maintenance based on their purpose: corrective, adaptive, perfective, and preventive maintenance.

- **Corrective** maintenance is concerned with repairing wrong behavior of the system and fixing bugs [31, 206].
- **Adaptive** maintenance describes the adaption of the system to changes in its context including infrastructure, new legal requirements, or new interfaces of partner systems [31, 206].
- **Perfective** maintenance implements new requirements. While adaptive maintenance originates from changes in the ecosystem, perfective tasks focus on the functionalities of the system itself [31, 206].
- **Preventive** maintenance aims to maintain a high quality in the future. In contrast to adaptive maintenance, there is no specific trigger from the outside. Typical examples are refactorings, which do not change the functionality of a system, but rather aim to avoid technical debt [31].

The ease to perform such tasks on a system is referred to as its maintainability. High maintenance costs mostly stem from the activities performed during maintenance tasks. To overcome shortcomings of other models, Deissenboeck et al. [46] and Broy et al. [25] focus on these activities instead of system characteristics. This can help in controlling maintenance costs and emphasizes the influence of activities such as concept location and impact analysis [25, 46].

One primary aspect of maintenance is the comprehension of a program [215]. Different cognitive models are used at different abstraction levels to understand the existing code base. Still, to what extent the program is understood depends on the analyst and their experience [215]. Pennington [163] further emphasizes the importance of code comprehension. The majority of code to be maintained has been authored by other programmers. That is why, as she concludes, understanding source code plays an important role in program maintenance.

The terms comprehensibility and understandability refer to the same concept. However, readability must be distinguished from this. Other authors refer to the readability of code as its legibility [25]. Readability hence describes the ease of syntactically parsing the tokens [172]. In contrast, understandability focuses on the ease to capture the semantics of the code [215].

2.1.3. Assessing and Controlling Maintainability

The economic importance of maintenance results in the need to assess and control the maintainability of a system. This includes both continuous control during development as well as post-development reviews. If severe maintainability problems are found, strategic realignments may be necessary to ensure the economic viability of a system [139, 164]. These may include full or partial migrations or even re-developments [164].

Software does age and decay over time [54, 157], thus posing the need for regular assessments even after the initial release. Furthermore, how quality is perceived is subject to the contemporary state of practice. What was considered best practice at the time of the initial development may be considered an anti-pattern only a few years later. Thus, these assessments have to be performed regularly. This results in the need for fast and reproducible analysis techniques.

Static Code Metrics

Early approaches implementing objective and fast maintainability assessments are static software metrics. Fenton and Bieman's book on software metrics [57] summarizes a large body of measurements for a variety of different use cases. In the remainder of this work, we differentiate between intra-class and inter-class aspects of a software class. Intra-class metrics describe characteristics of the class itself, for example, its size or the number of its methods. In contrast, inter-class attributes consider the relationships to other classes such as inheritance or coupling. Over the past decades, several metric suites have been proposed specifically for maintainability assessment. The most popular instances include:

- Halstead's metrics [10, 75] describe different viewpoints on the number of operators and operands in a program. Based on these intra-class metrics, Halstead [75] defined different formulas to calculate the implementation difficulty, implementation effort in seconds, or the comprehensibility of a program [10]. However, as early as 1994, Shepperd and Ince [197] stated that Halstead's assumptions about programs are no longer up-to-date. Representing software only in terms of operands and operators neglects important aspects such as data flow and module structure. It is reasonable to assume their criticism is still valid today as current software has evolved even more in the direction of large, complex systems.
- Chidamber and Kemerer [34, 35] proposed another metric suite. It is often referred to as the CK metrics suite. It was developed as a first foray to analyze the then novel object-oriented software systems. It contains both intra- and inter-class aspects and consists of the following metrics:
  - WMC: Weighted Method Count
  - DIT: Depth of Inheritance Tree

  - NOC: Number of Children
  - RFC: Response for a Class
  - CBO: Coupling between Objects
  - LCOM: Lack of Cohesion in Methods

  However, important implementation details are missing. In particular, the description of CBO lacks details which types of coupling are considered [195]. Similarly, it is not clear how to compute the lack of cohesion in methods. Although Chidamber and Kemerer presented a formula to calculate LCOM [34], their definition has been observed to be imprecise [85, 195].

- In 1994, Oman and Hagemeister [153] proposed the Maintainability Index. The goal of this index is to achieve comparability between systems by reducing multi-faceted maintainability assessments to a single metric. In the same year, Coleman et al. [37] validated a connection between the proposed index and manual evaluations. It is used as a proxy measurement for maintainability, e.g., in [100] and [144]. The Maintainability Index applies polynomial functions with fixed parameters to aggregate several metrics. In contrast to their work, we apply machine learning instead of fixed parameters to combine code metrics into a maintainability rating.

In general, literature offers contradictory views about the usefulness of static code metrics for maintainability assessments: Li and Henry [126] showed a strong correlation between the number of changed code lines and eleven code metrics, including the CK metrics. Coleman et al. [37] showed a relation between qualitative expert judgment and numerical static metrics such as the Maintainability Index. In contrast, contemporary research tends to be more critical about software quality metrics and static analysis tools. Interestingly, several metrics supposed to describe a certain quality attribute fail in doing so, according to a survey by Sharma and Spinellis [195]. A majority of metrics are not empirically validated and shown *not* to relate to software maintainability [147, 150, 195, 199]. A notable exception is cognitive complexity [27], which was shown to reflect the understandability of code by Muñoz Barón et al. [147]. In general, static metrics can only measure the structure of software and are thus insufficient to measure all aspects of software quality [214]. One metric, that has received criticism in particular is the depth of the inheritance tree. Several studies examined its influence on maintainability. Interestingly, it did not yield a significant relation to software maintainability [29, 41, 42, 168]. Furthermore, most code smells did not pose a correlation with the maintainability of a system [200]. Sjøberg et al. [199] found, that only the size of a system and its cohesion are strongly related to maintenance effort. This matches the finding of Basili et al. [13]. They report they could not find a metric that is a better indicator for fault proneness than the size. The same statement was made by Dagpinar and Jahnke [41] considering the maintenance frequency of code. Furthermore, the existence of code smells seems to correlate mostly with the size of a system [222].

Gap: Code Metrics Proposed metrics are often not validated to relate to software maintainability or refer to obsolete programming styles. Notable exceptions are cognitive complexity and the size of a program. There is a need to provide measurements or indicators that correspond to maintainability as perceived by software engineers.

Static Analysis Tools

Static code metrics are often calculated by analysis tools. Surprisingly, the computed results frequently differ between tools when computing the same metric [18]. This discrepancy hinders switching between tool suppliers and is likely to create a vendor lock-in [18]. Furthermore, it is common that users feel overwhelmed by the plethora of information offered by many tools [94]. If metrics are used to assess the quality of software, clear strategies for selection, aggregation, and interpretation of the metrics are necessary to use them effectively [17].

To lower the bar for the user, some tools consolidate measurements into *findings*, *issues*, or *warnings*. These warnings are raised if, e.g., certain metric thresholds are violated. However, Oliveira et al. [152] warn against using fixed thresholds. Exceeding a threshold might be acceptable in some contexts, but poses a critical issue in a different setting. This view is confirmed by Beck and Fowler [15]. They dedicatedly refrained from providing thresholds for code smells and stated *"you will have to develop your own sense of how many instance variables are too many instance variables and how many lines of code in a method are too many lines"* [15, p.63].

In general, static analysis tools tend to provide too many false positives [94, 113, 213, 226]. The use of fixed thresholds might be one driving factor. Also, their findings are often presented in an incomprehensible way [94]. Configuring a tool to personal or project-specific needs usually requires enormous effort [213]. This decreases the motivation to apply such metric-based analysis tools [213].

Throughout this thesis, the following tools will be mentioned and applied: Continuous Quality Assessment Toolkit (ConQAT) [44], Designite [194], PMD [169], SD Metrics [220], Sonarqube [28], Sourcemeter [131], and Teamscale [84]. The open-source tool ConQAT is used both in research [98] and in industrial contexts, e.g. in HCs at itestra GmbH. Teamscale is the industrial successor of ConQAT. It also offers historical analyses, a variety of testing-oriented analyses, and supports more programming languages.

Gap: Analysis Tools While several tools exist to assist quality control, there are recurring drawbacks observed: They provide either a large number of metrics to be interpreted by the user or present warnings. Often, there are too many false-positive warnings [94, 113, 213, 226], tedious configuration is needed [213], or the warnings are incomprehensible [94]. To foster maintainability assessments, tools should identify problematic code with high precision and present the warnings in an intuitive way.

Code Smells

The term Code Smell was coined by Beck and Fowler [15] in 1999. While maintainability is hard to measure and assess, Beck and Fowler observed that problematic design decisions often manifest in recurrent forms. They consolidated them into 22 anti-patterns, for which they provide guidelines on how to refactor them and mitigate potential future problems.

Among the most popular instances are duplicated code, god classes, long methods, and feature envy. *Duplicated code* increases the maintenance effort, as changes have to be applied to all siblings of a code snippet [98]. Also, the code base gets unnecessarily large, thus hindering its analyzability, too. A *god class* or *blob* describes a class, which centralizes the system's logic. It is responsible for a broad variety of features with only little cohesion. *Long methods* are considered to yield similar problems like god classes. The *feature envy* smell is a symptom of subpar encapsulation. It occurs, e.g., if an object requires a lot of data from another entity to perform a computation. Furthermore, feature envy can be observed at the function-level. A method often calls methods from a second entity. In both cases, the design would benefit if the required functions or data were refactored and moved to the first object. [15]

There exist several approaches to detect code smells based on heuristics or ML. An overview of 84 smell detection tools was provided in 2016 in [58]. Kaur et al. [102] provide an overview of various ML-based detection approaches for code smells. In Section 2.2, we will elaborate on the usage of ML for maintainability prediction in more detail. This section will also cover ML-based code smell detection.

There is still uncertainty, however, about the influence of code smells on software maintainability. While widely accepted as indicators for potential problems, they do—by definition—not necessarily pose a maintainability issue and have to be interpreted [15]. However, they *may* lead to problematic behavior when maintaining or extending the code base. Beck and Fowler themselves explicitly refrained from providing thresholds when a maintainability issue is present or not [15]. Still, components affected by smells are found to be more error-prone and more likely to be changed frequently [105, 106, 154]. If several smells are present simultaneously, the comprehensibility of the source code suffers [1].

On the other hand, there exist contradicting results, too. The existence of code smells is mostly correlated to the size of a system [222]. Furthermore, the occurrence of smells cannot be linked to higher maintenance efforts in general [1, 74, 151, 200]. Notable exceptions are the *long method* and *long class* smells [200]. Abbes et al. [1] investigated the two smells Blob and Spaghetti Code in detail. They found the presence of one smell does not decrease maintainability, but maintenance effort increases if both smells occur simultaneously. Furthermore, Yamashita et al. [223] observed that some aspects of software maintainability are not captured by code smells. Hence, relying on code smells alone for maintainability assessments is problematic.

Gap: Code Smells Code smells can be valuable hints for maintainability issues. However, most code smells do not yield a correlation with the actual maintenance effort in general [74, 151, 200]. If a smell is detected, it must be followed up to understand its implications on maintainability. Thus, we hypothesize that predicting the maintainability judgment of experts is more beneficial for maintainability assessments than code smell detection.

Expert-based Assessments

Code reviews can help to evaluate and control the quality of programs [16, 141]. Expert-based maintainability assessments are often very accurate [95]. In comparison to code smells, they can predict maintenance effort more precisely [222, 223]. Experts are also shown to be more accurate than formalized effort estimation methods [111]. In fact, expert-based effort estimations are the most frequent form found in a meta-study by Jørgensen et al. [95] in 2002. Their work summarizes 15 publications on expert-based effort estimation.

Yet an effective method, manual assessments are time-consuming and tedious [95]. Another drawback of expert-based assessments is their inherent bias. Intuitively, the assessment is biased by the subjective opinion of the expert conducting the assessment. Interestingly, the experience of an expert is not an indicator for more precise assessments [96]. To account for this uncertainty, Rosqvist et al. [177] proposed to incorporate the confidence of an analyst into their assessment. Considering these confidences, a consensus of several experts is found after discussion.

An early framework for quality assessments was introduced by Cavano and McCall [30]. They propose using measurable software characteristics and combining them hierarchically until the quality-in-use is reached at the very top of the hierarchy. In [164], Pizka and Panas introduced Software Health Checks (HCs), the expert-based assessment method used at our industrial partner itestra GmbH. During the analysis of a software system, experienced experts focus on specific goals and adapt the analysis to the context of the system. Eventually, their evaluation helps to define adequate improvements and to reduce risks and costs [164, 204]. To take action and perform changes to the system, stakeholders need to be convinced first [204]. Hence, tangible facts are important to illustrate maintenance problems [139, 204]. All recommendations and conclusions should be based on these facts, too [139, 164]. Also, using these facts as a basis for the evaluation makes the assessment less subjective [139]. Ideally, the collected facts are deduced from an underlying quality model [139]. These can include static measurements such as size metrics and the ratio of duplicated code, an assessment of the quality of comments, or even code snippets illustrating the violation of best practices.

Gap: Expert-based Assessments Expert-based maintainability assessments are accurate but expensive and time-consuming. To mitigate potential subjectivity, it is important to base the assessment on objective facts. Tools are mentioned as a key element for assessments,

as they can extract certain facts automatically [139]. Still, the assessment remains time-consuming, as these facts have to be analyzed manually and not all facts can be extracted by tools [139].
To improve the state of the art, it is necessary to understand how experts conduct maintainability assessments and which processes and tools are applied in detail. This allows us to develop tool support that reduces their manual work and facilitates the established process.

2.2. Using Machine Learning to Predict Maintainability

Machine Learning (ML) has emerged as a key instrument to support various tasks in other disciplines of software engineering: Examples include auto-completion, pattern mining, code translation, and synthesizing programs from specifications [4]. Furthermore, there exists a multitude of research concerned with the prediction of functional quality attributes like error-proneness (e.g. [24, 33, 93, 108, 191, 196, 217, 219, 221]). However, these works aim to solve a different problem, use different datasets, and interpret their findings with respect to different evaluation baselines. Interestingly, Shepperd et al. [196] analyzed 41 studies on defect prediction and found the reported performance depends mostly on the researchers conducting the experiments. The influence of the chosen techniques apparently did not have a significant influence [196].

Considering maintainability assessments, ML offers the potential to predict the maintainability of software without tedious manual assessments. The following subsections summarize the most important studies investigating this problem. Here, the most discriminant factor is the choice of the predicted attribute. While some studies aim to predict the judgment of human experts, others refer to code smells, task completion time, or the number of observed changes as a surrogate for maintainability.

2.2.1. Investigating Correlations With Maintainability

A study by Daly et al. [42] in 1996 referred to the time needed to perform typical maintenance tasks on a system. They investigated the relationship between the task completion time and the inheritance tree of the study systems. In 1998 and 2003, similar experiments were conducted by Cartwright [29] and Prechelt et al. [168], resp. Interestingly, the studies report different results. Daly et al. and Cartwright found no influence of the depth of the inheritance tree on the maintainability of a system, while Prechelt et al. found inheritance mattered in some cases. Still, Prechelt et al. conclude maintenance effort is hardly correlated with inheritance depth.

In [200], professional software engineers were hired to perform maintenance tasks. Here, the task completion time was correlated to code smells. No significant relation between the time spend and the presence of code smells was found. Abbes et al. [1] considered both task completion time but also whether the task was performed correctly, and the

NASA task load index. They then investigated the influence of two code smells on the maintainability of a system. Misra [144] referred to the Maintainability Index [153] as a surrogate for maintainability. Then, ML algorithms are trained to predict the Maintainability Index. However, this approach evaluates the performance of a newly developed ML model towards an already existing polynomial formula with known parameters and weights. Hence, the benefits of this approach over using the Maintainability Index directly remain unclear.

Kádár et al. [100, 101] assessed whether refactorings had a positive or negative effect on the quality of a system. They created a dataset of applied refactorings and measured the code quality with the Maintainability Index and code complexity metrics. They found that refactorings led to improved quality, i.e. better metrics. In particular, they found that refactorings often decreased complexity and clone metrics. Still, the used notion of quality is inapplicable for ML for reasons stated above.

Dagpinar and Jahnke [41] referred to historical data and analyzed the commit history of a system. They referred to the frequency of maintenance activities to capture the maintainability of a system. Notably, they found that size and import coupling are important factors, while inheritance and export coupling are not. However, they did not perform experiments to predict the maintainability label from the code metrics.

Gap: Investigating Correlations With Maintainability Problematically, some studies referred to existing code metrics to quantify the maintainability of a system. This is unsatisfactory concerning the prediction of maintainability, as the predictions cannot improve against using those existing metrics. Others analyzed actual maintenance activities to quantify maintenance effort and thus maintainability. However, they only investigated the relationship between dependent and independent variables and did not perform prediction experiments in the sense of ML. Unfortunately, the data is not available to use in further studies. As their approach requires developers to actually perform maintenance tasks, it is rendered too costly to replicate in this study.

2.2.2. Predicting Code Changes

A large number of studies to predict maintainability uses the number of changed code lines as a proxy measurement for maintenance effort [3, 92, 103, 121, 126, 137, 138, 173, 209, 212, 227, 229]. One group of studies uses data provided by Li and Henry in [126], while others created their own dataset following a similar methodology.

Studies Based on Data by Li and Henry

The most prominent research in this area was conducted by Li and Henry [126] in 1993. They performed regression experiments to predict the number of changed lines in a code file based on static code metrics. In their study, they extracted the metrics and the number of revised code lines from two commercially developed systems in Classic-Ada. In sum, the dataset consists of 110 data points.

This dataset has been used by several related studies: Van Koten and Gray trained Bayesian Networks on these data [212], Kumar et al. utilized neuro-genetic algorithms [121], Thwin et al. even applied neural networks [209], while others investigated genetic algorithms and probabilistic neural networks [137].

In contrast to predicting the number of code changes, Reddivari and Raman [173] categorized the numerical data provided by Li and Henry into ordinal classes. Then, they used various algorithms including Random Forests and Decision Trees to predict that category.

A literature survey by Kaur and Kaur [103] found a total of 27 studies using this dataset in ML experiments. Collectively, all these studies suffer from limitations introduced by the used dataset.

Gap: Studies Based on Data by Li and Henry Though used in a variety of experiments, relying on the Li-Henry data is problematic for studies aiming to predict maintainability. At the time of writing this thesis, this data is 29 years old and corresponds to software written in Classic-Ada. It is unclear how these results transfer to modern software systems and modern programming languages. Furthermore, the dataset contains only 110 data points. Still, many ML algorithms are known to perform best when trained on a large number of samples. Also, the generalizability of any results obtained on such small datasets is questionable. Moreover, the use of changed code lines as a surrogate for maintainability yields some limitations.

Number of Revised Lines of Code

Other studies refer to the number of revised lines as a surrogate for maintenance efforts. However, they create their own, larger dataset to increase the validity of their results. For instance, Zhang et al. [227] followed the methodology of Li and Henry [126] to create a dataset from 8 systems. They combined the number of changed lines with the number of commits to capture the effort to change a method. This is supposed to be a better approximation of the actual maintenance effort than relying on revised code lines alone [227]. Al Dallal [3] used a similar approach and investigated both the number of revisions and the number of changed lines. Jain et al. [92] also assembled their own dataset from four open-source projects and applied a genetic algorithm to predict the change between two project versions.

In contrast, Malhotra and Lata [138] used a slightly different approach. First, they extracted the revised lines of codes per file. Second, they modeled the prediction as a binary classification problem. Therefore, they split the data into easy and hard-to-maintain files based on the number of changes. Unfortunately, they do not report the used threshold to create the binary split. Furthermore, their study focuses mostly on the effects of various data preparation methods.

Gap: Using Revised Lines of Code as a Surrogate for Maintainability A variety of researchers created larger datasets from state-of-the-art study systems, thus partially closing the gap left by the Li-Henry dataset from 1993. Still, some conceptional drawbacks remain unattended. Researchers have not discussed the relation between revised lines of code and maintainability in much detail. Using this proxy measurement suffers from several assumptions: It inherently assumes all changed lines require identical effort. In general, it does not consider the complexity of the maintenance task. This means large changes are assumed to originate from supposedly bad maintainability, not from the complexity of new requirements. Also, change does not necessarily occur at the spot of the underlying problem. We observed that historical code can evolve into an incomprehensible and virtually unchangeable form. Thus, it is likely that this code file will never be changed; instead, the neighboring classes are adapted to cope with that. Furthermore, the most severe limitation is the lack of attention to any maintenance activities but code writing. However, understanding the code before changing or enhancing it is an integral part of maintenance [215], especially if the code was written by a different person [163]. These efforts must not be neglected in maintainability assessments.

2.2.3. Predicting Code Smells

A third family of studies aims to predict code smells as a surrogate for software maintainability. There exist several static tools to detect code smells heuristically, e.g. Decor [145], Ptidej [71], JDeodorant [211], and PMD [169]. A comparative study has been conducted by Pecorelli et al. [161] contrasting heuristic detection approaches and ML-based approaches for five code smells. Both approaches are shown to have different strengths. While the heuristic approach slightly improves upon the ML approach, their performance leaves room for future work. It yields an F-Score between 0.16 and 0.44 depending on the code smell.

In two studies, Arcelli Fontana et al. [7, 59] trained and compared several ML algorithms on an imbalanced, binary dataset. They reached F-Scores of over 0.95 and found only little performance differences between the different classifiers. Using the same dataset, Mhawish and Gupta achieved similarly good results [142]. The study design and data used by Arcelli Fontana in [7, 59] were later on analyzed and criticized by Di Nucci et al. [50]. They replicated the study with a refined design and different evaluation techniques, which account for the imbalanced dataset. Interestingly, they found major differences between the single classification algorithms. Notably, no algorithm exceeded an F-Score of 0.5.

In contrast to these works, Sharma et al. [192] applied code2vec to detect code smells. One advantage of this technique is the transferability of the trained models to other languages [192]. The highest performance in their experiments was observed for detecting the long method smell. It is at an F-Score of 0.64 and an mcc of 0.67.

What sticks out in this research area is the number of considered smells. Contemporary tools just offer detection mechanisms for a limited range of smells [58]. The same observation holds for research. Most studies on machine-learned code smell detection only aim to detect a small subset of smells. Out of 20 studies reviewed by Kaur et al. in [102], five studies focused on just one smell, ten works considered three or four smells, and only five publications aimed to predict five or more smells. Notable exceptions are [106, 105, 154]. Hybrid detection approaches, in general, consider more code smells than ML-based studies [102].

Gap: Predicting Code Smells There is a growing body of research on using ML to detect code smells. However, the applicability of current approaches for maintainability assessments is limited. Most studies focus on a small selection of smells, thus leaving important aspects unattended. Furthermore, the accuracy of contemporary approaches rarely exceeds the performance of heuristic-based tools. Eventually, one major limitation of code smell detection originates from the concept of smells itself. Since code smells represent indicators for potential maintainability issues, every finding has to be further analyzed manually. Hence, the maintainability of the analyzed code, in general, remains speculative.

2.2.4. Predicting Expert Judgment

The judgment of experts is a more accurate predictor for maintenance effort than the presence of code smells [222, 223]. In addition, expert judgment is found to be superior to formalized estimation methods [111]. Therefore, we argue predicting expert judgment is the most promising path toward ML-supported maintainability assessments.

Unfortunately, human-labeled data is subject to bias or imprecision. To account for the imprecision of human beings, Pizzi et al. [165] adjusted the labels provided by experts before using them. After these adjustments, the performance of the trained models increased.

After finding conflicting expert ratings in a previous study [81], Hegedűs et al. switched to surveying only a single expert to avoid conflicts [82]. In the earlier study [81], 35 experts were asked to rate the changeability of ten methods each on a three-point scale. Using static code metrics as input, their classification models achieved moderate performance. Considering accuracy, it outperformed the distribution-based baseline by only 10 percentage points.

In a consecutive study [82], Hegedűs et al. again referred to a threefold scale but converted it to a numerical scale after collecting the labels. Then, regression algorithms were trained to predict the numerical representation of that class. Furthermore, they experimented with different datasets and data collection methodologies. They conclude that creating a useful dataset is a major and ongoing challenge.

Hayes et al. [79, 80] used the perceived maintainability rating on a scale from 1 to 10 as the dependent variable. They had students perform maintenance tasks on an application created by students, too, and asked for the students' maintainability judgment.

A different approach to creating manually-labeled data was employed by Zhong et al. [228]. As the first step, they used unsupervised learning to cluster a large number of code classes. Then, as the second step, one representative of each cluster was manually analyzed and labeled by an expert. This label was then applied to all members of the cluster. In contrast to this thesis, they considered only binary classification and focused on the error-proneness of code. Also, we refrain from automatically assigning the same label to all members of the same cluster.

Corazza et al. [38] predicted the quality of code comments, which have been manually labeled. The problem of diverging expert opinions was observed in this domain as well. Similarly, Hegedűs et al. [82] recognized the subjective bias in manually labeled data as a major limitation of their studies, too. To cope with this challenge, Jørgensen proposed to rely only on the most accurate expert [95]. However, the experience of an expert is no indicator of more accurate predictions [96]. Hence, their experience is an insufficient selection criterion. If no data on the accuracy of the experts is a-priori available, Jørgensen proposed to use the aggregation of several experts [95]. In contrast, Tokmak and colleagues [210] hint that study participants should not be treated equally. Their study indicates that experts and novices tend to evaluate software differently. However, their work focuses on quality-in-use evaluated by end users of an application.

Gap: Predicting Expert Judgment Predicting the maintainability judgment of an expert is arguable a promising way to support maintainability assessments. However, the quality of an ML application depends on the quality of the utilized data. All studies mentioned in this subsection did not share their manually labeled datasets with the scientific community, hence limiting the reproducibility of their results. Also, most datasets are biased towards a single expert, a small group of experts, or refer to students. Due to the tedious manual labeling, the used datasets are small and thus challenging for ML. There is a need for a large, expert-labeled dataset denoting the consensus of several qualified experts.

2.3. Machine Learning Background

Machine Learning (ML) applications map input to a specific output based on rules which are inferred from example data. To develop ML applications, two concepts have to be considered: A dataset describing the desired output for a given input, and the representation of the input. In our context, we need to represent source code in a form that is suitable to be fed into ML algorithms; and a dataset denoting the maintainability of the source code.

Publicly available datasets exist for a variety of related research areas, including

- the number of changed code lines and static metrics for 110 code files provided by Li and Henry in [126]
- several datasets containing revised lines of code, following the methodology of Li and Henry
- various code smell datasets such as the MLCQ dataset by Madeyski and Lewowski [135], the Landfill dataset by Palomba et al. [155], and the QScored dataset by Sharma and Kessentini [193]

- several datasets focusing on functional quality, as summarized by Wahono [217]

However, there is no publicly available data concerning the expert assessments of code files.

ML research distinguishes between different experimental settings based on the dependent variable to be predicted, i.e. the output. This can be a nominal characteristic, ordinal categories, binary classes (as the presence of a code smell), or numerical values (as the number of revised code lines). If there are more than two possible outputs, the task is called a multiclass classification. As we will explain later in more detail, the main focus of this thesis lies on an ordinal multiclass prediction problem.

2.3.1. Feature Engineering

In the context of ML, the term *features* refers to the independent variables, i.e. the input based on which the output is predicted. The selection and preparation of the features are crucial for the success of ML applications [205, 207]. Using only a few input features with high predictive power is likely to mitigate overfitting and to increase the performance of an algorithm at the same time [110, 120]. The objective of feature engineering is to extract factors from raw data with a high predictive power towards the predicted attribute. In our case, the raw data corresponds to the source code and the predicted attribute to its maintainability. However, source code in general is arbitrarily long, while most algorithms require inputs of fixed length.

Static Code Metrics as Features

Most studies mentioned in Section 2.2 use static code metrics as input. They are easy to extract, are hypothesized to correspond with maintainability at least to some extent, and allow to map source code to a fixed number of features. Frequently employed metric suites include Halstead metrics [10, 75], the Maintainability Index and measurements contributing to its calculation [153], the Chidamber-Kemerer metric suite [35], or the MOOD [77] metrics. Studies relying on these metric sets are biased by the chosen selection. Other metrics might provide higher predictive power and improve the prediction performance. However, these features are excluded a-priori.

In contrast, we propose to investigate a broad spectrum of metrics containing different types of metrics such as size, coupling, inheritance, code duplication, code style violations, and tool warnings. Then, colinear metrics are removed to avoid distorted analysis results [8, 51, 205]. Eventually, we evaluate the remaining candidates for their predictive power towards a given label and use only the most promising candidates.

There exist several techniques to quantify the predictive power of a feature candidate. In this study, we rely on *mutual information* [118, 178], as it does not make assumptions about the distribution of the metric values. Alternative measurements are Pearson's Correlation

Coefficient [159], Spearman's Correlation Coefficient [202], and the ANOVA F-value. If there are no computational constraints, it is also possible to exhaustively investigate all subsets of features without ranking them.

Other Input Features

In addition to code metrics, there exists a variety of other possibilities to represent source code for ML. These include statistical language models, such as *n-grams* or *SLAMC* [149]. Both approaches consider source code as a sequence of tokens, which are generated by lexical analysis. N-grams calculate the probability that a given token will follow a specific sequence of previous tokens. The parameter *n* specifies the number of previous tokens to take into account. This technique is used, among others, by [86, 156] and [190]. SLAMC is an extension of n-grams for source code [149]. It also considers different types of tokens, e.g. whether a token denotes a literal, a variable, or an operator.

In a preliminary study, we investigated the use of language-based transformer models such as *BERT* [49], *CodeBERT* [56], and *RoBERTa* [130]. Our initial findings on using these techniques to predict maintainability are reported in [190]. Furthermore, we report on our experiments using image classification to predict the quality of source code in [190]. Screenshots of syntax-highlighted code files are fed into both Neural Networks and Support Vector Machines (SVMs) to classify their readability, understandability, or complexity. The idea stems from the observation that human reviewers often build a strong intuition about the quality of code at a first glance and without actually reading the code. Still, these contributions are not the focus of this dissertation and are only briefly touched upon in Section 9.2.

2.3.2. Evaluating Machine Learning Models

Performance Metrics

ML algorithms are trained on one partition of the data, the train set, and evaluated on a disjoint partition, the test set. Different views on the confusion matrix provide different metrics to quantify the performance of the algorithm. Popular measurements are the *accuracy* of the algorithm, i.e. the ratio of correctly classified instances, its *precision*, *recall*, and *F-Score*. On imbalanced datasets, as they are typically found in software quality prediction [138, 50], these metrics can be misleading [50]. Here, the *Matthews Correlation Coefficient (mcc)* [67] is the preferable measure [224] as it accounts for imbalanced distributions. It measures the agreement between two ratings, i.e. the predicted label and the true label. An mcc ($mcc \in [-1; 1]$) value close to 0 indicates no agreement or agreement only happening by chance, while $+1$ corresponds to the perfect alignment of both ratings. Negative values describe inverse correlations. In the multiclass case, the calculation of mcc is slightly adapted to consider the total number of correct and incorrect predictions across all classes [123].

In this thesis, predicting the maintainability of code will be formulated as an ordinal multiclass classification problem. For binary classification problems, Receiver Operating Characteristic (ROC) [55] and its Area Under Curve (AUC) are preferable [22]. However, they are not recommended in a multiclass setting [201]. For regression experiments, different performance metrics are used due to the continuous scale. The mean squared error, the explained variance, and the R^2-score are commonly applied. As they are not suitable to evaluate classification results, comparing the performance of regression and classification models is difficult.

Naturally, the observed performance of an ML model can be biased by the chosen train-test split of the available data. Cross-validation is one technique to avoid such bias by repeating the experiment several times with varied data splits. In k-fold cross-validation, the dataset is split into k partitions. Then, the experiment is performed k times with a different partition used as the test set in every run. The performance of an ML model is then calculated against the union of all runs.

In the majority of our ML experiments, the data is sampled from nine study objects (cf. Chapter 5). This allows for using project-wise cross-validation: Instead of random sampling, we split the data along the study objects. Thus, each model is trained on data from eight projects and evaluated on data from one project; this procedure is repeated nine times. Other splits like six projects for training and three for testing would also be possible, but we want to use as much as possible of the limited amount of data for training. Hence, we use eight projects for training.

Aggregating Multiclass Performance Metrics

In Chapters 6 and 7, multiclass experiments are performed, which aim to classify data points on a four-part scale. Similar to binary classification, several performance metrics can be calculated to evaluate multiclass prediction models. To facilitate the comparison of classifiers, it is useful to evaluate their performance across all labels. There exist two ways to compute the aggregated performance: *macro*-averaging and *micro*-averaging.

Macro-averaged metrics represent the unweighted average of the respective metric for each class [124]. Given a dataset with four labels, the performance scores for all four classes are computed first. Then, their average is reported as the performance of the classifier. This way, all labels are weighted equally. In contrast, micro-averaging maintains equal weights for each data point, independently of the label distribution [124]. It considers the complete confusion matrix across all labels when computing the performance score. True positives and false positives are counted across all classes before the performance metric is computed based on these values. Unless stated otherwise, we will refer to micro-averaged performance metrics when comparing classifiers.

An interesting side effect of the micro-average is, that *"micro-averaging in a multiclass setting will produce precision, recall and F[-Score] that are all identical to accuracy."* [124]. This does not mean that they describe the same concepts, but their calculation results in the same numerical score. We illustrate this using precision. Precision is defined as the ratio of true positives divided by the sum of true positives and false positives, i.e. how often the class was predicted by the classifier.

$$Precision := \frac{TP}{TP+FP}$$

In the multiclass case, this unfolds into the following fraction: the numerator is the sum of all true positives; the denominator is the sum of how often each class was predicted (i.e. true positives and false positives of each class). Consider the four labels A, B, C, and D:

$$Precision\_multiclass := \frac{TP(A)+TP(B)+TP(C)+TP(D)}{(TP(A)+FP(A))+(TP(B)+FP(B))+(TP(C)+FP(C))+(TP(D)+FP(D))}$$

As each data point has to be classified as either A, B, C, or D, the denominator contains every data point. Its value is equal to the total amount of data points. Furthermore, the numerator corresponds to the amount of correctly classified instances. Thus, micro-averaged precision results in the same value as accuracy, which is the ratio of correctly classified instances. The definition of recall can be extended analogously and results in the same formula. Since recall and precision share the same value, the F-Score is also identical[1]. Thus, accuracy, precision, recall, and F-Score yield the same value in this case. In our experiments, we refer to the implementation in *scikit-learn* [162], a state-of-the-art ML library. Their user guide [124] provides the calculation rules for all used metrics.

