Machine Learning for Software Refactoring: a Large-Scale Empirical Study

MSc. Thesis Computer Science

Jan Gerling

Machine Learning for Software Refactoring: a Large-Scale Empirical Study

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jan Gerling
born in Hamburg, Germany

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.se.ewi.tudelft.nl

Machine Learning for Software Refactoring: a Large-Scale Empirical Study

Abstract

Refactorings tackle the challenge of architectural degradation of object-oriented software projects by improving its internal structure without changing the behavior. Refactorings improve software quality and maintainability if applied correctly. However, identifying refactoring opportunities is a challenging problem for developers and researchers alike. In a recent work, machine learning algorithms have shown great potential to solve this problem.

This thesis used RefactoringMiner to detect refactorings in open-source Java projects and computed code metrics by static analysis. We defined the refactoring opportunity detection problem as a binary classification problem and deployed machine learning algorithms to solve it. The models classify between a specific refactoring type and a stable class using the metrics as features. Multiple machine learning experiments were designed based on the results of an empirical study of the refactorings.

For this work, we created the largest data set of refactorings in Java source code to date, including 92800 open-source projects from GitHub with a total of 33.67 million refactoring samples. The data analysis revealed that Class- and Package-Level refactorings occur most frequently in early development stages of a class, Method- and Variable-Level refactorings are applied uniformly during the development of a class. The machine learning models achieve high performance ranging from 80% to 89% total average accuracy for different configurations of the refactoring opportunity prediction problem on unseen projects. Selecting a high Stable Commit Threshold (K) improves the recall of the models significantly, but also strongly reduces the generalizability of the models.

The Random Forest classifier (RF) classifier shows great potential for the refactoring opportunity detection, it can adapt to various configurations of the problem, identifies a large variety of relevant metrics in the data and is able to distinguish different refactoring types. This work shows that for solving the refactoring opportunity detection problem a large variety of metrics is required, as a small set of metrics cannot represent the complexity of the problem.

Author: Jan Gerling
Student id: 4807367
Email: jgerling@student.tudelft.nl

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS (SERG), TU Delft
University supervisor:	Dr. M. Aniche, Faculty EEMCS (SERG), TU Delft
Committee Member:	Dr. Z. Erkin, Faculty EEMCS (CSG), TU Delft

Preface

First of all, I would like to thank my supervisor **Maurício Aniche** for the great support and advice he gave me during my thesis. I could always count on his great expertise, interest and enthusiasm. Such great support cannot be taken for granted and thus, I am very grateful to him.
Also, I want to thank my committee members Prof. dr. Arie. van Deursen and Dr. Zekeriya Erkin for their interest in my thesis and for being part of my committee. Furthermore, I want to thank Prof. dr. van Deursen for his great lecture "Software Architecture", which strengthened my interest in Software Engineering and taught me a lot.
Special thanks goes to Sarah for all her support, care and love. I am extremely happy to have you in my life. I would also like to thank my parents and sister for their love, patience and support.

Jan Gerling
Delft, the Netherlands
November 27, 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Refactorings are a cornerstone of modern day software development. Software projects evolve over time: As they grow in size and complexity, the internal design degrades, thus the overall quality of the software declines. Software refactorings tackle this problem, by enhancing the software quality without affecting its behavior [19]. Refactorings have been proven to improve software quality metrics [67, 13, 19, 36, 37] and are widely applied in practice [62]. Various tools support refactoring activities, by detecting refactoring opportunities and offering refactoring templates. Nonetheless, developers face various challenges in practice. They are concerned about the safety of a refactoring [37, 59], in adequate recommendations of refactorings [59] or the associated costs of a refactoring [36, 37, 59]. Developers struggle to identify refactoring opportunities and lack the support of superiors to perform refactorings [59]. Developers need adequate assistance in the identification of refactoring opportunities and simple strategies to convince their superiors of the necessity to perform a refactoring. Currently, developers rely on their intuition or tooling they describe as inadequate to do so.

1.1 Recommending Refactorings

Recommending refactorings is not a new field of research, various different approaches have been applied in the past. One approach using code smell heuristics was presented by Marinescu [44] in 2004. The author's approach consisted of a set of rules that can be applied to source code fragments in order to identify refactoring opportunities. Another approach presented in 2003 by Tourwe and Mens [68] was to detect code smells with logic meta programming and to combine these results with a static framework to suggest refactorings. Search-based algorithms have been used for the detection of refactoring opportunities as well as for ordering the refactorings to maximize the output. The detection is stated as an optimization problem, which is solved by identifying the best solu-

tions that increase a fitness function comprised of various heuristics, e.g. code smells [43]. Machine learning has not only been applied to various challenges in software engineering, such as code smell detection [6] or [15], but also for the prediction of software refactorings. More recent studies analysed the performance of machine learning algorithms for software refactoring detection or prediction. In 2017, Kumar and Sureka [39] created a machine learning pipeline with Least-Squares Support Vector Machine (LS-SVM) and Synthetic Minority Over-sampling Technique (SMOTE) to detect refactorings at class level from source code. A statistical model using cohesion measures for predicting the need of move method refactorings in a class was suggested by Al Dallal [1].

Lately, research by Aniche et al. [4] with the aim to *"evaluate the feasibility of using supervised machine learning approaches to identify refactoring opportunities"* was conducted. Their research showed the great potential of supervised machine learning algorithms for the prediction of software refactorings in Java source code. Their approach was to detect refactorings in open-source projects and extract source code metrics together with them. These metrics were then utilized for the training of six different supervised machine learning algorithms. The authors collected instances of classes that were not refactored but changed during the last 50 commits as negative training samples. The research showed that (i) the models generalize well to other contexts, in this case different developer communities and unseen projects, and often reach an accuracy above 90%. Furthermore, (ii) the research identified the great importance of process- and ownership metrics for the prediction quality, and (iii) the random forest classifier outperformed all other classifiers. The approach by Aniche et al. [4] differed from the other approaches due to the large data set considered in their research and the variability of different refactorings.

Nonetheless, the work by Aniche et al. [4] leads to various open questions and issues which include (i) data quality and scale, (ii) lack of understanding of the importance of process- and ownership metrics, (iii) superficial understanding of the source code metrics and refactorings, and (iv) further exploration of the utilized algorithms and their potential. Based on these findings this thesis pursues three main objectives which are explained in the following section. This work solely focuses on production code, as it is inherently different from test code.

1.2 Challenges and Approach

To accomplish the objectives of this research, various challenges were identified and the approach to overcome them is presented in this section. The three objectives of this thesis are:

1. **Create a large scale data set of refactorings**

2. **Analyze the feature distribution**

3. **Further explore supervised machine learning algorithms for refactoring prediction**

1.2.1 Data Collection

Extending up on the work of Aniche et al. [4], the first objective of this thesis is the creation of a large-scale refactoring data set. The data set will allow the in-depth analysis of the importance of process- and ownership metrics for the models and bolster the exploration of the algorithms potential. Researchers can further utilize this data to reproduce this thesis or build up on this work. Additionally, various questions regarding refactorings in open source projects can be tackled with this data set. In the initial approach, data collection suffered from various issues:

- **Incorrect process- and ownership metrics:** Class renames or moves were not tracked for process- and ownership metrics, thus a class was assigned new process- and ownership metrics after every *rename* or *move* refactoring. This raises questions regarding the importance and actual distribution of these metrics.

- **Unhandled exceptions:** A variety of unhandled exceptions occurred during data collection such as the code metrics extractor would fail on invalid classes or the over use of memory by RefactoringMiner.

- **Performance:** The extraction of a refactoring instance took roughly 1,61 seconds and about 3,34 seconds for a stable instance. This data mining speed does not allow the creation of a much larger data set with comparable resources.

- **Data sparsity:** For various refactorings only a few thousand instances were available, e.g. only 4.744 instances of *Extract Variable* refactoring or 7,273 instances of *Extract And Move Method* refactoring were detected in the largest subset. This might explain the lower performance of the classifiers for these refactorings.

Improving the existing data collection tool[1] by solving the before mentioned and additional issues, will allow the creation of a large-scale, high quality refactoring data set.

[1] https://github.com/refactoring-ai/predicting-refactoring-ml

1.2.2 Feature and Refactoring Type Distribution are largely unknown

None of the prior research on refactorings analysed the distribution of features (metrics and attributes) among different refactoring types or levels. An analysis of the feature distribution in the created data will be used to optimize the outcomes of the classifier training, asses the validity of the models and further analyze their outcomes. Furthermore, the research by Aniche et al. [4] collected stable commit instances with a thresholds of 25, 50 and 100 commits. The selection of the stable commit threshold is of great importance for the classifier, as it defines the negative samples for the training. Thus, an analysis of the feature distributions per commit stability threshold will help to identify risk and opportunities for the model training. The distribution of Stable-Instances within the data and the distribution of features is unknown for the different commit thresholds.

1.2.3 Major Factors for the Quality of the Model

As the prior research focused on evaluating the overall potential of machine learning algorithms for refactoring prediction, many questions regarding the quality and relevant factors arise. This thesis faces the following challenges:

- **Reproduction:** The prior experiment by Aniche et al. [4] will be repeated on the new/ larger data set. This will serve as a baseline for the evaluation of various changes made to data collection and machine learning pipeline. Also, analysing feature importances for field- and variable level refactorings might yield interesting insights.

- **Benefits of process- and ownership metrics**: The prior research identified the importance of process- and ownership metrics, but did not evaluate the benefits these metrics provide for the models. Furthermore, the question arises if a high-quality model for the prediction refactorings can be created without using process- and ownership metrics.

- **Selecting a stable commit threshold:** The selection of the commit threshold for negative samples might have severe consequences for the validity and quality of the models. Based on the results of the analysis of the feature distribution, different thresholds for commit stability are selected and their performance for creating models for refactoring prediction are evaluated.

- **Data set balance:** Imbalanced data sets are considered problematic for various machine learning algorithms [78], as they often perform best for

equally balanced classes. On the other hand, research on code smell detection questions the performance and quality of balancing approaches [54]. Thus, evaluating the effects of data-balancing on the classification performance might yield interesting new insights.

1.3 Research Questions

From the previously explained goals and challenges, the following research questions were developed and are addressed in this thesis:

Data Collection and Analysis

- **RQ 1: How are features distributed among refactoring types and levels?**
- **RQ 2: How are features distributed among Stable-Instances?**
- **RQ 3: What are implications of the distribution of process- and ownership metrics?**

Machine Learning

- **RQ 4: How does the prior approach perform with the new data set?**
- **RQ 5: How does the selection of the stability threshold effect classifier performance?**
- **RQ 6: How is the performance of the classifier for Field-Level refactorings?**
- **RQ 7: What are the effects of imbalanced training on the prediction quality?**

1.4 Contributions

This section outlines the major results and contributions of this thesis. The three main contributions to the field of Software Engineering of thesis are the following:

- A mature tool for large-scale refactoring mining for analysis and machine learning purposes was developed. This tool builds up on earlier work by Aniche et al. [4] which was extended and improved. The main extensions made to the tool are (i) the integration of RefactoringMiner (RM) v.2.x, (ii)

significant performance improvements by reworking the commit processing, (iii) significant improvements of the stability and reliability of the tool and (iv) extensive testing to ensure the validity of the results. Additionally, a great number of bugs was fixed, focusing on the metrics. CK.[2]

- A large scale data set of refactorings coupled with descriptive code metrics in open-source projects, containing 92800 projects for both test and production code was generated. A total of 28.93 million refactoring instances for production code and 4.75 million instances for test code were collected. Furthermore, the data set contains more than 65.96 million instances of stable classes.

- An in-depth analysis of the data set focusing on class metrics and attributes for Refactoring- and Stable-Instances and the distribution of process- and ownership metrics was conducted. The data analysis revealed that Stable-Instances are a unique subset of classes, which are highly maintained and have good class metrics.

- An in-depth analysis of RF and LR for the purpose of identifying refactoring opportunities and recommending refactorings was carried out using three experiments. We reproduced a prior experiment in the field, explored the selection of Stable Commit Thresholds (K) and imbalanced training data. The experiments show that the selection of the K has a great impact on the precision and recall of the prediction, that RF can indeed achieve great results in modeling the refactoring recommendation problem and that imbalanced training does not affect the model performance significantly.

1.5 Structure of this thesis

This thesis contains three main chapters regarding the results (i) Large-Scale Refactoring Data Collection (Chapter 3), (ii) Exploratory Data Analysis (Chapter 4) and (iii) Recommending Refactorings via Machine Learning (Chapter 5). Furthermore, the next chapters deals with related work (Chapter 2), whereas the last two chapters of this thesis focus on threads to validity Chapter 6, and conclusions and future work Chapter 7. The data collection chapter details the methodology used to create the large-scale refactoring data set and gives an overview of the data collection process. Furthermore, the chapter gives an overview of the data set and a brief explanation on how to use it. In the data analysis chapter we explore the refactoring operations, Stable-Instances and process- and ownership metrics. We analyze key metrics, identify clusters in the data and depict highly relevant data characteristics. Finally, we answer the

[2] https://github.com/mauricioaniche/ck

Research Questions 1 to 3 in the chapter. In the machine learning, we explore the use of machine learning for refactoring recommendation. We lay out the details of the classifier training for refactoring recommendation, describe three experiments, discuss their results and answer Research Questions 4 to 7.