Evaluation Baselines

Despite the availability of a variety of evaluation metrics, it is vital to put the observed performance into context. As Di Nucci et al. [50] point out, high values can often be deceptive. Frequently, the distribution of the labels in the dataset facilitates high accuracy values. Therefore, it is necessary to compare the performance of a classifier against baselines. One baseline is generated by a naive classifier, which, for example, predicts the most popular class in the training set. An alternative baseline classifier may randomly assign labels. For better comparability, the probability of each label is set according to its frequency in the training data. Since these naive classifiers do not infer any rules from the features in the dataset, they are sometimes referred to as Zero Rule or ZeroR classifiers. These classifiers shed light on how many correct classifications originate from the dataset distribution. However, as mcc takes the distribution of labels into account, they yield an mcc of 0.0 since all agreements happened by chance. While maybe harder to interpret than F-Score, this illustrates nicely why we value mcc over F-Score. Later in this thesis, we will introduce a human-level baseline, too. This is particularly interesting for tasks on which humans are likely to fail or human performance is incoherent. Depending on the use case, it might be utopian to aim for perfect prediction accuracy. Instead, a reasonable first step is to converge

[1]An extended version of this explanation is available by the developers of the Weka ML library: `https://waikato.github.io/weka-blog/posts/2019-02-16-micro_average/`

towards average human performance. The classifier can then be used to mimic a second human opinion and support the person actually performing the task.

In general, the context of the application must not be neglected. Depending on the criticality of misclassifications in the given use case, there exist situations where high precision is crucial but small recall is acceptable. In other cases, it might be preferable to aim for high recall and accept low precision.
In conclusion, we note that relying solely on performance metrics is insufficient to evaluate the quality of an ML classifier. The results have to be put into the context of the application area and compared against reasonable baselines.

2.3.3. Popular Architectures for Machine Learning

There is an increasing number of ML algorithms and architectures available. This section introduces those mentioned in the remainder of this thesis and explains them briefly. Among others, we applied K-Neighbor classifiers, Random Forests [23], Gradient Boosting [62], Ada Boosting [78], Extremely Randomized Trees [65], Logistic Regression [88], and code2seq [5]. However, to keep the focus on those algorithms, which are most relevant for this thesis, we refrain from explaining all techniques in detail.

Tree-based Algorithms Decision trees are among the most popular ML algorithms. They are easy to understand, simple to use, and do not require long training periods. However, they are prone to overfitting as small changes in the features can lead to significantly different results [114, 162]. One solution to this problem are ensemble classifiers, which consist of a large number of weak classifiers. In Random Forests [23], several internal decision trees are trained and their predictions are aggregated. They introduce randomness so that each instance of a tree uses a different subset of the features and the training sample. Thus, overfitting is mitigated and the performance is increased. Eventually, the votes of all trees contribute equally to the final prediction. Extremely Randomized Trees [65] slightly differ from Random Forests considering how the decision trees are constructed but follow the same principles.

Boosting is a technique to improve the performance of ensemble classifiers by adding new internal classifiers which focus on mitigating previous errors [61]. Consequently, boosted trees take the concept of Random Forests one step further. Instead of initializing and training all trees in parallel, the trees are built consecutively. Each tree focuses on the weaknesses of the already existing trees. Two implementations of this principle are Ada Boosting [61, 78] and Gradient Boosting [62].

Code2Seq In addition to these generic learning algorithms, some techniques specialize specifically in source code as input. For instance, code2vec [6] and its successor code2seq [5] utilize the Abstract Syntax Tree (AST) as input. In the past, they have been applied to detect code smells [192], predict bugs [167, 218], localize faults [127], summarize code [5, 6] and predict method names [76, 125], as well as in autocompletion [109] and vulnerability prediction [19].

The code2seq framework implements the following principle: First, the framework generates the AST of a method and extracts a random subset of paths from the tree. Second, an encoder-decoder architecture predicts a property of the code based on the paths. Here, the encoder defines an individual vector representation for each path. The resulting encoding is then equivalent to the sequence of nodes in that path. Then, this embedding is used to predict the label. Originally, code2seq targets code attributes at the method level. In order to predict code maintainability at the level of code classes, we need to adapt the framework. Most importantly, we need to change the granularity of the AST from methods to classes. This results in a significantly larger number of paths in the tree. Consequently, several adaptions to the AST are necessary. We will elaborate on them in detail in Chapter 7.

2.4. Industrial Perspective

This thesis aims to support expert-based maintainability assessments. To contribute to the state of the art both scientifically and practically, it is necessary to establish close links with industry. The perspective taken on in this thesis is highly inspired by our industrial partner itestra GmbH. With branch offices in several European countries, itestra GmbH brings nearly 20 years of industrial experience to this joint research. In addition to software engineering and consulting projects, itestra GmbH is well-known for its post-release maintainability assessments. These are often referred to as Software Health Check [164]. They are based on scientific principles [139] such as concise and consistent naming [45] and activity-based maintainability models [25, 46, 216]. Many industrial big players rely on their software assessments [139]. Furthermore, they have been known for several years to contribute to the state of practice by publishing industrial experiences and insights from research projects, e.g. [44, 83, 97, 139, 164, 189, 198, 204, 216].

Hence, we believe they are qualified to provide informed opinions on current challenges in industry and on the potential benefits and disadvantages of our solution.

Part II.

Software Maintainability Assessments

3. Software Quality Assessments in Practice

Parts of this chapter have previously appeared in a peer-reviewed publication ([188]) co-authored by the author of this thesis.

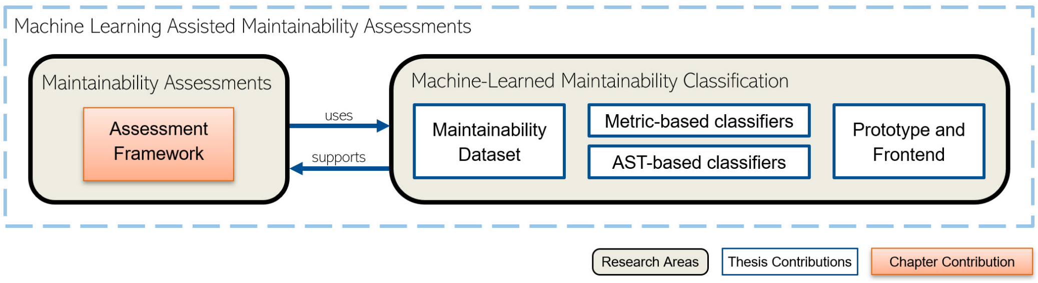


Figure 3.1.: Contribution of this chapter, i.e. an assessment framework, in the context of the thesis

This chapter contributes an assessment framework, that models how maintainability assessments are performed at our industrial partner. Figure 3.1 illustrates the contribution of this chapter in the context of the dissertation.

Assessments by experts are arguably the most precise way to evaluate the maintainability of a software system [111, 164]. Still, the required manual work results in high costs and significant time spent. Little is known about how these assessments are performed in detail. However, insights into the state of practice are a prerequisite to improve and support these assessments. To close this gap, we participated in two HCs at itestra GmbH. Our findings and the experience contributed by the industrial partners were consolidated into a structured assessment framework. Although tools are used to guide the analyst, a majority of the analysis is performed manually.

Problem: HCs are an effective but time-consuming and tedious way to analyze the maintainability of a software system. In order to facilitate them, it is necessary to understand the activities performed during an HC. Also, it is crucial to learn how experts aggregate their observations into a maintainability judgment. Specifically, there is little evidence on the role of tools currently used during the assessment.

Solution: The goal of this chapter is to make software quality assessments systematic and structured. We participated in two real-life assessment projects and captured our understanding of the state of practice as a structured framework. The framework is based on 15 years of our industrial partner's practice, our own observations, and a literature review. It includes the following activities: comprehension of software artefacts, building hypotheses, performing analyses, collecting evidence, and devising conclusions. Together with our industrial partner, we conducted quality assessments of two real-world systems using this process model.

Contribution: We contribute several insights about maintainability assessments. Static analysis tools are necessary to guide the assessment of large systems, but manually analyzing code is still an integral part of the assessment. To select which files to analyze in depth, it is important to build hypotheses early. Furthermore, tangible evidence of problematic maintainability is crucial to refine the hypotheses and eventually convince stakeholders of quality problems.

Limitations: The experience of our industry partner forms the basis of this work. Naturally, the results are biased towards the experts, who participated in this study, and their personal experiences. Furthermore, all partners are from the same company and are thus likely to share certain opinions. Future research could focus on the automation or support of activities that are currently performed manually by experts. In particular, we hypothesize the collection of evidence and identification of problematic code files can be improved.

3.1. Case Studies

3.1.1. Case Study A

In this case study, an offer management system used by an insurance company was evaluated. The owner of the system wanted to know the current quality status and asked for recommended short-term actions that can be deduced from these insights. According to the owner, the time to market provided by the system is non-satisfactory. The introduction of new tariffs or products requires high effort and is time-consuming. The system has been maintained for more than 15 years and was technically developed based on Model Driven Development. The system maintenance is outsourced to an offshore provider with approximately 50 developers working on it. Table 3.1 shows more detailed information about *Case Study A*.

Table 3.1.: Case Study A and B (as appeared as *Table 1* in [188])

Items	**Case Study A**	**Case Study B**
Domain	*Financial (Insurance)*	*Financial (Insurance)*
Purpose	*Offer Management System*	*Offer Calculation System for Brokers*
Dev. Language	*Java*	*Delphi*
Release (year)	*2000*	*2000*
Development	*Outsourced*	*In-house*
Maintenance	*Outsourced*	*In-house*
Provided Artefacts	*Source Code, Architecture Description, Data Model, User Manual*	*Source Code, Architecture Document, Data Model*
Size	*Medium-Large (1.8M SLOC)*	*Medium (486k SLOC)*

3.1.2. Case Study B

This system is an offer calculation system for brokers developed and used by an insurance company. It has been developed for the past 15 years by a group of 3 developers using Delphi. Now, the management considers replacing it with state-of-the-art technology. Therefore, the owner asked for a structured review of the technical state of the program, a comparison with similar systems, and a draft of recommended actions for mid-term improvement. The rightmost column of Table 3.1 summarizes *Case Study B*.

3.2. Quality Assessment Framework

Combining our industrial partner's 15 years of experience and our own insights, we composed a framework that describes a structured, activity-based assessment process for software systems. Such an assessment is different from a code review and can rather be seen as a general post-release software audit. Its purpose is to evaluate a given software system in order to find potential quality defects and recommend actions to improve the quality of the system. This is sometimes referred to as Software Health Checks (HCs) [164]. Before the assessment process started, there were several preliminary activities conducted, such as a kickoff meeting, a brief demonstration of the system, and handing over of the artefacts. The quality assessment framework is illustrated in Figure 3.2.

3.2.1. Input

As shown in Figure 3.2, we grouped the input information into three categories. One category captures system and project artefacts. Typical examples of system artefacts are source code, data model, documentation, and other material needed to reproduce the technical project.

For context comprehension, we collect context information that is not stated in the software artefacts as well. Among these are current issues of the system, observed quality in use deficits, the expectations of the stakeholders, the background and history of the system, and more. The third category of input information is the system owner's motivation for the assessment, i.e. known problems, questions to be answered, and the trigger for the assessment.

3.2.2. Process

The quality assessment process consists of three phases: overview, building hypotheses, and analysis.

Overview: This phase aims at acquiring a high-level understanding of the software by mapping the functionality of the system and the software structure. The functionality of the system is derived from the documentation that was provided by the system owner. Also, we conducted an informal interview with a quality consultant that is familiar with the software domain to capture several common functionalities of such systems.

Building Hypotheses: Figure 3.2 visualizes that the phase of building up hypotheses consists of two aspects: *Initial hypothesis* and *working hypothesis*. The *initial hypothesis* is an assumption about the quality of the system and its problems that is purely based on intuition and not on evidence from any detailed analysis. Quality consultants implicitly build a first hypothesis based on the information from the kickoff meeting, the problems motivating the assessment, and their knowledge about common quality defects. In contrast, the *working hypothesis* characterizes an assumption about the quality of the system, that is at least partially based on evidence but still needs to be confirmed through further analysis. *Working hypotheses* represent refinements of *initial hypotheses*.

Analysis: The actual analysis is guided by the hypotheses and is divided into three activities.

- *Selection*: This task relates quality models to the assessment. The quality consultants specify the system parts to be analyzed and choose what facts are to be observed in these files. This selection is driven by a quality model, in our case the fact-and-activity matrix presented in [25]. In this model, cost-intensive maintenance activities are correlated with observable facts within a software system. Sometimes a lack of information becomes obvious at this stage, forcing the quality consultants to inquire additional input from the system owner.

- *Assessment*: The presented framework allows quality consultants to use any quality assessment method. Depending on the quality attributes and the available information, they can apply either automated, semi-automated, or manual methods. In this task, the system is evaluated with respect to the specified criteria.
- *Evidence Collection*: In order to provide valuable feedback to the owner, it is important to augment the observations and conclusions with tangible evidence. We select evidence that represents average findings as well as very severe instances. Code fragments, for example, can illustrate performance issues, or concrete instances of misleading comments provide proof for statements about the quality of comments. Even measurements from automated tools can be useful evidence if they are put in the right context and are supported by concrete examples. As the evidence provides indication either in favor of the current hypothesis or against it, the working hypothesis is adapted according to the new insights and the iterative analysis is continued.

3.2.3. Output

After several rounds of iteration, the quality consultants converge towards a conclusion. This conclusion is combined with the context information about the system to identify possible root causes for the observed quality defects. Adequate actions to improve the quality are deduced and proposed to the owner. This last step has a more strategic nature and is therefore omitted in this study, as we focus on the technical review. In addition to a final presentation, intermediate presentations were delivered to keep the owner up-to-date and provide an opportunity to discuss the next steps.

3.3. The Quality Assessment

This section summarizes the results and findings of our quality assessment activities.

3.3.1. Findings from Case Study A

Input: The list of inputs for *Case Study A* can be found in Table 3.1. Overall, the size of the provided system files exceeded 2 GB and included more than 10,000 files labeled as documentation. The source code was mostly written in Java and consisted of approx. 16,000 Java files. There was only little information provided about current costs, open requirements, and the bug history. The data model was provided as a graphical representation spread over several documents. As described in Section 3.1, we also knew that the software faces a time-to-market problem, due to time-consuming implementation changes needed to introduce new products.

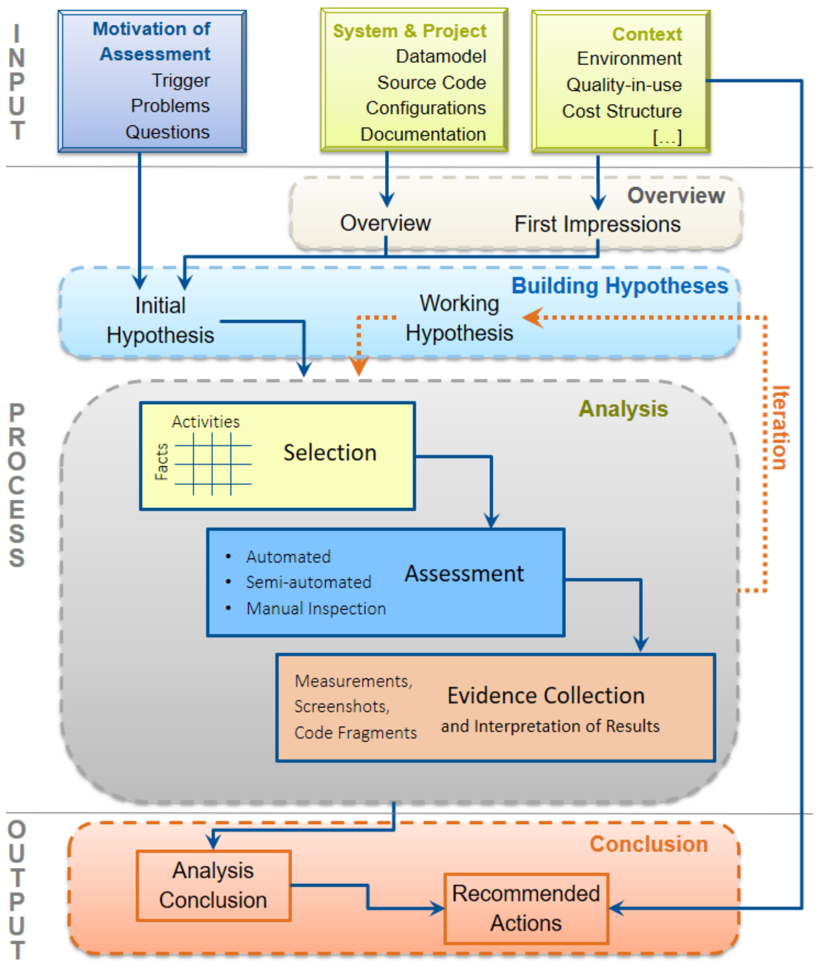


Figure 3.2.: Quality assessment framework (as appeared as *Figure 1* in [188])

Table 3.2.: Case Study A: Initial Hypotheses, Assessment Activities, Evidence, and Findings (as appeared as part of *Table 2* in [188])

	Case Study A
Item	Description
Hypothesis 1	**Poor program structure obstructs the program comprehension.**
Selection	Source Code, Architecture Document
Assessment	(i) Reverse engineer source code into UML diagrams; (ii) Extract object-oriented design metrics using SD Metrics [220]; (iii) Evaluate the size and coupling. metrics.
Evidence	Structural information and KPIs about the system; Examples of nested packages and operations with many parameters.
Key Findings	181 out of 1027 packages (17.62%) were found to have a high nesting value ($\geqslant$ 6 levels). Out of these 16126 classes, 633 classes (3.93%) show symptoms of god classes. 9.6% of 65522 operations have a long parameter list ($\geqslant$ 6 parameters).
Hypothesis 2	**The software suffers from code cloning that decreases the software maintainability.**
Selection	Source Code
Assessment	(i) Calculate cloning ratio using ConQAT; (ii) Identify generated code; (iii) Re-calculate cloning ratio; (iv) Evaluate the code cloning measures.
Evidence	Duplication KPIs from ConQAT that describe the code clones and ratio; Examples of duplicated code that illustrate a copy and paste policy in a specific package; Examples of generated code.
Key Findings	With 34.4%, the cloning ratio is considered high; The amount of generated files was surprisingly high.
Hypothesis 3	**The identifier naming convention is either not existing or not maintained which affects the program comprehension.**
Selection	Source Code
Assessment	(i) Randomly select source files from key modules; (ii) Observe identifier naming convention patterns; (iii) Discover identifier naming convention violations.
Evidence	Examples of inconsistent naming; Examples of different languages used for identifiers.
Key Findings	A global naming convention could not be identified; Multiple languages were used to name identifiers.
Hypothesis 4	**The poor quality of code comments affects the program comprehension.**
Selection	Source Code
Assessment	(i) Examine all comments in the key modules and randomly selected other modules; (ii) Discover issues related to those comments.
Evidence	Examples of commented-out code; Examples of confusing comments; Examples of comments with mixed languages.
Key Findings	A lot of source code was found in comments; Comments pointing to known problems (i.e. *'todo'* and *'fixme'* annotations) were found in productive code; Comments are sometimes ambiguous or mingle multiple languages.
Hypothesis 5	**There are issues in the data model that lead to a decreased modifiability of the system.**
Selection	Data Model
Assessment	(i) Acquire a high-level comprehension of the data model; (ii) Discover the high-level data model concept or pattern; (iii) Investigate low-level issues of the data model such as duplication, missing normalization, or inconsistencies.
Evidence	Examples of repeating attributes; Instances of missing normalization or standardization.
Key Findings	Attributes appear in several tables. Partially, these attributes have inconsistent field types. The schema was not normalized.

Process: The results and findings of this phase are the following:

Overview: Since the software documentation only showed parts of the software design, we rediscovered the low-level software structure by constructing UML diagrams from the source code using the tool Enterprise Architect CASE [1]. From these diagrams, it is almost explicitly shown that the modules of this software are structured according to the system's functionality. Hence, the mapping between functionality and structure is straightforward.

Building Hypothesis: From a product quality viewpoint, the software faces a modifiability issue that decreases its maintainability. Therefore, we formulated the initial hypotheses based on the maintainability matrix from [46] and our own experience. For example, redundancy affects the modifiability of a system. The quality of code comments and identifier naming influences the maintainability as well. Hence, the initial hypotheses pertain to these aspects.

Analysis: The activities in this stage were performed based on the formulated hypotheses. Table 3.2 summarizes the initial hypotheses that were refined over time, the assessment activities performed in each iteration step as well as findings and evidence. For reasons of conciseness, we only explain five hypotheses.

Output: In the context of this case study, the quality of a system describes the degree to which the fitness-for-purpose is reached while the running costs of the system are kept minimal. Running costs can be maintenance costs or operating costs. The maintenance effort is determined by maintenance activities, that can be related to observable facts inside the product [25] whereas operating costs are negatively influenced by ineffective implementations. Hence, a software system is considered *good* if the software fulfills its purpose at minimal running costs. Source code is considered *good* if the way it is written does not increase maintenance costs compared to other solutions and there is no reasonably achievable way to make it more efficient.

Table 3.2 only illustrates a small part of the findings. More activities had to be performed to evaluate the complete system. At the end of the assessment, we suggested two alternatives to the system owner: (i) Develop a new system that incorporates new technology for better scalability; or (ii) Buy and customize an available standard product. This conclusion was mainly influenced by the comparison of the maintenance costs and the perceived cost for replacing the existing software.

3.3.2. Findings from Case Study B

This subsection describes the activities and highlights the crucial results for *Case Study B*, which are summarized in Table 3.3.

Input: The list of inputs for *Case Study B* is denoted in Table 3.1. Even though the artefacts provided were fewer than in *Case Study A*, the artefacts were up to date and easy to

[1]http://www.sparxsystems.com/products/ea/

Table 3.3.: Case Study B: Initial Hypotheses, Assessment Activities, Evidence, and Findings (adapted from *Table 2* in [188])

	Case Study B
Item	Description
Hypothesis 1	**Bad coding practices such as duplicating code and hardcoding values decrease the quality of the system.**
Selection	Source Code
Assessment	(i) Browse source code from key modules and identify hardcoded identifiers; (ii) Discover code duplication using ConQAT; (iii) Evaluate the findings.
Evidence	Examples of duplicated code and its location; Examples of hardcode values.
Key Findings	Code duplication is a practice in this project; 4.4% of all code lines are part of a verbatim clone. Hardcoded values like paths and email addresses were found in the source code.
Hypothesis 2	**The poor quality of code comments affects the program comprehension.**
Selection	Source Code
Assessment	(i) Examine all comments in key modules and randomly selected other modules; (ii) Discover issues related to those comments.
Evidence	Examples of commented-out code; Examples of useless comments.
Key Findings	A lot of source code was commented out; Comments were used for informal discussions or as optical delimiters, provided no additional information, or used ambiguous language.

walk through. The assessment was supposed to answer whether the system is technically sustainable and suitable for future requirements.

Process: The results and findings of this step are the following:
Overview: In this case study, we refer to the software architecture and the source code to get a high-level understanding of the system's functionality and program structure. Although the software architecture was presented in a simple way, it was sufficient to understand the program structure.

Building Hypotheses: The system has been maintained by the same three in-house developers for 15 years. Due to a high risk of lost knowledge after staff turn-overs, we concentrated on the maintainability and understandability of the source code. Therefore, the initial hypotheses focus, for example, on code comments and bad coding practices.

Analysis: In Table 3.3, we illustrate two examples of hypotheses, the activities conducted in each iteration, and the corresponding findings and evidence.

Output: The information illustrated in Table 3.3 is only a small part of the findings from this assessment. The quality issue regarding the maintainability of the system is not severe and only improvement actions with minimal effort are recommended. However, this system is suggested to be replaced for strategic reasons. In order to be ahead of their competitors, the owner needs to acquire new state-of-the-art software that offers more functionalities and has better technical sustainability.

3.4. Discussion and Lessons Learned

The presented framework is regarded as a general description. Due to the project-specific nature of quality assessments, the framework is kept tailorable and flexible. Nevertheless, we are working on concrete guidelines on how to apply each step of the framework in detail and will provide them at a later stage of our research. The remainder of this section explains our lessons learned after assessing the quality of the study objects and discusses possible threats to validity.

3.4.1. Assessing Software Structure

When we conducted the software structure review, we observed several metrics. However, even though some of those metrics indicate an issue of the software, we experienced it differently when searching for concrete evidence. In many cases, the metrics were just misleading. For example, the tool identifies classes with more than 60 attributes and operations as ’god classes’. This term describes a class that tends to centralize the intelligence of a system or has multiple responsibilities [151]. Looking at the concrete instances, we found that the calculation of attributes and operations also includes constants, getters, and setters. In our point of view, those do not add any significant responsibilities to a class. Thus, the tool led to many false positives which had to be excluded.

3.4.2. Interpreting Redundancy Ratios

In our study, we observed very high cloning ratios. But when examining concrete instances, we found that lots of the files accountable for this number were generated files. Usually, quality analyses omit generated files, but in *Case Study A*, many generated sources were not labeled as such and were mixed with hand-written code. After identifying and removing these sources, the observed redundancy was significantly lower. Another example from our case study are files that define constants. Being almost identical from a structural point of view, these files were categorized as clones although their content is unrelated.

3.4.3. Providing Evidence

Measurements are meaningless unless they are backed up by tangible, convincing, and representative evidence. Even though several measurements can provide some sort of benchmark, providing evidence is necessary to prove the reliability of the measurement. In addition, evidence is helpful to demonstrate the negative consequences of found quality defects. For example, developers are more likely to accept cloning as a dangerous practice if they are presented with concrete code snippets together with the observed statistics.

3.4.4. Analyzing Identifiers

Browsing source code and performing program comprehension, we experienced the importance of identifiers for this task. We can confirm the claims of Deissenboeck and Pizka in [45], who showed that a concise and consistent naming of attributes is crucial for software understandability. From our observations, we can support that finding and add another dimension to it: All names should be taken from the same language. In our case study, we found identifiers derived from English, French, and German. Since we performed this task manually, we did not collect statistic information about the naming anomalies. The list of found anomalies was evidence enough to explain the issue to the product owner.

3.4.5. Evaluating the Data Model

In *Case Study A*, the data model was evaluated manually since only data model diagrams were provided as images. In *Case Study B*, the data model was provided as an SQL script. Browsing the models, we uncovered several anomalies. For example, it was impossible to store more than one telephone number per person. In addition, there was no strategy to manage history data - a task crucial for insurance companies. Our lesson learned here is that the data model is a valuable source of information. It is created to support the current functionality and therefore a lack of flexibility in the model can point to problems when introducing new functionalities.

3.4.6. Assessing Code Comments

Comments are a useful possibility to augment source code with additional information. However, we experienced that comments are often misused. The use of comments to discuss open issues instead of using an issue tracking tool indicates space for improvement on the process level. The same point can be made about *'todo"* and *'fixme'* tags in productive systems. Also, we saw a lot of code being commented out instead of removed. This code is likely to distract maintainers and therefore increases the maintenance effort.

3.4.7. Threats to Validity

The internal validity of our studies can be threatened by the small group of quality consultants (2) from just one company and the small number of researchers from academia (2) performing the assessment. In addition, it has to be mentioned that both examined systems were taken from the insurance domain, which can be seen as a threat to external validity. During the analysis, all analysts actively tried to avoid any bias and keep an open mindset. But as the systems were selected for a quality assessment by the owner, the quality consultants knew there was a certain likelihood to find quality deficits. Due to the motivation of the performed assessments, both case studies focused on maintainability.

3.5. Conclusion

This study aims to relate abstract quality models to concrete quality assessments. We present our tailorable framework to perform structured quality assessments. Applying it to real-world systems in two case studies with our industrial partner, we can conclude that blindly using automated assessment tools is not sufficient, though we can still use their output. Other lessons learned are, e.g., that (i) evidence collection is crucial not only to convince the system owner, but also the software developer, and (ii) design and maintainability of the data model may reflect the flexibility of a system.

This is a part of our early work to bridge the gap between quality characteristics, measurements, and assessments. We see several ways to improve this work such as (i) refining the framework and applying it to other systems from other domains; (ii) further research on convincing evidence from the perspective of various stakeholders; (iii) automatization of the activities in this approach and (iv) building a taxonomy that shows the relation between quality characteristics, quality measures, and assessment activities.

Part III.

Using Machine Learning to Predict Expert Maintainability Ratings

4. Pre-Study and Initial Experiment

Parts of this chapter have previously appeared in a peer-reviewed publication ([187]) co-authored by the author of this thesis.

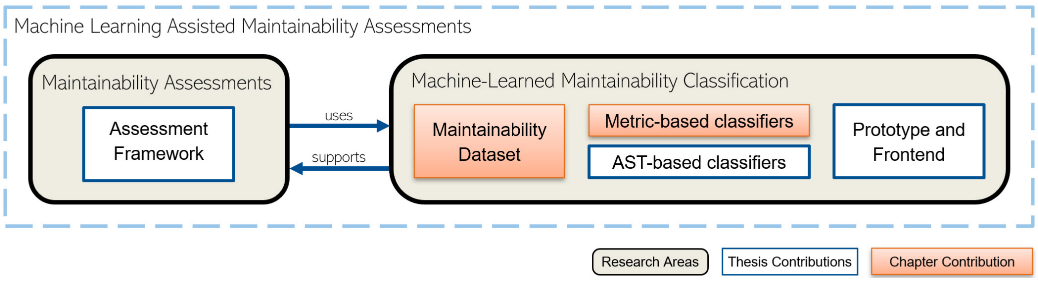


Figure 4.1.: Contribution of this chapter, i.e. an initial dataset and metric-based classification models, in the context of the thesis

This chapter presents our first set of experiments toward machine-learned maintainability classification. It features both metric-based classifiers as well as an initial dataset. Figure 4.1 illustrates the contributions of this chapter in the context of the proposed framework.

The previous chapter introduced industrial maintainability audits and our framework to conduct them. Taken together with the results of the initial experiment, the structured model allows choosing which aspects to support with automation and which tasks should remain in the hands of experienced consultants. While it seems utopian to automate strategic recommendations by experienced analysts, our research targets to identify problematic code. These code files can then be analyzed and interpreted by the expert. In this chapter, we demonstrate that machine learning can be useful to detect hard-to-maintain code and thus may contribute to the automation of health checks. However, it should only be used to support the expert and not to replace them.

Problem: Although HCs employ static tools, they are still time-consuming and require lots of manual work. Using ML to predict the maintainability of code as it is perceived by experts can facilitate the assessment. Only few related studies used machine learning to predict the judgment of experts; most refer to a proxy measure such as the number of changed code lines.

Solution: This prestudy evaluates if ML techniques can be used to predict the maintainability judgment of an expert instead of surrogate labels. With the help of software analysts working at itestra GmbH, we manually analyzed source code from three different projects, accounting for 115,373 lines of Java code. The experts labeled the corresponding Java classes for their maintainability to create a labeled dataset. Then, we retrieved the output of three static analysis tools for these classes and attached the labels. Eventually, we trained supervised machine learning algorithms to assess the maintainability of source code based on static measurements.

Contribution: Using a three-fold maintainability label and stratified 10-fold cross-validation, one classifier was able to predict the expert ratings with an accuracy of 81%. We interpret this performance both optimistically and pessimistically. Although the observed performance is promising, we conclude ML classifications cannot replace evaluations by experts. However, they can be used to create overviews and to support the analyst in selecting which parts of the system to focus on.

Limitations: The major limitation of this work stems from the used data. Large, well-crafted datasets with sophisticated sampling and labeling procedures are a priority for future research. The used dataset consists of a convenience sample of just 345 data points. However, there was no clear definition of the assigned labels and the labels were only informally discussed among the analysts.

4.1. Experiment Setup

In contrast to other studies, this study does not measure task-completion-time to refer to comprehensibility [182] or the number of revisions to refer to maintainability [120, 146]. Instead, we work together with professionals from industry and their experience-based definition of maintainability. For this purpose, we define maintainability as the *ease of change*, leading to two sub-characteristics:

1) As a developer, can I understand what the code does and identify where certain aspects are implemented?

2) As a developer, do I have to worry about hidden dependencies of the code I am currently modifying?

While the first aspect addressed the need to comprehend the source code, the second one focuses on where else the developer has to apply changes. For example, duplications of the code snippet have to be found and modified as well.

Provided the expert judgment, this research answers the following questions:

- Is it possible to predict a human intuition of the maintainability of source code based on tool measurements?

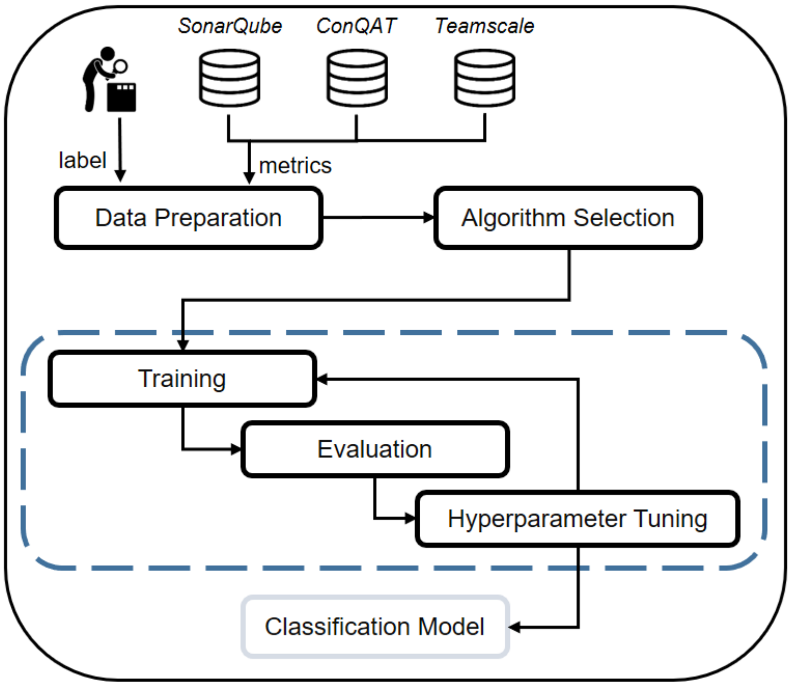


Figure 4.2.: Overall approach (as appeared as *Fig. 1* in [187])

- Are there relations between metrics and expert judgment, and which metrics have the highest predictive power?