Chapter 2

Related Work

2.1 Refactorings

Software refactorings were formally introduced in the early 1990s by Griswold [24] and Opdyke [52]. Later, Fowler introduced a collection of 72 refactorings to improve the design of existing code [19]. Fowler defined software refactorings as *"the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."* [19, p. 9]. This definition is used for refactoring in this work.

Software refactorings have two main purposes: First, to improve maintainability by improving the internal structure, e.g. architecture and design patterns [67, 13, 19], and to improve extensibility by adding flexibility and simplifications [58]. Software developers, on the other hand, have a large range of motivations to perform refactorings:

- readability [37]
- maintainability [37, 53]
- extensibility [37, 53, 62]
- bug removal [37, 53, 62]
- removal of code smells [37]

2.2 Refactoring Detection

The history of refactoring detection from multiple revisions of source code dates back to 2000 when the first heuristic-based approach was published by Demeyer et al. [14]. Other early approaches include the detection of clones to identify *Rename* and *Move* refactorings by Van Rysselberghe and Demeyer [75] and origin analysis for *Merge* and *Split* refactorings by Godfrey and Zou [22]. All of

the early approaches suffered from generalization issues and mediocre performance, thus later approaches for refactoring detection mostly focused on structural changes in the source code. This approach was first proposed by Xing and Stroulia [77] in 2006 and was based on their UMLDiff algorithm [76]. UMLDiff detects structural changes between two revisions of source code, such as renames, moves or removals. Another structural approach is deployed by Ref-Finder which detects a large variety of refactorings (63) from the syntax-tree of two source code revisions by using logic invariants defined for each refactoring [35]. RM was proposed by Tsantalis et al. [71] and Silva et al. [62]. It uses UMLDiff to detect changes in the syntax tree and it uses rules to detect refactorings instead of heuristics. The RefDiff algorithm presented Silva and Valente [61] uses heuristics on code metrics computed by static analysis to detect 13 refactorings.

Comparison of Refactoring Detection Tools In 2019 Tan and Bockisch [66] evaluated four refactoring detection tools: (i) RefactoringCrawler, (ii) Ref-Finder, (iii) RM v.1 and (iv) RefDiff. Their evaluation shows that RM v.1 is by far the best tool with issues on *Move Class* and *Rename Package* refactorings. The results of Tan and Bockisch [66] show very high precision for both RefactoringCrawler (95.9%) and RM (93.2%) clearly declassing RefDiff (40.3%) and Ref-Finder (61.1%). RM shows also the highest recall of the four evaluated tools with 72.9%. A more in-depth analysis of the individual refactorings detected by RM the authors revealed that RM has very low recall for the *Move Class* (33.8%) and *Rename Package* (11.4%) refactorings, which the authors explain by the confusion of these two refactorings. *Move Class* refactorings were often classified as *Rename Package* and the other way around.

RefactoringMiner v.2.x In 2020, Tsantalis et al. [73] released version 2.0 of RM, this version exceeds the performance of the earlier version with an average precision of 99.4% and a recall of 93%. Furthermore, the new version solves the issues reported by Tan and Bockisch [66] for *Move Class* and *Rename Package* refactorings and supports 55 refactoring types. Additionally, RM v.2.0 is the fastest refactoring detection tool with a median of 44 milliseconds and mean of 253 milliseconds to process a commit, in comparison to version 1.0 of RM, which takes on median 101 milliseconds or RefDiff 2.0 with a median commit processing time of 113 milliseconds [73].

RefactoringMiner detects refactorings for two revisions of Java source code, e.g. a commit and its parent commit from a git history, and returns a list of detected refactorings [72]. RM detects refactorings by matching statements in the source code and applying a multitude of rules for the detected changes between the matched statements. In the first phase of the matching, code fragments with a similar signature are matched in a top-down approach, starting at the class

level proceeding to the method and field level. In the second phase, the remaining elements are matched bottom-up. On these remaining elements, a multitude of rules is applied to detect refactorings and classify them correctly. In order to achieve an optimal outcome, the refactoring types are ordered by their spatial locality. Refactorings not moving code are processed first, e.g. *Rename Class* or *Change Return Type*, followed by refactorings moving code within the same entity, e.g. *Extract Variable* and lastly between entities, e.g. *Extract Superclass*.

2.3 Refactoring Recommendation

Refactoring Recommendation solves two subproblems of the field of automatic software refactoring: (i) the identification of refactoring opportunities and (ii) the selection of the correct refactoring [46]. Various approaches to recommend refactorings have been proposed by researchers, three major classes of these approaches are (i) heuristics or rules, (ii) search-based recommendation and (iii) machine learning. Many tools detect code smells and see these as a refactoring opportunity for specific refactorings [44, 28, 63, 42, 18].

Metrics-Based and Rule-Based Approaches Metrics-based smell detection uses manually defined thresholds for code metrics that were collected via static analysis from the source code to identify specific code smells [44, 60]. In 2009, Moha et al. [48] proposed DECOR, a framework to define specifications for code smells, and DETEX, a detection technique based on DECOR. A popular tool using a combination of rules and metrics to detect code smells and recommend refactoring is JDeodorant developed by Fokaefs et al. [16] in 2007, which was further extended in the following years. In its latest state, JDeodorant can be added as a plugin to the Eclipse IDE and detects feature envy [16], long methods [69], god classes [70], duplicated code [38], and handles state and type checking issues [70].

Search-Based Approaches Search-based refactoring recommendation applies the concepts of search-based software engineering (SBSE), which were defined by Harman and Jones [28] in 2001. In the SBSE, many problems are defined as optimization problems, which are solved with meta-heuristic search algorithms such as genetic, hill climbing or tabu search algorithms [27]. Popular implementations of SBSE in practice are refactoring, and software maintenance [30]. A variety of search-based refactoring recommendations have been developed in recent years [29, 47, 51]. An early application of SBSE for refactorings was suggested by Harman and Tratt [29] with an algorithm applying the concept of pareto optimality. A refactoring would only be selected if at least one relevant metric would improve and no metric decrease. A dynamic refactoring

recommendation tool was developed by [47]. The tool from Mkaouer et al. [47] makes suggestions to developers based on recent code changes and uses their feedback on those suggestions to improve it. Research has shown that the definition of the fitness function for the search-based problems is often a key issue, as the function might be ill-defined and results in unwanted and/ or biased outcomes [3, 42, 63]. Simons et al. [63] found that using only metrics for a fitness function for refactoring automization does not yield sufficient results and they strongly recommend including the engineer in the process.

Machine Learning Approaches Fontana et al. [17] suggested machine learning algorithms for code smell detection, the authors argue that machine learning algorithms more objectively evaluate the relevance of a code smell. In a later work Fontana et al. [18] evaluated various machine learning algorithms on their performance on code smell detection (data class, large class, feature envy and long method). The authors found that already a few hundred samples are sufficient to achieve an accuracy of 95% on the selected code smells and that RF perform best in this task. An approach to utilize modern deep learning techniques to detect feature envy and recommend *Move Method* refactorings was proposed by Liu et al. [42]. They used a neural network to map code features with code smells. The features included textual and structural features that were converted with Word2Vec. Aniche et al. [4] identified the great potential of machine learning algorithms for refactoring recommendations by creating models able to predict 20 different refactoring types. For their work they trained six binary classifiers with code metrics gathered from refactorings and separated Refactoring-Instances from stable classes. They created data sets of refactorings combined with code metrics describing the source code at the moment of the refactoring for 11149 open-source projects. They defined the concept of a stable class as a class that was changed in 50 commits but not refactored. The results of Aniche et al. [4] show an accuracy of 90% for the best performing classifier, RF, in separating between Refactoring-Instances and stable classes. Furthermore, this approach incorporates the software engineer's motivations for refactorings into the refactoring recommendation.

Chapter 3

A Large-Scale Refactoring Data Collection

This chapter describes the data collection process and gives an overview of the generated large scale refactoring data set. First, the tool developed for the data collection is explained in detail and afterwards the actual data collection. Second, the validation of the data set is described. Third, an overview of the refactoring data set is given, by showing descriptive statistics for Refactoring- and Stable-Instances. Last, a brief explanation for the data usage is done.

3.1 Methodology

The first objective of this research was the creation of a large refactoring dataset, see Section 1.2. The details of the selected approach are presented in this section. The data set of refactorings was generated in four steps: (i) Project Selection, (ii) Refactoring Detection, (iii) Metric Extraction and (iv) Data Merging and Storage. An overview of the research methodology for the classifier training is given in Fig. 3.1.

3.1.1 Selection of Projects

As of June 2019, GitHub has more than 22 million public projects containing Java source code [23].[1] These are too many projects to mine all of them, thus we selected **100.000** public projects with the highest watcher count. For this study, we only considered open-source projects, because their data and history can be accessed freely. GitHub users can watch a repository, which means they receive updates on discussions, releases and events. In contrast to starring a project on GitHub, the user receives detailed information on issues and the development process. Thus, we assume the user is more likely interested in

[1] Computed with GHTorrent 2019

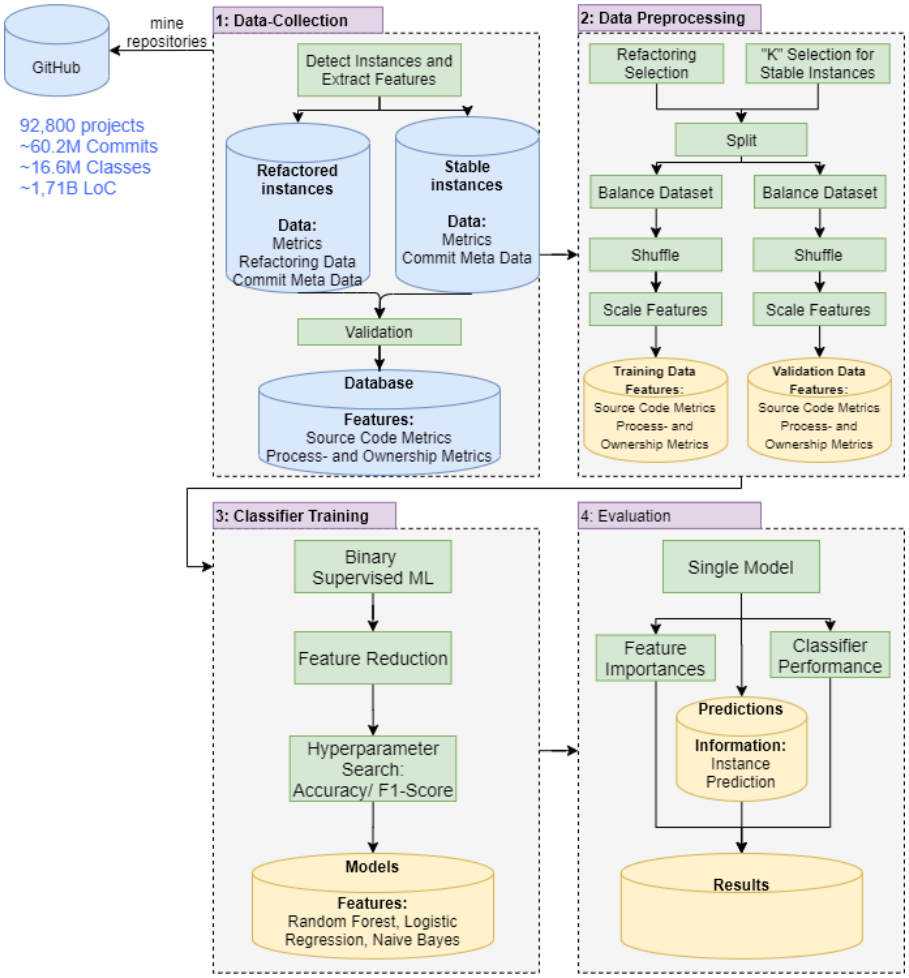


Figure 3.1: Overview of the Classifier Pipeline used in this research, excluding Data Analysis

the development and progress of the project. Based on this, we assumed that the project is relevant to many active software developers. For this reason, we chose to select projects based on the watcher count. Furthermore, we used the latest release, at the time, of GHTorrent [23] (June 2019) to collect the projects, which only provided the watcher count for projects.

We only selected GitHub projects for this data set, as GitHub is by far the largest hoster of open-source software projects in the world, which are hosted for free. [2] Furthermore, Aniche et al. [4] already generated refactoring databases for projects on F-Droid[3] and the Apache Software Foundation (ASF). Many of those projects are also hosted on GitHub.

[2] https://github.com
[3] https://f-droid.org/

3.1.2 Refactoring Detection

Detecting refactorings is a key component for the data set generation. We used version 2.0 of RM (RM) to fulfil this task [73]. RefactoringMiner detects refactorings for two revisions of Java source code, e.g. a commit and its parent commit from a git history, and returns a list of detected refactorings [72]. RM detects refactorings by matching statements in the source code and applying a multitude of rules for the detected changes between the matched statements. RefactoringMiner is currently the best refactoring detection tool for Java code, with version 1.0 having an average precision of 94% for its 15 supported refactorings [66]. This is exceeded by Version 2.0 with an average precision of 99.4% and a recall of 93%[73]. Additionally, version 2.0 of RefactoringMiner supports more than 40 refactoring types and fixed various issues that occurred on version 1.0, e.g. the confusion of "Rename package" and "Move Class" refactorings. Also, RefactoringMiner 2.0 is the fastest refactoring detection tool with a median of 44 milliseconds and mean of 253 milliseconds to process a commit, in comparison to version 1.0 of RM, which takes on median 101 milliseconds or RefDiff 2.0 with a median commit processing time of 113 milliseconds [73]. RefactoringMiner is the fastest and by far the most accurate refactoring detection tool, which combined with its wide variety of detectable refactorings (55) are the main reasons why we choose it. For more details on refactoring detection tools and RefactoringMiner see Section 2.2. For the data collection we used **version 2.0.1** of RefactoringMiner.

3.1.3 Metric Computation

The key features to the classifier training are these metrics: (i) source code metrics, (ii) process metrics and (iii) ownership metrics. These types of metrics have been used in previous research for prediction tasks in software engineering [33, 2, 40, 15, 57, 4].

Source Code Metrics. Source code metrics comprise a multitude of metrics with the goal to describe and measure the software in question. The fields of interest include among other fault prediction, code quality, code complexity and maintainability. More than 300 source code metrics are known [50], e.g. simple ones such as Number of Public Fields, Number of Methods or more complex ones such as LCOM or RFC.

We used CK[4] to extract the source code metrics. CK is a static code analysis tool which extracts a large variety of source code metrics and attributes from Java source code on four different levels (class, method, variable and field) [5]. Furthermore, we implemented two additional cohesion metrics LCC and Tight

[4] https://github.com/mauricioaniche/ck

Class Cohesion (TCC) for classes to CK. All in all, we extracted **70 metrics and attributes** from Java source code on four different levels for the training.

Level	Count	Features
Class	**47**	**Metrics and Attributes (40):** AnonymousesQty, AssignmentsQty, Cbo, ComparisonsQty, LambdasQty, LCOM, LOC, LCC, LoopQty, MathOperationsQty, MaxNestedBlocks, Nosi, NumberOfAbstractMethods, NumberOfDefaultFields, NumberOfDefaultMethods, NumberOfFields, NumberOfFinalFields, NumberOfFinalMethods, NumberOfMethods, NumberOfPrivateFields, NumberOfPrivateMethods, NumberOfProtectedFields, NumberOfProtectedMethods, NumberOfPublicFields, NumberOfPublicMethods, NumberOfStaticFields, NumberOfStaticMethods, NumberOfSynchronizedFields, NumberOfSynchronizedMethods, NumbersQty, ParenthesizedExpsQty, ReturnQty, RFC, StringLiteralsQty, SubClassesQty, TryCatchQty, UniqueWordsQty, VariablesQty, WMC, TCC, isInnerClass **Process Metrics(3):** qtyOfCommits, bugFixCount, refactoringsInvolved **Ownership Metrics(4):** qtyOfAuthors, qtyMajorAuthors, qtyMinorAuthors, authorOwnership
Method	**68** (21 + 47) method + class	AnonymousClassesQty, AssignmentsQty, Cbo, ComparisonsQty, LambdasQty, Loc, LoopQty, MathOperationsQty, MaxNestedBlocks, NumbersQty, ParametersQty, ParenthesizedExpsQty, ReturnQty, Rfc, StringLiteralsQty, SubClassesQty, TryCatchQty, UniqueWordsQty, VariablesQty, Wmc, startLine
Variable	**69** (1 + 21 + 47) variable + method + class	Appearances
Field	**48** (1 + 47) field + class	Appearances

Table 3.1: Features collected for the classifier training with their total count per level

Process- and Ownership Metrics. Instead of code quality metrics, process metrics (PM) were found to perform very well for defect prediction [15, 57, 56, 49]. Process metrics, also referred to as change metrics, incorporate information about the history of changes instead of information about

the changes themselves. They include the following metrics:

- **Quantity of Commits** - number of commits affecting a class file
- **Refactorings Involved** - number of detected refactorings for this class file, each refactoring is counted individually without considering the refactoring type, thus multiple refactorings can be accounted for on the same commit
- **Bugfix Count** - number of assumed bug fixes. A simple pattern matching algorithm attempts to detect bug fixes in the commit message, by checking for these keywords: `bug, error, mistake, fault, wrong, fail, fix`
- **Lines Added** - the total number of lines added in the latest commit for the class file.
- **Lines Deleted** - the total number of lines deleted in the latest commit for the class file.

Both lines added and deleted were not collected, because the metrics were often found to be incorrect during testing.

Process metrics are often coupled with ownership metrics, ownership metrics give further information about the state of ownership of a class file. We used the definition of ownership metrics by Bird et al. [9]:

- **Quantity of Authors** - total number of authors with commits affecting the class
- **Quantity of Major Authors** - number of authors with more than than 5% of all commits changing the class
- **Quantity of Minor Authors** - number of authors with more less than 5% of all commits changing the class
- **Author Ownership** - percentage of commits of the main contributor to this class

All process- and ownership metrics were extracted for a specific class file. Contrary to the previous research of Aniche [5], the process- and ownership metrics were still tracked in case the class file was moved to another directory or renamed.

3.1.4 Data Collection Tool

In short, our data collection tool processes all repositories by merging the data from RefactoringMiner, CK and the other metrics into refactoring or Stable-Instances, and stores them in a database. The data collection process is similar for each repository, every repository is handled individually. Thus, the data collection can be easily parallelised in order to mine many repositories at the same time. Please note, the data collection tool can log many details of the data extraction, but for the sake of simplification logging will not be covered in this section. The main components and processing steps of the data collection tool will be explained in the following.

Entire Repository

A simplified overview of the major processing steps for a single repository is shown in Fig. 3.2. A repository is processed in multiple phases, from which the first one is the initialization of the current project.

The repository is cloned to the local machine with its entire history, relevant meta-data is extracted (project name, GitHub URL, first and last commit id, size in bytes, commit count) and the history is reversed. Therefore, the oldest commit comes first and the latest commit last. In case an exception occurs during this phase, e.g. cloning fails because the repository is no longer (publicly) available, the repository will not be processed further and all relevant details are logged.

In the second phase each commit of the history is processed individually, by iterating over the entire history. For each commit, all Refactoring- and Stable-Instances are extracted and merged with relevant meta data and metrics. To store this data, a new database transaction object is created, which contains all extracted instances of the current commit, and if the commit was processed successfully the transaction is committed to the database. If an unhandled exception occurs during the commit processing, e.g. RefactoringMiner times out after 120 seconds, the transaction is rolled back and the next commit is processed. Most exceptions are handled during processing of a single commit and are not escalated up to this level. The processing of a single commit is explained in more detail later in this section.

In the last phase, once all commits were processed, the project data in the database is updated with a finishing date, the repository is removed from disk and relevant information is logged.

Single Commit

The processing of each commit consists of four main steps: (i) refactoring detection, (ii) refactoring metric extraction (iii) process- and ownership metrics up-

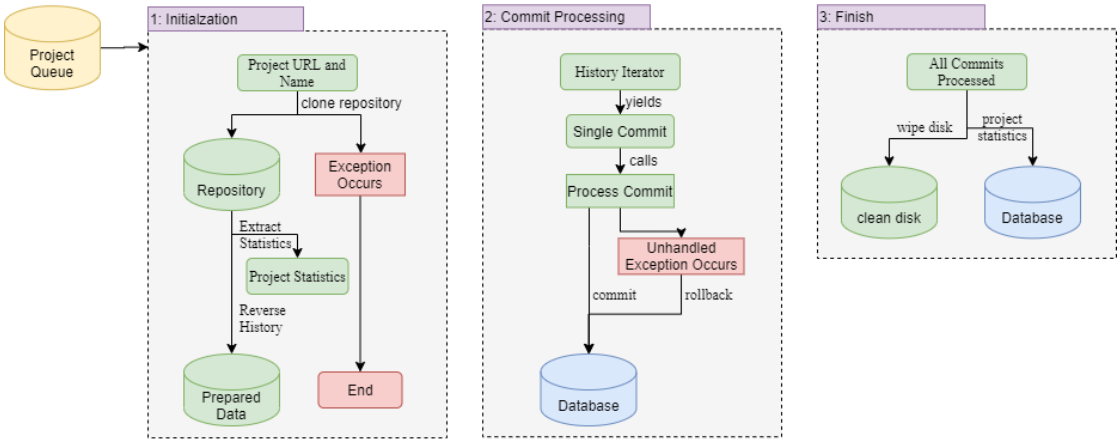


Figure 3.2: Simplified flow diagram of the data collection main loop describing the processing of an entire repository

date, and (iv) stable instance detection. A simplified overview for this is given in Fig. 3.3. During this phase of the data collection, refactoring- and Stable-Instances are extracted. An instance contains all metrics and relevant meta data describing the state of the source code and change history at a certain revision.

In the first step, RefactoringMiner detects the refactorings made by the current commit, compared to the previous one. Therefore, the tool passes the Java source code of the current commit and the previous one to RefactoringMiner, and a list of refactorings is returned. The first commit is skipped in this step, because RefactoringMiner could not detect any refactorings on this. In very rare occasions, RefactoringMiner would take very long, more than 600 seconds to handle a single commit. Thus, we set the time out for RefactoringMiner to 120 seconds, which proved to be a very reliable upper boundary during various test runs.

A second step happens if RefactoringMiner found some refactorings in commit. A new refactoring instance is created and persisted for each detected refactoring with the following data:

- **Source code metrics**: CK extracts all source code of the previous revisions code.
- **Process- and ownership metrics**: these are tracked by the ProcessMetricsTracker class.
- **Meta data**: the commit meta data is added to the instance.

After the refactorings are processed, the process- and ownership metrics are updated. Furthermore, if a class was renamed or moved, the commit tracker is

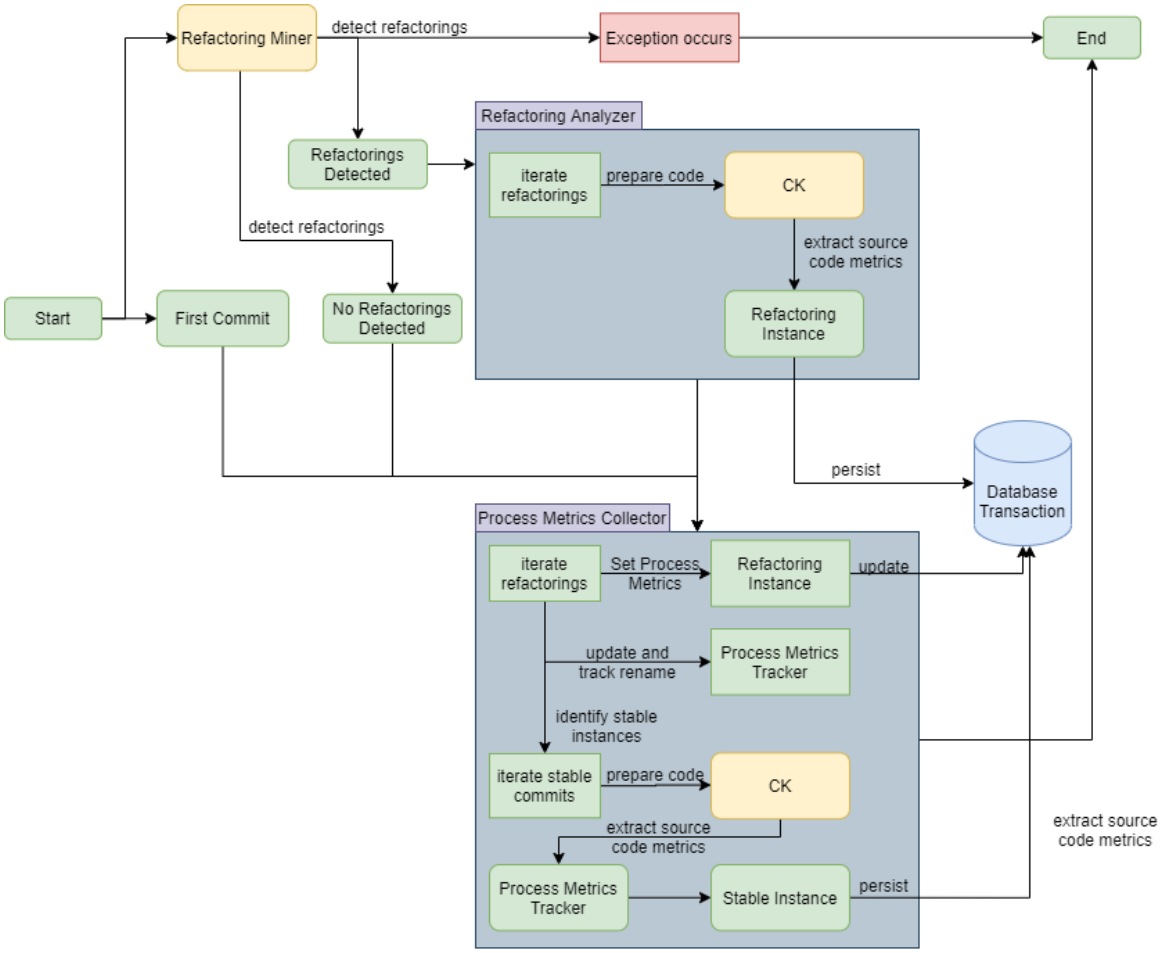


Figure 3.3: Simplified flow diagram of the processing of a single commit

updated, because the process- and ownership metrics are tracked via the fully classified file path of a class. If a class is renamed it is therefore very important, to also update the tracker.