4.1.1. Overall Approach

Figure 4.2 depicts the overall framework of our approach. In the data preparation phase, we extracted metrics from the code sample using static analysis tools, performed data cleaning, and combined the metrics and the label. Next, we train and validate the models. We selected a diverse set of 21 algorithms representative of different approaches. For each classifier, the iteration *train* $\rightarrow$ *evaluate* $\rightarrow$ *parameter tuning* continued until all possible parameter combinations were evaluated. At this stage, we also analyzed the predictive power of the features.

4.1.2. Study Objects

To evaluate the approach, a dataset of source code and its evaluation has to be created. We took our sample from three software systems written in Java. The chosen sample includes 115,373 Lines of Code (LoC), distributed over 345 classes. To ensure a high diversity among the study objects, we chose one small project with approx. 45k LoC, one medium-sized system with around 380k LoC, and one large project with more than 3M LoC. The age of the

Table 4.1.: Analyzed Source Code (as appeared as *Table I* in [187])

	System A	**JUnit 4 (4.11)**	**System C**
Domain	*Insurance*	*Software Dev.*	*Insurance*
Purpose	*Offer Management System*	*Testing Framework*	*Damage Evaluation System*
First Release	*2000*	*2014*	*2014*
Development	*Outsourced*	*Open-source*	*In-house*
Size	*3.1M LOC*	*44.6k LOC*	*380k LOC*
Chosen Sample	*65k LOC* *160 Classes*	*10k LOC* *75 Classes*	*41k LOC* *110 Classes*

systems lies between 4 and 19 years. The projects cover in-house, off-shore, and open-source development. Two of the systems are industrial projects located in the insurance domain. The third system is the software testing framework JUnit 4 (Version 4.11). Table 4.1 provides an overview of the systems.

4.1.3. Static Analysis Tools for Data Collection

Static code analysis tools analyze source code without actually executing it. Their measurements serve as input for our experiments. We targeted to use both commercial and free-to-use tools and to integrate both basic measurements as well as complex metrics. Among the various available tools we chose the following three:

- *ConQAT:* The tailorable, open-source framework integrates clone detection and structural assessments [44].
- *Teamscale:* This commercial tool evaluates both structural properties and code style to identify code anomalies. These anomalies are called findings and are automatically categorized according to their severity [84].
- *SonarQube:* The open-source tool offers tailorable quality gates. It also provides aggregated measurements like code smells and potential vulnerabilities or bugs [28].

Examples of the extracted attributes are the following:

- *Size*: Lines of Code, Source Lines of Code, Method Lines of Code, Number of Conditions
- *Structural*: Max. Method Length, Avg. Method Length, Max. Block Depth, Loop Length, Max. Loop Depth
- *Cloning*: Clone Coverage, Clone Units

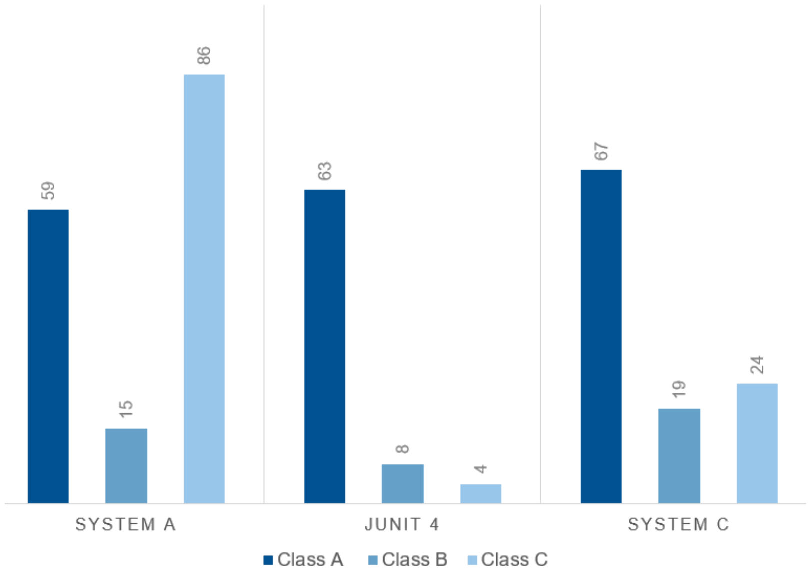


Figure 4.3.: Label distribution per system (as appeared as *Fig. 2* in [187])

- *Complex Measurements:* Cognitive Complexity, Code Smells, Teamscale-Findings (i.e. the number of quality violations identified by Teamscale)

4.1.4. Labeling

In order to learn from our code base, the source code is analyzed and labeled by two researchers and two quality analysts from itestra GmbH. On one hand, it is impossible to evaluate source code without context. On the other hand, we had to draw a line between what to take into account and what to omit from the analysis. Hence, we chose a class-level granularity. The possible classification is threefold: *Label A*, *Label B*, and *Label C*.

- **Label A** indicates the absence of indicators for maintainability problems with respect to the ease of change.
- **Label B** covers classes with some room for improvement.
- **Label C** is assigned to code that is hard to maintain and requires high effort to be changed.

Our experiment aims to capture the experience of professional experts. Therefore, it is imperative to label the data according to that expertise. Although this limits the size of the dataset, we still managed to label 345 classes, representing more than 115k Lines of Java Code that had to be inspected and evaluated. In this context, it is not advisable to automatically label large datasets with, for example, a rule-based script. The ML algorithm would not capture the expert opinion, but would simply reverse engineer the rules used for the automated labeling. During the joint assessment of the study objects, both the quality consultants and the researchers evaluated the source code. The judgment of the researchers was then discussed in joint validation sessions, ensuring the provided label matched the opinion of the experts. The labeling procedure resulted in 182 instances out of 345 (52.75%) being assigned *Label A*. 51 instances (14.78%) were labeled as *Label B* and 112 instances (32.46%) are categorized as *Label C*. The distribution of the labels among the single projects is shown in Figure 4.3.

4.2. Experiment

Though our dataset covers more than 115k Lines of Code, it accounts for just 345 instances. To avoid bias introduced by splitting the 345 data points in fixed training, validation, and test sets, we used 10-fold stratified cross-validation. Since we are using a threefold label and thus face a multiclassification problem, we use accuracy, precision, recall, and F-Score to evaluate the performance of the algorithms as suggested by Sokolova [201]. In addition, we analyzed differences in the performance between *Label A, B*, and *C*.

4.2.1. Prediction Results

Our experiments are implemented using the Waikato Environment for Knowledge Analysis (Weka) [73] Version 3.9.3. Every algorithm was run once in its default configuration before hyperparameter optimization was applied. The results discussed in the remainder of this subsection correspond to the best observed performance of each classifier.

Table 4.2.: Experiment Results (as appeared as *Table II* in [187])

Classifier	**Accuracy**	**Precision**	**Recall**	**F-Score**
J48	0.8087	0.7967	0.8087	0.8009
LMT	0.7971	0.7693	0.7971	0.7757
SimpleLogistic	0.7971	0.7577	0.7971	0.7566
...	...	...	...	...
OneR	0.7102	0.6430	0.7101	0.6540
...	...	...	...	...
Multilayer Perceptron	0.6667	0.6667	0.6667	0.6667
ZeroR	0.5275	n/a	0.5275	n/a

The algorithm with the best results was J48, an algorithm based on decision trees. It was able to classify 279 instances (81%) correctly. It achieved a macro-averaged precision of 79.7% with a recall of 80.9%, combining for an F-Score of 80.1%. The performance of the best classifiers and baseline comparisons are denoted in Table 4.2. The table also shows that J48 outperforms the other classifiers in all four performance measures.

Table 4.3.: Result Differences per Label (as appeared as *Table III* in [187])

	F-Score		
Classifier	*Label A*	*Label B*	*Label C*
J48	0.874	0.449	0.842
LMT	0.859	0.290	0.862
SimpleLogistic	0.860	0.161	0.860

In addition to the performance over the whole dataset, we also investigated performance differences between labels. Table 4.3 denotes the F-Score per class for the three best-performing classifiers. Indeed, a significant drop for files with *Label B* can be observed. J48 only achieves an F-Score of 44.9%, while the performance is even worse for LMT and SimpleLogistic with 29.0% and 16.1%, respectively.

4.2.2. Attribute Evaluation

Given the combination of the tool output and the labels assigned by manual inspection, we analyzed the resulting matrix for the most influential features. We applied six different feature selection algorithms to our data. Table 4.4 lists the results of the algorithms *InfoGain* and *OneR Attribute Evaluation*. For conciseness reasons, we list the results of neither all algorithms nor all 67 attributes. For presentation reasons, we count the number of times a feature was part of the 10 highest-ranked attributes. This number is provided in the right-most column of Table 4.4. Features with less than four votes are omitted from the table. The attribute evaluation identified clone coverage as one of the most predictive features. Also, clone units were selected by five out of six techniques, whereas Teamscale-Findings and the maximum size of a method are selected in four of the six cases. Hence, these characteristics are considered the most influential features.

4.3. Discussion

This experiment uses a threefold label as we think a threefold classification captures the expert understanding of maintainability best. We did not compare the results with other labels such as a twofold label. Binary labels do not reflect the real world, and, even more importantly, do not reflect the way experts perceive quality. For the very same reason, we

Table 4.4.: Most Influential Features (as appeared as *Table IV* in [187])

Attribute	InfoGain Score	OneR AttrEval	Top10 Appearances
Clone Coverage 50NN(i)	*0.3070*	*69.86*	*6*
Clone Units 50NN(i)	*0.2633*	*71.01*	*5*
Teamscale-Findings	*0.2777*	*66.38*	*4*
Max. SLOC per Method	*0.2415*	*64.03*	*4*
Max. LOC per Procedure	*0.2226*	*65.80*	*4*
Max. LOC per Method	*0.2164*	*65.51*	*4*

(i) non-normalized, minimum length of 50 units

decided to use a classification model instead of regression models as implemented in [82]. From our experience, a numerical value does not correspond to the way experts perceive quality. Quality analysts do not target to retrieve a numerical value but aim to develop a general understanding of existing problems.

4.3.1. Interpretation of the Prediction Results

The results presented in Section 4.2 show that the assigned label corresponds to the experts' categorization in 80.87% of the cases. The classifiers J48, LMT, and SimpleLogistic delivered the best results in our experiment. They clearly outperform baseline classifiers such as ZeroR by a large margin. The best-performing algorithm, J48, is based on C4.5, a decision tree algorithm described in detail in [170]. It achieved an accuracy of more than 80% and an F-Score greater than 80% as well. LMT, the second-best performing algorithm, also implements a decision tree. In contrast to J48, LMT uses logistic functions at the leaves [122].

Analyzing the performance of these three algorithms, we observed significant differences between code with *Label A*, *B*, and *C*. As illustrated in Table 4.3, the F-Score for *Label B* just ranged from 16% to 45% while being above 84% for all other classes. We interpret this finding as follows. Our prediction approach is able to identify classes with good quality and classes with bad quality. It performs poorly for mediocre labels. To solidify this interpretation, we analyzed the false positives. Using J48, 7 instances with *Label A* were erroneously classified as *Label C* (4%), 12 instances were mistaken for *B* (7%), and 163 instances (89%) were labeled correctly. In contrast, 10 instances of *Label C* were misclassified as *A* (9%), 6 instances (5%) were classified as *B* and 96 instances (86%) were labeled correctly.

Given these observations, the classification results can be interpreted both optimistically and pessimistically. The goal of industrial software quality assessments is to identify which parts of the system suffer from bad quality. Based on the identified issues, measures are taken whether to rebuild the system, renovate certain components or restructure the development team [188]. The analysis of the false positives shows that the automated approach is not yet suitable to replace the human expert in finding these trouble spots. Not only does it assign wrong labels 19% of the time, but the produced false positives are actually severe. Hard to maintain code was misclassified as easy to maintain in 9% of the cases. Missing that number of potential trouble spots prohibits relying on the classification in critical software assessments. System owners should not derive far-reaching actions based on a classification with just 81% accuracy.

However, we still consider the achieved results the first step towards automated quality analysis. Static analysis tools are not only used for external quality assessments, but also for continuous quality control. Using the tools SonarQube, Teamscale, and ConQAT, one obtains 67 different measurements, making it hard to reason about maintainability at a glance. This work presents a method to aggregate different metrics in a way that is learned from experienced experts. Though it is not comparable with human experts, the automated classification helps developers to identify a great share of the code with maintainability issues.

4.3.2. Interpretation of Attribute Evaluation

As mentioned earlier, static code analysis tools analyze source code and report the measured characteristics. The user must draw conclusions and interpret the metrics based on his experience and expertise. In this research, we created a dataset of source code, the static tool output for this code, and its expert evaluation. The most influential metrics presented in Table 4.4 now allow developers insights into the expert evaluation. Hence, we believe our feature analysis provides valuable guidance for developers which metrics to focus on to predict the expert opinion. The first two metrics to be taken into account are cloning coverage and clone units since they have the highest correlation with the expert judgment. Then Teamscale-Findings should be respected, as well as the maximum method length. This does not mean that all other metrics should be ignored, but this set already offers a good indication of code maintainability.

In the context of this research, maintainability was defined as the ease of change, i.e. a combination of comprehensibility of the code itself, and understandability of which dependencies have to be updated as well. Clone coverage and clone units refer to code duplications. Modification of a code snippet with a duplicate in another place forces the developer to search for the clone and apply the change here as well [98]. With code duplications hence leading to decreased maintainability, it is not surprising that cloning measurements show high predictive power. Interestingly, as opposed to the size of a method, the size of a class is not amongst the most influential features. Teamscale and other static analysis tools automatically rate large classes with more than 750 Source Lines of Code as hard to maintain [40]. We did not apply such fixed thresholds and actually rated every program class manually. Hence, we consider the results of the feature analysis a valuable contribution to research, as it reverse engineers the intuition of the human experts.

4.3.3. Threats to Validity and Future Work

In this study, the maintainability of Java classes was evaluated manually. To mitigate the threat to internal validity, validation sessions were performed to discuss the evaluation of researchers and software quality consultants. Still, the analysts are from just one company. We notice that our dataset consists of only three systems, covers just two domains, and only includes Java code. Also, the used dataset is imbalanced with *Label A* and *Label C* dominating the data distribution.

Furthermore, there is one major limitation to the chosen approach. While inspecting and evaluating the source code, we observed that several negative findings are of semantic nature. While static code analysis tools have their strengths in assessing structural characteristics, they cannot detect semantic flaws. For example, discrepancies between implemented behavior and documentation lead to lower perceived comprehensibility but are not reflected by structural metrics. This highlights the limitations of assessments restricted to code structure and emphasizes that expert-based reviews are needed for a holistic maintainability assessment.

This initial study presents preliminary findings and promising results on using static analysis metrics to classify the maintainability of source code. In the following chapters, we will explore the possibility of using metrics derived from the source code as well as alternative representations of code. Provided that the available data allow it, investigating the influence of inter-class metrics to complement intra-class observations is part of our long-term plan. In the meantime, reducing the number of features and increasing the size of the dataset is our priority in order to reduce the risk of overfitting and increase the reliability of the classification model.

4.4. Conclusion

The goal of this study is to model the experience of professional quality analysts using ML. Therefore, a sample of 115,373 Lines of Code was selected from three study objects, including two industrial systems. In joint work with professional quality analysts, the source code was inspected and evaluated at the class-level. The evaluation is based on the ease of change, i.e. the comprehensibility of the source code, and the ease to understand which external dependencies have to be updated after a change. The participants assigned *Label A*, *B*, or *C* to each file and discussed their assessment with each other. *Label C* indicates the code is hard to maintain, while *Label A* corresponds to the absence of indications for low maintainability. While manual assessments are a well-established method to evaluate the quality of software, many developers use static analysis tools to monitor quality. In this study, we used metrics emitted by such tools to learn and predict expert judgment. For our experiment, we extracted the metrics emitted by three such tools for each of the labeled program classes. We then used these attributes to learn and predict expert judgment. The algorithm J48 achieved an accuracy of 81% across all labeled code files. We consider this approach to be a promising first step toward automated software evaluation. While this performance is sufficient for quick assessments, it is not yet suitable to replace an expert review. In addition, we analyzed the used features and investigated their predictive power. We found that clone coverage and clone units are the most influential features. Teamscale-Findings, i.e. the number of identified quality violations as computed by Teamscale, and the maximum method length also have high predictive power. This result provides guidance which metrics to prioritize for maintainability evaluations, based on the correlation with the expert judgment.

5. Defining a Software Maintainability Dataset

Parts of this chapter have previously appeared in a peer-reviewed publication ([183]) co-authored by the author of this thesis.

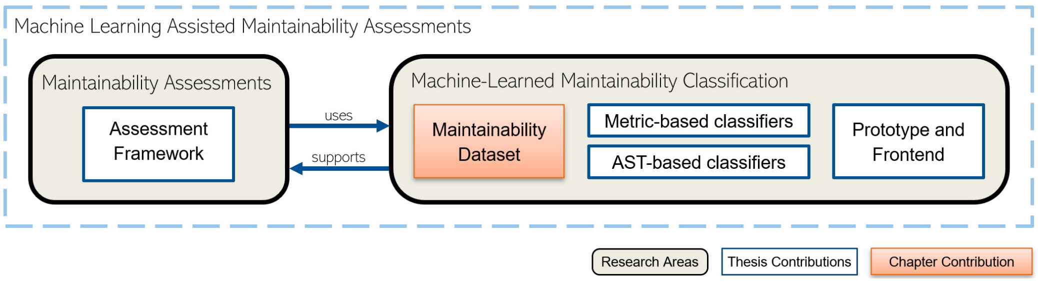


Figure 5.1.: Contribution of this chapter, i.e. the software maintainability dataset, in the context of the thesis

This chapter contributes a software maintainability dataset and describes the methods used to collect the data. Figure 5.1 illustrates the contribution in the context of the proposed framework.

Our initial results to predict expert maintainability ratings showed promising results. However, the need for a well-crafted dataset became evident. We, therefore, built an online labeling platform that allows assessing code and computes a weighted average of the experts' ratings. The assessment covers the overall maintainability judgment as well as several subdimensions of maintainability, including readability, understandability, or complexity. Sophisticated selection techniques were used to keep the dataset both representative and insightful despite the unpredictable number of submitted ratings. Every code file was analyzed by at least three experts. Interestingly, our work revealed that disagreement between analysts occurs frequently and considerably.

Problem: To develop more reliable machine learning models, a large software maintainability dataset is needed. Existing datasets are inapplicable, as they either use the number of revised lines of code or code smells as a proxy for maintainability, contain only a few manually labeled data points, or are biased towards the opinion of a single expert.

Solution: An online labeling platform displays code to the study participants and collects their evaluation on a Likert scale. A novel prioritization algorithm accounts for a realistic distribution of the labeled dataset towards all available data points. Also, it ensures to include outliers, which are particularly interesting in the context of quality analyses. Eventually, the Expectation-Maximization-Algorithm determines the most probable consensus of the experts.

Contribution: In this chapter, we present a large-scale study that creates a robust software maintainability dataset based on expert evaluations. In total, 70 professionals from different companies and universities submitted almost 2000 manual assessments. This results in 519 labeled data points, which are sampled from nine study projects. The consensus ratings and code of the open-source projects are shared with the scientific community. Dissent between the professionals, who rated the same code files, was commonly observed. This verified the need to have several participants evaluate the same code and eventually find a consensus.

Limitations: Although we took a variety of measures to mitigate potential bias, it is impossible to eliminate it entirely. One possible source is the selection and prioritization of code files. The code is sampled from five open-source and four industrial projects but does not cover all possible application domains. Only selected participants were allowed to read and evaluate the code of the commercially developed projects. Eventually, a consensus of the experts was calculated as a weighted average. While consensus building through discussion is preferable, it was not feasible in this case due to the number of participants and their limited time. A natural progression of this work is to use this dataset as ground truth to automatically identify code with problematic maintainability.

5.1. Research Questions

The primary goal of this research is to create a large and robust software maintainability dataset. To maximize the potential of the dataset for future research, this chapter answers the following research questions:

It is known that maintainability is determined by several subaspects. We hypothesize that these subaspects all contribute to the overall evaluation of maintainability, but they do not contribute equally. Thus, we pose our first research question: *Which subdimension of maintainability do experts value the most?* To put the results of this question into context, we extend the focus of the data: *Which other aspects do experts consider to assess the quality of code?*

Next, we investigate dissent and consensus between experts. Based on the fact that quality is inherently subjective, we put forward the hypothesis that expert opinions may differ. Consequently, we investigate: *In which cases do experts disagree and how severe is the dissent?* Assuming that experts may disagree, the need to find a consensus between them emerges. This motivates the next question: *How can we automatically aggregate the opinions of several experts, not knowing up-front which expert is right?* Eventually, our insights

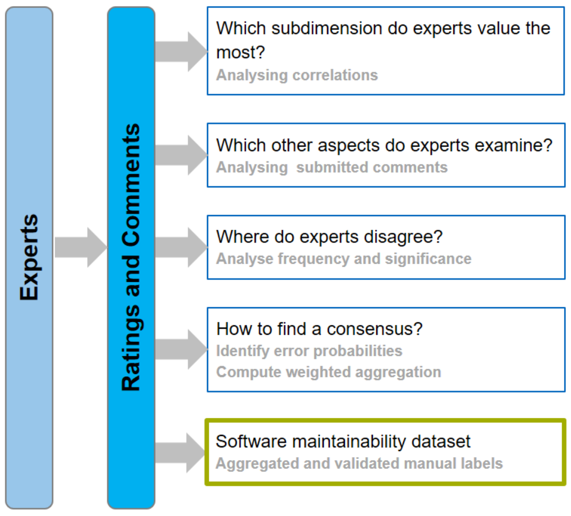

Experts
Ratings and Comments
Which subdimension do experts value the most?
Analysing correlations
Which other aspects do experts examine?
Analysing submitted comments
Where do experts disagree?
Analyse frequency and significance
How to find a consensus?
Identify error probabilities
Compute weighted aggregation
Software maintainability dataset
Aggregated and validated manual labels

Figure 5.2.: Research questions and analyses (as appeared as *Fig. 1* in [183]).

are used to construct a useful and robust software maintainability dataset. The findings from this study make several contributions to the current literature: (1) we introduce a dataset for future use in the maintainability research community; (2) an adaption of an algorithm for automatically finding consensus between experts; (3) several insights into expert judgments of maintainability, including the diversity of opinions and weighting of subcharacteristics; and (4) empirical evidence on what makes code less maintainable, including commented-out code, hardcoded values and quality of the comments.

5.2. Study Design

This section introduces the study objects, i.e. the analyzed source code, the recruited experts, and which labels they will assign. As Figure 5.2 shows, our research questions and analyses are based on the ratings and comments provided by experts. An insightful dataset is necessary to answer the research questions. This dataset should consist of transparent labels provided by qualified experts. Also, the systems should cover diverse domains to mitigate domain-specific bias. To foster the practical relevance, the systems should be written in a modern and often used programming language. Though open-source and closed-source software share common characteristics, they also differ in some aspects [158]. To achieve better generalizability of the results, the dataset should contain projects of both types.

5.2.1. Study Objects

We took code snippets from nine different projects. These projects are both open-source projects and commercially developed products. All chosen projects are developed in Java.

The sample covers a diverse range of domains and release dates. We believe this to reflect real-world software systems where quality reviews are applied.

The commercial systems are developed by two different vendors and are taken from two separate software ecosystems. Therefore, we consider these nine projects to portray a corpus with high representativity. However, the study objects do not cover every possible domain and only the Java programming language. We decided to stick with one programming language only to avoid context switching during the labeling process. Java is considered one of the most popular programming languages and there is a high demand for Java in industry[1]. Hence, it is a reasonable choice. We selected the following projects as sources for the code snippets:

- Open-source projects:
  - **ArgoUML[2]:** Tool to design, simulate and generate code from UML diagrams
  - **Art of Illusion[3]:** 3D Modeling and Rendering Studio
  - **Diary Management[4]:** Multi-user calendar tool
  - **JUnit 4[5]:** Testing framework for Java programs
  - **JSweet[6]:** Transpiler to convert Java code into Javascript or Typescript code
- Commercial projects, anonymized:
  - ***xApp*:** App used by insurance damage assessors
  - ***xBackend*:** Backend of an insurance system
  - ***xDispatch*:** Planning of personnel dispatch
  - ***xPrinting*:** Printing and layouting of documents

In total, these systems consist of 15,714 Java files and account for 1.4 million source lines of code (SLOC), i.e. lines of code without comments and blank lines. Table 5.1 lists more details about the single projects including the release year. Since Java is a rapidly evolving programming language, we wanted to include code that does use newer features of Java as well as older code that sticks to basic features. Furthermore, we deliberately did not use the latest release for every project, e.g. ArgoUML. This enables future research to compare the maintainability, i.e. the maintenance effort predicted by the experts, to the actually required effort.

Each data point in our dataset represents one *.java* file, which - following the Java conventions - is equivalent to one programming class. Too short code snippets might not

[1]`https://cce.fortiss.org/trends/radar/compare?technologies=java`
[2]`https://github.com/argouml-tigris-org/argouml`
[3]`http://www.artofillusion.org/`
[4]`https://sourceforge.net/projects/diarymanagement/`
[5]`https://junit.org/junit4/`
[6]`http://www.jsweet.org/`

Table 5.1.: Included Sample Projects (as appeared as *Table I* in [183])

Project	**Domain**	**Release**	**Size in SLOC**	**Files**
ArgoUML	UML Models	2010	177 k	1,904
Art of Illusion	3D Modeling	2013	118 k	470
Diary Management	Calendar	2014	17 k	131
JSweet	Code Transpiler	2019	79 k	1,933
JUnit 4	Software Testing	2014	25 k	383
xApp	Insurance	2018	28 k	223
xBackend	Insurance	2018	82 k	637
xDispatch	Scheduling	2019	472 k	5,192
xPrinting	Printing	2019	435 k	4,841
In total:			**1.43 M**	**15,714**

provide enough context for the analyst to reason about e.g. complexity or understandability. Analyzing larger chunks of software such as packages, on the other hand, is very time-consuming and not suitable to collect many evaluations. A class-level analysis is a reasonable compromise between displaying context and the time needed to assess the data. Most static analysis tools also support class-level analysis, and this granularity is chosen by related work as well [2, 17, 121, 126, 187, 228].

5.2.2. Study Participants

The target group of our study are software quality analysts, researchers with a background in software quality, and software engineers that are involved with maintaining software. Some participants have up to 15 years of experience in quality assessments. In sum, 70 professionals and researchers participated. First, we invited selected experts to participate in the study. Second, we asked them to disseminate the study to interested and qualified colleagues. The survey was also promoted in relevant networks. The participants are affiliated with companies including Airbus, Audi, BMW, Boston Consulting Group, Celonis, cesdo Software Quality GmbH, CQSE GmbH, Facebook, fortiss, itestra GmbH, Kinexon GmbH, MaibornWolff GmbH, Munich Re, Oracle, and three universities. However, 7 participants did not want to share their affiliation.

5.2.3. Label Definition

The label represents the perceived quality of a code snippet. To be relatable, it should be as objective as possible. This is a non-trivial task given that the perception of quality may vary. We are meeting this challenge with two measures: finding the consensus of several opinions and limiting the possible perspectives under which quality is assessed. Consequently, we have to explicitly define which quality attributes the experts should focus on during the survey.

This study focuses on maintainability. Maintenance costs are the biggest contributor to the economic effectiveness of software systems [164], thus making maintainability a relevant research area. Also, expert assessments are particularly valuable here as maintainability evaluations cannot be obtained automatically. In the context of this work, we understand maintainability as the estimated future maintenance effort of a program class. This effort is influenced by several factors. During quality assessments, analysts investigate different aspects of quality. To give a final assessment, they negotiate with themselves a weighting of the observed dimensions. In our study, we reproduce this thought process. The study participant first evaluates the code with regard to various more fine-grained criteria before giving a final statement on maintainability. To record these ratings, a labeling platform was implemented. Section 5.3.1 describes it in detail.

In favor of a faster labeling process, we limited the survey to five dimensions: readability, understandability, complexity, adequate size, and overall maintainability. The more dimensions are to be considered, the more effort is needed for the evaluation. The same argument holds for inter-class characteristics and architectural context. Analyzing these aspects is prohibitively time-intensive and negatively impacts the amount of data that can be labeled in a given time. Besides, a more tedious and complex labeling process might decrease participant motivation.

It is unlikely that any set of questions can cover the entire spectrum of software maintainability. Therefore, we dedicate one question to the overall judgment. This means that all subaspects that have not been covered by an explicit question can still influence an expert's overall evaluation. In the following, we explain the chosen dimensions and reasons to select them: One key activity in software maintenance is reading the existing source code [172]. Readability captures how easy it is to syntactically parse the code. It is concerned with e.g. indentation, line length, and identifier length. After reading the code, all maintenance activities require maintainers to comprehend the semantics of the code they are going to adapt [215]. Therefore, we are interested in assessing the understandability of code. This attribute captures the effort to understand which concepts the code implements and to easily identify at which point a desired change must be made. This concept is known to affect maintainability [12]. Understandability and readability are related but need to be treated separately [166]. The distinction becomes clearer if we consider very short variable names of just one character. Those are easily readable but might not yield enough information to comprehend their meaning. In the next dimension, we attempt to measure the complexity of a particular piece of code. This is especially interesting, since most other approaches like e.g. McCabe's cyclomatic complexity [140], try to formalize complexity. In contrast, we explicitly refrain from that and treat complexity as a concept-by-intuition. The last considered criterion is the adequate size and intra-class modularity of a program class. In our study, participants can express whether they think code should be split up into smaller snippets. This applies to both the program class itself as well as its methods. The fifth label is the overall maintainability of the code snippet according to the personal intuition. In the further course of the thesis, we will mainly refer to this label.

For each dimension, we try to capture the expert opinion as directly as possible. Consequently, we refrain from formal definitions for any evaluated quality attribute and model them as concepts-by-intuition. This term describes complex concepts for which the human intuition can be elicited with a single question without having to break the concept down into sub-aspects [181]. One way to capture opinions about concepts-by-intuition is to pose statements and ask the participants whether they agree or disagree [181]. We advocate a four-part Likert-scale ranging from *strongly agree* to *strongly disagree*. From our point of view, binary values do not reflect the way experts perceive quality. In contrast, too many different values to choose from might overwhelm the experts and borders between the single categories will blur. Besides, a four-part scale deprives participants of the possibility of choosing a neutral position. Experts are forced to at least indicate a tendency. Some studies [63] suggest asking study participants how confident they are that their answer is correct. This is rendered obsolete by the scale chosen, as the experts can express their confidence with a clearer label on the four-part scale.

5.2.4. Availability of the Dataset

The interest in automating software quality evaluations is growing. Using publicly available data fosters the reproducibility of studies and enables fair comparisons. The results of this study, i.e. the analyzed open-source code and the corresponding labels, are shared in [184]. Please note that we only distribute the code of the non-proprietary systems. The archive also contains a summary of the dataset's threats to validity.

5.3. Data Collection

Before we can answer the research questions, we have to create a meaningful dataset. Such a dataset should consist of representative data points and reliable labels. One of the first things to consider is which projects to choose the samples from and which experts to recruit for the evaluations. We elaborated on that in Section 5.2, whereas this section focuses on how exactly we collected the data. The interaction of all data collection activities is visualized in Figure 5.3.

The first step is to extract code snippets from the study projects and define in which order

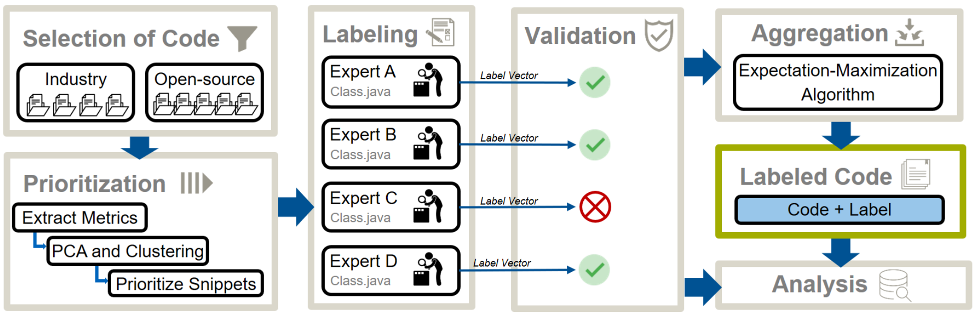


Figure 5.3.: Activities during the dataset creation: selection of code, prioritization of samples, labeling, validation, aggregation, and analysis (as appeared as *Fig. 2* in [183]).

the snippets will be analyzed. The study participants then evaluate the code and assign labels. Every code snippet is rated by several experts. Afterward, the collected labels are validated. Eventually, we apply an aggregation step. In this step, error probabilities for each participant are computed and their consensus is determined.