In the final step of the commit handling, Stable-Instances are detected. A commit is considered stable if it was not refactored during the last k commits changing it. We selected a large variety of K's *(15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80, 90, 100)* that could be considered stable, in order to later investigate when a class can be considered stable. If a class was considered stable for multiple k's they all refer to the same commit. This is either the first commit introducing the file or the last commit with a refactoring on it. Therefore, the metrics for a stable instance for class "A" are equal for `K=15` and `K=20`.

The data collection tool utilizes Hibernate to maintain its data model. Hibernate is connected to an external SQL database via the jdbc connector.

3.2 Data Collection Process

In this section, the execution of the data mining is described in detail. Descriptive statistics, the mining environment and encountered issues are presented. During the data collection, **100000 projects** were mined in two batches of 50.000 projects each. All in all, more than **60,2 million commits** were processed. In total, for all machines combined, the data collection took about **69 weeks** (11594,3 hours). The average processing time of a commit was **692 milliseconds**.

3.2.1 Mining setup

We used 20 Ubuntu 18.04 LTS VMs with each 1 GB Ram, 1 CPU Core and 20GB disk space for the data collection. The data was stored on another VM with 1 CPU Core and 500GB disk space, which was extended during the data collection process from 150GB to 500GB to handle the huge amount of data. The individual data collection processes were encapsulated in Docker container.

3.2.2 Errors and Exceptions

The data collection process was thoroughly logged with log4j[5] and these logs were analyzed in order to ensure the success of the data collection and to evaluate the robustness of the created tool. The generated logs were automatically analyzed with a custom tool[6] focusing on the progress of the data collection and important errors and exceptions during the data collection. Despite the efforts, a small fraction of the logs was lost, about 9-12% of the logs were either overwritten or otherwise corrupted. About 7.200 projects were not processed, because either the URL for the projects was incorrect or in most cases the project was no longer (publicly) available. The reasons for why a project was no longer available were not explored. For 1.261 projects of the 92.800 projects (1,36%), the data collection was not successfully finished. Thus, after the failing commit no new data was added to the database. This was partly caused by 30 OutOfMemoryErrors (OOME), which caused the entire tool to crash. The crashed projects differ in commit- and file count, and also many other characteristics. We noticed docker container rarely failed due to unknown reasons, and we would have to restart them.

Exceptions occurred at three major levels (i) Docker container, (ii) main loop and (iii) commit processing, for details on the levels see Section 3.1. In case the tool crashed, the application was restarted by a bash script and a new project was processed. Combined, 16992 exceptions occurred in the main loop or escalated up to there. This number was extracted from the database and is not

[5] log4j
[6] LogAnalyzer

affected by the lost logs. The most prevalent issue on this level were time out exceptions caused by RefactoringMiner (15282 exceptions), because RefactoringMiner would not finish processing a single commit after 600 seconds. From the logs, 211 other exceptions were identified that were escalated up to the App level. The most prevalent exceptions and their handling are described in the following:

- **RefactoringMiner and CK miss-match (93903)**: The class names or method identified in a refactored class by RefactoringMiner could not be found by CK in the java source code. In this case, the refactoring was skipped, but the process metrics tracker was still updated.

- **Class name extraction (4799):** The tool failed to extract the canonical name of a class and could therefore not fetch the source code for the analysis by CK. The affected refactorings were skipped.

Level	**Refactoring**	**Total Count**	**Unique Count**	% **Unique**
Class	Total Unique	4260567	4250266	99.76%
Method	Total Unique	9756226	9494440	97.32%
Method	Change Parameter Type	3264206	3030073	92.83%
Variable	Total Unique	7810164	7640260	97.82%
Variable	Parameterize Variable	96467	92881	96.28%
Field	Total Unique	3580129	3571855	99.77%
Other	Total Unique	4802696	4802449	99.99%

Table 3.2: Unique entities refactored per refactoring level and with single entity refactorings below 97% unique entities

Overall, the validation of the collected refactorings yields three main results: (i) the confidence in the collected refactorings is high due to the very low frequency of detected incorrect instances, (ii) the "Change Parameter Type" refactoring is the only refactoring for which we detected inconsistent data and (iii) the Stable-Instances show no signs of errors or flawed data points.

3.3 Data Cleaning and Validation

The validity of the data was ensured in a multi-step process: (i) rigorous testing of the data collection tool, (ii) data-cleaning, (iii) manual validation, and (iv) inconsistency validation.

3.3.1 Data Cleaning

Due to a mistake in the project selection, forks of projects were included in the data collection. These forks were later identified by using GHTorrents database to check if a project is a fork and these forks were cross-validated and complimented with the mined history of the projects. Duplicate entries in the project history were detected by checking for two or more projects sharing at least one commit, meaning commit id and date are equal. In total **7269** out of the total 92800 projects were marked as forks in the data set. All refactoring- and Stable-Instances were moved into distinct tables and thereby, separated from the main data set. In later occasions always the main, non-forked data is referenced. The removal of forks affected **15.39 million refactoring** instances and **46.24 million stable** instances.

Furthermore, we detected Refactoring-Instances with a very high frequency of refactoring on the same class file. This either represents anomalies or an error by RefactoringMiner. An instance classified for this criteria, if a single class suffered from more than 50 refactorings on the same commit. All of the identified **19.44 million** (5.77%) instances were then marked as invalid and are in later references to the data set not included. An analysis shows a small fraction of refactorings is affected heavily by this measure, especially "Inline Method" (47.85%), "Rename Attribute" (35,10%), "Move And Inline Method" (35.26%) and "Push Down Method" (25.90%). The selection of 50 refactorings is somewhat arbitrary, it was selected based on the distribution of refactorings per class on a single commit.

As mentioned in Section 3.2.2, for a small fraction of projects the data collection do not finish. The collected samples from these projects, 12.68% of the non-forked refactorings and 27.12% of the Stable-Instances, were not removed. The collected refactorings are still valid and so are the stable-instances. Additionally, we could not detect a bias towards specific refactorings or refactoring levels in the set of unfinished projects. For the Stable-Instances, the fraction of instances from unfinished projects for the method, variable and field is very similar ranging from 27.14% to 28.4%, but for class level instances it is 21.85%.

3.3.2 Data Validation

The data set was manually verified by checking 40 randomly selected refactoring and Stable-Instances each. During the manual inspection, the individual instances and all its metrics were inspected in detail. First, the actual instances were verified, e.g. the refactoring was detected and stored correctly, then the metrics and attributes were inspected for anomalies, and finally a random selection of metrics and attributes was recalculated manually in order to verify their correctness. The manual inspection of the refactoring- and Stable-Instances did not yield any incorrect samples or metrics, except for the Lines Added and Lines

Deleted process- metrics which were subsequently removed from the data and also from the data collection tool. For these two metrics, there was a strong miss-match between reported and detected line deletions or additions. [7]

In addition to the manual validation and removal of forks, the data set was further validated by evaluating the fraction of unique entities affected by a refactoring type. Most of the refactorings mined only affect one entity[8], e.g. "Change Parameter Type" only affects a single method, thus we validated if the same entity suffered multiple times the same refactoring on a commit in the same project. The analysis shows [9] that for class, field and the "other" level, the fraction of unique entities for single entity refactorings is above 99,76%, see Table 3.2. For method level refactorings it is slightly lower with 97.32%, due to many incorrect samples for the "Change Parameter Type" refactoring. Also for the variable level refactorings, the fraction of unique entities refactored reaches 97.82%. This analysis shows further that the number of duplicate entries is very low, as the Refactoring-Instances rarely affect the same entity multiple times.

3.4 Refactorings

This section describes the mined refactorings in the data set. In the further data description we are only considering valid refactorings and exclude data points from forked projects.

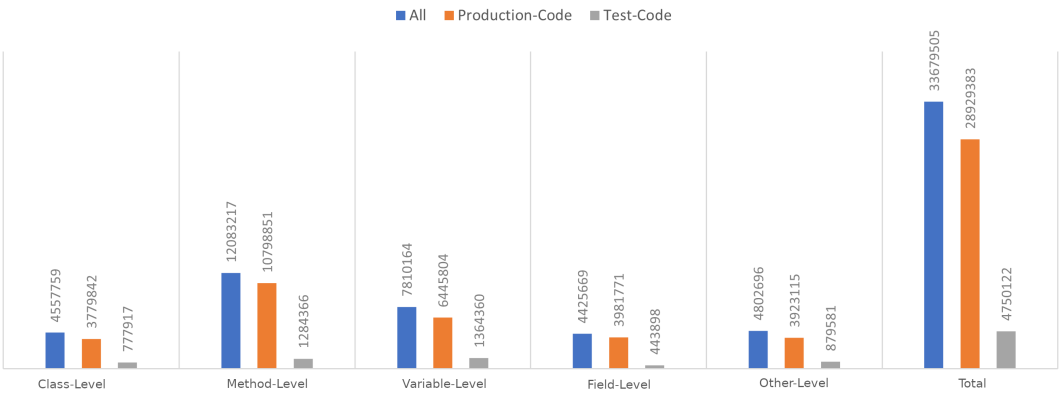


Figure 3.4: Count of all valid refactorings per level and in total

In Fig. 3.4 the total count of valid refactorings per level is given for the entire code base, and production and test code separately. Overall, **33.68 million refactorings** were extracted from the data, by far the largest fraction of these refactorings was identified in production code with 85.9% of all refactorings.

[7] GitHub issue on the data collection repository
[8] Overview of Refactored Entities
[9] Detailed Analysis of Refactored Entities

The fraction of the test code differs quite strongly for the five refactoring levels: 17.1% for Class-Level, 10.6% for Method-Level, 17.5% for Variable-Level, 10.0% for Field-Level and 18.3% for Other-Level. Refactorings on the Method- and Variable-Level are the most likely refactorings in test code with 27.0% and 28.7%. This differs from the overall ratio of refactorings. Method-Level refactorings are the most prevalent refactorings with 35.9% of all refactorings, followed by Variable-Level refactorings with 23.2%. The other three levels have a similar fraction of refactorings with an average of 13.6%. Compared to the research by Aniche et al. [4], the total number of identified refactorings is **16.14 times higher**.

The overall count of identified refactorings is given in the Appendix in Table A.1. The instance count of different refactorings per level differs widely, e.g. only 24851 instances of *Extract Subclass* are in the data, but more than three million instances of *Move Class*, which comprises 67.4% of all Class-Level Refactoring-Instances. A similar pattern can be observed for the other Refactoring-Levels as well, the Method-Level refactorings are dominated by *Change Parameter Type* (27%), *Change Return Type* (21.5%), *Extract Method* (11.2%) and *Rename Method* (18.4%) refactorings, which constitute 78.0% of the Method-Level refactorings. The three refactorings *Change Parameter Type* (41.2%), *Change Variable Type* (23.7%) and *Rename Parameter* (19.9%) make up 84.8% of the Variable-Level refactorings, and the Field-Level refactorings are dominated by *Change Attribute Type* (44.2%) and *Rename Attribute* (26.5%). This was hidden by the previous analysis of the different Refactoring-Levels. Furthermore, the analysis of the different refactorings per level shows that *Split Variable* (2988), *Replace Attribute* (4409), *Move And Rename Attribute* (6175) and *Split Parameter* (7538) were very rarely detected in the GitHub repositories.

A direct comparison between the data set extracted by Aniche et al. [4] and the new data set reveals that the new data set contains 32.7 times more Class-Level refactorings, 4.92 times more Method-Level refactorings and 6.12 more Variable-Level refactorings. A comparison of each refactoring is given in Fig. 3.5, noteworthy are the *Move And Rename Class*, *Rename Class* and *Move Class* refactorings, as they increase by factors of 379.2, 161.95 and 61.67. Also, the number of *Extract And Move Method* refactorings in the database is with a total of 435262 samples increased by a factor of 44.77. Overall, the comparison of the refactorings extracted between the original data set created by by Aniche et al. [4] and the new data set shows a substantial increase in the available data especially for *Move* and *Rename* refactoring types.

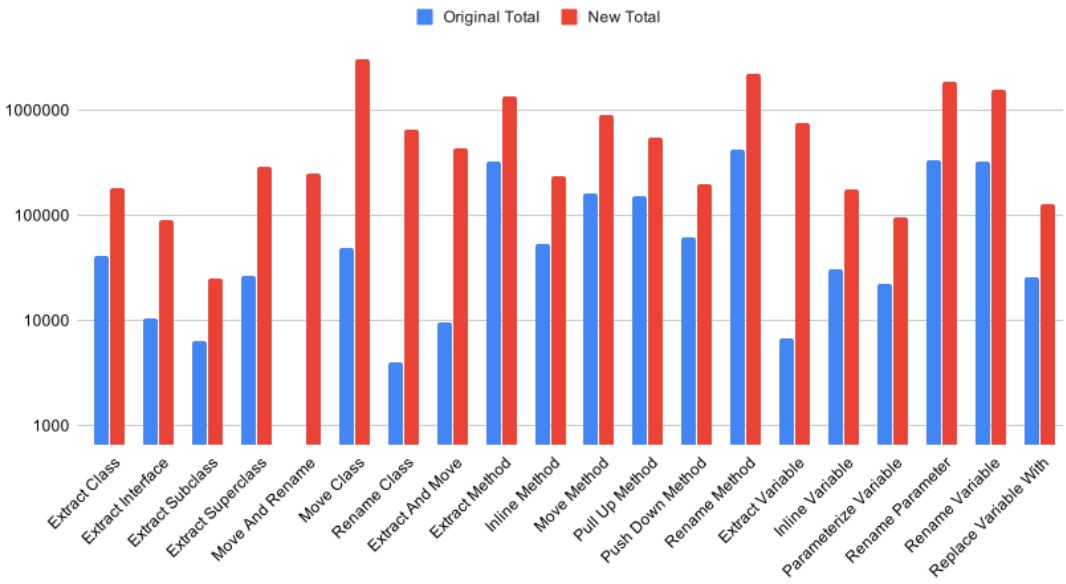


Figure 3.5: Comparison of the total count of refactorings of the original and this work, log-scale on the Y-axis.

3.5 Stable-Instances

In addition to refactorings, the data set also contains samples of non-refactored (stable) classes. Whenever a class was not refactored, but changed during the last K commits, it is considered a Stable-Instance and added to the data set, for more details see Section 3.1. All in all, **65.96 million Stable-Instances** were mined for all levels and commit thresholds, of these instances 81.1% were detected in production code and 18.9% in test code, see Table 3.3. Similar to the distribution of refactorings per level, the distribution of Stable-Instances differs widely for the different levels. Most Stable-Instances were extracted on the Variable-Level with a total of 38.48 million instances (58.34%) and the second most instances were collected for the Method-Level with 18.32 million instances (27.77%). The rarest Stable-Instances are Class-Level instances with a total count of 2.95 million (4.47%).

In Fig. 3.6, the distribution of Stable-Instances per K (commit threshold) is displayed for the four levels. K is the number of commits in which a class file was changed but not refactored, for more details see Section 3.1. The decline of the Stable-Instances detected with an increasing K is very steep, see Fig. 3.7. Only 6.88% of the total Stable-Instances that were collected for `K=15` are still stable for `K=50`. Already the decline of Stable-Instances from `K=15` to `K=20` is immense, as only 46.42% of the instances prevail. Furthermore, Fig. 3.7a shows that Class-Level Stable-Instances decline slower than all other levels, e.g. for `K=70` still 7.39% of Class-Level instances are considered stable in contrast to

Level	All Count	Production Code Count	% All	% Production	Test Code Count	% All	% Test
Class	2,948,846	2,045,472	69.37%	3.82%	903,374	30.63%	7.24%
Method	18,321,289	14,406,772	78.63%	26.93%	3,914,517	21.37%	31.39%
Variable	38,481,843	31,510,082	81.88%	58.91%	6,971,761	18.12%	55.90%
Field	6,212,929	5,530,418	89.01%	10.34%	682,511	10.99%	5.47%
Total	65,964,907	53,492,744	81.09%	100.00%	12,472,163	18.91%	100.00%

Table 3.3: Total count of Stable-Instances for each level, the fraction per level of the total Stable-Instances and the fraction per level of it's own code class

4.22% for the total instances. A detailed analysis of the distributions of the Stable-Instances focusing on test- and production code reveals that the decline for production code is more pronounced and that for production code all four levels show a very similar distribution, see Fig. 3.7b. Further, the analysis shows for test code the decline is not so pronounced and especially class level instances are more stable, see Fig. 3.7c. For example, for `K=50` only 7.3% of the Stable-Instances of the Class-Level remain for production code in contrast to 28.52% for test code.

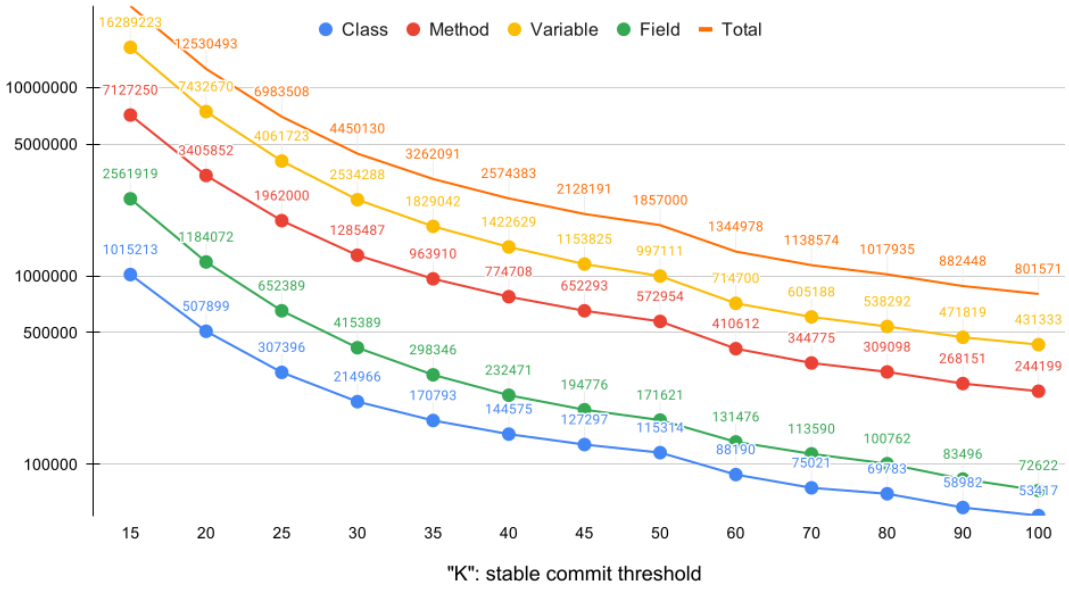
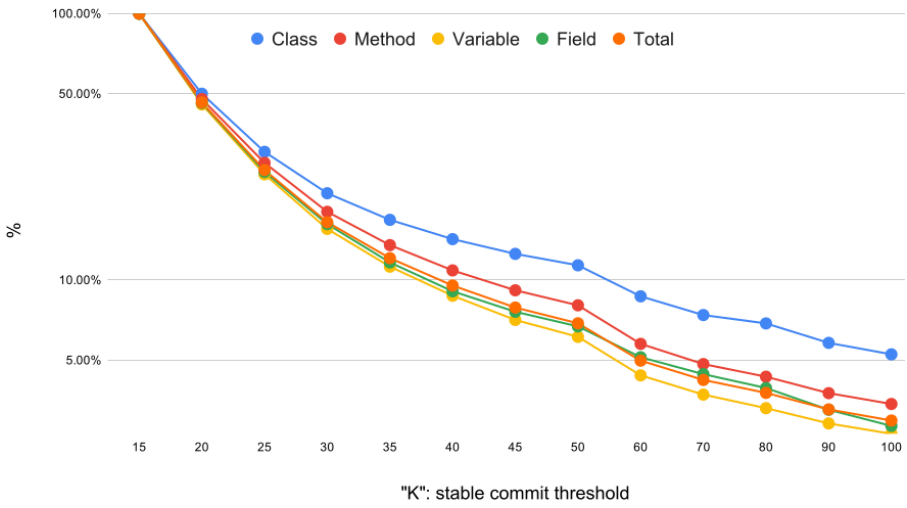


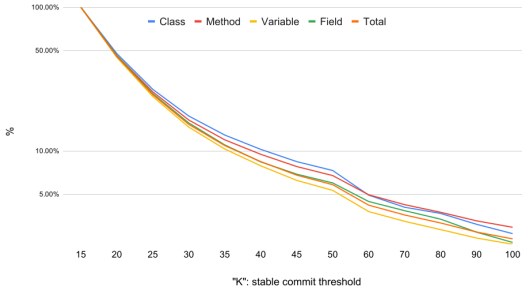
Figure 3.6: Total count of Stable-Instances per level and K

3.6 Open Data

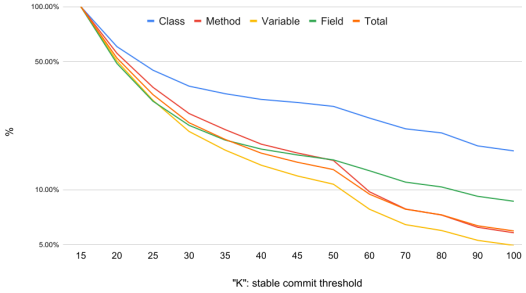
We hope other researchers will utilize the created data set for their research and explore it in more detail, e.g. it might yield precious insights on the refac-



(a) All Code



(b) Production Code



(c) Test Code

Figure 3.7: Fraction of the `K=15` instances for each K for each level and the total of Stable-Instances

toring habits in open source projects. For this reason, in this section a brief introduction to the database structure, the most important columns and tables, and important remarks is given. A simplified overview of the main tables and their relations is given in Fig. 3.8. The database contains 129.2GB of uncompressed data spread over 11 tables. These tables can be separated into three

categories: (i) project data, (ii) instance data and (iii) metrics and attributes. The entire data set can be found in the online appendix [21].

In the first category is the project table, this table contains all projects with relevant meta data. The most important columns in this table are:

- **commit count thresholds:** The K for this Stable-Instance, it can range from 15 to 100.
- **data set name:** The data set was divided into three subsets: training, test and validation. These names are only relevant for the classifier training, as all projects were gathered on GitHub and other important attributes, e.g. if a project is a fork, are stored in other columns.
- **exceptions count:** Number of exceptions handled in the main loop during data-collection, see Section 3.2.2.
- **finished date:** Date when the data mining was successfully completed for this project, can be empty in case the data collection failed.
- **is fork:** Indicates whether or not this project is a fork. If the project was marked as a fork, all related Refactoring- and Stable-Instances were transferred into separate tables.

The second category comprises of four tables, two for Stable- and Refactoring-Instances each. There is one table for Refactoring-Instances from not forked projects called `RefactoringCommit` (30.2GB) and one for forked ones called `RefactoringCommit_Forked` (13.5GB). Both tables have the same structure. These tables contain all instances of collected refactorings with references to metrics and attributes. The most relevant columns are:

- **file path:** File path for the class affected by the refactoring starting in the project's root folder. In contrast to the class name, this field can be used to distinguish between classes.
- **refactoring:** The refactoring of this instance as a string, e.g. "Move Class".
- **refactoring summary:** Summary of the refactoring, as generated by Refactoring Miner. This column contains detailed information about the refactoring.
- **isValid:** Shows if an instance was marked as invalid during data cleaning, see Section 3.3.1.
- **isTest:** Indicates whether the instance was detected in test code.

The structure of the tables for Stable-Instances is similar to the one for Refactoring-Instances. The table of instances from non-forked instances `StableCommit` has a size of 23.6GB and the one for forked instances `StableCommit_forks` of 24.7GB. The main difference from the table structure to refactoring is the addition of commit threshold column, indicating the K for the current instance, also the columns "refactoring" and "refactoring summary" are missing.

The last category of tables contains metrics, attributes and meta data for the individual instances. The `CommitMetaData` table (2.5GB) contains relevant meta data for each commit, e.g. commit data or commit id. Every commit in this table has a unique id for every project, thus the commit meta data id can be used to filter for unique commits. The same relation does not hold for the ids of the other metric tables, e.g. the id of a class metric cannot be used to group by class queries, as the metrics are added individually for each refactoring and stable instance extracted, see Fig. 3.8. The other tables in this category are `ClassMetric` (10.6GB), `FieldMetric` (0.84GB), `MethodMetric` (10.7GB), `ProcessMetrics` (9.5GB) and `VariableMetric` (3.0GB).

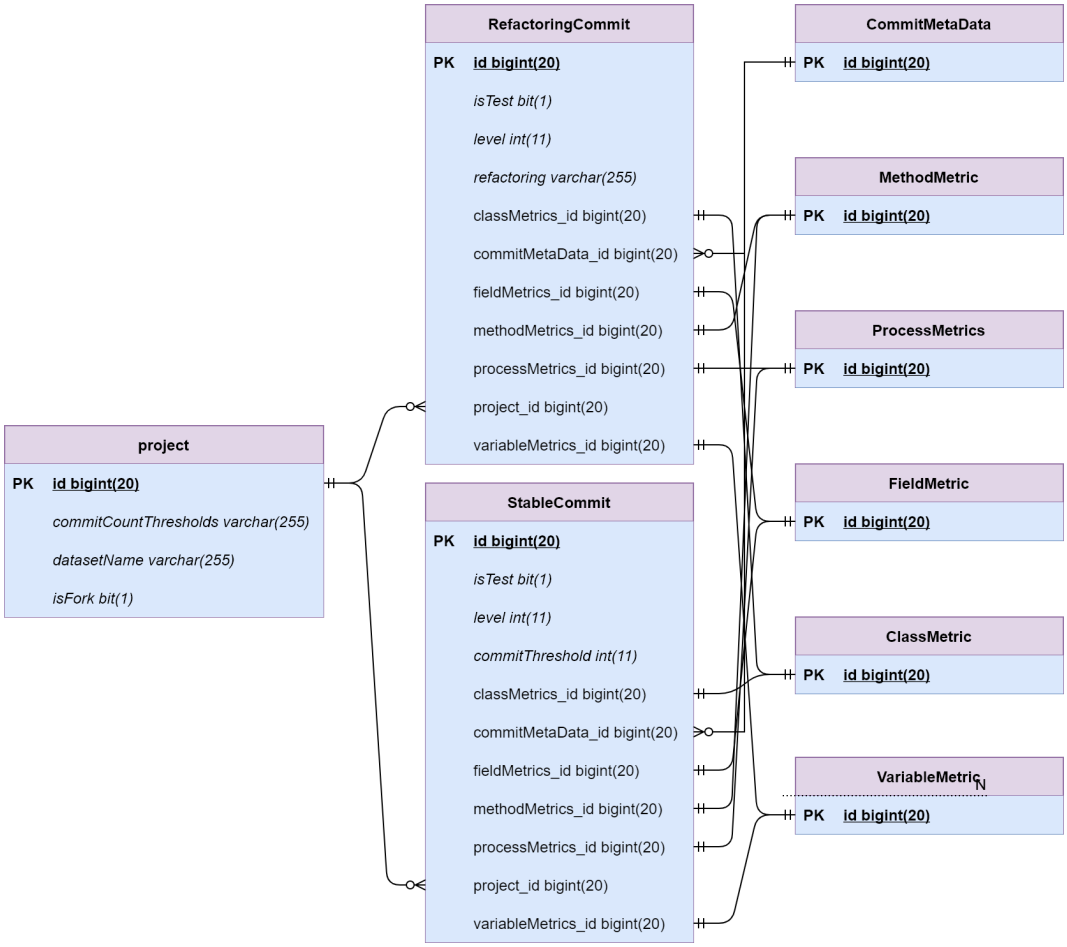


Figure 3.8: Simplified representation of the database schema, displays the relations between tables and selected columns

Chapter 4

An Exploratory Analysis of Refactoring Operations

In this chapter the large-scale refactoring data we gathered in the previous chapter is analysed in detail. The overall goal is to understand the data set and its components in order to later explore, validate and analyse the results of the machine learning chapter. Furthermore, the results of this analysis guide the decisions for the various experiments in the machine learning chapter. The research questions cover Stable- and Refactoring-Instances, and the distribution of process- and ownership metrics in particular, see Section 1.3. This chapter contains the methodology for the data analysis, the results of the analysis, and a discussion of the results with answers to the research questions 1, 2 and 3. It is important to mention that this analysis has an exploratory character, as the analysed data was unexplored and many relations were still unknown. Therefore, the main focus of this analysis is on the identification of trends, potential patterns and to get an overview of the data. All charts and statistics made for the data analysis can be found in the online appendix [20].