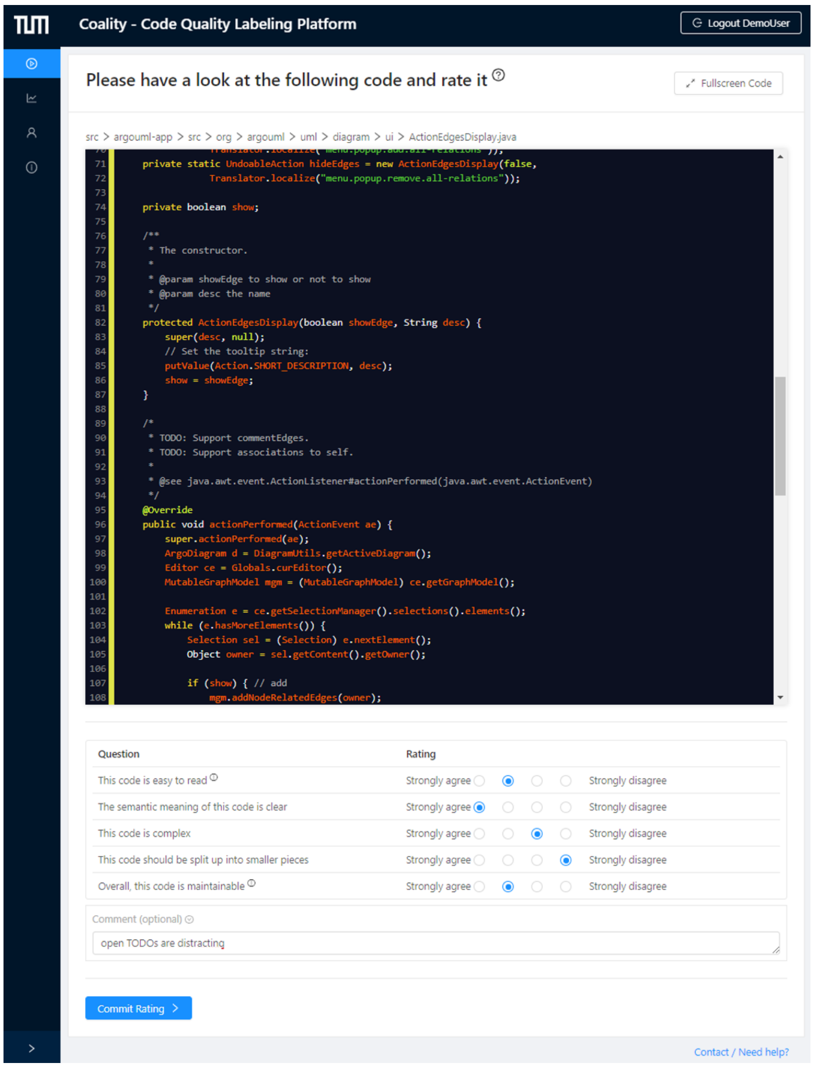
5.3.1. Labeling Platform

The labels are collected per an easy-to-use online tool. Since the study participants are volunteers, it is crucial to keep them motivated. The code evaluation platform provides a modern and intuitive frontend. No training period is needed. Every participant owns a password-protected account. This enables a sophisticated permission concept. The code of the commercial systems is restricted to employees of that company and to users owning explicit rights to inspect the code. Open-source code can be rated by every participant. After logging in, a code snippet is presented to the user. To read the code as conveniently as possible, users can select their favorite syntax highlighting theme.

We ask the participants to rate each code snippet in five dimensions: readability, understandability, complexity, adequate size, and overall maintainability. For every dimension, we post one statement and the analysts express their opinion on a four-part Likert-scale:

- *This code is easy to read*
- *The semantic meaning of this code is clear*
- *This code is complex*
- *This code should be split up into smaller pieces*
- *Overall, this code is maintainable*

Besides, the platform allows to add free text comments as well.



Please have a look at the following code and rate it

src > argouml-app > src > org > argouml > uml > diagram > ui > ActionEdgesDisplay.java

```java
    private static UndoableAction hideEdges = new ActionEdgesDisplay(false,
            Translator.localize("menu.popup.remove.all-relations"));

    private boolean show;

    /**
     * The constructor.
     *
     * @param showEdge to show or not to show
     * @param desc the name
     */
    protected ActionEdgesDisplay(boolean showEdge, String desc) {
        super(desc, null);
        // Set the tooltip string:
        putValue(Action.SHORT_DESCRIPTION, desc);
        show = showEdge;
    }

    /*
     * TODO: Support commentEdges.
     * TODO: Support associations to self.
     *
     * @see java.awt.event.ActionListener#actionPerformed(java.awt.event.ActionEvent)
     */
    @Override
    public void actionPerformed(ActionEvent ae) {
        super.actionPerformed(ae);
        ArgoDiagram d = DiagramUtils.getActiveDiagram();
        Editor ce = Globals.curEditor();
        MutableGraphModel mgm = (MutableGraphModel) ce.getGraphModel();

        Enumeration e = ce.getSelectionManager().selections().elements();
        while (e.hasMoreElements()) {
            Selection sel = (Selection) e.nextElement();
            Object owner = sel.getContent().getOwner();

            if (show) { // add
                mgm.addNodeRelatedEdges(owner);
```

Question	Rating					
This code is easy to read	Strongly agree	○	●	○	○	Strongly disagree
The semantic meaning of this code is clear	Strongly agree	●	○	○	○	Strongly disagree
This code is complex	Strongly agree	○	○	●	○	Strongly disagree
This code should be split up into smaller pieces	Strongly agree	○	○	○	●	Strongly disagree
Overall, this code is maintainable	Strongly agree	○	●	○	○	Strongly disagree

Comment (optional)

open TODOs are distracting

Commit Rating

Figure 5.4.: Screenshot of the labeling platform showing code and questionnaire (as appeared as *Fig. 3.* in [183]).

Figure 5.4 shows a screenshot of the survey tool. To foster participation and keep the users motivated, the study implements gamification elements. For example, confetti animations and motivational messages are displayed once the user has committed a certain amount of labels. Ambitious users can compare their performance with others on a public scoreboard. To follow data privacy rules, participants must explicitly opt-in to appear on this scoreboard.

5.3.2. Prioritization of Samples

To make efficient use of the experts' time, the snippets are labeled in a specific order. First, the extracted metrics are used to cluster the code. Then, we prioritize them by iterating through these clusters. The next code snippet to be labeled is the snippet with the highest priority, from a codebase to which the user has access rights, and that did not yet receive enough valid ratings.

The combined size of the sample projects exceeds 1.4 million source lines of code and contains more than 15,000 files. The participation happens voluntarily, thus making it illusional to aim for a dataset of 15,000 manually evaluated classes. Paying study participants is beyond the possibilities of this study due to the required expert knowledge. Since we cannot reason about the time the experts will dedicate to labeling, it is not possible to determine a feasible sample size apriori. Hence, we apply prioritization instead of sampling.

From our experience, quality reviews target to identify spots that are worth further investigation. A useful dataset must hence include such cases. However, focusing too much on suspicious data points would jeopardize the representative nature of the dataset.

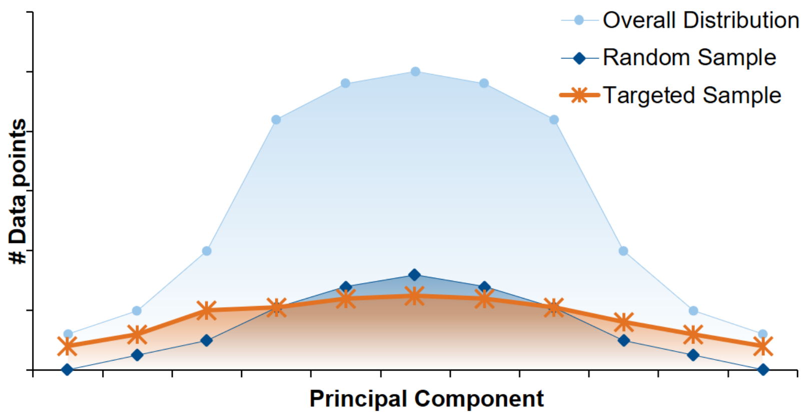


Figure 5.5.: The distribution of the targeted sample in comparison to a random sample (as appeared as *Fig. 4* in [183]).

Figure 5.5 illustrates sample distributions plotted towards an arbitrary dimension. The total distribution, which is approx. normally distributed, is visualized as the bell-shaped light blue curve. A random sample of, e.g., 25%, would probably correspond to the dark blue curve. In the diagram, one can see that this sample would hardly contain any edge cases, neither on the small nor on the large end of the axis. From the quality assessments described in Chapter 3, we know that classes with extraordinary values provide the most insights to the analysts. Therefore, we target a distribution that resembles the orange curve. This curve has two main characteristics: First, it contains significantly more edge cases than the random sample. Second, it still roughly resembles the overall distribution and does still contain many non-edge instances.

Table 5.2.: Illustration of the Prioritization Algorithm (as appeared as *Table II* in [183])

Cluster	**# Files**	**Priorities assigned to files in this cluster**	**# Labeled files**	**# Labeled files if random**
Cluster 0	10	1; 7; 21; 35;…	4	1
Cluster 1	46	2; 8; 22; 36;…	4	4
Cluster 2	55	3; 9-10; 23-24; 37-38;…	7	6
Cluster 3	68	4; 11-12; 25-26; 39-40;…	7	7
Cluster 4	109	5; 13-15; 27-29; 41-43;…	10	11
Cluster 5	182	6; 16-20; 30-34; 44-47;…	15	18

The theoretical foundation of our approach is as follows: Unsupervised ML, i.e. clustering, groups data points together that are more similar than others. The clustering algorithm uses static software metrics as characteristics to define the clusters. Code metrics were chosen as our pre-experiment showed they are a useful asset to predict maintainability in our pre-experiment. In sum, we collected 162 code metrics for each data point. These include structural metrics concerned with nesting or size as well as more complex metrics like the number of code smells. The reason for the variety and high number of metrics is that we do not know—at this point—which metrics have the strongest predictive power towards maintainability. Therefore, we must avoid arbitrarily limiting ourselves to a particular metrics suite. A list of the used metrics and tools is published with the final dataset in [184]. Values are normalized and Principal Component Analysis avoids that clusters are distorted because of metrics measuring related properties. Then, *k-Means* is run on the principal components with *k* determined by the Expectation-Maximization Algorithm [48]. Some data points are more insightful than others because they somehow differ from the average sample. We hypothesize that those interesting cases can most probably be found in clusters with only a few other members. Consequently, we prioritize the data points starting with the smallest clusters. But, taking only the potentially interesting files would lead to a dataset that does not reflect the generality. Therefore, we only add a relative share of $max(1, 0.03 * sizeofcluster)$ points of each cluster to the prioritization queue. To ensure that at least one sample of every cluster is labeled, we perform a start-up round. Here, we label exactly one class from every cluster.

We illustrate the effect of this using the Art of Illusion sample project. There were six clusters identified in this project. One of the clusters contains only 10 files; we refer to this as *Cluster0*. Let us assume that the experts manage to label 10 percent of the files in this project, i.e. 47 files. If we chose randomly which files to label, the probability that at least one of the files from *Cluster0* is among the labeled files is $1 - \frac{\binom{47}{0}\binom{460}{47}}{\binom{470}{47}} = 65.5\%$. With our approach, the probability to have at least one file from *Cluster0* in the sample is 100%. In fact, there will be exactly four files from *Cluster0* in the sample.

Table 5.2 illustrates the algorithm. In the start-up round, one class from every cluster is labeled, starting with the smallest cluster, i.e. *Cluster0*. In the next iteration, we select one sample from *Cluster0*, one from *Cluster1*, two from *Cluster3* and *Cluster4* resp., three from *Cluster4*, and five from *Cluster5*. This is performed iteratively until all files are prioritized. The number of labeled data points per cluster is denoted in the third column of Table 5.2. The right-most column shows how the distribution would look like on average had we used random prioritization. It becomes obvious that our approach shifts the focus from the large clusters to the smaller clusters, where the most interesting files are located. Simultaneously, the overall distribution of data points is respected.

5.3.3. Validation and Seriousness Checks

Every assessment is validated before it is included in our dataset. The target group of our study are professional software analysts, developers, and researchers. But since the labeling platform is publicly accessible, the motivation of a participant in general remains unknown and every participant needs to be treated agnostically. Analyzing the data for inconsistencies can help to identify unserious participation [174]. In our study, we take the combination of quality dimensions into account and compare the values to known illicit combinations. E.g. if a participant states the code snippet was easily maintainable, while at the same time expresses it was neither readable nor understandable and highly complex, that rating is considered implausible. In that case, the plausibility score $p \in [0; 1]$ is set to 0, and 1 otherwise.

The time needed to complete a task can be used to filter out noise by spammers or unserious contributors as well [89, 104, 136]. The labeling platform measures the time taken to create a rating. The number of characters an expert assesses per second arguably follows a normal distribution. We assume the faster an evaluation was finished, the more implausible it becomes. If a rating was created significantly faster than the average, i.e. more than two standard deviations faster than the average, this rating is discarded. If it has been created only slightly faster than the mean, we assign a linearly interpolated credibility score $c \in [0; 1]$. There is no punishment if the assessment took longer than the average.

5.3.4. Aggregation of Ratings

Quality is a viewpoint-dependent characteristic. As we will present in Section 5.4, we found significant differences between the expert assessments. To avoid biasing the results towards the opinion of one expert, we aim to find a consensus between them. Reaching a consensus usually involves convincing others with arguments. But we refrain from finding a consensus per discussion as proposed by Rosqvist [177]. To save time, we target a fully automatable process. In this work, we use the term consensus to refer to the result of a weighted vote. Since the study participants have varying backgrounds and experiences, it appears natural to not treat every submission equally. Instead, we use an aggregation that assigns weights to each rating. The result of the aggregation is then considered the

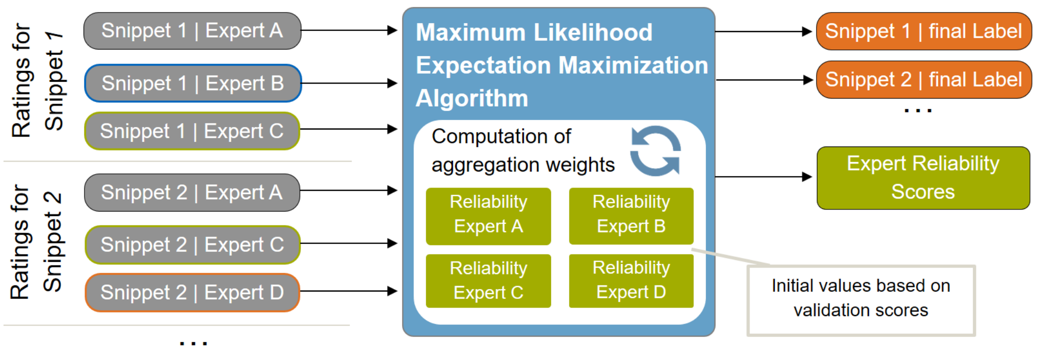


Figure 5.6.: The rating aggregation approach (as appeared as *Fig. 5.* in [183]).

consensus. The Maximum Likelihood Expectation Maximization Algorithm (EM algorithm) is a statistical approach to determine such unknown weights iteratively [48]. Dawid and Skene [43] popularized this algorithm for problems where different observers may report different interpretations of the same yet unknown classification. They use the example of several clinicians diagnosing a patient, which transfers directly to our study where analysts diagnose a code snippet. In both cases the true label is unknown. The algorithm jointly maximizes the likelihood of experts' error rates, i.e., their reliability, and calculates the most probable classification of the code or patient, respectively. This way, the algorithm determines the consensus of the evaluators. The error rates are stored and later on reported to the users as part of a gamification approach. An overview of the approach is visualized in Figure 5.6.
The quality of the aggregation result depends on the chosen starting values of the reliabilities [43, 48]. The initial weights incorporate knowledge from the validation steps described in Section 5.3.3. The product of the plausibility score p and credibility score c forms the first estimate of the weight.

5.4. Results

In total, 70 experts participated in this study and submitted 1976 ratings. Eventually, the labeled dataset consists of 519 distinct code snippets. Here, we only consider code that received three valid ratings.

5.4.1. Influences on the Perceived Overall Maintainability

Strong evidence was found that the understandability of code is valued more than its readability, complexity, or adequate size. In our study, every submitted evaluation comprises an overall maintainability label, a label for each of the four subdimensions, and an optional comment. Every label is a rating on a four-part Likert-scale.

The correlation between the perceived maintainability and its subaspects is tested using Pearson's Correlation Coefficient [159]. For this analysis, we convert every label to an integer value between 1 and 4. The highest coefficient was observed for understandability (0.80), followed by adequate size (-0.74), complexity (-0.73) and readability (0.72). The questionnaire uses an inverted scale for adequate size and complexity, thus their correlation is negative. Notably, all scores lie in a similar range. To refine the results, we also apply the Relief algorithm modified for regression [110, 176]. This approach confirms understandability (0.18) as the most influential subaspect. Readability (0.11), adequate size (0.07) and complexity (0.04) seem to be less expressive predictors. If we treat the labels as nominal values instead of ordinal ones, the results are still valid. We analyzed the influence of the subdimensions using information gain [208], information gain ratio [208], and the performance of a simple One Rule classifier using only that single feature [87]. All three approaches confirm the previous result and rank understandability higher than the other three attributes.

The labeling platform provides the opportunity to submit free text comments about the assessed code. This option was used 198 times. A majority of participants used it to summarize the evaluated code snippet or their assigned labels. Other prominent topics were content or domain-specific remarks arguing about the specific implementation of a function. Inconsistencies within the code snippet and hardcoded values were perceived to hinder maintenance. Concerns regarding the violation of coding conventions were also widespread. Another recurring theme were comments in the evaluated code. Missing comments in places where the code is difficult to understand without further explanation was often perceived as negative. Code in comments, bad quality of comments, and *'todo'* or *'fixme'* annotations were found irritating, too. Furthermore, some participants felt distracted by long copyright statements.

5.4.2. Dissent and Consensus between Experts

We find that disagreement between experts happens often and significantly. In our study, every code snippet was assessed by at least three experts. There exist 2872 distinct rating pairs, i.e. pairs of ratings for the same code by different participants. In total, we find disagreement in 2107 of these pairs (73.4%). This expresses that the experts disagreed in at least one observed aspect. To distinguish between negligible and significant dissent, we calculate the sum of the differences. Please note that there were five questions asked. A cumulative deviation greater than five therefore means that the individual ratings deviate on average by more than one point in each question. We observe such significant differences in 493 cases (17.2%). In 36 cases (1.2%), the difference was even greater than ten. This threshold corresponds to an average deviation greater than two in each question. In one actual example, expert *A* assigned the ratings *[1, 1, 3, 4, 1]*, while expert *B* rated almost completely the opposite: *[4, 3, 1, 1, 4]*. This example will be examined further in the discussion. Similar dissent can be found in many instances. One snippet, e.g., was evaluated by four experts. Two experts agree in pairs, but their opinion contradicts that of the other pair. Not only does the evaluation of the single aspects vary, but we also identify differences regarding the final judgment. Experts assign identical ratings for the subdimensions but differ in the overall judgment. We noticed this in 24 cases.

5.4.3. Analysis of the Computed Consensus Probabilities

To overcome the observed disagreement, a consensus should be reached. In this study, the participants were not available for discussions, therefore, the consensus has to be found differently. We apply the EM algorithm to aggregate the opinions of several participants. An in-detail description can be found in Section 5.3.4. The algorithm computes the probability for each label to be correct. Eventually, the label with the highest probability can be considered the final label. To shed light on the confidence of this calculated consensus, we analyze how large these probabilities are. Figure 5.7 illustrates the values for the overall maintainability dimension as a box plot.

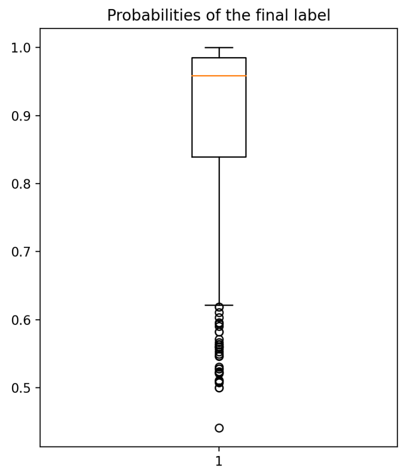


Figure 5.7.: Boxplot of the probabilities for 'overall maintainability'

For 339 of the 519 labeled code files (65%), the probability for the final overall maintainability label was greater or equal to 90%. For 451 data points (87%), the probability was above or equal to 70%. On average, the final label is assigned with a probability of 89% to be the true label for this data point. At 95%, the median is even higher. The lowest confidence is observed at 44%. This is the only instance with the label probability being below 50%. For this Java class, the EM algorithm provides the following probability vector: $[0.006, 0.343, 0.441, 0.209]$. Compared to the other data points, the probability is more evenly distributed among three classes. Nevertheless, the probability for the third label to be true is still 10 percentage points higher than the probability for the second label. In summary, we can observe the EM algorithm has converged to final ratings with high confidence.

5.4.4. The Software Maintainability Dataset

Table 5.3.: Distribution of Label 'Overall Maintainability' per Project (as appeared as *Table III* in [183])

Project	Strongly Agree	Weakly Agree	Weakly Disagree	Strongly Disagree
ArgoUML	34	25	11	4
Art of Illusion	10	20	25	18
Diary Management	7	2	2	0
JSweet	63	5	2	3
JUnit 4	60	12	1	0
xApp	21	10	3	1
xBackend	18	11	5	1
xDispatch	32	24	10	7
xPrinting	31	21	14	6
Across Projects	**276**	**130**	**73**	**40**

For the remainder of this section, we interpret the class with the highest probability as the final label. Table 5.3 shows the distribution of the final labels for the overall maintainability of every study project. The last row of the table summarizes the distribution across all projects. From this data, it can be seen that positive assessments outweigh negative evaluations. In total, 78% of the examined files are reviewed as positive or very positive. Only 22% of the classes are considered hard or very hard to maintain. In 8 out of 9 projects the majority of the assessed classes are perceived as easily maintainable. Then, the number of classes in each category diminishes with decreasing maintainability. The Art of Illusion project is the only exception to this observation. Here, the majority of the files are considered difficult to maintain.

Figure 5.8 visualizes the overall distribution across all projects. These pie charts also show the distribution across all open-source projects in comparison to the closed-source projects. In total, the dataset includes 304 open-source classes (59%) and 215 classes from closed-source systems (41%). The evaluated open-source code and its labels are shared in [184].

5.5. Discussion

In this section, we will discuss how the created dataset can be utilized in future research, the results of the empirical analyses presented in Section 5.4, and the applied data collection

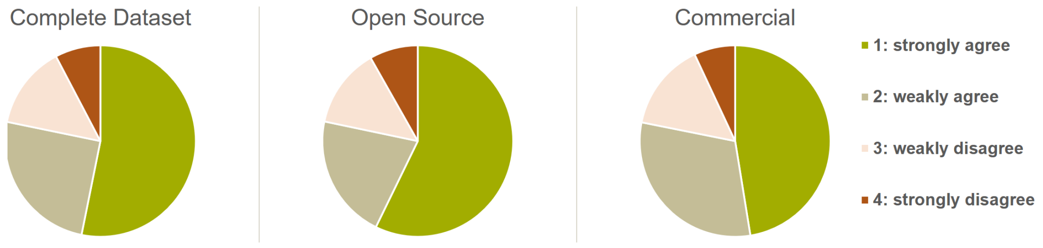


Figure 5.8.: Distribution of the label 'overall maintainability' per project type (as appeared as *Fig. 6* in [183]).

technique.

5.5.1. Discussing the Results

RQ1: Our study showed that understandability has the highest impact on the overall maintainability of software. This finding is of immediate interest to software developers since understandability is an actionable characteristic. Understandability refers to the ease to identify concepts behind the code and identify where a certain concept is implemented. Developers can actively aim to improve this attribute of their code and thus improve its maintainability.

Our results do not justify completely neglecting any of the examined characteristics. All examined aspects show a correlation with maintainability and can thus be considered reasonable subcharacteristics. However, trade-offs should be in favor of understandability.

Finally, it is worth repeating that this survey focuses on intra-class characteristics. This means only aspects observable in the source code of the Java class were respected during the labeling process. Consequently, other aspects may gain importance when additionally considering the relationships between classes.

RQ2: Missing necessary comments or comments of bad quality were found to decrease the maintainability. This confirms that comments are important to comprehend code [171]. Also, it supports our finding that understandability is the most important quality characteristic. The negative statements about commented-out code support this, too. There are several reasons why code is stored in comments and not deleted [225]. However, the participants are afraid this code has to be analyzed thoroughly to figure out why it has not been deleted. Since the code was taken from released products, *'todo'* and *'fixme'* annotations were perceived negatively for similar reasons. Here, the maintainer must analyze whether the issue was resolved or the code was released with incomplete features.

RQ3: Although the participants of this study are qualified software analysts, engineers, or researchers and are affiliated with renowned companies, their judgment is not unanimous. We found the dissent within a group does not deviate much from the dissent within all participants. However, we found exceptions when focusing on *significant* dissent. Within one company, we observe significant dissent in only 9.9% of the cases, while across groups it occurs in 17.2%. Other user groups do not show such deviations. One explanation could be that dissent is mostly caused by personal preferences. Another explanation might be bias introduced by various domains. A software engineer who is mostly concerned with automotive software might evaluate code from the insurance domain differently than an engineer who is familiar with the topic. To make reliable statements about the influence of the domains, we would need a larger amount of ratings per snippet. Otherwise, it is hard to reason that differences occurred due to the domain and not by personal preferences.

To overcome the observed dissent, a consensus has to be found. One instance, for example, was rated almost opposite by some experts. The snippet contains both complex and simple methods. Variable names are not self-explanatory, but the code is well documented. In fact, one can find arguments for positive and negative statements in every rating dimension. This instance exemplarily shows why it is important to investigate expert evaluations in depth.

RQ4: The EM algorithm dynamically aggregates the experts' ratings. Using predefined weights is not possible since we do not know upfront which experts are more reliable than others. We refrained from using the job description and experience of a participant here for three reasons: First, users need to input this information themselves and there is no possibility to verify their statement. Second, it is not trivial to define how, for example, five years of experience as a developer should count against three years as an analyst. Third, the approach suffers from the assumption that people gain expertise through age only. While discussing the idea with industrial partners, we learned that long years of experience are not necessarily an indicator of higher reliability. Therefore, the initial reliability score does not incorporate this information.

The result of the aggregation is a probability vector for each label to be the correct label. In most cases, the EM algorithm converged to clear results: Considering the overall maintainability, the most probable label can be assigned with a confidence of 70 percent or higher for 87% of the data points. These EM probabilities are calculated based on the ratings provided by the survey participants. They must therefore be interpreted with caution. If an analyst had voted differently, or another expert had participated instead, the values would be different. Later in this thesis, we will use both the label with the highest probability as well as the reported probabilities to predict the maintainability of source code.

5.5.2. Discussing the Dataset and Its Usage in Future Work

The created dataset can be used to design tools that predict software maintainability. Furthermore, our dataset contains data about the perceived understandability, readability, adequate size, and complexity as well. Instead of aiming at maintainability as such, future work can also focus on the automatic assessment of these attributes.

One of the most interesting characteristics of the created maintainability dataset is its unbalanced distribution. The majority of the files were evaluated to be easily maintainable, while only a few files were actually negatively perceived. This is not an ideal basis for building classifiers. However, subsequent trimming of the data to a balanced form would lead to too few data points to make reliable statements. Furthermore, that trimmed dataset would not be representative anymore. Nevertheless, the reported label distribution has implications on the design of approaches to automate assessments. In our sample, only 22% of the Java classes are considered problematic. In software quality assessments, it is important to precisely identify hotspots of bad quality. The challenge now is to fabricate tools that identify these rare but important instances reliably. The evaluation of tools should therefore give special thought to the classification of these cases instead of aiming for high overall precision and recall.

Although the label distribution is similar in 8 of the 9 analyzed systems, it deviates considerably in the Art of Illusion project. This emphasizes the importance to consider data from several projects to base tool development on. Additional studies can add more labels from further projects and improve the generalizability of the results. Another progression of this work is to compare the expert evaluations to the observed maintenance effort. This work focuses on the maintainability of software. Further studies focusing on other dimensions can contribute to a holistic software quality dataset.

5.5.3. Discussing the Methodology and Threats to Validity

To fully exploit the potential of prediction algorithms, a well-researched dataset is needed. All studies reviewed so far, however, suffer from at least one of the following drawbacks: They consider only obsolete programming languages [126], use a formal definition as the ground truth [144], keep the data confidential and hinder replicability of the results [165, 187], or lack generalizability due to a small number of sample projects and experts [165, 187]. To mitigate the effect of biased evaluators, we collaborated with 70 participants from 17 different companies. However, only selected participants were allowed to inspect commercially developed software. To avoid domain or project-specific bias, we included code from nine projects, both open and closed-source. Still, the analyzed code is only written in Java. Limiting the study to one language eliminates the need to frequently accustom to new contexts. We chose Java because of its high relevance in industry.

The prioritization impacts which classes are labeled first and are thus part of the dataset. This is necessary because of the size of the corpus. Given enough time, the prioritization would not affect the created dataset at all. The algorithm is based on code metrics. Their correlation with quality is known from our initial experiment in Chapter 4 and related works [37]. Using a broad variety of measurements avoids bias towards specific metrics. However, the problem to predict maintainability from metrics is not yet solved. To build useful tools, the dataset has to contain all typical constellations of metrics. The clustering groups data with similar constellations together. Iterating through these clusters ensures that representatives of all typical constellations are selected.

The labeling platform presents code snippets to experts who evaluate the code. The granularity of Java classes was chosen to keep the labeling effort feasible. This implies that only intra-class characteristics can be evaluated. We are aware that major aspects of software quality are inter-class attributes such as cloning and coupling. These are not characteristics of one isolated code snippet, but a class and its wider context. We limited the scope of the assessments to intra-class characteristics to foster the practicality and scalability of the labeling. Had we included the context of a class in the assessment, the labeling would have been far too time-consuming and complex to collect a reasonable amount of labels. An interesting continuation of this work would be to include more context and inter-class relationships.

The labeling platform only displays the code and its location inside the package structure of the project. One could argue that static analysis metrics are used in quality assessments in practice [188] and should therefore be presented here as well. Actually, we desist from that to avoid biasing the results. The goal is to capture the experts' opinions without any other influences. Displaying metrics can take the focus away from the code itself. Experts might be biased by implicit thresholds, e.g. for the size of a class, and draw conclusions based on that metric without actually reading the code. In fact, showing metrics might lead to the wrong assumption that they *must* be taken into account. The selection which metrics to display and which not might consequently bias the labeling even more. Most static measurements such as, e.g., nesting depth can be observed directly from the code as well. One notable exception that is indeed hard to identify manually is intra-class cloning.

One important aspect of software quality evaluations is the fitness-for-purpose or adequacy of the software. Software that implements complex calculations in a highly specialized domain will reflect this complexity, at least to some degree, in its source code. One could argue that negative maintainability ratings are unjustified in this case and that code from such systems should not be compared to code from supposedly simpler systems. However, we assume that, even in complex systems, the complexity is not uniformly distributed across all code files. Each class represents a controllably small part of the complex system. Therefore, it is fair to include classes from different systems and different domains in the same dataset. If source code appears hard to maintain, a negative rating is justified in any case. Later on, this dataset will be used to train ML classifiers to identify potentially problematic code. Hard-to-maintain code is worth being reported and examined by the experts even if its negative characteristics may originate from external influences.

Most existing manually labeled quality datasets rely on one single expert and his evaluation. Therefore, that data is highly biased towards that expert's subjective opinion. In our study, every code snippet was evaluated by three participants. A code file may receive more than three ratings if several experts are evaluating the same file simultaneously. Eventually, the EM algorithm iteratively assigns weights to each rating and converges towards a consensus for each data point. For 451 code files (87%) the overall maintainability label was determined with more than 70% confidence. For 339 data points, the probability for a specific label was even 90% or higher. In addition to using the most probable label, the probability of the label itself can be used as the dependent variable in regression experiments.

Besides an overall judgment, we ask participants to assess four selected subaspects of quality. The selection of these dimensions is discussed in detail in Section 5.2.3. The analysis of the submitted comments showed which aspects the participants took into account in the evaluation. However, the analysis could only respect those aspects that the experts found worth mentioning. Therefore, the list is not exhaustive.

5.6. Conclusion

This study investigates expert evaluations of software maintainability and creates a robust maintainability dataset that can be used to develop reliable prediction tools. In our survey, 70 professionals assessed code from 9 open and closed-source software projects. The submissions included ratings of the readability, understandability, complexity, adequate size, and overall maintainability of the code. Assessing the inter-class modularity with respect to the surrounding architecture was prohibitively expensive in this survey. It would increase the time needed to label each data point and decrease the number of data points labeled by each analyst. The same argument applies to the selection of only four maintainability dimensions. While it would have been possible to include more dimensions, we decided in favor of a faster labeling process.

The projects at hand contain more than 15,000 files and account for 1.43 million source lines of code. A sophisticated prioritization algorithm defined the order in which the data points were labeled. This keeps the dataset both representative and insightful. The resulting dataset contains the consensus assessment of 519 Java classes. Interestingly, our work revealed that disagreement between experts occurs frequently and considerably. Although we narrowed down the number of perspectives the code has to be evaluated from, we found significant dissent in 17% of the cases. Small deviations are observed in 73%. Consequently, we argue that a consensus between the experts has to be found before relying on their evaluations. For this reason, we presented an aggregation algorithm. Based on the submitted ratings and the evaluators' error probabilities, it determines which judgment is most probably correct. On average, the most probable overall maintainability label is assigned with an internal confidence of 89%. Based on this label, the majority of the evaluated files are considered to be easily maintainable.

Our analysis of the assessments revealed that understandability has the highest impact on the overall perceived maintainability. Whilst this attribute has the highest influence neither readability nor complexity nor adequate size should be neglected. Furthermore, we identified which other aspects of the code the experts took into account. Among the most reported issues are violations of coding conventions, commented-out code, *'todo'* or *'fixme'* annotations, and missing or unhelpful comments.

Until now, most other software quality datasets are either not based on expert judgment or rely on a small group of experts without further analysis. This study, however, provides in-depth insights into how experts perceive quality. Moreover, we showed that this perception can vary and that aggregating ratings is therefore necessary and useful. Finally, we present a robust dataset as the basis for building precise and useful quality assessment tools. In the spirit of open science, the evaluated open-source code and its ratings are shared with the scientific community.

6. Software Maintainability Prediction Based on Static Code Metrics

Parts of this chapter have previously appeared in a peer-reviewed publication ([188]) co-authored by the author of this thesis.

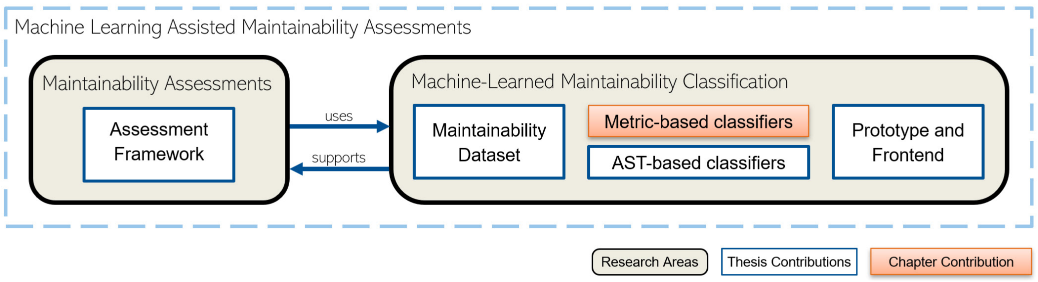


Figure 6.1.: Contribution of this chapter, i.e. metric-based classification models, in the context of the thesis

This chapter focuses on maintainability classification using static code metrics as input. Figure 6.1 illustrates the contribution in the context of the proposed framework.

The initial experiments in Chapter 4 showed the potential of using static code metrics as features to predict the maintainability of source code. The recently created maintainability dataset (cf. Chapter 5) paves the way for elaborated machine learning experiments to predict expert maintainability judgment. The dataset contains the consensus of several analysts. In general, the average expert's opinion deviates from the consensus in some cases. This allows for a new evaluation baseline: On average, the alignment of analysts with the consensus is at an accuracy of 70.1% and mcc of 0.529. With an accuracy of 72.9% and mcc of 0.525, our models achieved predictions that are as close to the consensus rating as an average human analyst.

Problem: Recently, machine learning has emerged as a useful tool to integrate human experience and automation. However, past approaches did not yet reach operational maturity. They either target legacy programming languages, are biased toward the subjective opinion of a small expert group, and often do not contain a practicality-oriented evaluation.

Solution: In this study, we use the recently described software maintainability dataset (cf. Chapter 5), that contains maintainability labels for Java classes on an ordinal, four-part scale. In contrast to the pre-study presented in Chapter 4, the following chapter investigates several ordinal machine learning approaches including multiclass classification, binary decomposition, regression-based models, and novel tailorable meta-classifiers. Then, we compare the deviation of the predictions from the ground truth to the average deviations of the experts from the ground truth, i.e. their consensus. Starting with more than 170 static code metrics extracted by seven different tools, we identified the features with the highest predictive power towards the maintainability label.

Contribution: All results were obtained on the software maintainability dataset in project-wise cross-validation and are averaged over 30 random seeds. The highest observed Matthews Correlation Coefficient is at 0.525, which is only 0.004 below human performance. Notably, our created models are (i) more often consistent with the consolidated label, and (ii) make less severe mistakes than the average human expert. In addition, we found rather simple code metrics such as the size of the class, its cognitive complexity, maximum and average method length, and the total size of all methods to yield the highest predictive power.

Limitations: To increase the expressiveness of the results, we used project-wise cross-validation and a human-labeled dataset. Still, the evaluation took place on a dataset and the developed models have not been deployed in an industrial context. So far, only static code metrics have been tested as input features.

6.1. Experimental Design

This study systematically examines the use of ML to predict software maintainability based on expert judgment. One of the greatest challenges in prediction experiments is the definition of independent and dependent variables. The research data in this study is drawn from a recently published and manually labeled dataset. A key issue in our research is to use easily available features that are already widely used in quality assessments. We therefore craft our features from static code metrics. By employing elaborated evaluation metrics and cross-validation of the prediction models, we provide a reasonable and realistic experiment setting. To demonstrate the practicality of our approach, we put the performance metrics into context and compare them to meaningful baselines. Specifically, we will answer the following questions in this chapter:

- **RQ1:** Which static code metrics yield the highest predictive power in maintainability prediction?
- **RQ2:** How well can classifiers distinguish between easy and hard-to-maintain code?

- **RQ3:** How good is the performance of the ML models considering an ordinal label?

6.1.1. Dataset and Response Variable

The response variable and factors, i.e. the labeled data used in our experiment, are drawn from a recently published dataset [183, 184]. In this study, we focus on the attribute *overall maintainability judgment*. The code files were sampled from five open-source and four commercial software systems in Java, including *ArgoUML*, *Art of Illusion*, *Diary Management*, *JUnit 4*, and *JSweet*. The data collection methodology is described in detail in Chapter 5. In brief, several experts assigned ratings on a four-class Likert-scale to each Java class. Their evaluation focused on maintainability aspects that can be identified within a Java class without additional knowledge of its surroundings. Then, the opinions of the participants were aggregated into a consensus judgment. Eventually, the rating of a Java file is reported as the probability of each label on the Likert-scale to be the correct answer. To enhance readability, we refer to these labels as EASY, RATHER EASY, RATHER HARD, and HARD TO MAINTAIN.

One way to craft a label from the available data is to compute a weighted average of each class and its probability. Converting ordinal classes to numbers is not unusual and common practice in ordinal classification [117]. However, arithmetic operations like multiplication are only defined on ratio scales and should be avoided on ordinal scales [60]. Therefore, we choose the class with the highest probability as the response variable in our experiment. This results in the following label distribution: 276 (53.2%) Java files are considered EASY, 130 (25.0%) RATHER EASY, 73 (14.1%) RATHER HARD, and 40 (7.7%) HARD TO MAINTAIN.

Although the dataset is very recent, it bears several characteristics that are challenging for ML. As a manually labeled dataset, the size is limited to 519 data points. The measurement uses an ordinal scale, thus posing an ordinal classification problem. Furthermore, the dataset is imbalanced in its distribution. However, imbalanced data is a common problem in quality prediction [138].

6.1.2. Feature Engineering and Independent Variables

The objective of feature engineering is to extract factors from raw data with a high predictive power towards maintainability. In our case, the raw data is the source code of the study objects. Static code metrics are numerical representations, that capture certain characteristics of code. Since metrics are known to be valuable indicators for maintainability [37, 121, 153], they are a natural choice to be used as features. Using metrics as ML input has led to promising results in previous research [126, 137, 187, 212].

To avoid bias towards a specific tool or metric suite, e.g. CK [35] or MOOD [77], we proceeded as follows: First, we extracted metrics from a variety of static analyzers that are used in quality control, namely ConQAT [44], Designite [194], SD Metrics [220], SonarQube [28], Sourcemeter [131], Teamscale [84], as well as some cohesion metrics from an unpublished tool. This offers a variety of different metrics types such as size metrics, code cloning information, code style rule violations, coupling and inheritance information, as well as aggregated data such as *findings* or cognitive complexity. Second, we eliminated redundant metrics that are measured by more than one tool. For example, the size of a class in lines of code was measured by four tools. Third, we removed collinear metrics. One assumption for successful ML is the absence of high inter-correlations among the input variables [205, 207]. In fact, collinearity can lead to distorted results of the feature selection [51] and should be eliminated [8]. Following [205], we use Pearson's Correlation Coefficient [159] and remove features with a value greater than 0.8. For example, this eliminates *source lines of code* from the candidate features due to its 0.99 correlation with *lines of code*.

Another popular technique to reduce the feature space is the elimination of almost constant features. Due to their low variance, such features do not offer a high information gain. However, in this specific case, we did not apply a variance threshold. The chosen static analyzers are designed to detect quality flaws. Thus, we hypothesize a metric might be inconspicuous most of the time and only show high peaks in rare cases, i.e. when a quality defect occurs. Such a metric indeed offers high predictive power towards quality assessments and must not be excluded based on its low variance.

The result of the feature engineering is a list of 132 potentially useful candidate features. The complete list is available in our supplemental material [186]. Feature selection, which we elaborate on in Section 6.2.1, determines which subset is used in every experiment run. Please note, normalization and oversampling are not part of feature engineering, but of a data preprocessing step.

6.1.3. Evaluation Criteria

In this study, we consider research questions related to the performance of prediction models. Technically speaking, these models solve an ordinal multiclass classification problem based on an imbalanced, small dataset. To evaluate their performance, the predicted labels are compared against the true labels. One intuitive way to do so is accuracy, which describes the ratio of correctly classified instances among all instances. Other popular performance metrics are precision, recall, F-Score, Receiver Operating Characteristic (ROC) [55], and the Area Under Curve (AUC). Though ROC and AUC are viewed as superior choices in binary classification tasks [22], they are not recommended in a multiclass setting [201].

To evaluate the performance of multiclass classifiers across all labels, we use micro-averaging (cf. Section 2.3.2). In this case, the evaluation metrics accuracy, precision, recall, and F-Score result in the same number. For better readability, we will thus refer to this score only as *accuracy* in the remainder of this study.

Regarding the ordinal character of the label, it is particularly interesting how far off wrong predictions are. Mistaking excellent code for moderately good code should be punished less than mistaking excellent code for very poor. Therefore, we apply the mean squared error (MSE) to evaluate the prediction. To apply this concept to the ordinal case, we define all intervals on the ordinal scale to be equal and of size 1. Furthermore, we use Cohen's Kappa (Cκ) [36] and Matthews Correlation Coefficient (mcc) [67] to respect the imbalanced label distribution. Both metrics represent the alignment of two raters while also taking into account that agreement might happen by chance. For balanced datasets, $C\kappa \in [-1; 1]$ applies. Values below or equal to zero indicate randomness, larger values indicate higher agreement. However, the maximum reachable value is lower for imbalanced datasets, depending on the exact distribution. For mcc ($mcc \in [-1; 1]$) a value of 1 indicates perfect alignment as well. In contrast to Cohen's Kappa, an mcc of -1 corresponds to perfect inverse alignment.

All classification experiments are performed with 30 different random states. The results reported here correspond to the average performance of a classifier and its configuration. In Section 6.2.2, we introduce novel ordinal classifiers, which build upon internal regression models. To optimize these regression models, we use the mean squared error as well.

Project-Wise Cross-Validation

ML algorithms are trained on a set of data and evaluated on a hold-out set that has not been part of the train set. There is a variety of heuristics on how to split the available data into training and test sets. In this study, we respect the use case of quality analyses where the experience of past projects is applied to a new project. Therefore, we split the data along the study objects, i.e. the data points in the test data are drawn from a different project. However, the measured performance depends heavily on the chosen test project and the generalizability of the results to other projects remains unclear. One way to overcome this is k-fold cross-validation. Here, the dataset is split into k partitions. Then, the experiment is performed k times with a different hold-out set used in every run. In this study, we use a project-wise cross-validation. Each model is trained on data from eight projects and evaluated on data from one project. Hence, all experiments are performed nine times with each project being the test set in one run.

Baselines for Evaluation

To put the results into context, we establish three performance baselines: Two naive classifiers and the human experts, who participated in the creation of the labeled dataset. The first naive classifier always predicts EASY, the most common label. In this imbalanced dataset, it achieves an accuracy of 54%. Due to its constant output, both Cκ and mcc are 0. The second baseline classifier randomly selects one of the four available labels. Here, the randomness is based on the distribution of labels as well. Averaged over 30 runs, it achieves an accuracy of 38%. The random character becomes visible as Cκ and mcc are close to 0.

The most interesting baseline is the performance of human experts. As explained in Section 6.1.1, the ground truth for a particular data point is defined as the aggregated judgment of the experts. Our models are then trained towards that ground truth. However,

Table 6.1.: Performance Baselines (as appeared as *Table 1* in [185])

Ordinal Classification

Baseline	**ACC**	**MSE**	**Cκ**	**MCC**
Consensus of experts	1.0	0.0	1.0	1.0
Average expert	0.701	0.414	0.528	0.529
Always EASY	0.544	1.394	0.0	0.0
Distrib.-based guessing	0.384	1.775	0.075	0.063

Binary Classification

Baseline	**F-Score**	**AUC**	**Recall**	**Precision**
Consensus of experts	1.0	1.0	1.0	1.0
Average expert	0.881	0.831	0.881	0.881
Always (EASY or RATHER EASY)	0.794	0.500	0.794	0.794
Distrib.-based guessing	0.644	0.542	0.677	0.614

the individual expert's judgment in general deviates somewhat from that aggregated consensus. Analyzing the deviations between their individual ratings and the eventual consent label, we establish a reasonable baseline. On average, the study participants agreed with the consensus in 70% of the cases, thus resulting in an accuracy of 70%. Analogously, we observe an MSE of 0.41, Cκ of 0.53, and mcc of 0.53.

For the binary classification of code snippets as targeted in RQ2, the experts achieved an F-Score, precision, and recall of 88% each and an AUC of 83%. In contrast, the constant base classifier yields an AUC of 50% and an F-Score of 79%. The random classifier achieves a similar AUC (54%) but less precision (61%), recall (68%), and F-Score (64%). Table 6.1 summarizes these baselines.

6.2. Experiment Execution

The goal of our experiments is to identify problematic program classes. This section elaborates on the experiment execution, data preprocessing, and ML models. In an initial experiment, we investigated which data preprocessing techniques are applicable in our setting and identified the most promising prediction algorithms. Then, we trained ML models on the engineered features and labels. Eventually, we evaluated the created models with respect to the defined evaluation criteria.

For a fair comparison, we repeated every experiment 30 times and fed 30 different random seeds to the classifiers. The reported results correspond to the average of these runs. All models are implemented using *scikit-learn* [162]. The hyperparameters of the models are tuned using grid search with integrated cross-validation. However, in the pre-experiments, only one random seed was used.

6.2.1. Data Preprocessing

In the feature engineering phase, we chose static code metrics as features. One key issue with static metrics is their varying value range. Hence, normalization and standardization techniques can facilitate the extraction of information for learning algorithms. Balancing the dataset can compensate for a learning bias introduced by the unbalanced distribution of labels. However, first experiments showed that no technique has an exclusively positive effect on all evaluated models. While they can improve the performance of one model, the performance of others might be decreased. Related research has already made similar observations [115, 138, 160, 161]. The data preprocessing is therefore configurable and not applied in every experiment run.

Balancing We paid close attention to the imbalanced label distribution when selecting the evaluation criteria. However, the distribution should also be taken into account in data preprocessing.

Undersampling and oversampling are techniques to balance out uneven distributions. In undersampling, data points whose label is overrepresented are omitted. Due to the already small dataset size, we refrain from this method and focus on oversampling. Here, artificial data points of the minority class are added. One of the most popular algorithms is SMOTE, the Synthetic Minority Oversampling Technique [32]. Malhotra et al. [138] identified this technique as particularly applicable to maintainability prediction. A comparison of several state-of-the-art algorithms is provided in [115]. There exist hybrid methods combining undersampling of the majority class with oversampling of the minority class as well. In our setting, this harmed the results for all models. In our pre-study, we found the most useful algorithm in our case is k-means-SMOTE [52]. We used the implementation described in [116] and set $k = 2$ due to the small dataset. After performing k-means clustering within the minority class, two members of the cluster are chosen and a synthetic example is created as a new data point in the feature space between those two examples [52, 115]. However, this does not mean a new code snippet is created – only the numerical features used for the prediction are synthesized. For example, two code snippets have a size of 1000 and 1100 lines of code, respectively. Then, a synthetic entry with 1050 lines of code will be added to the training set.

Normalization and Standardization Some ML algorithms are prone to features spanning several orders of magnitude. We, therefore, use scikit-learn's `StandardScaler` to rescale numerical features around their respective mean. Another possibility is normalization, which converts all features to an interval $I = [-1, 1]$. This is implemented using scikit-learn's `MinMaxScaler`. Please note, both options can lead to improved results as well as to a performance decrease. Hence, we implemented them as optional configuration settings, which are not permanently applied.

Feature Selection In this experiment, there are 132 features available that are related to software maintainability. However, using all available features is not recommended in ML. Using only a subset can increase the performance of ML algorithms while simultaneously making them less prone to overfitting [110]. Accordingly, we rank the available features and only use the most relevant ones as input. For classification tasks, we utilize *mutual information* [118, 178] to quantify the dependency between a metric and software maintainability. For the regression-based classifiers, which we will introduce in Section 6.2.2, we use the mutual information in a version for a continuous target variable. In each experiment, we start with a subset of five features. Then, we enlarge the input set in increasing steps.

6.2.2. Machine Learning Models

Since related works did not provide clear answers to which algorithm offers the best solution for maintainability prediction, we performed a pre-experiment to identify promising algorithms. Therefore, we selected a diverse set of 25 algorithms representative for different approaches, including support vector machines, logistic regression, neighborhood-based and Bayesian algorithms, decision trees, neural networks, and ensemble classifiers. In the pre-experiment, we evaluated them with the default parameters provided by scikit-learn.

Next, we applied an exhaustive grid search for the six most promising algorithms to find the best hyperparameters. These include Gradient Boosting [62], Ada-Boost [78], Extremely Randomized Trees [65], Random Forests [23], as well as Logistic Regression [88] and the K-Nearest Neighbor classifier. However, these classifiers are designed for multiclass settings. Although they are applicable to ordinal settings as well, there is a possible information loss [60]. Frank and Hall propose splitting the ordinal classification task into binary tasks that specialize in problems of the form *"is it worse than a given label?"* [60]. We refer to their original approach as binary decomposition.

Similar to them, we built naive decision trees based upon the output of binary base classifiers. We implemented this in three versions: (i) traversing the base classifiers from the best class to the worst class, (ii) traversing from worst to best, and (iii) starting at the median. These three meta-classifiers are illustrated in Figure 6.2.

In their proposed approach, Frank and Hall utilize the prediction confidence of binary base classifiers to find the most probable label [60]. Instead of extracting the prediction probability from binary predictors, we can directly use the probability of each label as it is reported in the dataset (cf. Section 6.1.1). Then, we train regression models towards these probabilities. We implemented this concept in two meta-classifiers: (i) predicting the probability of each label directly, and (ii) predicting the probability of the label to be worse than another label as implied by [60]. These two algorithms are visualized in Figure 6.3. Eventually, the label with the highest probability is assigned.

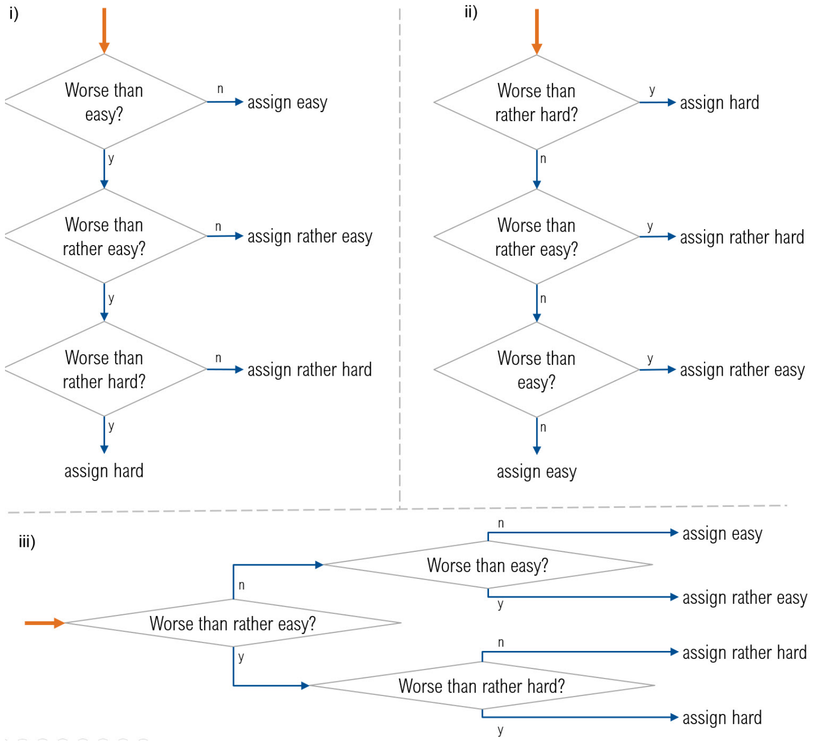


Figure 6.2.: Ordinal classification based on binary decisions: i) best-to-worst traversion, ii) worst-to-best, and iii) refinement starting at the median (adapted from *Figure 1* in [185])

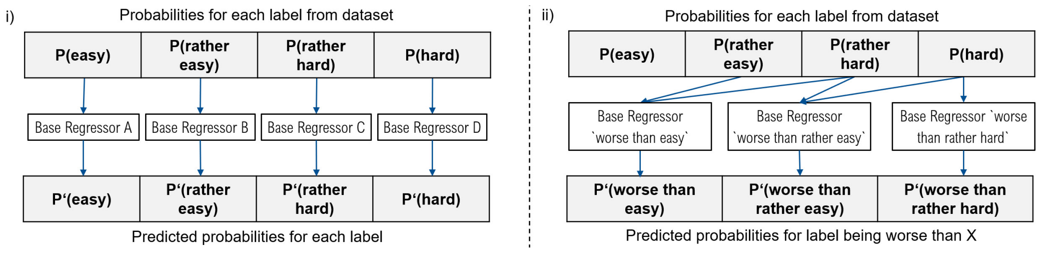

i)
Probabilities for each label from dataset
P(easy)
P(rather easy)
P(rather hard)
P(hard)
Base Regressor A
Base Regressor B
Base Regressor C
Base Regressor D
P'(easy)
P'(rather easy)
P'(rather hard)
P'(hard)
Predicted probabilities for each label
ii)
Probabilities for each label from dataset
P(easy)
P(rather easy)
P(rather hard)
P(hard)
Base Regressor `worse than easy`
Base Regressor `worse than rather easy`
Base Regressor `worse than rather hard`
P'(worse than easy)
P'(worse than rather easy)
P'(worse than rather hard)
Predicted probabilities for label being worse than X

Figure 6.3.: Meta-classifiers based on label probabilities: (i) predict the probability of each class, (ii) predict the probability of the class being worse than a given label (adapted from *Figure 2* in [185])

Besides, we modeled the ordinal classification as a regression problem. Therefore, we defined the distances between the ordinal classes to be equal and mapped the classes {EASY, RATHER EASY, RATHER HARD, HARD} to integers $\{1, 2, 3, 4\}$. Then, we applied regression towards that numeric label, rounded the predicted number to the nearest integer, and converted it back to the ordinal classes. We call this rounded regression. However, oversampling of the minority class is not possible in this case since there is no notion of a class in regression.

In summary, this leaves us with six multiclass classifiers, four ordinal classifiers based on binary decomposition, and three regression-based classifiers.

6.3. Experiment Results

We will now address the research questions and present the results of the performed experiments. If not explicitly mentioned otherwise, the prediction results are averaged over 30 random seeds and obtained in project-wise cross-validation. For increased readability of this report, we refrain from listing the hyperparameters of the mentioned models. These can be found in our supplementary material as well [186].

6.3.1. Metrics With the Highest Predictive Power

The first research question aimed to identify the static code metrics with the highest predictive power. To quantify the relationship between a metric and the maintainability label, we use *mutual information* [118, 178]. This provides a non-negative number that characterizes how much information about one variable, i.e. the maintainability, can be learned from a second variable, i.e. the code metric. To compare the effects of the mentioned data preprocessing techniques (Section 6.2.1), we computed the mutual information for all three techniques and ranked the features accordingly. We indeed observed differences in whether normalization and standardization were applied before the analysis. Table 6.2 shows the ranks of the metrics. For space reasons, we only show the ten metrics with the highest mutual information. A more detailed table is available in our supplemental material [186].

It is apparent from this table that the first ten ranks in each setting are occupied by the same ten metrics. This is a strong argument these ten features provide high prediction power independently of the data preprocessing. As the results show, five metrics are consistently ranked higher than others. These metrics are the maximum size of a method, the size of the class itself, Sonar's cognitive complexity metric, the amount of code in methods, and the average size of a method. All four size-related metrics use lines of code as their unit of measurement. Cognitive complexity aims to measure the understandability of the control flow [27].

6.3.2. Binary Classification Performance

The second research question is targeted towards the binary separation of problematic and unproblematic Java classes. This provides a first, less fine-grained impression about the quality of source code. The dataset defined EASY and RATHER EASY as agreement that code is maintainable, and RATHER HARD and HARD as disagreement. Hence, RQ2 can be modeled as a binary classification whether the code is (EASY or RATHER EASY) or (RATHER HARD or HARD) to maintain.

Table 6.2.: Ranks of the most influential code metrics (as appeared as *Table 2* in [185])

Code Metric	**Rank using preprocessing**		
	None	Normal.	Standard.
Max size of methods	1	1	1
Lines of code	2	2	3
Cognitive complexity	3	3	2
Lines of code in methods	4	4	4
Avg size of methods	5	5	5
Max length of loops	6	10	8
Loop count	7	8	10
Nesting level	8	6	6
Loc with depth greater 4	9	8	7
Max depth of loops	10	7	8
...	...	...	...

Using just five features, Extremely Randomized Trees [65] achieve an F-Score of 91.29% with identical values for precision and recall. As pointed out earlier, the Area Under the Curve (AUC) of the Receiver Operating Characteristic is a common and adequate choice to evaluate binary classifiers. The highest AUC (82.29%) was observed for the K-Nearest Neighbor classifier using seven standardized features.

6.3.3. Ordinal Classification Performance

For conciseness, we only report the results achieved by the best classifier with the best configuration and hyperparameter set. In this context, we refer to a *configuration* as the chosen data preprocessing method and the number of input features. The exact configurations are available in the supplemental material. Table 6.3 summarizes the best results and the baselines introduced in Section 6.1.

The ordinal meta-classifiers reached significant results. They achieved accuracy values between 70.4% and 72.0%. The binary-based approaches achieved an MSE between 0.3538 and 0.3562, both probability-based approaches an MSE of 0.3509. For the meta-classifiers, Cκ ranges between 0.4906 and 0.4970, and mcc between 0.5015 and 0.5154. Rounded regression yielded similar values for accuracy and Cκ, and slightly better values for mcc and MSE.

Table 6.3.: Ordinal Classification Results (as appeared as *Table 3* in [185])

	ACC	**MSE**	**Cκ**	**MCC**
Baselines				
Guessing (distr.-based)	0.3840	1.775	0.075	0.0633
Always EASY TO MAINTAIN	0.5445	1.394	0	0
Average expert	**0.7010**	**0.4140**	**0.5278**	**0.5291**
Meta-classifiers				
Binary Decomposition	0.7203	0.3562	0.5062	0.5154
Chained binary	0.7162	0.3538	0.4931	0.5039
Chained binary inverse	0.7162	0.3538	0.4954	0.5039
Chained binary median	0.7162	0.3538	0.4932	0.5007
Binary probabilities	0.7099	0.3509	0.4906	0.5015
Individual probabilities	0.7175	0.3509	0.4906	0.5015
Other algorithms				
Rounded regression	0.7209	0.3416	0.5097	0.5230
Multiclass models	0.7294	0.3135	0.5137	0.5252

The last section of the table summarizes the results for multiclass classifiers. The highest accuracy was achieved by Gradient Boosting. Using 25 features and input normalization, it labeled 72.94% of the data correctly. The lowest MSE, which we use to measure the severeness of misclassifications, was 0.3135. This was observed for Extremely Randomized Trees using 18 features and no data preprocessing. Balancing the dataset with SMOTE led to Cκ and mcc results of 0.5137 and 0.5252, respectively. A Random Forest classifier achieved these values with five features and without data preprocessing. Using the same configuration (SMOTE-balancing, five features), Random Forest models also achieved an accuracy of 71.55% and MSE of 0.3477. Both values are better than those reported for the human study participants. However, we did not find a classifier and preprocessing combination that dominates all others in every dimension. Using stratified 10-fold cross-validation instead of project-wise cross-validation, we even achieved accuracies of 75%, mean squared errors smaller than 0.3, and Cκ and mcc of 0.6.

Analyzing the intermediate results within the cross-validation, we observed the performance varies strongly depending on the project chosen for validation. In particular, we note a decreased performance if *Art of Illusion* is chosen as the validation project. This project contributes most of the negatively labeled data. Thus, these points are missing in the training set. Hence, the label distribution in the training set becomes even more imbalanced which enhances distribution-based bias.

6.4. Interpretation

This section includes an overview of the results and their interpretation along the research questions.

6.4.1. RQ1: Which static code metrics yield the highest predictive power?

In Section 6.3.1, we reported the ten most influential metrics according to their mutual information. Interestingly, the most influential metrics are identical independently of the data preprocessing technique, thus making a strong argument for their high predictive power. This is supported by the results of RQ3. The highest accuracy, Cκ, and mcc scores were achieved by models using just five metrics. Interestingly, this set consists of four rather basic size metrics and just one aggregated measurement. This finding was unexpected given the supposedly advanced measurements we fed into the feature selection. Still, similar observations have been made in the past. In a study by Sjøberg et al. [199], neither code smells, coupling, inheritance, nor the Maintainability Index showed correlations with the actual maintenance effort. The only aggregated metric within the ten highest-ranked features in our study is cognitive complexity. In contrast to cyclomatic complexity [140], which focuses on paths in the control flow of a program, cognitive complexity aims at the understandability of the control flow [27]. Its usefulness has been validated in a survey by Muñoz Barón et al. [147] as well. The two most influential features in this study are the maximum size of methods and the overall size of the class. These results are consistent with those reported by related work: Sjøberg et al. [200] found long classes and long methods to increase maintenance effort as well, and Schnappinger et al. [187] identified the size of the longest method as one of the most influential features in their experiment, too. Though these two metrics correlate with each other, their correlation coefficient is below the threshold to exclude them from the feature set.

Furthermore, it is known from related work that cloning metrics such as the clone ratio, length of clones, and number of clone classes are valuable predictors for maintenance effort [98, 179, 187]. Contrary to expectations, this study did not find high correlations between cloning metrics and the maintainability label. This discrepancy could be attributed to the labeling procedure. The data labeling focused on manually-detectable intra-class characteristics and excluded inter-class characteristics for efficiency reasons. Thus, cloning was not respected by the participating analysts [183]. Consequently, cloning metrics do not yield much information towards the reported label. This limitation of the dataset explains the low predictive power of coupling and inheritance metrics as well. We do not deny the usefulness of these metrics. Because we aim to identify problematic code classes, we value maintainability aspects more that are directly observable and repairable within that class. However, the effects of cloning, coupling, and inheritance are difficult to determine and remedy at the class-level.

6.4.2. RQ2: How well can classifiers distinguish between easy and hard-to-maintain code?

With respect to the second research question, it was analyzed how effectively prediction models can discriminate between classes with high and low maintainability. For the experiment, the ordinal classes EASY and RATHER EASY were combined as well as RATHER HARD and HARD. Due to the imbalanced label distribution, the naive classifiers reached F-Scores of 64% and 79%. Precision and recall are of the same magnitude. However, they yield an AUC-Score of only 50% and 54%. The best performing classifiers achieved an F-Score of 91.29% and an AUC of 82.29%, clearly outperforming the baseline classifiers. To interpret this result in the context of our use case, we compare it to human performance. The AUC of the experts (83.16%) is less than 1 percentage point greater than the AUC of our model. This still leaves room for improvement but provides a good approximation with an accuracy similar to experts. However, with our approach, there is an increase of 3.2 percentage points concerning the F-Score. Accordingly, this binary prediction, though less fine-grained than a multiclass analysis, raises the possibility to get a valuable first overview with just a small risk of misclassification.

6.4.3. RQ3: How good is the performance considering the ordinal label?

The current study evaluated the use of ML algorithms to predict the maintainability of source code on an ordinal scale. To evaluate the results, we referred to the mean squared error, Cohen's Kappa, and Matthews Correlation Coefficient (mcc). We also report the accuracy score, which in this setting is identical to precision, recall, and F-Score. The achieved results are summarized in Table 6.3. To put them into context, we defined three baselines. The most relevant baseline is built upon the experts involved in labeling the data. The final label is set as the consensus of the participants. Thereby, although we are using a dataset labeled by humans, the individual experts did not reach perfection. In fact, the average participant was congruent with the final label in only 70% of the cases. Using our novel meta-classification concepts, we exceed this performance in terms of accuracy and MSE. Regarding Cκ and mcc, they are slightly below it. Interestingly, the results of these approaches are very close to each other and no concept with clear advantages can be identified.

Notably, the best-performing multiclass models are ensemble classifiers. This means they use multiple internal classifiers and combine their results using averaging or boosting. In our experiments, a Gradient Boosting model reached an accuracy of 73%. Surprisingly, that exceeds the human baseline by 3 percentage points. It also outperforms naive classification baselines by a large margin (18.5 and 35 percentage points, respectively). Similar results are found for the mean squared error, which quantifies the severeness of ordinal deviations. The best performing model, Extremely Randomized Trees, reaches a value of 0.3135, which is 0.1 lower than the human baseline. The naive baselines are clearly outperformed concerning MSE, Cκ, and mcc as well. Although our best model achieved remarkable results of

$C\kappa = 0.5137$ and $mcc = 0.5252$, it did not reach human-level performance. For the average expert, we report $C\kappa = 0.5278$ and $mcc = 0.5291$. This results in a delta of 0.0141 and 0.0039 in favor of the human baseline. Overall, we can summarize that ML models exceeded human-level performance in two dimensions, and are very close in two others. These results allow for more efficient quality control as classes can be prioritized according to their predicted quality.
A comparison with related work is difficult. To the best of our knowledge, as of today, no other studies are using the same dataset or a comparable ordinal classification setting.

6.5. Discussion

In the following, we discuss the study design, the practical implications of our results, its limitations, and future work.

6.5.1. Applicability of the Study Design

As discussed in Section 6.1.3, the models were evaluated in project-wise cross-validation. Had we applied stratified 10-fold cross-validation across all projects, the achieved performance had been above the human baseline in all four dimensions. For the sake of a more realistic evaluation, we focus on the results of the project-by-project cross-validation – even if this leaves us slightly below the human baseline in two metrics.

One notable finding of our analysis is the varying prediction performance within the cross-validation. This implies a strong influence of the chosen train-test split on the performance. The lowest performance was observed when evaluated on *Art of Illusion*. This project contributes a majority of the data with the underrepresented labels RATHER HARD and HARD. This finding validates our experimental design. First, it suggests that transfer learning, i.e. training a model and applying it to a completely different context, is not trivial. Hence, a project-wise validation is realistic and more insightful. Second, the deviating performance justifies the use of cross-validation. Third, the exceptionally high loss concerning *Art of Illusion* implies an influence of imbalanced train sets. Hence, using balancing as a preprocessing step is reasonable. However, it did not improve the results of all evaluated ML models. This is in line with the conclusion of Pecorelli et al. [161], who made a similar observation concerning code smell detection.

6.5.2. Practical Implications

Notwithstanding the limited dataset, this work offers valuable contributions to maintainability assessments. In this study, we identified the software metrics with the highest predictive power towards maintainability. This result is helpful for developers using static analysis tools and provides practical guidelines on which measurements should be given the most attention. To quickly get an overview of the state of a project, the code can be divided into unproblematic and problematic program classes. Our binary model allows this to be done with little risk of incorrect predictions. For a more fine-grained analysis, the ordinal multiclass model can be used. Maintainers can utilize the predicted labels as a prioritization of which classes should be improved first. Though not perfectly accurate, our models provide the same quality of evaluations concerning the ground truth as an average analyst. However, the threats to validity described in the following section apply.