4.1 Methodology

First, the sub division of the research questions into smaller questions and topics is motivated. Afterwards, the selection of features for the analysis is explained. Lastly, the utilized statistical techniques for analysis are briefly described and explained. For the visual analysis, a variety of visualization algorithms is deployed including scatter plots, box plots, violin plots, heatmaps and line plots. The following three Research Questions are relevant for this chapter:

- **RQ 1: How are features distributed among refactoring types and levels?**
- **RQ 2: How are features distributed among Stable-Instances?**

- **RQ 3: What are implications of the distribution of process- and ownership metrics?**

4.1.1 Focus of the Analysis

The three research questions are difficult to answer, because the data set is large and its dimensionality is enormous with a great variety of refactoring types, metrics and attributes, Stable-Instances and commit thresholds. The Refactoring- and Stable-Instances are the two main entities among which the data is divided. Furthermore, together with the process- and ownership metrics they are crucial for the success of the refactoring prediction models.
For all research questions we pursue the following topics in the analysis:

- *Statistical distribution of the features*
- *The overall spread of features*
- *Identification of subsets*

As the analyzed entities have different characteristics and are utilized differently in the classifier training, we explore individual issues for the three of them. For the Refactoring-Instances, we explore how they evolve over time. For the Stable-Instances, we attempt to answer the following additional sub-questions:

- *Are the Stable-Instances inherently different from the Refactoring-Instances?*
- *How does the increasing commit threshold affect the features?*

For the process- and ownership metrics we pursue the following sub-questions:

- *How do the distributions of process- and ownership metrics differ from the other analyzed metrics?*
- *What might explain the importance of the process- and ownership metrics for the success of the classifier training?*

4.1.2 Feature Selection

The total number of different metrics (features) is 69, analyzing all these features in-depth is not possible. Thus, we analyze process- and ownership metrics, because they are highly relevant for the success of the classifier. We further analyze the six metrics attempt to capture abstract quality measures of classes. Additionally, we analyze the six other metrics that were considered relevant by the classifiers in the research by Aniche et al. [4], see Table 4.1. We later refer to them as class attributes, in order to more easily distinguish them from the abstract class metrics.

Class Metrics	Class Attributes
Coupling between Object (CbO)	Number of Methods
LCC	Number of Public Fields
RFC	Quantity of String Literals
LCOM	Number of Unique Words
TCC	Number of Variables
Weight Method Class (WMC)	SLoC

Table 4.1: Class metrics and attributes considered in the data analysis

4.1.3 Statistical Distribution

The statistical distribution of the features is analysed in multiple steps. First, a Kolmogorov–Smirnov test (K-S test) is performed to evaluate if a feature is normally distributed. To test the normality of the distribution, the null hypothesis is set to a standard normal distribution and the samples are also normalized[41]. The Shapiro-Wilk test is used to test the features for normal-distribution [64]. Positive results are verified by a visual inspection of the histograms of the metrics. Furthermore, the skew of a metric is measured [1] and can be used to further evaluate the overall distribution of the features. Skewness describes the asymmetry of a distribution from its average [25]. Moreover, in order to find functions best describing a metric, they are visually analysed. The aggregate functions of SQL in combination with specific characteristics of the data are utilized to calculate various relations in the data, e.g. the fraction of unique class metrics of Stable-Instances is computed by summing up unique class metric ids and grouping them by commit thresholds and levels.

4.1.4 Identification of Sub-Sets

For the identification of sub-sets, we apply two approaches (i) visual inspection of features and (ii) clustering algorithms. We utilize KMeans and DBScan to detect clusters in the data. KMeans is a distance based clustering algorithm where samples are assigned to the nearest cluster based on the distance to its center. In contrast, DBScan is a density based technique, which creates clusters in high density areas separated by low density zones. Both techniques are very different and can yield different results, thus we expect to identify a variety of potential clusters. Also, both median and mean are used to describe the features to the clustering algorithms. We use the clustering algorithms implemented in the latest version of scikit-learn [2,3].

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.skew.html
[2] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
[3] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html

4.2 Refactoring-Instances

In this section, we explore the distribution of features among Refactoring-Instances in the collected data in order to answer ***RQ 1: How are features distributed among refactoring types and levels?***. We analyse the statistical distribution of the class metrics (see Fig. 4.1), we identify clusters (Table A.2, Fig. 4.3) and outliers and explore the evolution of refactorings over time (Fig. 4.5).

Observation 1: The features are non-normally distributed and right skewed. The Class-Level metrics and attributes are non-normally distributed among Refactoring-Levels and individual refactoring types. All but two of the features are right skewed, indicating that the overall distribution is right fat tailed. The skew ranges from 4.72 for the Class-Level CbO metric to 253.85 for the LCOM metric of the Other-Level, with most of the metrics having a high skew exceeding 15. The distribution of the skew is coherent for the various Refactoring-Levels and features. Other-Level refactorings are Only the class metrics TCC and LCC are exceptions, as they are either distributed slightly left (-0.09) or right (0.66) to their mean across the various Refactoring-Levels. Additionally, the distribution of TCC and LCC metrics is highly similar for all Refactoring-Levels and the differences between refactoring types are very limited. Only the Field-Level Refactoring-Instances differ to some degree, their TCC and LCC 25% quartile are around 50 instead of 0 for all other Refactoring-Levels, see Fig. 4.1. The feature distribution indicates that the mean does not describe the features well. The statistical analysis showed, that the mean for most metrics and attributes is around the 75% quartile or even exceeds it.

Observation 2: Similar refactorings have very similar metrics. Highly similar refactorings often possess very similar class metrics and attributes, especially refactorings addressing the same issues, e.g. *Merge Parameter* and *Split Parameter* or *Extract Method* and *Inline Method*. An example for this is displayed in Fig. 4.2.

Observation 3: Features cover a wide range Many Refactoring-Levels have outliers in their feature distribution. For the Field-Level refactorings, the features are distributed highly cohesive, except for the *Move Attribute* and *Rename Attribute* refactoring. The average number of fields is considerably higher for these refactorings with averages of 5.51 and 8.07 compared to averages of 1.7 to 2.7 for the other Field-Level refactorings. This example is particularly interesting, as the disparity to the very similar *Move and Rename Attribute* refactoring is strong (mean 1.7).

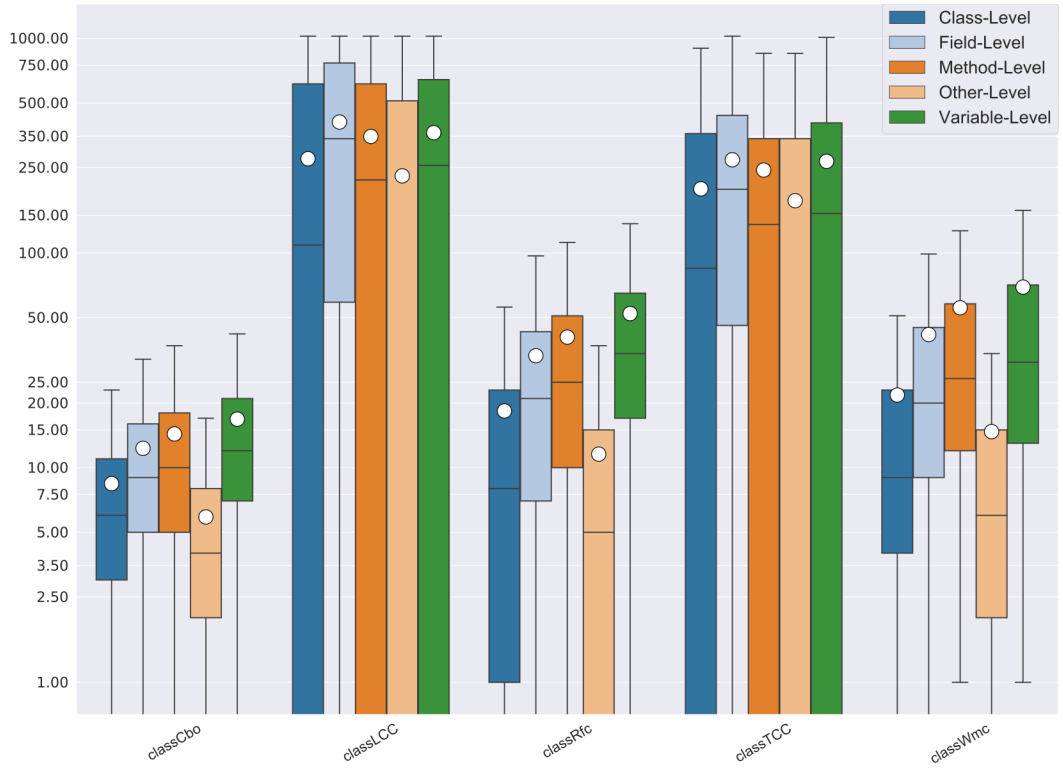


Figure 4.1: Class metrics without LCOM for all Refactoring-Levels (log-scale)

Observation 4: Refactorings of Class- and Other-Level are applied to simple classes. Class- and Other-Level refactorings have significantly lower metrics compared to the other Refactoring-Levels e.g. CbO, RFC and WMC, see Fig. 4.1. Furthermore, class attributes are also significantly lower compared to the other Refactoring-Levels, e.g. the median SLoC of a class for Method-Level refactorings is about twice as high (148,5) as for Class-Level refactorings (50), similar is the relation for the Number of Unique Words in a class. Thus, these refactorings predominantly are applied to simpler, less evolved classes. A likely explanation is that these refactorings are applied mostly in early development stages of a class. *Extract class* is the only refactoring of the Class-Level, that shows a similar feature distribution as Method- and Variable-Level refactorings.

Observation 5: Refactoring-Levels can form different clusters. The Refactoring-Levels Class- and Other-Level form a cluster, with a very similar distribution of features, so do Method- and Variable-Level refactorings, and finally Field-Level refactorings comprise a third cluster, see Fig. 4.3. Analyzing the Refactoring-Levels in more detail, reveals that often smaller subsets of refactorings strongly influence the overall characteristics of a Refactoring-Level mostly due to the imbalance in instance counts, e.g. for the Class-Level these are *Move*

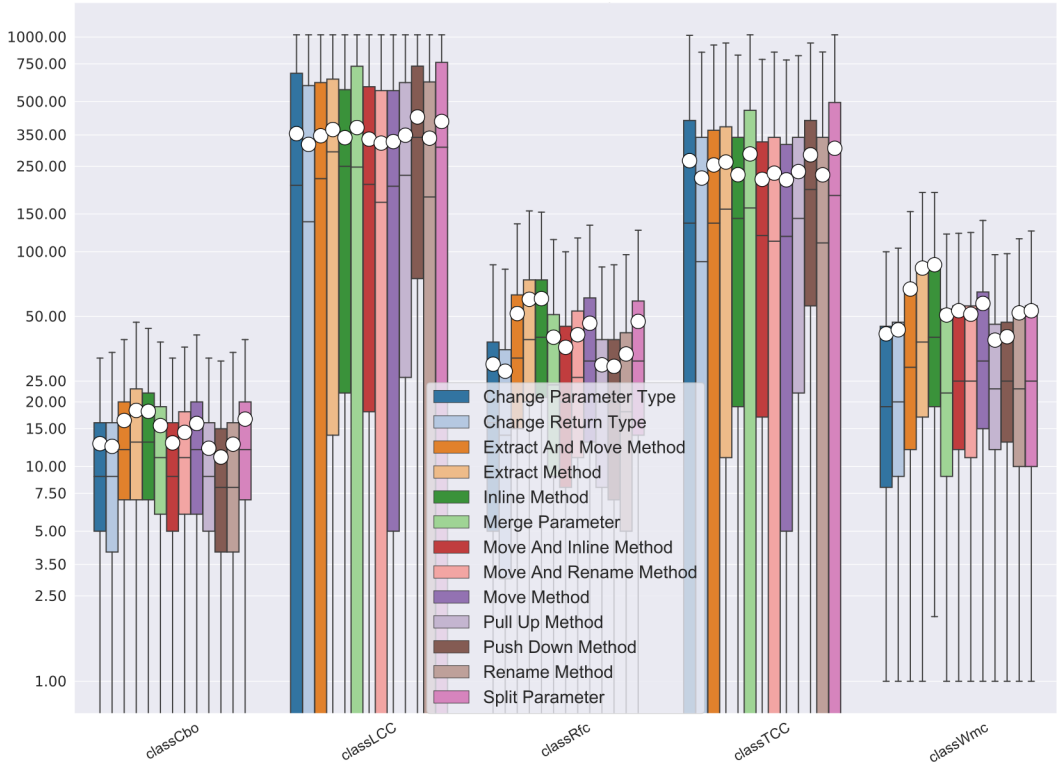


Figure 4.2: Class metrics without LCOM for the Method-Level (log-scale)

Class and *Rename Class* refactorings, see Fig. 4.4 and Table A.1.