In Section 6.2.2, we introduced meta-classifiers for ordinal classification. Using these models, we achieved an accuracy of 72.0% and an MSE of 0.3538. This reflects an accuracy improvement of 1.9 percentage points concerning the human baseline and an MSE that is 0.06 smaller. However, in this setting, they could not match the results of the multiclass ensemble classifiers Gradient Boosting and Extremely Randomized Trees. In the ordinal classification study by Frank and Hall [60], this behavior can be observed in several settings as well. Our meta-classifiers decompose the ordinal multiclass problem and use base predictors to solve subtasks. Then, the results of the internal models are combined to assign the final label. One advantage of this approach is that the base predictors can be optimized independently. In this study, we applied a default decision threshold of 0.5 for the binary classifiers. Analogously, the regression-based meta-classifiers strictly assign the label with the highest predicted probability. To avoid overfitting on the dataset, we deliberately did not temper the decision thresholds in this study. However, in a practical setting, this can easily be adapted to personal preferences. In a recall-oriented setting, a code snippet can already be reported as HARD TO MAINTAIN if the prediction probability reaches a threshold of, for example, 0.3. Accordingly, the precision can be improved by demanding higher confidence in the prediction.

6.5.3. Threats to Validity

External Validity We used a recently published, manually labeled dataset. Nevertheless, the label does not depend on the subjective opinion of a single expert but represents an aggregation. Still, the limitations of the dataset as described in [183] apply. In particular, the definition of the label must be taken into account. Because of the manual labeling procedure, it is limited to quality aspects directly visible in the code, as we elaborated on in Section 6.1.1 and Section 6.4. To evaluate ML models, the data needs to be split into train and test sets. As bias might be introduced by a fixed split, we applied cross-validation. Here, we chose project-by-project cross-validation to reflect the practical use case, where files from a project are evaluated without any data from that project being part of the training. The data is drawn from nine different projects from a variety of domains. Still, not every domain is covered. Besides, the dataset does only contain Java code. Furthermore, some models such as Random Forest internally use random processes. To mitigate the risk of random bias, we repeated every experiment with 30 different random seeds. This number of repetitions is a reasonable trade-off [175]. The biggest threat to external validity is the relatively small number of data samples. Adding new manually labeled data points was out of scope for this study, although we introduced synthetical data using SMOTE balancing. Nevertheless, the dataset contains only 519 original data points.

Internal Validity For our experiments, we extracted static code metrics from the study objects. The results of the study might thus be biased by the selected metrics and tools used. Although we covered a wide range of tools and metric types including structural metrics, design metrics, size metrics, and supposedly more expressive metrics like code smells and rule violations, we do not claim completeness of this set. A similar point can be made about the evaluated ML models. An in-depth evaluation of all known classification models is not feasible. Therefore, we performed a pre-study to identify the most promising ones. Please note, several recent algorithms like neural networks are unlikely to perform well on small datasets. The predictive power reported in Section 6.3.2 is characterized by mutual information. We are aware of other measures including Pearson's Correlation Coefficient [159], ANOVA F-value, or Spearman's Correlation Coefficient [202]. In our pre-study, we compared the reported results to those by Spearman's Correlation Coefficient and ANOVA F-value and found only small deviations. However, we focus on the results of mutual information as it is a generalized measure that does not make assumptions about the analyzed variables.

6.5.4. Future Work

The size of the training set is a key factor in ML. Expanding the dataset at hand or creating new ones should be a top priority for the maintainability community. For comparison purposes, we repeated our experiment using just a subset of the available data. Based on the observed poorer results, we hypothesize the performance of the presented models will grow with larger datasets. It is noteworthy that the dataset at hand contains evaluations of more fine-grained quality aspects such as readability as well. Further research might explore the prediction of these sub-dimensions, too. To make the results of the analysis more actionable, Explainable Artificial Intelligence can be used to reason about factors leading to the assigned label. An overview of methods can be found, for example, in [9]. However, the limitations of metric-based prediction remain to be elucidated. If the automation of quality assessment is to be moved forward, other input features should be evaluated as well. Depending on the chosen input, data augmentation can be used to inflate the available training data.

6.6. Conclusion

Internal quality control is essential in software engineering to avoid exploding costs. Thus, rapid and accurate quality evaluations are crucial. Yet fast, static analysis tools provide a myriad of metrics that need to be interpreted by developers. In contrast, expert evaluations are precise but time-consuming and expensive. Several approaches exist to automatically predict the result of an expert assessment. The main drawbacks of existing approaches are potentially outdated study objects, missing data preprocessing, bias towards a small group of experts, or evaluations that are not practice-oriented.

This study examines the *ordinal classification* of source code maintainability. The presented ML models are based on commonly used static code metrics. We analyzed more than 170 metrics from seven different tools. Our analysis found supposedly simple metrics contain the greatest predictive power. In our prediction experiments, the label corresponds to the consensus of several experts. Frequent disagreement between experts was reported during the labeling process. Hence, a comparison with the average study participant provides a practicality-oriented evaluation. In this chapter, we showed binary discrimination between high and low maintainability is possible with an F-Score of 91.3% and AUC of 82.3%. Considering a four-part ordinal scale, we achieve predictions that are as close to the consensus of experts as the single ratings of an average human being. Regarding Cohen's Kappa and Matthews Correlation Coefficient, our trained models are just 0.014 and 0.004 short of average human performance. More specifically, we exceed human accuracy by 3 percentage points and yield a mean squared error that is 0.1 smaller. Our results show ML can play an important role in effective and efficient quality control.

7. Software Maintainability Prediction Based on the Abstract Syntax Tree

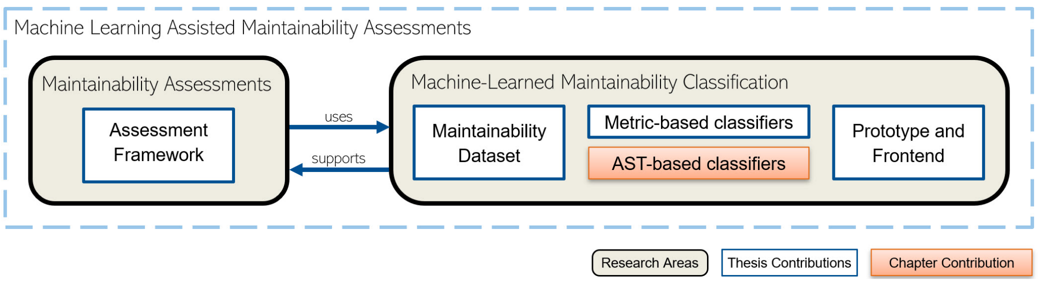


Figure 7.1.: Contribution of this chapter, i.e. AST-based prediction models, in the context of the thesis

This chapter examines AST-based classification models to predict software maintainability. Figure 7.1 illustrates the contributions of this chapter in the context of the proposed framework.

Trained and evaluated on a human-labeled maintainability dataset, machine learning classifiers achieved promising experimental results when using code metrics as input. Next, we compared the results of these metric-based models to those of the code2seq framework, which uses embeddings of the AST. In our experiments, this approach achieved a maximum mcc of 0.473 and could thus not reach the performance of the metric-based model.

Problem: So far, approaches to predict software maintainability mostly utilize static code metrics to represent the analyzed code. However, they focus on static code metrics and do not investigate other input types. In particular, recent developments in ML enable to use embeddings of the AST as input features, too. This approach has shown promising results in related disciplines [5, 6, 19, 76, 125, 127, 192], but was not yet applied to predict software maintainability ratings.

Solution: We contribute novel AST-based predictors and compare them to previously discussed metric-based classifiers regarding the prediction of maintainability ratings. We adopted the code2seq approach to predict maintainability ratings based on the AST of a program. Apart from relying on static metrics or ASTs separately, we also examined combinations of the AST and static code metrics as input.

Contribution: Evaluated in cross-validation on the software maintainability dataset presented in Chapter 5, the code2seq framework significantly outperforms naive baseline classifiers. The AST-based approaches achieved an mcc of 0.450 and could not equal the performance of the metric-based models. Using static code metrics as additional information, the performance can be increased by 0.023 but stays below the performance of the metric-based models, too.

Limitations: Neural Networks like code2seq typically assume large training datasets. Although the results are promising, code2seq might not yet play to its full strength due to the rather small dataset. Furthermore, the network can only process a certain number of paths from the AST. These paths are randomly selected and may thus hinder the transferability of the model to new data points.

7.1. Experiment Design

7.1.1. Dependent Variable: Maintainability Label

This study set out to complement recent metric-based ML models with a novel AST-based approach. Typically, ML research exploits the availability of a large number of examples to learn from. Considering maintainability, research to date has tended to either use the number of changed lines [3, 92, 103, 121, 126, 137, 138, 173, 209, 212, 227, 229] or expert ratings [79, 80, 81, 82, 185, 187] as surrogates for maintainability. Using the number of changed lines paves the way for large datasets, but has received criticism. If only the extent of the changes is considered, it is impossible to assess whether a high number of changes originates from bad quality or stems from the inherent complexity of the maintenance task. Also, it assumes identical efforts per revised code line and does not consider maintenance activities such as understanding the code or analyzing the impact of changes [46]. On the other hand, datasets containing expert ratings are typically small and thus challenging for training large ML models.

In this study, we use the manually-annotated maintainability dataset introduced in Chapter 5. Due to the applied aggregation algorithm, the dataset provides probabilities for each rating to be true. For the remainder of this chapter, we refer to the rating with the highest likelihood as the true label. In total, there are four possible classifications: EASY, RATHER EASY, RATHER HARD, and HARD to maintain. This refers to the maintainability of a Java code file, which typically contains one Java class. Unfortunately, the dataset is imbalanced, with easy-to-maintain code outweighing problematic code. In total, 276 Java classes are rated as EASY, 130 as RATHER EASY, 73 as RATHER HARD, and 40 instances are considered HARD [183]. As Di Nucci et al. point out, the realistic and potentially imbalanced distribution of a software quality dataset is of utmost importance [50], although it may be challenging for ML tasks.

7.1.2. Evaluation and Performance Metrics

Cross-Validation To evaluate the performance of a trained ML model, the available data is split into subsets for training and evaluation. Analogous to the metrics-based experiments, we utilize that the data is sampled from nine different projects and apply a project-wise split of the data. The performance of the model is then calculated against the union of all runs. The training of a model is repeated nine times with each project serving as the test set once. This provides insights into the performance on a new study project compared to the manual label.

Performance Metrics Literature employs a myriad of performance metrics, which quantify different views on the confusion matrix. Receiver Operating Characteristic and its Area Under Curve gained popularity for binary predictions, while Mean Squared Error is often used to assess regression models. In our use case, an ordinal multiclass classification, the Matthews Correlation Coefficient (mcc) [67] is a suitable metric [224]. It quantifies the agreement of two ordinal ratings, i.e. the predicted label and the true label, while taking the imbalanced distribution of the data into account. An mcc of 0 indicates all agreement happened by chance while a value close to 1 shows the perfect alignment of the ratings. As in the previous chapter, we will additionally report the F-Score. For conciseness and easier comparability, we use micro-averaging to aggregate the performance across all classes (cf. Section 2.3.2).

Baselines To interpret the observed performance metrics, we consider two baselines: Constant classification and human performance. Recalling the distribution of the dataset, we find a constant predictor, which optimistically classifies every data point as EASY, to yield an F-Score of 0.544. This baseline is known as Zero Rule classification. As the mcc considers the effect of imbalanced data distributions, that classifier achieves an mcc of 0.0. While harder to interpret than F-Score, this illustrates nicely why we value mcc over F-Score when evaluating classification algorithms.

The second baseline is the average human performance. One unanticipated observation is, that the human mcc baseline is only 0.004 higher. Considering the F-Score, the Random Forest even outperforms the average human by a small margin of 2 percentage points. The highest possible value for mcc is $+1$, which indicates perfect alignment. Unintuitively, the human performance baseline is at 0.529. This stems from the labeling procedure. The dataset considers labels by at least three participants per code file. Then, their *consensus* is considered the final label. Hence, deviations between the final label and the single experts occur. Quantifying the alignment of the humans and the final label, the average human performance is at an mcc of 0.529 [185].

7.2. AST-Based Maintainability Classification

One drawback of metric-based maintainability classification is its dependence on certain tools to calculate the input metrics [18]. In contrast, the structural composition of source code is agnostic of specific tools. Hence, utilizing its structure as a feature for ML models is a natural choice. The AST of a code class is one way to represent this structure. In the past, it has been successfully leveraged in a variety of ML applications, including functionality classification [26], code summarization [5, 6], or bug prediction [167, 218].

7.2.1. The Code2Seq Framework

A contemporary framework focusing on the syntactic structure of programming languages is *code2seq* [5]. It follows an encoder-decoder architecture to predict certain properties of code based on the AST. After building the AST, pairwise paths between terminals of the tree are examined. The encoder defines an individual vector representation for each path. The resulting encoding is then equivalent to the sequence of nodes in that path. Eventually, a decoder maps the embedded features to the output label. Here, an attention mechanism assigns weights to the embeddings of the different input features, i.e. AST paths, and infers the label based on the most relevant inputs. Common applications include method name prediction and code summarization [5, 76, 125], or code creation and autocompletion [109]. The predecessor of code2seq, *code2vec* [6], was used, for example, in fault localization [127] and vulnerability prediction [19]. Code2vec follows a similar AST embedding approach [6]. According to the main author of both approaches, code2seq *"is a better model and less sparse than code2vec"*[1], and *"has a better encoder than code2vec"*[2].

The main difference is that code2vec can neither represent nor generate unseen relations [5]. It embeds paths monolithically and thus cannot handle unseen paths adequately. In contrast, code2seq represents AST paths as sequences and treats the paths node-by-node [5]. This leads to greater generalizability of the learned models. Because of the better generalization and performance of code2seq, only code2seq is examined in detail here.

7.2.2. Adopting Code2Seq

Originally, code2seq targets code attributes at the method level. To predict the maintainability of code files, we had to adapt the model in the following way. First, we changed the AST parser to build ASTs starting at the root of the class instead of a method. Second, we had to modify how the labels are processed. This includes retrieving the maintainability ratings from the dataset, splitting the data into project-wise partitions for the cross-validation, and changing the classification from a nominal to an ordinal prediction. Third, we adjusted the framework to conform with the class-level features. This is necessary, as a class-level

[1] `https://github.com/tech-srl/code2vec/issues/26`
[2] `https://github.com/tech-srl/code2vec/issues/42`

AST features both more and longer paths than a method-level AST. Thus, using the default parameters of code2seq delivers subpar results. In the following, we explain our adoptions:

Network Size The encoder consists of a Bidirectional Recurrent Neural Network with two Long Short-Term Memory (LSTM) cells of equal size. One cell handles the forward pass, while the other is responsible for the backpropagation. In our experiments, we found sizes of 128 and 256 to yield the best results.

Context Sampling Code2seq only considers a subset of all AST paths, with the size of the selected set being configurable. The paths are randomly selected. As the class-level AST contains substantially more paths than method-level ASTs, we increased the default value by a factor of 10. Otherwise, the probability of containing the paths that are most relevant for maintainability prediction would be very low. In general, we observe better performance if more contexts were considered. Unfortunately, examining even larger models in the grid search was infeasible due to the available hardware. Still, we employ up to 2000 paths per data point.

Nodes per Path Eventually, code2seq analyzes the sequence of nodes in a path. Intuitively, class-level AST paths tend to be longer than those at the method level. However, the length of a class-level path can be approximated as the length of a method-level path with one additional node, i.e. the class declaration. Hence, the consequences of the granularity change are neglectable in this case. Larger path lengths did not lead to higher prediction accuracy. The default setting of nine nodes per path seems to be a reasonable value for the class-level scenario, too.

7.2.3. Augmenting the AST with Static Metrics

Apart from relying on static metrics or ASTs as inputs separately, we also combine both feature types. So far, the code2seq model depends solely on the information provided in the AST. However, these data potentially can be augmented with additional information. Some information is only implicitly available or not included in the AST at all but can be captured by static code metrics. Providing this information explicitly is likely to improve the prediction performance. To select which metrics to use, we refer to the feature selection methodology and metric suite used in the metric-based experiments. It is described in detail in Section 6.2.1. All feature candidates are ranked by their mutual information toward the maintainability label. A new hyperparameter specifies how many code metrics will be used as additional features. If no code metrics are selected, we fall back to the variant using only AST features. Otherwise, the neural network receives the specified number of metrics as well as the AST paths as input. The number of considered paths is then reduced by the number of metrics to maintain the same number of input features.

Table 7.1.: Experimental Results and Baselines

Classification	**MCC**	**F-Score**
Average Human	0.529	0.701
Zero Rule Classifier	0.000	0.544
Code2Seq - AST only	0.450	0.653
Code2Seq - AST and Metrics	0.473	0.669
Random Forest - Metrics	0.525	0.720
Gradient Boosting - Metrics	0.510	0.729

Apart from that, the structure remains the same: The encoder encodes the input features, i.e AST paths and metrics, and the decoder predicts the label based on the most important features.

7.2.4. Training Code2Seq

We use project-wise cross-validation and hyperparameter grid search to determine the best model configuration. Furthermore, we include the number of static code metrics to be added to the AST as a searchable parameter, too. Our models are built on top of the Tensorflow-2 implementation[3] of code2seq. It is trained on two Nvidia RTX 2070 GPUs with CUDA 11.0 and 8 GB of memory each. For each experiment run, we initialize a new model. Using a pretrained code2seq model – as it is provided, e.g., by the developers of code2seq – and fine-tuning it to our classification task, is not possible because of the different AST granularities. One particular challenge in our experiments was overfitting. While the models often reached perfection on the train set, the performance on the test sets initially was on the same level as random classification. This phenomenon is typical when training large models on small datasets [66]. Since increasing the number of data points is beyond the scope of this study, we controlled the overfitting with increased dropout [203] and early stopping, i.e. limiting the training time [66].

7.3. Results

Table 7.1 summarizes the aggregated results of the experiments and contrasts them with the baselines and previously presented results. Figure 7.2 visualizes the *mcc* values as a bar chart. The values reported in this chapter represent the aggregated and micro-averaged measures after the cross-validation.

[3]`https://github.com/Kolkir/code2seq`

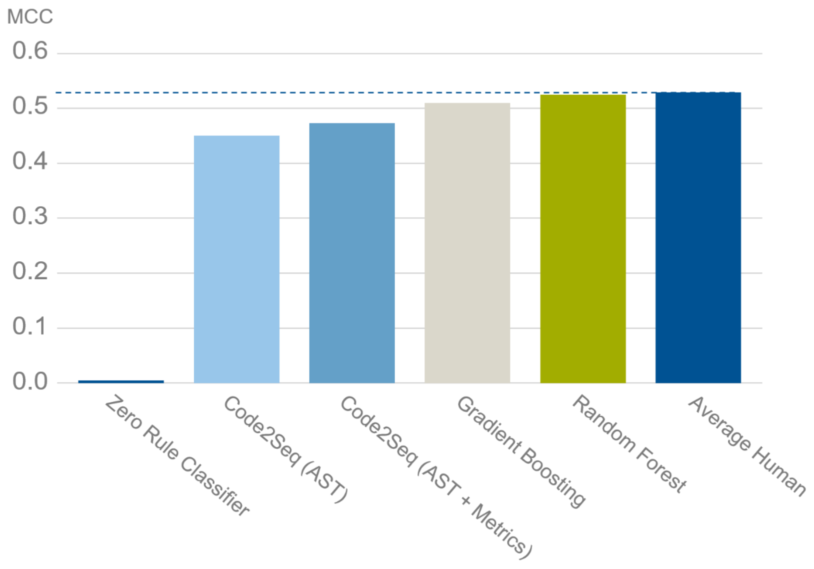


Figure 7.2.: Aggregated mcc scores of the classifiers and baselines.

7.3.1. Prediction Performance

This study provides one of the first investigations into how AST-based ML techniques can be used to predict software maintainability at the class level. Using embeddings of the AST paths, our customized code2seq model achieved an F-Score of up to 0.653 and an mcc of 0.450. Interestingly, these are observed for two different models. Both are of the same size (two LSTM cells of size 128) but apply different dropout probabilities. The model achieving the highest F-Score used a dropout keep probability of 0.5, while the model providing the highest mcc used a keep probability of 0.1. Also, they differ in terms of allowed maximum training epochs.

If we complement the AST paths with static metrics, the code2seq model achieved an F-Score of 0.669 and mcc of 0.473. Here, the best performance was found using three additional metrics, namely the length of the longest method, the size of the class, and its cognitive complexity. These metrics are a subset of those used by the metric-based Random Forest. This model uses a dropout keep probability of 0.4 and consists of two LSTM cells of size 128 each.

7.3.2. Result Interpretation

Considering the use case of predicting the maintainability of Java classes, the present results indicate that AST-based code2seq models perform worse than models based on static code metrics. However, using both metrics and AST paths is found to increase the performance of code2seq compared to AST paths alone. Despite this improvement, the code2seq models are outperformed by metric-based classifiers.

To put the results into context, we compare the results to the evaluation baselines introduced in Section 7.1.2. We consider the metric-based Random Forest the best-performing model. It achieved an mcc of 0.525, which is 0.075 higher compared to the code2seq model and 0.525 above the Zero Rule classifier. Interestingly, a metric-based Gradient Boosting model slightly exceeded the Random Forest considering the F-Score (0.729 vs 0.720) but yielded a lower mcc (0.510 vs 0.525). As we value mcc over F-Score, we consider the Random Forest the superior model.

Concerning the distribution-based baseline, the code2seq model outperforms the naive Zero Rule classifier by 10 percentage points and reaches an F-Score of 0.65. Considering mcc, it achieved a value of 0.45 in contrast to 0.00 by the baseline classifier. Concerning the F-Score, our code2seq approach is just 5 percentage points off the human baseline. Using both AST and metrics, the F-Score of the model is increased by 1.6 percentage points and the mcc by 0.023. However, it could not reach human-level performance. The mcc is 0.056 lower compared to that of an average analyst. As mentioned earlier, the used maintainability label represents the consensus of at least three humans per code file. Thus, the deviation of the individual ratings from the ultimate consensus can be used to calculate the performance of each expert. Analyzing the dataset in depth, we found an average study participant to correspond to an F-Score of 0.701 and mcc of 0.529 [183, 185].

The overall performance of the code2seq models is diminished by misclassifications. The detailed analysis of the confusion matrix revealed that several misclassifications exist but, in general, their severity is not critical. Out of the 40 data points considered HARD in the dataset, 17 were correctly predicted. 18 Java classes were misclassified by one step as RATHER HARD, and only 2 resp. 3 were labeled as EASY or RATHER EASY.

Collectively, our experiments provide important insights: First, we can state that it is possible to utilize class-level ASTs to predict the maintainability of code classes after adapting the original code2seq framework. Second, we find it is possible to enhance syntactic AST paths with static code metrics to increase the prediction performance. Interestingly, it is not beneficial to add many additional features. The best results were obtained by adding only three metrics, namely the length of the longest method, the total size of the file, and its cognitive complexity. Noteworthy, the additional metrics contributed only 1.6 percentage points for F-Score and increased the mcc by just 0.023. As we will elaborate on in the threats to validity section, the size of the available dataset might be a limiting factor for the performance of code2seq.

In general, we find larger code2seq models to perform better than smaller models, and more sampled paths to improve against smaller sample sizes. Apart from these attributes, we observe only little performance differences which can be attributed to the hyperparameters of the framework. Sharma et al. [192] report similar results in their study on detecting code smells with encoder-decoder models.

7.4. Discussion and Threats to Validity

7.4.1. Discussion of the Results

In this study, we close a gap in recent studies, which focused only on static code metrics to classify software maintainability. Therefore, we adapted code2seq, a framework to predict nominal attributes of code at the method level, to predict ordinal maintainability ratings at the class level. One of our most significant findings is that rather simple metric-based models (cf. Chapter 6) outperformed these supposedly more sophisticated approaches based on embeddings of the AST.

Although neural networks typically require large datasets, our code2seq model outperforms distribution-based classifiers even when trained on only few hundred samples. Human performance, on the other hand, still poses an upper limit. Furthermore, the performance of code2seq could not improve against the results of metric-based models. Augmenting the input of code2seq with static metrics increases its mcc by 0.023 and its F-Score by 1.6 percentage points. This shows the potential of this approach for future research, as it enables the combination of syntactical features with static measurements. However, the gains in our experiments were only marginal.

The relevance of feature selection is corroborated by our findings as well. Using a high number of metrics as input did not increase the performance of a model. The best results are achieved using only three metrics to augment the AST embeddings in code2seq.

The maintainability label can adopt four ordinal values. A common approach for ordinal labels is to decompose the classification into a set of binary decisions, as proposed by [60, 129, 185]. We performed experiments using such a binary decomposition, too. Here, we investigated both binary code2seq models and the combinations of code2seq models with metric-based models for different steps of the decomposition. However, we did not find any improvement over the multiclass setting or the binary composition using only metric-based internal classifiers. Most likely, this can be attributed to the size and distribution of the dataset. For instance, there are only 40 positive samples available to train the binary decision *'is the label worse than* RATHER HARD'. This is particularly problematic for the large code2seq model.

7.4.2. Applicability of AST-based Maintainability Classification

Despite their success in other disciplines, AST embeddings could not compete with supposedly simpler metric-based models when predicting class-level maintainability. Notwithstanding this negative result, it is important to state that we do not question the applicability of code2seq in general. Our observations only refer to the given task and dataset. For the given ordinal multiclass classification problem and trained on a rather small, manually labeled dataset at the granularity of code classes, it could not achieve the same level of performance as reported for other tasks. Similar findings are reported by Sharma et al. [192], who examined the detection of code smells. Here, code2vec, the predecessor of code2seq, fell short of expectations, too.

Although the results are promising, it is possible that code2seq cannot yet play to its full strength due to the rather small dataset. As mentioned earlier, neural networks typically harvest large training sets. In some use cases, it is possible to reuse models, which have been trained on large datasets, which do *not* represent the task at hand. Later on, the model is fine-tuned to perform a different task, for which only a small dataset exists. Although pretrained code2seq models are available, downstream fine-tuning is not applicable in our case: First, we consider a different type of ASTs as input and also embed static metrics. Second, the composition of our adopted model differs from the default setup. Section 7.2.2 describes the adaptions in detail. Oversampling the data to increase the number of training samples, e.g. using SMOTE (cf. Section 6.2.1), is not possible with non-numerical features such as AST paths.

The usage of code2seq, in general, comes with several drawbacks. The network can only process a certain number of paths. By default, it considers 200 paths. When transitioning to class-level ASTs, the number of paths in the tree increases exponentially. As a consequence, the grid search had to consider significantly larger values of up to 2000 paths, i.e. increasing the previous default by a factor of 10. This is a trade-off. For large Java classes, this sample size is still very small. The largest class in the training data results in 14 million AST paths. Consequently, training and prediction are very dependent on the selection of these paths. Increasing the number of considered paths even more results in very sparse input for small Java classes. Also, a high number of input features makes the model prone to overfitting and results in longer training times. This poses a major barrier to using code2seq in this context. If there were an unlimited amount of data, one could train different models for classes with similar numbers of paths. This could mitigate the influence of path sampling and avoid sparse inputs. Unfortunately, this is not possible with the available data.

Taken together, this chapter provides important insights into using neural networks for maintainability classification. If the body of available maintainability ratings is increased in the future and the influence of the path sampling is mitigated, code2seq might develop into the most promising solution. As of now, the applicability of AST-based maintainability prediction is limited.

7.4.3. Threats to Validity

The biggest threat to validity for ML experiments is the used data and its train-test split. To avoid bias introduced by the split, we use cross-validation. As the models are later on used to analyze new projects, we mimic this scenario and use project-wise splits. To assist expert-based assessment, it is reasonable to refer to expert-labeled data. However, manually labeled data are prone to threats to construct validity, especially considering vaguely defined concepts such as maintainability. Also, the distribution of labels in the dataset has a major impact on the significance of the results [50]. To mitigate both data-related threats, we refer to the dataset introduced in Chapter 5 and available in [184]. It features a realistic label distribution and eliminates subjective bias by referring to the consensus of at least three experts per data point [183]. The average deviation of an expert from the consensus allows for an illustrative human-level performance baseline. Typically, datasets in this domain are very small. The used dataset contains only 519 samples, but still provides a significantly larger corpus than those used in related studies [103, 126, 137, 187, 212, 229].

In Section 7.2.2, we elaborated on the necessary adoptions to code2seq and its hyperparameters. Both code2seq and the metric-based models are optimized using a hyperparameter grid search. However, grid search can only find the best configuration within the specified grid.

7.5. Conclusion

Recent research has made advances to support maintainability assessments with machine-learned maintainability ratings. So far, these studies focus on classifiers based on static code metrics. To complement metric-based approaches, we implement an AST-based prediction concept, which has successfully been applied in other disciplines. In particular, we leverage code2seq, a framework originally designed to predict method names based on the AST. After adapting it to the ordinal ML task at hand, we compare its result to recent metric-based classifiers. Predicting maintainability ratings on a four-part scale, the AST-based classifiers reached an mcc of 0.45 compared to an mcc of 0.525 offered by metric-based models. However, the performance of the adopted code2seq model can profit from embedding both the AST and code metrics. Still, with an mcc of 0.473, it does not reach the performance of the solely metric-based model. Despite its success for some tasks in other disciplines, the present results indicate that AST-based classification models are hardly applicable to predict class-level maintainability. One limiting factor might be the size of the used manually-labeled dataset. Another problematic factor is the potentially high number of AST paths in program classes, which leads to a strong dependence on the sampled subset considered by the model.

Part IV.

Application in Practice

8. Machine Learning Assisted Maintainability Assessments: Application in Practice

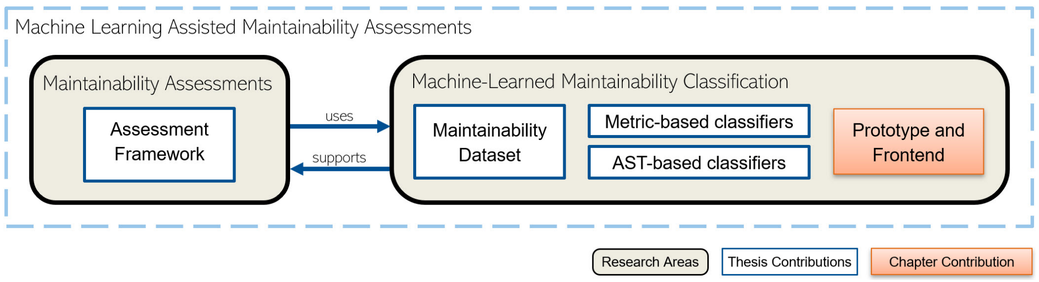


Figure 8.1.: Contribution of this chapter, i.e. prototypical tool support, in the context of the thesis

This chapter closes the last research gap and introduces prototypical tool support to be used in maintainability assessments. Figure 8.1 illustrates the contributions of this chapter in the context of the proposed framework.

In a variety of studies, ML has shown promising results to predict the maintainability rating of experts based on code metrics [81, 82, 165, 185, 187]. Our recent results suggest some models can almost be as accurate as average humans when predicting the consensus of several experts [185]. However, these studies miss an important facet: They do not demonstrate how and if their results transfer to real-life scenarios. In particular, it is unclear how the models can be integrated effectively into industrial maintainability assessments.

To verify the applicability of our approaches in practice, we constructed a tool prototype. It handles the data preparation, inference of the labels, and visualization of the results in a web-based frontend. Applying the tool to three industrial real-life projects, two experts from itestra GmbH verified that the provided machine-learned classifications are helpful to assess the technical condition of a system.

Problem: So far, approaches to predict software maintainability are only evaluated on a test dataset. In our experiments, the AST-based approaches achieved an mcc of 0.450 and could not equal the performance of the metric-based models, which achieved an mcc of 0.525. However, there is little evidence on whether this difference manifests in practice. Furthermore, it remains speculative how the models can be applied to facilitate maintainability assessments.

Solution: To shed light on how the results transfer to a real-world setting, we implemented a prototypical proof-of-concept tool. It integrates the extraction and preprocessing of the data, the inference of the maintainability ratings, and a web-based frontend. The tool features the best-performing metric-based (Chapter 6) and AST-based (Chapter 7) prediction models and offers several visualizations of the predicted maintainability. Eventually, we applied it to three commercial software systems provided by our industrial partner itestra GmbH.

Contribution: The application in practice revealed several insights: The experts report that both models could identify instances of subpar quality with high precision. However, they tend to have more confidence in the metric-based model as it found more instances of problematic quality. This corroborates the results observed in the previous experiments. In comparison to the state of practice, the experts reported faster overviews of the technical state of a system and easier prioritization of code files for further analysis.

Limitations: So far, the provided tool support is only prototypical and was only applied in exemplary case studies. In the future, this research could benefit from a large-scale case study to provide empirical evidence on the benefits of automated maintainability classification.

8.1. Selecting and Deploying Classification Models

In this chapter, we investigate the application of machine-learned maintainability classifiers in a real-world scenario. Here, we refer to Chapters 6 and 7 to select the most promising prediction models. Evaluated on our maintainability dataset, the best-performing model is a Random Forest classifier. It consists of 50 internal decision trees and uses SMOTE oversampling during training. Five static code metrics serve as the input: the size of the class, the length of the longest method, the average length of a method, the cognitive complexity of the class, and the total amount of lines within functions.

This metric-based model is complemented by the best AST-based code2seq model. It consists of two LSTM cells of size 128 each and uses a dropout keep probability of 0.6. In addition to the embeddings of 2000 paths from the AST, it also considers three code metrics: the length of the longest method, the size of the class, and the cognitive complexity. These configurations have been determined by the optimizations elaborated on in the previous chapters.

In Chapters 6 and 7, we used cross-validation to determine the best-performing model configuration. To employ the model in practice, this configuration is used to retrain the model on *all* available data points. To prevent the spilling of information from the new data into the prediction, every data-aware component has to be fitted purely on the training data. This also includes utilities such as the scaler used to normalize numerical metrics.

To store the code2seq model, we use Tensorflow's built-in functionality to save and restore checkpoints. For the Random Forest implemented in scikit-learn, we use pickle dumping and pickle loading.

8.2. State of Practice

Despite the promising results obtained in experiments on a dataset, there is still uncertainty about how machine-learned maintainability ratings can improve the state of practice. To evaluate the usefulness of the trained models in practice, we apply them to three commercial software systems in the context of expert-based maintainability assessments. This enables to investigate if the performance differences are reflected and relevant in practice. Furthermore, it allows for preliminary evidence on how ML classifications contribute to maintainability assessments.

Software quality HCs typically focus on the economic effectiveness and viability of a system [164]. At the end of the HC, the technical condition of the system, its ecosystem, infrastructure, and other factors are consolidated into long-term strategic recommendations [164, 188]. For the technical analysis, the expected maintenance effort is estimated by experienced experts. To do this, they gather tangible, reliable facts and concrete code examples to support their estimate. Here, the sheer size of the analyzed systems asks for an efficient analysis procedure. Quality analysts need fast and reliable results, which are easy to understand and ideally represent their own judgment. Common static analyzers are unsatisfactory, as they either provide only metrics or produces too many and hard-to-interpret warnings [94]. Although tools are used during expert-based HCs, a great share of code is still read and assessed manually.

The goal of this work is not to replace strategic advice with ML. However, we postulate that an automated classification may help prioritize and select code files for further examination. Drawing such samples is necessary, as the systems under consideration often reach several million lines of code in size. As the predicted classifications follow a four-part ordinal scale, they are more fine-grained than binary labels. This facilitates prioritizing the code files. Automated labeling can also help to rapidly create a rough assessment of the overall state. This is important to build early hypotheses on the condition of the system, which is helpful to steer the further process [188].

8.3. Study Setup

8.3.1. Study Projects

For this study, we applied the models to three commercial software projects, which are representative for quality audits at itestra GmbH.

Table 8.1 denotes the metadata of the projects, which are from the insurance and automotive domains. The shown size metrics consider only Java code and neglect test code or generated files. Notably, *Project-C* still accounts for 1.4 million lines of code. *Project-C* was developed between 2007 and 2012, *Project-A* is an ongoing development project started in 2021. Please understand that all names remain anonymized. Given the industrial environment of the application, it was not possible to include HCs of open-source software in this study.

Table 8.1.: Case Study Projects, anonymized

Alias	**Domain**	**# Java Files**	**LoC**	**Source LoC**
Project-A	Insurance	740	30k	24k
Project-B	Automotive	857	140k	104k
Project-C	Automotive	8,115	1,417k	731k

8.3.2. Methodology

The previous sections focused on the design and evaluation of the most promising ML models. In this section, we demonstrate how they can be deployed in an industrial context, for which tasks they can be applied, and how they can improve the state-of-the-art assessment process. However, the provided evidence can only be anecdotal. Exact quantifications of the observed improvements are not possible in this type of study. Hence, the selection of participants is crucial. For our study, we rely on two experts from the company itestra GmbH. Over the past two decades, the company has gained lots of experience in the field of software quality analyses. These analyses are performed by experienced engineers following scientific maintainability principles [164, 188] such as activity-based quality models [46] or concise and consistent naming [45]. The primary inclusion criterion for the participants was their experience. Both have participated as lead analysts in at least 15 maintainability assessments each. Through their empirical experience, they can give an informed opinion about the benefits and disadvantages of the tool prototype. However, for business reasons, we refrain from providing exact effort or monetary measurements of the analyses.

8.4. Tool Prototype

To efficiently use the trained models during an HC, we implemented a tool prototype. As of now, the tool is designed for file-level analysis of Java code. It features a web-based frontend that integrates the predicted classification, selected static metrics, and the code itself.

8.4.1. Toolchain

The toolchain consists of three stages: data preparation, inference of the labels, and visualization of the results. As both deployed models use static metrics as input, we need to analyze the code base first. Our tool takes the analysis report of ConQAT [44] as input. We chose this analyzer, as it is typically used by our industrial partner and thus does not induce any overhead. Unfortunately, the cognitive complexity metric is required by the metric-based models, but not part of the ConQAT metric suite. Though it is available via Sonarqube and other tools, we refrain from employing a second tool. Instead, we followed the specification of cognitive complexity provided in the appendix of [27] and implemented the metric on our own. Simultaneously, we create the AST for each Java file, sample the necessary paths, and store them temporarily. In the second phase, we instantiate the pre-trained models and utilities. They classify the Java code based on the static code metrics and preprocessed AST paths. Eventually, the predicted labels and used metrics are stored as a csv file for further inspection. Finally, the temporary files are deleted and the created csv file is loaded by an Angular web application.

In total, the preprocessing of a code file and the inference of the prediction takes less than 2.5 seconds on average on a local machine. However, this effort is required only once per analyzed project, similar to the overhead introduced through other analysis tools.

8.4.2. Frontend

The frontend visualizes the predicted maintainability labels and the code of the analyzed system. An online demo is available[1], which exemplifies the usage with *ArgoUML*, an open-source system. The frontend offers a project-level overview with aggregated KPIs, pie charts, and a treemap. Treemaps are commonly used to visualize two-dimensional, hierarchical data as rectangles. In our case, the size of a Java class corresponds to the size of the rectangle, the position of the rectangle corresponds to its location within the package structure of the project, and its color illustrates the predicted maintainability. Yellow color indicates one of the models classified the code as RATHER HARD, whereas red corresponds to at least one classifier predicting HARD or both predicting RATHER HARD. Also, two pie charts represent the proportion of code with problematic ratings. Hovering over a data point in the treemap displays the class name and its size; a left click opens the file. Code files can also be selected from a file tree, which uses the same color scheme to

[1] `https://markusschnappi_group.pages.gitlab.lrz.de/ml4swq-demo/`

highlight suspicious code files. When a file is opened, its code, the predicted ratings, and selected static metrics are shown. Furthermore, it offers the possibility to inspect which static metrics contributed to the Random Forest classification. The application offers several views, including a searchable and filterable table featuring the most important static metrics, e.g. size, cloning, and rating.

Figure 8.2 shows the treemaps of the study projects. For *Project-A*, most of its code has been rated inconspicuous and well-programmed. Meanwhile, those classes that should be inspected more closely are located in the same package and concentrated at the bottom right-hand corner. This allows efficient overviews and a convenient entry point to select which files to inspect in detail. For *Project-B* and *Project-C*, which are illustrated in the second and third treemaps, the first impression is far more pessimistic. In *Project-B*, quality flaws do not seem to be concentrated in specific hotspots and extend to the entire project. Using this information, the analysts can adapt the assessment process accordingly.

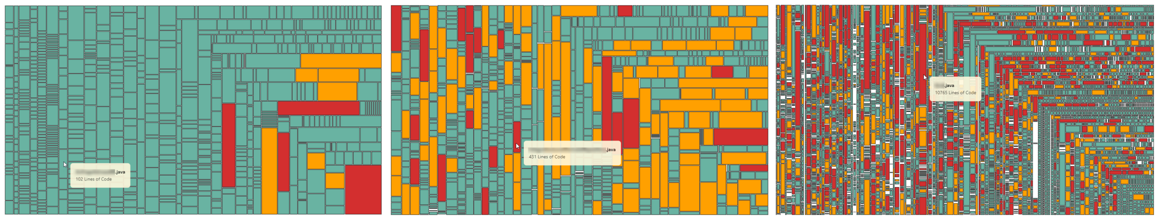


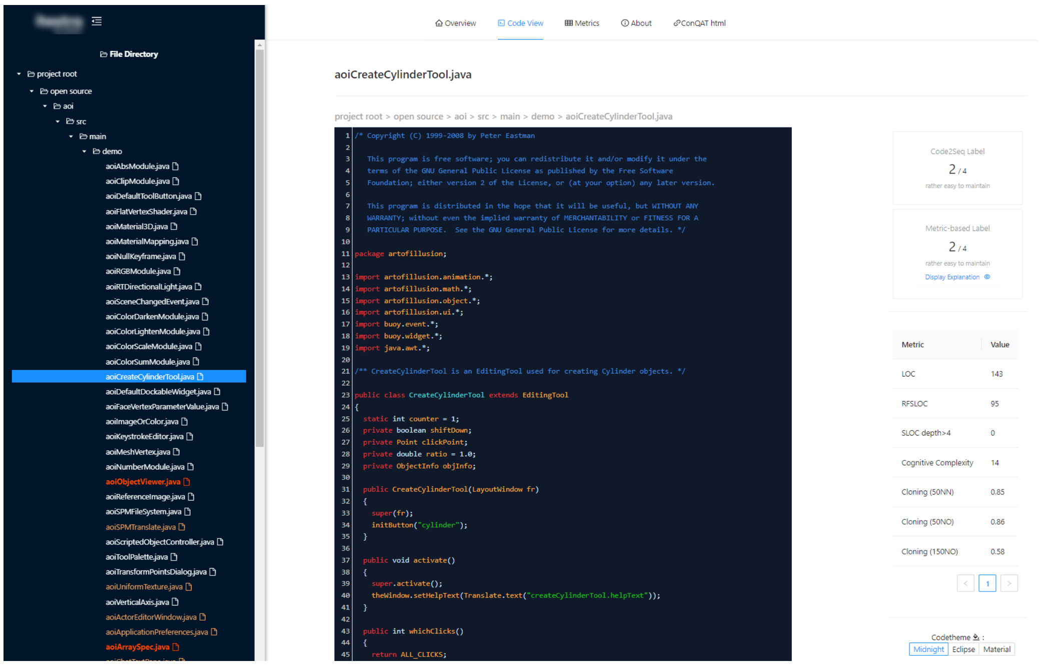
Figure 8.2.: Treemaps visualizing the rating and size of the code files in *Project-A*, *Project-B*, and *Project-C*, resp. (left to right).

Figure 8.3 shows a screenshot of the code view. In the file tree on the left, *aoiObjectViewer.java* and *aoiArraySpec.java* have been highlighted in particular, while some others have also been emphasized but are considered less critically. On the right, the classification results of the currently presented code are displayed.

The dataset used to train the algorithms considers only those aspects which are visible in the code of a Java class itself. Hence, the labels do not reflect, for instance, whether the class contains code clones. The importance of cloning for maintainability analyses is widely accepted [98, 179, 180]. Changes applied to one clone instance must be applied to all other instances as well, thus increasing the maintenance effort. Thus, for a holistic perspective, we supplement the predicted expert rating with cloning metrics.

8.5. Insights from the Application in Practice

The size of the audited systems, in general, stipulates the use of tools to guide which files to analyze manually during an HC. In this regard, the four-level rating scale is particularly useful, as it not only allows classification into good and problematic code but provides gradations to prioritize the files in a meaningful way.
Due to the size of the analyzed systems, the evidence collected during an HC must be



- project root
  - open source
    - aoi
      - src
        - main
          - demo
            - aoiAbsModule.java
            - aoiClipModule.java
            - aoiDefaultToolButton.java
            - aoiFlatVertexShader.java
            - aoiMaterial3D.java
            - aoiMaterialMapping.java
            - aoiNullKeyframe.java
            - aoiRGBModule.java
            - aoiRTDirectionalLight.java
            - aoiSceneChangedEvent.java
            - aoiColorDarkenModule.java
            - aoiColorLightenModule.java
            - aoiColorScaleModule.java
            - aoiColorSumModule.java
            - aoiCreateCylinderTool.java
            - aoiDefaultDockableWidget.java
            - aoiFaceVertexParameterValue.java
            - aoiImageOrColor.java
            - aoiKeystrokeEditor.java
            - aoiMeshVertex.java
            - aoiNumberModule.java
            - aoiObjectViewer.java
            - aoiReferenceImage.java
            - aoiSPMFileSystem.java
            - aoiSPMTranslate.java
            - aoiScriptedObjectController.java
            - aoiToolPalette.java
            - aoiTransformPointsDialog.java
            - aoiUniformTexture.java
            - aoiVerticalAxis.java
            - aoiActorEditorWindow.java
            - aoiApplicationPreferences.java
            - aoiArraySpec.java

aoiCreateCylinderTool.java

project root > open source > aoi > src > main > demo > aoiCreateCylinderTool.java

```java
/* Copyright (C) 1999-2008 by Peter Eastman

   This program is free software; you can redistribute it and/or modify it under the
   terms of the GNU General Public License as published by the Free Software
   Foundation; either version 2 of the License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful, but WITHOUT ANY
   WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
   PARTICULAR PURPOSE.  See the GNU General Public License for more details. */

package artofillusion;

import artofillusion.animation.*;
import artofillusion.math.*;
import artofillusion.object.*;
import artofillusion.ui.*;
import buoy.event.*;
import buoy.widget.*;
import java.awt.*;

/** CreateCylinderTool is an EditingTool used for creating Cylinder objects. */

public class CreateCylinderTool extends EditingTool
{
  static int counter = 1;
  private boolean shiftDown;
  private Point clickPoint;
  private double ratio = 1.0;
  private ObjectInfo objInfo;

  public CreateCylinderTool(LayoutWindow fr)
  {
    super(fr);
    initButton("cylinder");
  }

  public void activate()
  {
    super.activate();
    theWindow.setHelpText(Translate.text("createCylinderTool.helpText"));
  }

  public int whichClicks()
  {
    return ALL_CLICKS;
```

Code2Seq Label
2 / 4
rather easy to maintain

Metric-based Label
2 / 4
rather easy to maintain
Display Explanation

Metric	Value
LOC	143
RFSLOC	95
SLOC depth>4	0
Cognitive Complexity	14
Cloning (50NN)	0.85
Cloning (50NO)	0.86
Cloning (150NO)	0.58

Figure 8.3.: Screenshot of the tool used in the case study: Color-encoded file tree on the left, code at the center, additional static metrics on the right.

factual and reliable, but not necessarily complete. Hence, precision is preferred over recall when it comes to identifying quality flaws. If classes, which are considered RATHER HARD or HARD to maintain by the tool, do not actually pose maintainability issues, there is little value in the classification. Therefore, we asked the analysts to focus on suspicious code first and verify if the code indeed was problematic from their point of view. Afterward, a random sample of the remaining classes was analyzed and their predicted label was compared to the intuition of the experts.

Interestingly, for all manually analyzed code files which were predicted to be RATHER HARD or HARD, the users confirmed evidence of bad maintainability. In *Project-A*, which is a rather small project, all code samples with problematic ratings were manually validated. The experts echoed the ratings and confirmed quality flaws in all of them. For *Project-B* and *Project-C*, only samples were inspected due to the size of the projects. Here, the visualizations of the labels were particularly helpful to identify the most problematic parts of the systems. Then, we asked them to analyze random code samples and compare the predicted maintainability rating with their own assessment. After examining the provided maintainability ratings, both experts indicated a preference for the Random Forest model over the code2seq model. Though both models identify bad code with high precision, the Random Forest was observed to be less optimistic. Interestingly, their impression is backed up by empirical data. For 76% of all code files in the study systems, the predictions of both models agree, i.e. 7421 files out of 9712 were assigned the same rating by both models. In only 4% of the cases, the AST-based model suggests a worse rating than Random Forest. In contrast, the Random Forest is more pessimistic in 1858 cases, i.e. 19% of all files. The experts perceived this to correspond to a higher recall for quality flaws while offering comparable precision. Therefore, they had more confidence in the predictions of the metric-based model.

In total, the machine-learned classification supported the collection of evidence of bad maintainability and increased the efficiency of the analysis. It identified problematic code with little risk of misclassifications. This is in line with the results obtained on the dataset. Visualizing the ratings generates a rapid overview of the system's state.

8.6. Discussion

Until now, most studies on software maintainability prediction limited themselves to experiments on datasets. This study contributes insights from applying the models to three real-life commercial projects. The involved analysts report three key insights:

- the availability of the ratings facilitates the selection of files to inspect
- both deployed models can identify hard-to-maintain code with high precision
- the metric-based Random Forest model is preferred over the code2seq model

8.6.1. Application of the Prototype

During this doctoral thesis, a multitude of different classifiers has been investigated. However, the tool prototype employs only the most promising metric-based and AST-based models. Both models have been trained and evaluated on the same dataset and retrained for deployment following the state of the art. Employing more than two models may hinder the usability of the tool and distract the users. For this reason, we limited the scope to only two models.

Comparison of the models We applied both the code2seq model and the best performing metric-based model, i.e. the Random Forest, to the study projects. The experts confirmed maintenance problems in all manually-examined files with predicted maintainability issues, thus indicating a high precision toward problematic code. Interestingly, both analysts expressed a preference for the Random Forest and justified that with a perceived higher recall. However, given the amount of 9712 code files in this case study, there is no definite recall score available. The perceived recall is based on the observation that all investigated findings were correct, but that model provided more findings than the other.

This impression is validated by a comparison of the predictions. Across the three study systems, the Random Forest assigned a worse rating than the code2seq model to 1858 (19%)

of the code files. In contrast, code2seq predicted a worse rating for only 417 (4%) of the files. The remaining 7421 files were assigned the same label by both classification models. The experts did not know this statistical distribution when indicating their preferences.

Sampling However, given the size of the study systems, it was infeasible to validate the predictions for all 9712 code files. Therefore, as the use case stipulates the importance of precision over recall, the analysts focused on supposedly bad code first. Afterward, the experts analyzed a random sample of files and compared the predicted label with their own assessment. We argue this process is more congruent with their usual assessment and provides more relevant insights than just presenting a random selection of predictions. The latter was already accounted for in the experiments based on the dataset. The dataset is manually labeled by experts and a project-wise train-test split is used. Thus, the validation already corresponds to the scenario of showing a sample of predictions from a new project to experts. The main purpose of our study is to show how the models can be integrated into an established assessment process.

Frontend and Visualization Among others, this study set out to shed light on how ML maintainability classifications can support the assessment process. To make the classification results easily accessible to the analysts, we provide a web-based frontend. It shows both project overviews and the code of individual files. One visualization is the treemap. It incorporates the size and location of a Java class and its predicted maintainability. Yellow color shows that one model predicted RATHER HARD, red indicates one model considered the code HARD, or both predicted RATHER HARD. The color-encoded treemap was reported to be more beneficial than a treemap that only shows the size of a file. As it can be seen in Figure 8.2, there exist both large but unsuspicious instances as well as small yet hard-to-maintain code classes. In total, the treemap enables a rapid overview of the state of the system and an easy identification of problematic files. Furthermore, the possibility to open code directly from the treemap was appreciated by the users. As the code is viewed in the same frontend, there is no need to switch between a code editor and the tool report.

Selection of metrics to show The tool displays selected static metrics in addition to the classification. The experts decided on the selection of these metrics according to their company guidelines. Until now, they frequently switched between the report of static analysis tools and a code editor. To reduce the overhead, we display the code, predicted rating, and key static metrics in the same frontend. In consultation with the experts, we display the following measurements: size in LOC, size in redundancy free source LOC, source lines with nesting depth greater than four, cognitive complexity, and three variants of clone coverage. One expert also suggested showing the architectural context of a class. So far, this was not the focus of our work.

Version history As of now, the prototype analyzes only the current state of a project. This originates from the use case of HCs as post-release assessments. In contrast to continuous evaluations during development, there is often no history available. However, it is a straightforward task to run the analysis on different versions and monitor how the quality has evolved over time.

Explaining the predicted rating As a first foray to explain the predicted maintainability rating, we added Shapley Additive Explanations (SHAP) to the prototype. Section 9.1 elaborates on SHAP in more detail and explains how it can be used in the context of maintainability prediction.

In the present study, we only implemented this technique for the metric-based Random Forest model. Explaining the classifications of more complex models such as Neural Networks requires more complex explanations, too [9]. Even with the rather simple visualizations of the SHAP values for the Random Forest classifier, the quality analysts needed time to get familiar with them. One expert noted the provided information is interesting but does not sufficiently explain the decision for users without an ML background.

8.6.2. Threats to Validity

The employed ML models are trained to predict the maintainability of source code based on a human-labeled dataset. Inherently, manually labeled data is prone to subjective bias and tedious to create. To mitigate the effects of subjective opinions, we trained the classification models on a dataset that contains the consensus of three experts per data point [183, 184]. Unfortunately, this data collection leads to increased labeling effort and results in a limited amount of data. Still, the models are trained on 519 data points, which is a significantly larger corpus than those used by related works [81, 82, 103, 126, 137, 165, 187, 212, 229].

The generalizability of the insights from the application might be questioned as we only provide anecdotal, qualitative evidence, and rely on the feedback of only two analysts. Unfortunately, studies involving expert assessments do not allow for fully controlled environments. Ideally, we would have two identical teams analyze the same project, and vary the tool support used [14, 112]. However, this is infeasible for two reasons. First, single-project multi-team studies are economically challenging, especially if time-intensive expert work is required. Second, any observed efficiency differences could be attributed to the team members, not the used technique. Single-team multi-project studies are ruled out for similar reasons. Any observed difference could stem from differences in the analyzed projects. Due to these restrictions, our study focuses on a representative case study. Experienced experts are asked to evaluate the tool against a historical baseline when performing typical tasks, following the guidelines provided in [112]. Naturally, the validity of the conclusions is threatened by the choice of experts and study objects. The study participants are experts from itestra GmbH with a proven track record of more than 30 maintainability assessments. Still, we acknowledge experts from different companies might report different insights. Our study comprises only three projects but they are representative of typical audited systems at itestra GmbH.

8.6.3. Future Work

After the initial application of our prototype, we plan to establish it for regular use at itestra GmbH. This allows to consolidate the preliminary results and solidify them with long-term quantitative evidence. Explicit acceptance and rejection of predicted ratings can expand the used dataset. This way, the models can be periodically retrained with new data. While the provided tool support integrates seamlessly into the existing assessment process at itestra GmbH, integrating it into IDEs would enable using it during development, too. In addition to analytic feedback, the next step in quality evaluations could be automated refactoring to repair non-functional quality faults.

8.7. Conclusion

Using ML to predict the maintainability of source code has emerged as a potential alternative to time-consuming assessments by experts. However, most studies report results obtained on specific datasets and provide mostly theoretical arguments on whether and how such predicted ratings facilitate the assessment process. In particular, the previous chapters investigated the performance of metric-based and AST-based prediction approaches based on the same software maintainability dataset. Here, we observed a tendency of metric-based models to outperform code2seq models, which rely on the AST of a Java class.

To shed light on how these results transfer to a real-world setting, we implemented a prototypical analysis tool. It integrates the extraction and preprocessing of the data, the inference of the maintainability ratings, and a web-based frontend. This tool is applied to three commercial study projects in the context of maintainability assessments. The study projects span over 9.7 thousand Java classes with a combined size of 1.6 million lines of code. A team of two experts from itestra GmbH analyzed them following their established HC process. They validate that assessments are facilitated by the predicted ratings. Given the size of the analyzed systems, the experts used them to gain an overview of problematic code and to decide which files to focus their manual analysis on. In particular, they confirm that both models can identify hard-to-maintain code with high precision. However, the experts considered the metric-based model superior as it found more instances of bad quality. This corroborates the relevance of the performance differences observed in previous experiments.

Although it should not replace assessments by human experts, ML maintainability prediction has emerged as a valid alternative to generate overviews and prioritize which samples to inspect more closely.

Part V.

Conclusions

9. Emerging Results and Alternative Approaches

Parts of this chapter have previously appeared in a peer-reviewed publication ([190]) co-authored by the author of this thesis.

This chapter briefly discusses alternative approaches and results emerging from preliminary experiments to complement the main contributions of the thesis.

9.1. Explaining the Predicted Maintainability Rating

A major problem with the use of ML is the black-box nature of most algorithms [9]. Developers may even refuse to use a static analysis tool if the results of the tool lack understandability [94]. Especially in quality analyses, it is crucial to communicate problems in the system to the developers and the system owner with tangible evidence [164, 188]. Here, interpretable models can lead to increased trust in ML classifications by both the analyst and the customer [9].

9.1.1. Shapley Additive Explanations

Understanding why a prediction was made can help motivate actions that the prediction itself cannot. In particular, it is important to understand which features influenced the prediction, the magnitude of this influence, and the confidence of the model in its final prediction.

Shapley Additive Explanations (SHAP) can provide that information [133]. It is a model-agnostic technique to analyze a pretrained predictor without changing its structure or behavior. The analysis starts with the base value for the prediction probability, which would be used if no features were available. Then, additive feature attribution investigates how each feature manipulates the current output of the model, i.e. the prediction probability of a certain label. The result of impact analysis can then be visualized, for example, as bar charts or force plots to be easily interpreted by users [134].

9.1.2. Application in the Prototype

We offered these visualizations as a feature in the case study described in Chapter 8. Figure 9.1 shows the force plot of an exemplary classification. In this example, a code class was rated as hard to maintain. Overall, the confidence of the model is only 0.61. The main drivers are the cognitive complexity of the class and the length of the longest method, while the amount of LoC within functions had a mitigating effect. Maintainers can use this information as a starting point for improvements. In this case, they might prioritize refactorings that reduce the cognitive complexity and evaluate long methods for a lack of cohesion.

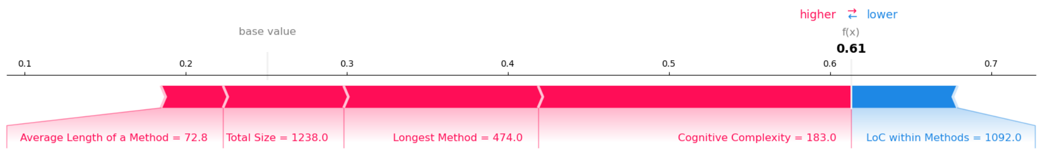


Figure 9.1.: Force plot visualizing the SHAP values leading to a class being rated as hard to maintain, the magnitude and direction of the feature influence, and the final confidence.

If the analysts were skeptical about a pessimistic rating at first, the SHAP visualization shed light on the prediction. In most cases, a high cognitive complexity value justified the classification. Inspecting the code more closely, the experts eventually confirmed the classification. The experts agreed that the SHAP values help to understand the predicted rating and indicate which aspects to examine the code for in more detail. Notwithstanding this advantage, the visualizations did not necessarily provide actionable advice on how the code should be changed to improve the rating.

We applied an implementation of SHAP for tree classifiers [132], which integrates well with the scikit-learn implementation of our Random Forest model[1]. Complex models, such as Neural Networks, require complex explanations [9]. Even ML experts often struggle with the interpretation of these explanations. Applying SHAP to large models with many input features, such as our AST-based code2seq model, is technically feasible. The target audience of such explanations are mostly the developers of the models with an extensive ML background [9]. This makes those techniques inapplicable to communicate explanations to project stakeholders. Please note, the code2seq model considers up to 2000 paths in the AST, while the Random Forest uses just five input features. To keep the models usable in practice, we refrained from adding interpretability mechanisms to the code2seq model. Thus, we only provide SHAP values for the purely metric-based model as a proof of concept.

9.1.3. Constructive Recommendations

Eventually, we derive constructive recommendations on how quality problems may be eradicated. As our first foray into such constructive feedback, we exploit the SHAP values of all four potentially predicted maintainability labels. This results in the following guidelines:

[1]`https://github.com/slundberg/shap`

- **A:** If a feature contributes positively to the current predicted rating and at the same time has a negative influence on better ratings, this feature should be specifically improved by refactorings.
- **B:** If there are no features where *A* applies, we recommend focusing on those having a positive influence on the data point being rated as HARD to maintain, independent of the current predicted rating.
- **C:** Analogously to *B*, we also recommend refactoring features revealing a negative influence on the data point being judged as EASY to maintain.

Please note, the goal of these guidelines is not to trick the prediction model into a more optimistic output but to pinpoint underlying quality issues in order to be genuinely addressed.

9.2. Using Image- and Text-Based Classification Algorithms

So far, this thesis focused on using either code metrics or the AST of a Java class to predict code quality. In a preliminary study [190], we investigated using text- and image-classification algorithms to classify the readability, understandability, and complexity of Java classes. Since readability and understandability are quality attributes of both natural language text and source code, we hypothesize these two attributes can be predicted by text classification algorithms. Furthermore, when analyzing and labeling the code used in our studies, software analysts already built a strong hypothesis based on their first impression of the code. Several experts confirmed they have been able to get an accurate intuition of the quality of a code snippet by looking at a visual representation of its overall structure, without going into syntactic or semantic detail. Training image-classification algorithm on images of syntax-highlighted mimics this process. Using the manually labeled dataset introduced in Chapter 5, we applied five text- and two image-classification algorithms to predict the readability, understandability, and complexity of source code. These are BERT [49], RoBERTa [130], CodeBERT [56]), text-based Naive Bayes, text-based SVM, image-based SVM, and AlexNet [119].

During the preliminary study, we identified several open challenges for future research. One major drawback of all approaches is a lack of semantic understanding of the code, which originates from the representation as images or text. Another challenge for the applicability of these approaches is currently posed by their need for fixed-size inputs. The image-classification algorithms assume a constant image size and the text classification models require text samples of fixed length. This requires a preprocessing of source code files of unbounded length and arbitrarily complex structure into fixed-size data points, which caused a deterioration of data quality. In particular, the partitioning of source code into fixed-length strings or fixed-size images did not match the granularity of the available labels. For these reasons, this dissertation did not investigate text-and image-classification in more depth.

10. Conclusion and Outlook

Parts of this chapter have previously appeared in other publications by the author of this thesis [183, 185, 187, 188, 189, 190].

This chapter concludes the doctoral thesis by summarizing the results and their limitations. Eventually, an outlook and suggestions for future work are provided.

10.1. Summary

This thesis proposes a framework to facilitate maintainability assessments through machine-learned classification models that predict the maintainability of source code based on the judgment of human analysts. Industrial quality assessments, referred to as HCs, analyze both the technical state of a system as well as its ecosystem and business context. Eventually, an experienced analyst proposes strategic recommendations to improve the long-term viability of the system. These can range from minimal refactorings to migration projects and complete redevelopments. Due to these far-reaching consequences, we refrain from fully automating the assessment process. Despite recent advances in ML, we believe the final assessment and strategic advice should remain in the hands of human analysts. Hence, the goal of this thesis is not to replace expert reviews but to support them with a framework for ML-assisted assessments. The proposed framework helps to quickly hypothesize the overall condition of a system and identifies examples of quality violations to focus the analysis on.

To develop reliable ML algorithms, a high-quality dataset is indispensable. This work presents a software maintainability dataset containing 519 human-labeled data points, i.e. Java classes. In total, 70 professionals from industry and academia assessed code from nine different projects and rated its readability, comprehensibility, adequate size, complexity, and overall maintainability. For efficiency reasons, the labeling was restricted to intra-class aspects, which are observable in the code of the evaluated Java class The submitted ratings reveal that disagreement between evaluators occurs frequently. To overcome these differences, we refer to the consensus of at least three participants per data point as the ground truth.

Using this dataset, we investigated several ML classifiers and different representations of source code as input. These models classify the maintainability of source code on a four-part ordinal scale. Models based on embeddings of the AST of a Java class could not reach the performance of models utilizing static code metrics as input. Supposedly naive metrics such as the size of a class and its methods are found to yield the highest predictive power towards maintainability. Collectively, a metric-based Random Forest model achieved the best classification results. It reached an mcc of 0.525 when predicting the consensus label in project-wise cross-validation. Analyzing the agreement of the individual experts and the consensus of all experts, we find the average human mcc is only marginally higher. In summary, our models achieved a prediction accuracy towards the consensus of the experts, which is comparable to an average human analyst.

To bring these experimental results closer to practice, we provide a prototype to be used in industrial maintainability assessments. It incorporates the trained models, explanations of the predicted label, and a web-based frontend. Applying it to three real-life software systems at our industrial partner confirmed our framework can facilitate the assessment process. Notably, hard-to-maintain code is identified with high precision. While the predicted maintainability ratings are useful to support human analysts, automated classification can and should not replace expert reviews entirely.

Overall Conclusion Expert-based maintainability assessments are accurate but tedious, while automated analyses are fast yet imprecise. This dissertation integrates expert-based maintainability assessments and machine-learned classification approaches. ML algorithms, especially metric-based models, can contribute to efficient expert-based maintainability analyses. Trained on manually labeled data, they can identify hard-to-maintain code with high precision and guide the experts on which code files to inspect in detail.

10.2. Limitations

Notwithstanding its contributions, this work is subject to certain limitations. These originate from the limited scope of a thesis and corresponding design restrictions, the used data, and the considered industrial perspective.

Scope We deliberately imposed some restrictions to keep the scope of this work concise. For instance, software quality consists of several subcharacteristics. In this thesis, we solely focus on the assessment of maintainability. In addition, we limit the assessment to source code, i.e., we refrain from taking the documentation, business model, data model, or deployment infrastructure into account. This is a necessary restriction, as the evaluation of these artefacts each represents a new research direction of its own. Considering the analyzed source code, we limited ourselves to Java code. Other programming languages offer different syntactical features, thus hindering the transferability of our learned models.

So far, we focused on the current state of a software system and did not consider its history. However, it is straightforward to apply our approach to past versions of the code and monitor how its maintainability has evolved over time. Until now, the proposed framework has only been applied in exemplary case studies offering anecdotal, qualitative evidence of its benefits. A natural progression of this work is to perform long-term studies and collect quantitative, empirical evidence. Due to the required time and resources, this was not possible during this dissertation.

Industrial Perspective This research was carried out as part of a joint research project of the Technical University of Munich and an industrial partner, itestra GmbH. Our view on the state of practice, the identified improvement potential, and the possible benefits of our contributions are highly influenced by this partner. We acknowledge experts from different companies might follow different processes, face different challenges, or report different insights. However, itestra GmbH has been performing software maintainability assessments for more than 15 years. Their experts regularly share their advances with the scientific community, e.g. in [139, 164, 204]. Apart from these reports, there is only little related work on the industrial practice of maintainability assessments (cf. Section 2.1.3).

Dataset Characteristics The biggest threat to the validity of our results originates from the used data. Typically, datasets concerning software quality are very small. Large datasets are difficult to create due to the required manual labeling. This thesis introduces a dataset consisting of 519 manually-labeled Java classes. Although this number is challenging for ML tasks, it still provides a significantly larger corpus than those used in related studies [81, 82, 103, 126, 137, 165, 187, 212, 229]. The code is sampled from nine different projects from both commercial and open-source development teams. For confidentiality reasons, only a subset of the study participants was allowed to analyze code from the commercial projects.