Observation 6: Individual refactorings can be separated into three to five clusters. The cluster analysis of the individual refactoring types revealed three strong clusters including at least 20 out of the 39 refactoring types collected. Additionally, two more potential clusters were detected, see Table A.2 and Fig. 4.4. The first and most obvious cluster is comprised of three Class-Level refactorings and the two Other-Level refactorings. These refactorings have highly similar distributions of class metrics and attributes. Their class metrics are considerably lower compared to all other refactorings, see Fig. 4.4. The second cluster is comprised of eight of the nine Variable-Level refactorings and the *Extract and Move Method* refactoring of the Method-Level, *Rename Parameter* is the missing Variable-Level refactoring. The class metrics are highly similar for these refactoring types with a median CbO of 12.4. A third cluster is comprised of six Field-Level refactorings *Move Attribute*, *Move and Rename Attribute*, *Pull Up Attribute*, *Push Down Attribute*, *Replace Attribute*, *Change Attribute Type*.

Observation 7: *Rename Parameter* refactorings have a unique feature distribution. Furthermore, the analysis showed that some refactoring types are not aligned with their level, especially *Rename Parameter* has a unique distributions of features. For the *Rename Parameter* refactoring, the four class attributes SLoC, Quantity of String Literals, Number of Unique Words, Number of Variables and three class metrics RFC, WMC and CbO are significantly lower than for all other refactoring types.

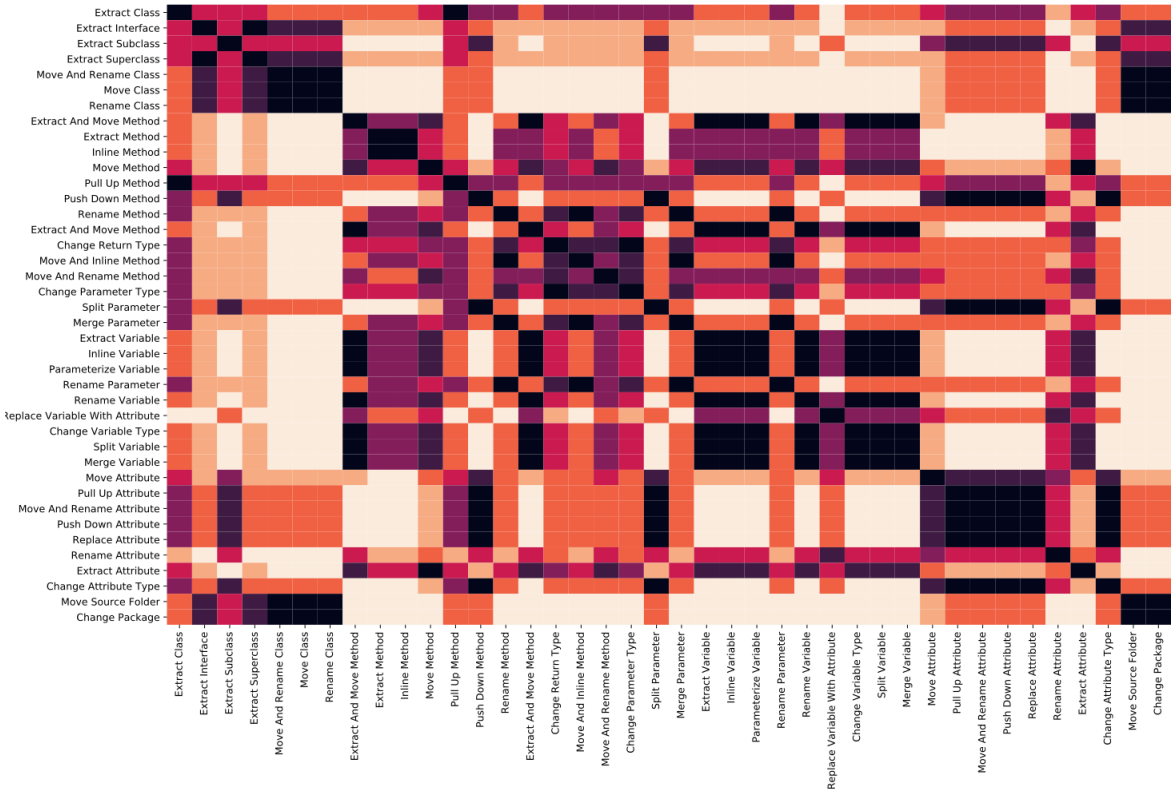


Figure 4.3: Heatmap of the likelihood of co-occurrence in a cluster for all refactorings clustered with KMeans with the mean and median of the class metrics and attributes

Observation 8: Class- and Other-Level refactorings are applied predominantly in early development stages. Fig. 4.5 displays the cumulative frequency of Refactoring-Instances occurring after a certain number of commits on a class file for the Variable- and Class-Level refactoring types. The refactorings in cluster 1 (Class- and Other-Level) occur early in the development process of a class, more than 85% of all of these refactorings were detected withing the first 10 commits of class. Moreover, 79% of cluster 1 refactorings are one of the first 5 refactorings of a class. For comparison, for all refactorings not in cluster

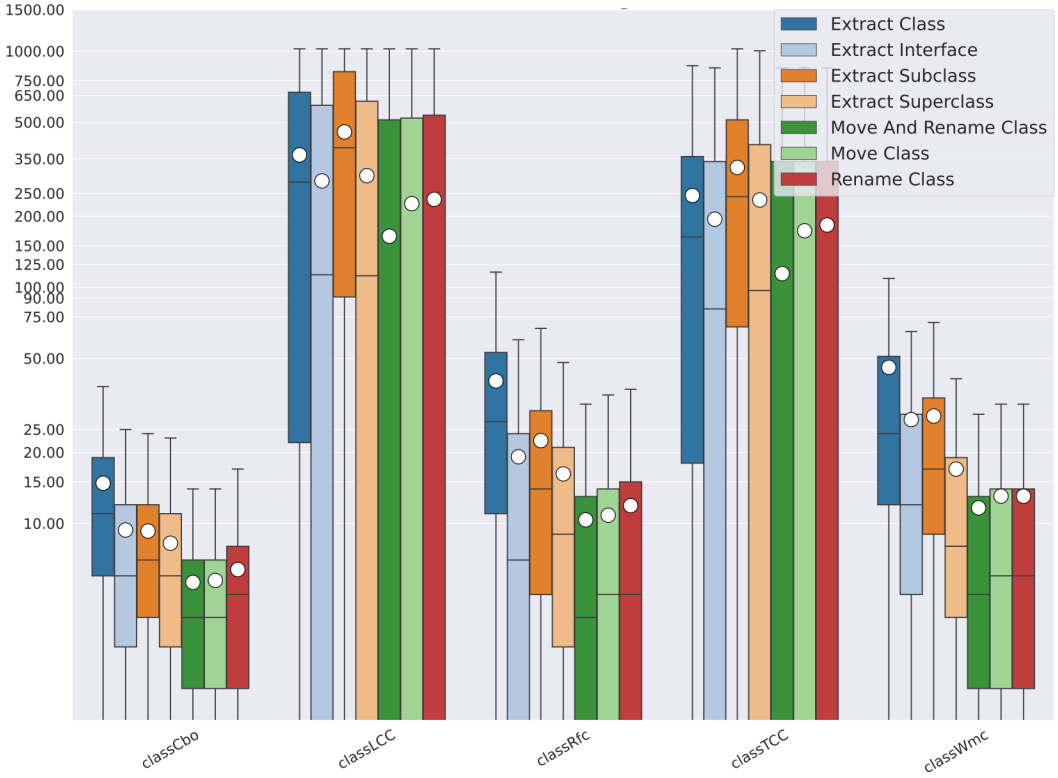
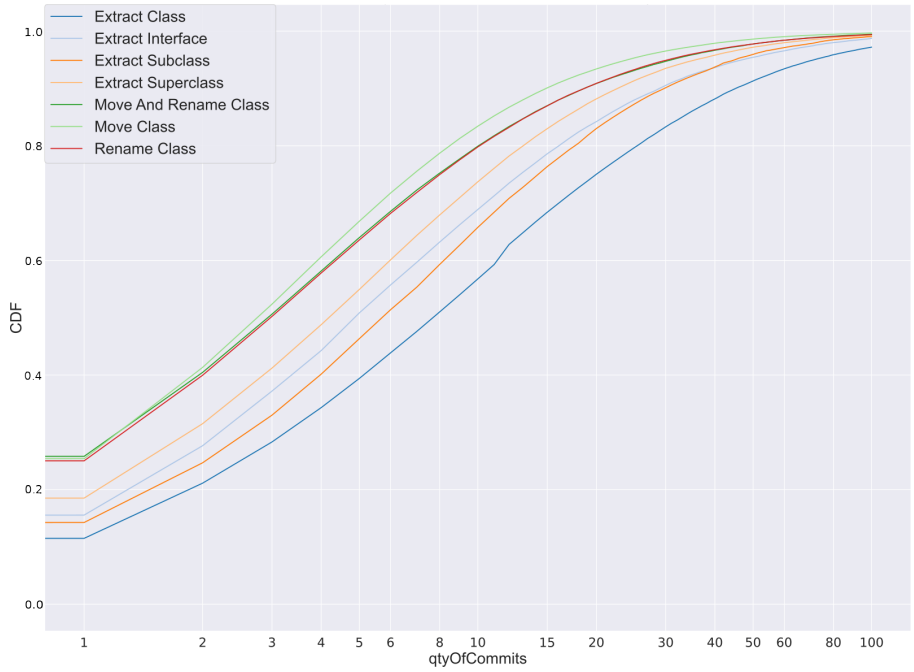


Figure 4.4: Class metrics for the class level refactorings (log-scale)

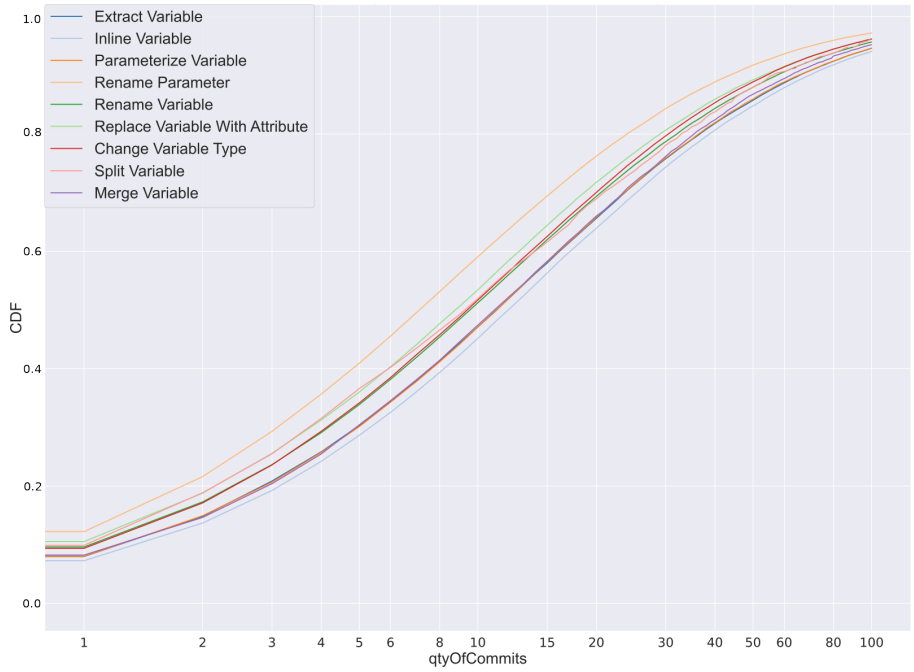
1 more than 50% of their Instances have a commit count higher than 5. Thus, it can be concluded that cluster 1 refactorings are applied early in the development of a class, and are increasingly unlikely for more developed and maintained classes. The other Class-Level refactorings, except *Extract Class*, are also occurring slightly earlier, than the refactorings of the Variable-, Method- and Field-Level. This combined with the over-representation of the *Move Class* refactoring explains the lower complexity metrics collected for the combined Class-Level refactorings. The refactoring *Extract Class* has a fairly similar distribution to the refactorings in cluster 3. All three other Refactoring-Levels have relatively homogeneous behaviour in regard to the commit count.