Using human-labeled data inherently is prone to threats to construct validity. In particular, vague concepts like software maintainability and its subattributes are susceptible to subjective bias. To mitigate this bias, we refer to the consensus of three study participants per data point. Although reaching a consensus by discussion is preferable, this was infeasible in our survey due to the limited availability of the study participants. Hence, we refer to an artificially calculated consensus. The Expectation-Maximization algorithm iteratively refines the weight of each submitted rating and its estimate of the final label until convergence. Concerning the overall maintainability label, it eventually assigns the consensus label with an average confidence of 89%. Still, its calculations are based on the subjective ratings submitted by the survey participants.

Granularity of the Maintainability Label In this thesis, we consider the maintainability of Java code files, which typically contain one Java class. This granularity is a trade-off. One could argue that ratings at the level of methods are more actionable, or that ratings at the module level are more holistic. However, it is difficult to evaluate a method without considering the class it is part of. Similarly, it is unclear how to aggregate lower-level findings into a module-level maintainability rating. In conclusion, we opted for class-level ratings for two reasons: First, it offers a reasonable compromise between the method level and the module level. Second, this granularity is used by most static analysis tools and is thus most familiar to the analysts.

The assigned maintainability label is based on the code of the class itself. Hence, it does not consider surrounding classes, export coupling, the inheritance hierarchy, or code cloning between different files. We are aware that these attributes may influence maintenance activities. However, we argue they are not properties of a class per se, but properties of the higher-level design. To facilitate the labeling process, we refrained from including this context for scalability and complexity reasons.

Focusing on Intra-Class Maintainability Our labeling procedure considers only intra-class characteristics directly observable in the code of a Java class. Inter-class aspects, which consider the broader context of a class, could not be taken into account as they would have made labeling more difficult and reduced the number of usable data points. Related studies fail to provide clear evidence to what extent inter-class aspects need to be respected in quality analyses. As several studies [29, 41, 42, 168] point out, the depth of the inheritance tree is an inadequate predictor for software maintainability. Dagpinar and Jahnke [41] also found little value in afferent coupling but acknowledge the influence of efferent coupling. The latter is visible in the code of the class and is thus covered by our maintainability label.

To corroborate the theoretical arguments for our granularity choice, we investigated the correlation between inter-class software metrics and maintainability in a Master's thesis [69] and a publication [70] advised and co-authored, resp., by the author of this thesis. We extracted several metrics focusing on the relationships between classes, such as coupling, inheritance, and cloning. Investigating the correlations between these measurements and the observed maintenance effort, we found mostly negative results. For some study projects and limited time intervals, selected inter-class metrics offered very high correlation coefficients. However, this only holds for certain time intervals and specific projects. Across all study projects, the correlation was moderate at most. This renders the examined inter-class relationships inappropriate predictors for software maintainability. Together with the facilitated labeling process and a higher number of labeled data points, we therefore consider it justifiable to focus mainly on intra-class maintainability.

Machine Learning Experiments The results of the presented ML experiments might be biased by the used dataset, chosen input features, and algorithms used. The limitations of the dataset are described above. In addition, the metric-based prediction is limited to the chosen tools and code metrics provided by them. Still, we employed a broad variety of tools and metrics from different families to avoid bias towards a specific metric suite.

In this thesis, we focused on those ML algorithms with the most convincing hypotheses and most promising results in our pre-experiments. Still, the hyperparameter grid search used to optimize the configuration of these classifiers did only consider the specified parameters.

10.3. Outlook and Future Work

Constructive Maintainability Repair Using explainable artificial intelligence techniques, future research can identify which parts of the code caused a bad rating. This information can be used to automatically suggest refactorings to improve the maintainability of the code. As a first foray into explaining the predicted rating, we implemented Shapley Additive Explanations [133] as part of the case study described in Chapter 8. Still, the feedback is analytical and provides little guidance on how the code should be adopted. It is a fruitful area for future research to augment the explanation of the current rating with constructive refactoring recommendations.

Aggregating Ratings at Different Granularities Future research may consider maintainability ratings on different levels. Fine-grained analyzers could identify specific code snippets causing high maintenance effort, while coarse-grained models evaluate the state of modules or complete systems. At each level, it can be examined how the labels at one level influence the label at higher levels. Can the quality of a project be judged if the quality of each class is known? More information on how to reason about the quality of packages or projects would help to establish a more holistic understanding of software maintainability.

Adequacy of the Implementation One drawback of our current classification approach is the lack of semantic understanding of the code. Though we can detect subpar code, our models cannot reason about its adequacy. Intuitively, the complexity of the task poses a lower limit to the complexity of the implementation. In the future, we suggest focusing more on semantic aspects of programs. Which problem is solved by the code, and is the code an adequate solution to this problem? In this work, the classification models report hard-to-maintain code without taking its adequacy into account. So far, this task remains with the human analysts.

Economic Sustainability The running costs of software systems include maintenance activities but are not limited to them. It is worthwhile to examine other relevant aspects and investigate the potential to automatically assess them. This includes the infrastructure and associated costs, performance bottlenecks, and general energy consumption. Future research may investigate which factors contribute to the sustainability of a software project. How can these factors be assessed, and what is their relationship to e.g. maintainability or functional correctness?

Architectures and Datamodels The quality of the data model is of utmost importance for the quality of a software system. In a Master's thesis supervised by the author of this thesis, we identified several antipatterns and best practices for the insurance domain [72]. These can be extended to other domains as well. Also, the architecture of the system can be evaluated in similar means. An interesting research question is to examine and validate the influence of these models and their quality on the source code and its quality.

Maintainability and Testability Most industrial HCs focus on non-functional quality attributes. Intuitively, hard-to-maintain code tends to be error-prone and hard to test as well. Future works could incorporate knowledge from defect prediction and testability research to reason about the maintainability of code. Furthermore, the intuitive relationships between non-functional and functional quality can be examined in more detail.

Bibliography

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15Th European Conference on Software Maintenance and Reengineering (CSMR*, pages 181–190. IEEE, 2011.

[2] Norita Ahmad and Phillip A Laplante. Employing expert opinion and software metrics for reasoning about software. In *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC)*, pages 119–124. IEEE, 2007.

[3] Jehad Al Dallal. Object-oriented class maintainability prediction using internal quality attributes. *Information and Software Technology*, 55(11):2028–2048, 2013.

[4] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4):1–36, 2018.

[5] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2018.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[7] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.

[8] Arthur Aron and Elaine Aron. *Statistics for the behavioral and social sciences*. Prentice Hall Press, 2002.

[9] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-López, Daniel Molina, Richard Benjamins, et al. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58:82–115, 2020.

[10] CT Bailey and WL Dingee. A software study using halstead metrics. In *Proceedings of the 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality*, pages 189–197, 1981.

[11] Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A probabilistic software quality model. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 243–252. IEEE, 2011.

[12] Rajiv D Banker, Srikant M Datar, Chris F Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–95, 1993.

[13] Victor R Basili and David H Hutchens. An empirical study of a syntactic complexity family. *IEEE Transactions on Software Engineering*, 1(6):664–672, 1983.

[14] Victor R Basili, Richard W Selby, and David H Hutchens. Experimentation in software engineering. *IEEE Transactions on software engineering*, 1(7):733–743, 1986.

[15] Kent Beck and Martin Fowler. Bad Smells in Code. *Refactoring: Improving the Design of Existing Code*, pages 75–88, 1999.

[16] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pages 202–211. ACM, 2014.

[17] Hans Christian Benestad, Bente Anda, and Erik Arisholm. Assessing software product maintainability based on class-level structural measures. In *International Conference on Product Focused Software Process Improvement*, pages 94–111. Springer, 2006.

[18] Sébastien Bertrand, Pierre-Alexandre Favier, and Jean-Marc André. Building an operable graph representation of a java program as a basis for automatic software maintainability analysis. In *The International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 243–248, 2022.

[19] Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar Çomak, and Leyli Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684, 2020.

[20] Barry W Boehm, John R Brown, and Hans Kaspar. *Characteristics of software quality*. North-Holland, 1978.

[21] Barry W Boehm, John R Brown, and M Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, ICSE '76, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press, IEEE Computer Society Press.

[22] Andrew P Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[23] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[24] Lionel C Briand, Jürgen Wüst, John W Daly, and D Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of systems and software*, 51(3):245–273, 2000.

[25] Manfred Broy, Florian Deissenboeck, and Markus Pizka. Demystifying maintainability. In *Proceedings of the 2006 International Workshop on Software Quality*, pages 21–26. ACM, 2006.

[26] Nghi DQ Bui, Lingxiao Jiang, and Yijun Yu. Cross-language learning for program classification using bilateral tree-based convolutional neural networks. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[27] G Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 International Conference on Technical Debt*, pages 57–58, 2018.

[28] G Ann Campbell and Patroklos P Papapetrou. *SonarQube in action*. Manning Publications Co., 2013.

[29] Michelle Cartwright. An empirical view of inheritance. *Information and Software Technology*, 40(14):795–799, 1998.

[30] J.P. Cavano and J.A. McCall. A framework for the measurement of software quality. In *ACM SIGMETRICS Performance Evaluation Review*, volume 7, pages 133–139, 1978.

[31] Ned Chapin. Do we know what preventive maintenance is? In *Proceedings of the 2000 International Conference on Software Maintenance (ICSM)*, pages 15–17, 2000.

[32] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[33] Lin Chen, Bin Fang, Zhaowei Shang, and Yuanyan Tang. Tackling class overlap and imbalance problems in software defect prediction. *Software Quality Journal*, 26(1):97–125, 2018.

[34] Shyam R Chidamber and Chris F Kemerer. Towards a metrics suite for object oriented design. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 197–211, 1991.

[35] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[36] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[37] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *IEEE Computer*, 27(8), 1994.

[38] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. Coherence of comments and method implementations: a dataset and an empirical investigation. *Software Quality Journal*, 26(2):751–777, 2018.

[39] José Pedro Correia, Yiannis Kanellopoulos, and Joost Visser. A survey-based study of the mapping of system properties to ISO/IEC 9126 maintainability characteristics. In *2009 IEEE International Conference on Software Maintenance (ICSM)*, pages 61–70. IEEE, 2009.

[40] CQSE GmbH. *Teamscale, the Official User Reference, Version 4.8*. CQSE GmbH, Munich, Germany, 2019.

[41] Melis Dagpinar and Jens H Jahnke. Predicting maintainability with object-oriented metrics-an empirical comparison. In *10th Working Conference on Reverse Engineering (WCRE)*, pages 155–155. IEEE Computer Society, 2003.

[42] John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2):109–132, 1996.

[43] Alexander Philip Dawid and Allan M Skene. Maximum likelihood estimation of observer error-rates using the em algorithm. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 28(1):20–28, 1979.

[44] Florian Deissenboeck, Elmar Juergens, Benjamin Hummel, Stefan Wagner, Benedikt Mas y Parareda, and Markus Pizka. Tool support for continuous quality control. *IEEE software*, 25(5):60–67, 2008.

[45] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[46] Florian Deissenboeck, Stefan Wagner, Markus Pizka, Stefan Teuchert, and J-F Girard. An activity-based quality model for maintainability. In *2007 IEEE International Conference on Software Maintenance (ICSM)*, pages 184–193. IEEE, 2007.

[47] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65(2):127–139, 2003.

[48] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[49] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *North American Association for Computational Linguistics*, 2018.

[50] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621. IEEE, 2018.

[51] Carsten F Dormann, Jane Elith, Sven Bacher, Carsten Buchmann, Gudrun Carl, Gabriel Carré, Jaime R García Marquéz, Bernd Gruber, Bruno Lafourcade, Pedro J Leitao, et al. Collinearity: a review of methods to deal with it and a simulation study evaluating their performance. *Ecography*, 36(1):27–46, 2013.

[52] Georgios Douzas, Fernando Bacao, and Felix Last. Improving imbalanced learning through a heuristic oversampling method based on k-means and smote. *Information Sciences*, 465:1–20, 2018.

[53] R. Geoff Dromey. Cornering the chimera [software quality]. *IEEE Software*, 13(1):33–43, 1996.

[54] S G Eick, T L Graves, A F Karr, J S Marron, and A Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[55] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[56] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[57] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[58] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–12, 2016.

[59] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V Mäntylä. Code smell detection: Towards a machine learning-based approach. In *2013 IEEE International Conference on Software Maintenance (ICSM)*, pages 396–399. IEEE, 2013.

[60] Eibe Frank and Mark Hall. A simple approach to ordinal classification. In *European Conference on Machine Learning (ECML)*, pages 145–156. Springer, 2001.

[61] Yoav Freund, Robert E Schapire, et al. Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning (ICML)*, volume 96, pages 148–156, 1996.

[62] Jerome H Friedman. Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378, 2002.

[63] Tony Gardner-Medwin and Nancy Curtin. Certainty-based marking (cbm) for reflective learning and proper knowledge assessment. In *Proceedings of the REAP International Online Conference: Assessment Design for Learner Responsibility. Glasgow: University of Strathclyde*, pages 29–31, 2007.

[64] D. A. Garvin. What does product quality really mean. *Sloan management review*, 25, 1984.

[65] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[66] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[67] Jan Gorodkin. Comparing two k-category assignments by a k-category correlation coefficient. *Computational biology and chemistry*, 28:367–374, 2004.

[68] Robert B. Grady and Deborah L. Caswell. *Software Metrics: Establishing a Company-wide Program*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[69] Lena Gregor. Investigating inter-class attributes for capturing software maintainability. Master's thesis, Technical University of Munich, University of Augsburg, Ludwig-Maximilian University Munich, 1 2022.

[70] Lena Gregor, Markus Schnappinger, and Alexander Pretschner. Revisiting inter-class maintainability indicators. In *2023 IEEE 30th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 805–814, 2023.

[71] Yann-Gaël Guéhéneuc. Ptidej: Promoting patterns with patterns. In *Proceedings of the 1st ECOOP Workshop on Building a System using Patterns.*, pages 1–9. Citeseer, 2005.

[72] Stefan Haas. Detecting smells in data models. Master's thesis, Technical University of Munich, 9 2021.

[73] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[74] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–39, 2014.

[75] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[76] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. Improved automatic summarization of subroutines via attention to file context. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, pages 300–310, 2020.

[77] Rachel Harrison, Steve J Counsell, and Reuben V Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496, 1998.

[78] Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. Multi-class adaboost. *Statistics and its Interface*, 2(3):349–360, 2009.

[79] Jane Huffman Hayes, Sandip C Patel, and Liming Zhao. A metrics-based software maintenance effort model. In *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR), 2004.*, pages 254–258. IEEE, 2004.

[80] Jane Huffman Hayes and Liming Zhao. Maintainability prediction: A regression analysis of measures of evolving systems. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 601–604. IEEE, 2005.

[81] Péter Hegedűs, Tibor Bakota, László Illés, Gergely Ladányi, Rudolf Ferenc, and Tibor Gyimóthy. Source code metrics and maintainability: a case study. In *International Conference on Advanced Software Engineering and Its Applications*, pages 272–284. Springer, 2011.

[82] Péter Hegedűs, Gergely Ladányi, István Siket, and Rudolf Ferenc. Towards building method level maintainability models based on expert evaluations. In *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*, pages 146–154. Springer, 2012.

[83] Konrad Heidler and Arnaud Fietzke. Remote sensing for assessing drought insurance claims in central europe. In *IGARSS 2019-2019 IEEE International Geoscience and Remote Sensing Symposium*, pages 7306–7309. IEEE, 2019.

[84] Lars Heinemann, Benjamin Hummel, and Daniela Steidl. Teamscale: Software quality control in real-time. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 592–595, 2014.

[85] Brian Henderson-Sellers, Larry L Constantine, and Ian M Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object oriented systems*, 3(3):143–158, 1996.

[86] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[87] R.C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63–91, 1993.

[88] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[89] Jan Marten Ihme, Franziska Lemke, Kerstin Lieder, Franka Martin, Jonas C Müller, and Sabine Schmidt. Comparison of ability tests administered online and in the laboratory. *Behavior Research Methods*, 41(4):1183–1189, 2009.

[90] ISO/IEC. ISO / IEC 9126-1 Software engineering – Product Quality – Part 1: Quality Model. Technical report, ISO/IEC, 2001.

[91] ISO/IEC. ISO/IEC 25010 - Systems and Software engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models. Technical report, ISO/IEC, 2010.

[92] Ashu Jain, Sandhya Tarwani, and Anuradha Chug. An empirical investigation of evolutionary algorithm for software maintainability prediction. In *2016 IEEE Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*, pages 1–6. IEEE, 2016.

[93] Yue Jiang, Bojan Cuki, Tim Menzies, and Nick Bartlow. Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 11–18, 2008.

[94] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[95] Magne Jørgensen. A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, 70(1-2):37–60, 2004.

[96] Magne Jørgensen, Dag IK Sjøberg, and Geir Kirkebøen. The prediction ability of experienced software maintainers. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR)*, pages 93–99. IEEE, 2000.

[97] Elmar Juergens, Florian Deissenboeck, Martin Feilkas, Benjamin Hummel, Bernhard Schaetz, Stefan Wagner, Christoph Domann, and Jonathan Streit. Can clone detection

support quality assessments of requirements specifications? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 79–88, 2010.

[98] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering (ICSE)*, pages 485–495. IEEE, 2009.

[99] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. Measuring software product quality: A survey of ISO/IEC 9126. *IEEE software*, 21(5):88–92, 2004.

[100] István Kádár, Péter Hegedus, Rudolf Ferenc, and Tibor Gyimóthy. A code refactoring dataset and its assessment regarding software maintainability. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 599–603. IEEE, 2016.

[101] István Kádár, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. A manually validated code refactoring dataset and its assessment regarding software maintainability. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, page 10. ACM, 2016.

[102] Amandeep Kaur, Sushma Jain, Shivani Goel, and Gaurav Dhiman. A review on machine-learning based code smell detection techniques in object-oriented software system(s). *Recent Advances in Electrical & Electronic Engineering (Formerly Recent Patents on Electrical & Electronic Engineering)*, 14(3):290–303, 2021.

[103] Arvinder Kaur and Kamaldeep Kaur. Statistical comparison of modelling methods for software maintainability prediction. *International Journal of Software Engineering and Knowledge Engineering*, 23(06):743–774, 2013.

[104] Frank Keller, Subahshini Gunasekharan, Neil Mayo, and Martin Corley. Timing accuracy of web experiments: A case study using the webexp software package. *Behavior research methods*, 41(1):1–12, 2009.

[105] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering (WCRE)*, pages 75–84. IEEE, 2009.

[106] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[107] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*, pages 305–314. IEEE, 2009.

[108] Taghi M. Khoshgoftaar and John C. Munson. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications*, 8(2):253–261, 1990.

[109] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE, 2021.

[110] Kenji Kira and Larry Rendell. A practical approach to feature selection. In *Ninth International Workshop on Machine Learning*, pages 249–256. Morgan Kaufmann, 1992.

[111] Barbara Kitchenham, Shari Lawrence Pfleeger, Beth McColl, and Suzanne Eagan. An empirical study of maintenance and development estimation accuracy. *Journal of systems and software*, 64(1):57–77, 2002.

[112] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE software*, 12(4):52–62, 1995.

[113] Ugur Koc, Parsa Saadatpanah, Jeffrey S Foster, and Adam A Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 35–42, 2017.

[114] Sotiris B Kotsiantis. Decision trees: a recent overview. *Artificial Intelligence Review*, 39(4):261–283, 2013.

[115] György Kovács. An empirical comparison and evaluation of minority oversampling techniques on a large number of imbalanced datasets. *Applied Soft Computing*, 2019.

[116] György Kovács. SMOTE-variants: a Python Implementation of 85 Minority Oversampling Techniques. *Neurocomputing*, 366:352–354, 2019.

[117] Stefan Kramer, Gerhard Widmer, Bernhard Pfahringer, and Michael De Groeve. Prediction of ordinal classes using regression trees. *Fundamenta Informaticae*, 47(1-2):1–13, 2001.

[118] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Estimating mutual information. *Physical review E*, 69(6), 2004.

[119] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[120] Lov Kumar, Aneesh Krishna, and Santanu Ku Rath. The impact of feature selection on maintainability prediction of service-oriented applications. *Service Oriented Computing and Applications*, 11(2):137–161, 2017.

[121] Lov Kumar, Debendra Kumar Naik, and Santanu Ku Rath. Validating the effectiveness of object-oriented metrics for predicting maintainability. *Procedia Computer Science*, 57:798–806, 2015.

[122] Niels Landwehr, Mark Hall, and Eibe Frank. Logistic Model Trees. *Machine Learning*, 95(1-2):161–205, 2005.

[123] Scikit learn User Guide. Matthews correlation coefficient. `https://scikit-learn.org/stable/modules/model_evaluation.html#precision-recall-f-measure-metrics`, 2022.

[124] Scikit learn User Guide. Precision, recall and F-measures. `https://scikit-learn.org/stable/modules/model_evaluation.html#precision-recall-f-measure-metrics`, 2022.

[125] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC)*, pages 184–195, 2020.

[126] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993.

[127] Yi Li, Shaohua Wang, and Tien N Nguyen. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 661–673. IEEE, 2021.

[128] Bennet P Lientz, E Burton Swanson, and Gail E Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[129] Yanzhu Liu, Adams Wai-Kin Kong, and Chi Keong Goh. Deep ordinal regression based on data relationship for small datasets. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2372–2378, 2017.

[130] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[131] FrontEndART Software Ltd. Sourcemeter. `https://www.sourcemeter.com`, 2020.

[132] Scott M Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. From local explanations to global understanding with explainable ai for trees. *Nature machine intelligence*, 2(1):56–67, 2020.

[133] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

[134] Scott M Lundberg, Bala Nair, Monica S Vavilala, Mayumi Horibe, Michael J Eisses, Trevor Adams, David E Liston, Daniel King-Wai Low, Shu-Fang Newman, Jerry Kim, et al. Explainable machine-learning predictions for the prevention of hypoxaemia during surgery. *Nature biomedical engineering*, 2(10):749–760, 2018.

[135] Lech Madeyski and Tomasz Lewowski. Mlcq: Industry-relevant code smell data set. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 342–347. ACM, 2020.

[136] Neil Malhotra. Completion time and response order effects in web surveys. *Public Opinion Quarterly*, 72(5):914–934, 2008.

[137] Ruchika Malhotra and Anuradha Chug. Software maintainability prediction using machine learning algorithms. *Software engineering: an International Journal (SeiJ)*, 2(2), 2012.

[138] Ruchika Malhotra and Kusum Lata. An empirical study on predictability of software maintainability using imbalanced data. *Software Quality Journal*, 28(4):1581–1614, 2020.

[139] Benedikt Mas y Parareda and Jonathan Streit. Software quality modelling put into practice. Technical report, itestra GmbH, 2008.

[140] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 4(4):308–320, 1976.

[141] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

[142] Mohammad Y Mhawish and Manjari Gupta. Predicting code smells and analysis of predictions: Using machine learning techniques and software metrics. *Journal of Computer Science and Technology*, 35(6):1428–1445, 2020.

[143] José P Miguel, David Mauricio, and Glen Rodríguez. A review of software quality models for the evaluation of software products. *International Journal of Software Engineering and Applications (IJSEA)*, 5(6):31–54, 2014.

[144] Subhas Chandra Misra. Modeling design/coding factors that drive maintainability of software systems. *Software Quality Journal*, 13(3):297–320, 2005.

[145] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2009.

[146] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. Software Quality Analysis by Code Clones in Industrial Legacy Software. *Science And Technology*, pages 87–94, 2002.

[147] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. An empirical validation of cognitive complexity as a measure of source code understandability. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2020.

[148] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality. In *2008 ACM/IEEE 30th International Conference on Software Engineering (ICSE)*, pages 521–530. IEEE, 2008.

[149] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 532–542, 2013.

[150] Mayra Nilson, Vard Antinyan, and Lucas Gren. Do internal software quality tools measure validated metrics? In *International Conference on Product-Focused Software Process Improvement (PROFES)*, pages 637–648. Springer, 2019.

[151] Steffen M Olbrich, Daniela S Cruzes, and Dag IK Sjøberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *2010 IEEE international Conference on Software Maintenance (ICSM)*, pages 1–10. IEEE, 2010.

[152] Paloma Oliveira, Marco Tulio Valente, and Fernando Paim Lima. Extracting relative thresholds for source code metrics. In *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR)*, pages 254–263. IEEE, 2014.

[153] Paul Oman and Jack Hagemeister. Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, 24(3):251–266, 1994.

[154] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188–1221, 2018.

[155] Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Landfill: An open dataset of code smells with public evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)*, pages 482–485. IEEE, 2015.

[156] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. A textual-based technique for smell detection. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

[157] D L Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 279–287. IEEE Computer Society Press, 1994.

[158] James W Paulson, Giancarlo Succi, and Armin Eberlein. An empirical study of open-source and closed-source software products. *IEEE transactions on software engineering*, 30(4):246–256, 2004.

[159] Karl Pearson. VII. Note on Regression and Inheritance in the Case of Two Parents. *proceedings of the royal society of London*, 58(347-352):240–242, 1895.

[160] Fabiano Pecorelli, Dario Di Nucci, Coen De Roover, and Andrea De Lucia. On the role of data balancing for machine learning-based code smell detection. In *Proceedings of the 3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation*, pages 19–24, 2019.

[161] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. Comparing heuristic and machine learning approaches for metric-based code smell detection. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 93–104. IEEE, 2019.

[162] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[163] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3):295–341, 1987.

[164] M. Pizka and T. Panas. Establishing economic effectiveness through software health-management. In *1st International Workshop on Software Health Management*, 2009.

[165] Nicolino J Pizzi, Arthur R Summers, and Witold Pedrycz. Software quality prediction using median-adjusted class labels. In *Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02*, volume 3, pages 2405–2409. IEEE, 2002.

[166] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, pages 73–82. ACM, 2011.

[167] Michael Pradel and Koushik Sen. Deep learning to find bugs. Technical report, TU Darmstadt, Department of Computer Science, 2017.

[168] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter Tichy. A controlled experiment on inheritance depth as a cost factor for code maintenance. *Journal of Systems and Software*, 65(2):115–126, 2003.

[169] PMD Open Source Project. Pmd source code analyzer. `https://pmd.github.io/`, 2020.

[170] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[171] Jef Raskin. Comments are more important than code. *ACM Queue Magazine*, 3(2):64–65, 2005.

[172] Darrell R Raymond. Reading source code. In *Proceedings of the 1991 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 3–16, 1991.

[173] Sandeep Reddivari and Jayalakshmi Raman. Software quality prediction: an investigation based on machine learning. In *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, pages 115–122. IEEE, 2019.

[174] Ulf-Dietrich Reips. Standards for internet-based experimenting. *Experimental psychology*, 49(4):243, 2002.

[175] John A Rice. *Mathematical statistics and data analysis*. Nelson Education, 2006.

[176] Marko Robnik-Sikonja and Igor Kononenko. An adaptation of relief for attribute estimation in regression. In Douglas H. Fisher, editor, *Fourteenth International Conference on Machine Learning (ICML)*, pages 296–304. Morgan Kaufmann, 1997.

[177] Tony Rosqvist, Mika Koskela, and Hannu Harju. Software quality evaluation based on expert judgement. *Software Quality Journal*, 11(1):39–55, 2003.

[178] Brian C Ross. Mutual information between discrete and continuous data sets. *PloS one*, 9(2), 2014.

[179] Chanchal Roy and James R Cordy. A survey on software clone detection research. Technical report, Queen's University at Kingston, Ontario, Canada, 2007.

[180] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[181] Willem E Saris and Irmtraud N Gallhofer. *Design, evaluation, and analysis of questionnaires for survey research*. John Wiley & Sons, 2014.

[182] Andrea Schankin, Annika Berger, Daniel V Holt, Johannes C Hofmeister, Till Riedel, and Michael Beigl. Descriptive compound identifier names improve source code comprehension. In *Proceedings of the 26th Conference on Program Comprehension (ICPC)*, pages 31–40. ACM, 2018.

[183] Markus Schnappinger, Arnaud Fietzke, and Alexander Pretschner. Defining a software maintainability dataset: Collecting, aggregating and analysing expert evaluations of software maintainability. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 278–289. IEEE, 2020.

[184] Markus Schnappinger, Arnaud Fietzke, and Alexander Pretschner. A software maintainability dataset. `https://figshare.com/articles/dataset/_/12801215`, Sep 2020.

[185] Markus Schnappinger, Arnaud Fietzke, and Alexander Pretschner. Human-level ordinal maintainability prediction based on static code metrics. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 160–169. ACM, 2021.

[186] Markus Schnappinger, Arnaud Fietzke, and Alexander Pretschner. Supplemental material: Human-level ordinal maintainability prediction based on static code metrics. `https://doi.org/10.6084/m9.figshare.14102843.v2`, June 2021.

[187] Markus Schnappinger, Mohd Hafeez Osman, Alexander Pretschner, and Arnaud Fietzke. Learning a classifier for prediction of maintainability based on static analysis tools. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC)*, pages 243–248. IEEE, 2019.

[188] Markus Schnappinger, Mohd Hafeez Osman, Alexander Pretschner, Markus Pizka, and Arnaud Fietzke. Software quality assessment in practice: a hypothesis-driven framework. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, page 40. ACM, 2018.

[189] Markus Schnappinger and Jonathan Streit. Efficient platform migration of a mainframe legacy system using custom transpilation. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 545–554. IEEE, 2021.

[190] Markus Schnappinger, Simon Zachau, Arnaud Fietzke, and Alexander Pretschner. A preliminary study on using text-and image-based machine learning to predict software maintainability. In *International Conference on Software Quality (SWQD)*, pages 41–60. Springer, 2022.

[191] Naeem Seliya and Taghi M Khoshgoftaar. Software quality estimation with limited fault data: a semi-supervised learning perspective. *Software Quality Journal*, 15(3):327–344, 2007.

[192] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software*, 176:110936, 2021.

[193] Tushar Sharma and Marouane Kessentini. Qscored: A large dataset of code smells and quality metrics. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 590–594. IEEE, 2021.

[194] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. Designite: A software design quality assessment tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, pages 1–4, 2016.

[195] Tushar Sharma and Diomidis Spinellis. Do we need improved code quality metrics? *arXiv preprint arXiv:2012.12324*, 2020.

[196] Martin Shepperd, David Bowes, and Tracy Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[197] Martin Shepperd and Darrel C Ince. A critique of three metrics. *Journal of systems and software*, 26(3):197–210, 1994.

[198] Tobias Simon, Jonathan Streit, and Markus Pizka. Practically relevant quality criteria for requirements documents. In *Software Engineering Research and Practice*, pages 115–121. Citeseer, 2008.

[199] Dag IK Sjøberg, Bente Anda, and Audris Mockus. Questioning software maintenance metrics: a comparative case study. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 107–110. IEEE, 2012.

[200] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2012.

[201] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information processing & management*, 45(4):427–437, 2009.

[202] Charles Spearman. "general intelligence" objectively determined and measured. *Studies in individual differences: The search for intelligence*, pages 59–73, 1961.

[203] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[204] Jonathan Streit and Markus Pizka. Why software quality improvement fails (and how to succeed nevertheless). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 726–735, 2011.

[205] Giancarlo Succi, Witold Pedrycz, Snezana Djokic, Paolo Zuliani, and Barbara Russo. An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. *Empirical Software Engineering*, 10(1):81–104, 2005.

[206] E Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, pages 492–497, 1976.

[207] Barbara G Tabachnick and Linda S Fidell. *Using multivariate statistics*, volume 5. pearson Boston, MA, 2007.

[208] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to data mining*. Pearson Education India, 2016.

[209] Mie Mie Thet Thwin and Tong Seng Quah. Application of neural networks for estimating software maintainability using object-oriented metrics. In *Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2003.

[210] Hatice Sancar Tokmak, Lutfi Incikabi, and Tugba Yanpar Yelken. Differences in the educational software evaluation process for experts and novice students. *Australasian Journal of Educational Technology*, 28(8), 2012.

[211] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 4–14. IEEE, 2018.

[212] Chikako Van Koten and AR Gray. An application of bayesian network for predicting object-oriented software maintainability. *Information and Software Technology*, 48(1):59–67, 2006.

[213] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25(2):1419–1457, 2020.

[214] Jeffrey Voas. Can clean pipes produce dirty water? *IEEE Software*, 4(4):93–95, 1997.

[215] Anneliese von Mayrhauser and A. Marie Vans. Program Comprehension During Software Maintenance and Evolution. *IEEE Computer*, 28(8):44–55, 1995.

[216] S. Wagner, K. Lochmann, L. Heinemann, M Kläs, A. Trendowicz, R. Plösch, A. Seidl, A. Goeb, and J. Streit. The Quamoco product quality modelling and assessment approach. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 1133–1142. IEEE Press, 2012.

[217] Romi Satria Wahono. A systematic literature review of software defect prediction. *Journal of Software Engineering*, 1(1):1–16, 2015.

[218] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[219] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[220] Jürgen Wüst. SD-Metrics: The software design metrics tool for UML. `https://www.sdmetrics.com/`, 2018.

[221] Fei Xing, Ping Guo, and Michael R Lyu. A novel method for early software quality prediction based on support vector machine. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 10–pp. IEEE, 2005.

[222] Aiko Yamashita and Steve Counsell. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software*, 86(10):2639–2653, 2013.

[223] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 306–315. IEEE, 2012.

[224] Jingxiu Yao and Martin Shepperd. Assessing software defection prediction performance: Why using the matthews correlation coefficient matters. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 120–129. ACM, 2020.

[225] Young Seok Yoon and Brad A Myers. An exploratory study of backtracking strategies used by developers. In *2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE)*, pages 138–144. IEEE, 2012.

[226] Ulaş Yüksel and Hasan Sözer. Automated classification of static code analysis alerts: a case study. In *29th IEEE International Conference on Software Maintenance (ICSM)*, pages 532–535. IEEE, IEEE, 2013.

[227] Wei Zhang, LiGuo Huang, Vincent Ng, and Jidong Ge. SMPLearner: learning to predict software maintainability. *Automated Software Engineering*, 22(1):111–141, 2015.

[228] Shi Zhong, Taghi M Khoshgoftaar, and Naeem Seliya. Unsupervised learning for expert-based software quality estimation. In *Eigth IEEE International Symposium on High Assurance Systems Engineering (HASE)*, pages 149–155. IEEE, 2004.

[229] Yuming Zhou and Hareton Leung. Predicting object-oriented software maintainability using multivariate adaptive regression splines. *Journal of systems and software*, 80(8):1349–1361, 2007.

List of Figures

List of Tables