4.3 Stable-Instances

In this section we present the results to answer ***RQ 2: How are features distributed among Stable-Instances?***. We analyze the overall distribution features of Stable-Instances, the distribution of metrics and attributes across various K and the four different Stable-Levels (Fig. 4.6, Fig. 4.7, Fig. 4.8, Fig. 4.9, Fig. 4.10, Fig. 4.11). Additionally, we analyze how many unique classes are



(a) Class-Level



(b) Variable-Level

Figure 4.5: CDF per commit count of a class for refactorings of the of Class- and Variable-Level, x axis in log-scale, capped at 100 commits

covered by Stable- and Refactoring-Instances (Fig. 4.6).

Observation 1: The features for Stable-Instances are non-normally distributed and right skewed. Similar to the Refactoring-Instances, the class metrics and attributes are also non-normally distributed and all but two features are are right skewed with a right fat tail.

Observation 2: Stable-Instances cover only 7.33% of all collected classes. In total 8.35 million unique classes were collected. Only 0.62 million unique classes were considered stable with `K=15`, this is 7.33% of all recorded classes in the database, in contrast to the 7.92 million unique classes being refactored (94.82%). Of the total 8.35 million unique classes, only 187000 classes were refactored at least once and considered stable, these are 30.27% of the unique classes of the Stable-Instances. In Fig. 4.6 the unique classes are analysed in more detail. Overall, the fraction of unique classes declines rapidly with increasing K, for `K=50` only 0.46% of all recorded classes are considered stable on the Variable-Level. Additionally, the fraction of unique classes is much lower for Field-Level Stable-Instances.

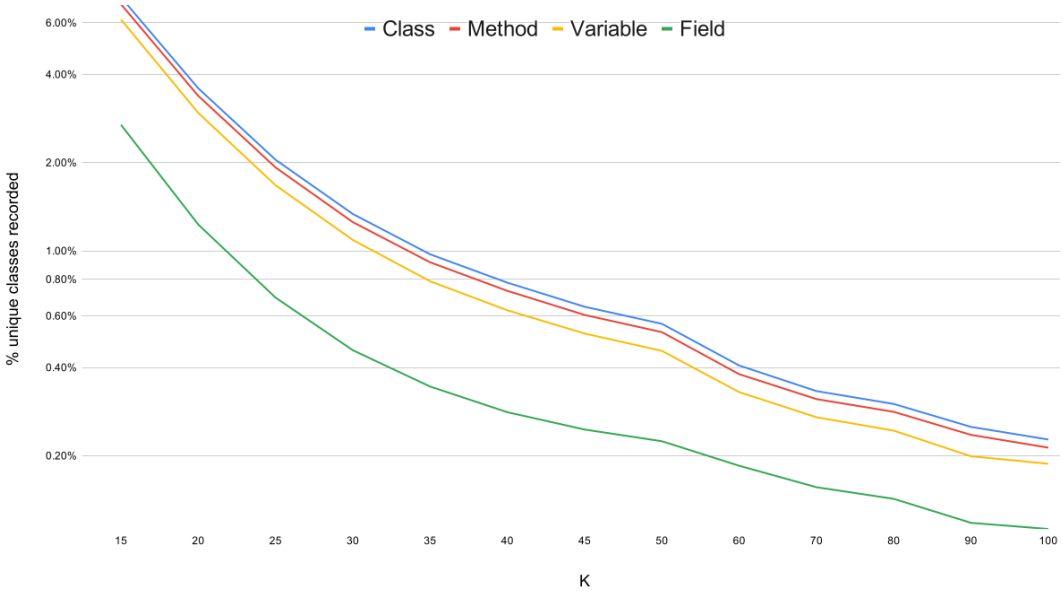


Figure 4.6: Fraction of unique classes of the total recorded unique classes for the Stable-Instances per K for production code

Observation 3: Classes with high method and variable counts are over-represented. Fig. 4.7 displays the total count of the Stable-Instances per level

and K. Comparing it with Fig. 4.6 reveals that for the Method- and Variable-Level, classes with many methods and variables are highly over-represented in the data set of Stable-Instances. For every method in a stable class and subsequently every variable in each method a new Method- and Variable-Level Stable-Instance is created, e.g. for a class with 10 methods with 3 variables each, a total of 10 Method-Level and 30 Variable-Level Stable-Instances were created, all sharing the same class metrics, the Field-Level instances suffer from a similar but smaller issue. To cope with this imbalance in the data, we only consider a single Stable-Instance for each level, reducing the number of Stable-Instances considered for the Method- and Variable-Level significantly, see Fig. 4.8.

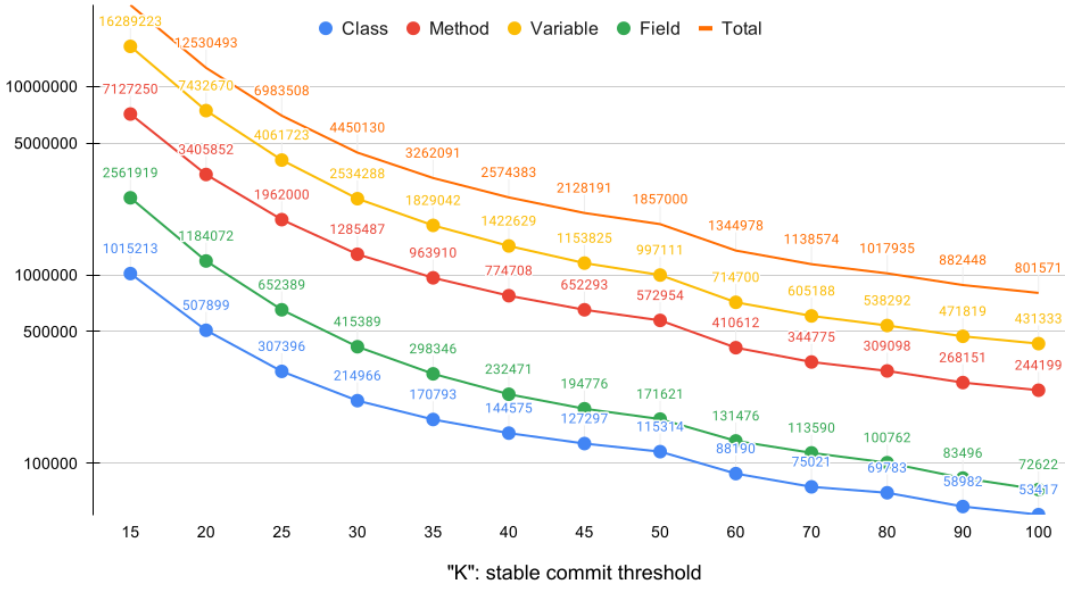


Figure 4.7: Total count of Stable-Instances per level and K

Observation 4: Stable-Instances form only two clusters. The Stable-Instances can be grouped into two clusters: one consisting of Class-, Method- and Variable-Level instances and the other of Field-Level instances, see Fig. 4.9 and Fig. 4.10. The elements of the first cluster have a similar set of class metrics, this comes at no surprise as they are subsequently built up on the same set of classes. The metrics improve mildly for the Variable- and Method-Level, but are within the same order of magnitude. The Field-Level Stable-Instances which have at least a single public field: tend to be larger, have more methods and variables as the other Stable-Instances. The Number of Methods for Field-Level instances has a median of 7 in strong contrast to a maximum of 2 for the other levels. Additionally, the class metrics of Field-Level instances are

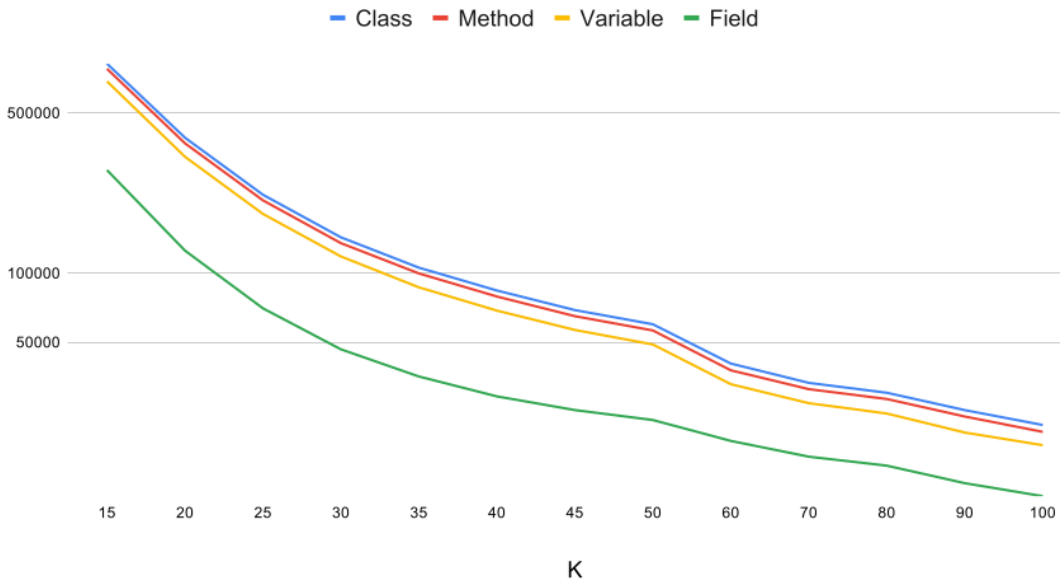


Figure 4.8: Total count of unique class metrics per Stable-Level and K

worse in regard to software quality, which can be explained by the strong focus of metrics on methods and fields.

Observation 5: Stable classes have low class metrics. A comparison of the class metrics reveals that the metrics differ strongly between Stable- and Refactoring-Instances. Stable-Instances have significantly lower class metrics (better in regard to software quality), except for the Field-Level instances. For example, the median CbO of Class-Level Stable-Instances is 2 and the 75% quartile at 6, for Refactoring-Instances of the Class-Level the median is at 6 and the 75% quartile at 11. Also, the WMC differs significantly from the Refactoring-Instances. Field-Level instances are the exception as they are very similar to their counter-part the Refactoring-Instances. A similar relation can be observed also for the class attributes. Stable-Instances of the Class-, Method- and Variable-Level have considerably smaller classes. For example, the 75% quartile for the number of methods is at 5 for Class-Level and 6 and 7 for Method- and Variable-Level Stable-Instances, in contrast to Refactoring-Instances, where it ranges from 11 up to 22. Field-Level Stable-Instances have a very similar distribution of class attributes as the Field-Level Refactoring-Instances.

Observation 6: Only few metrics are affected by increasing K Analyzing the distribution K shows three results. First, four of the six class attributes are stable across all K's. The class attribute Number of Public Fields increases significantly for cluster 1 of Stable-Instances, e.g. for the Class-Level Instances

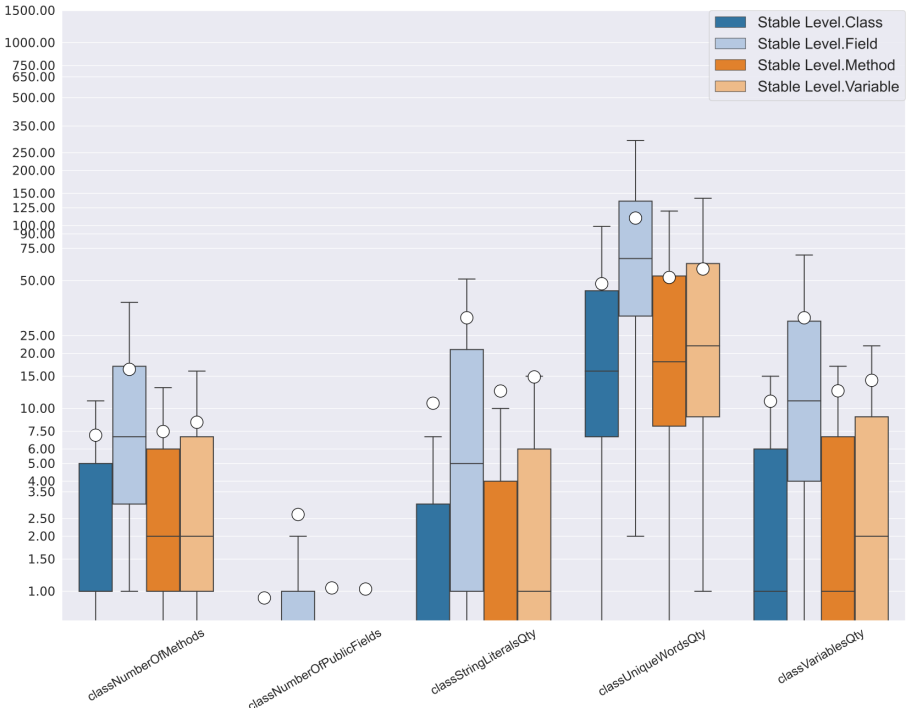


Figure 4.9: Distribution of the class attributes without SLoC for the Stable-Levels with `K=15`, y-axis with log-scale

the mean Number of Public Fields doubles from `K=15` to `K=30`. The Quantity of String Literals also increases from `K=15` to `K=30` and stabilizes afterwards. Second, the class metrics improve with increasing K across all Stable-Levels, see Fig. 4.11. The CbO starts at a mean of 6 - 7.3 (except Field-Level) and improves to mean of 4.5 to 5 (except Field-Level) at `K=60`. The changes are very subtle though and for `K>60` the changes are minor. These two pairs of metrics have almost identical distributions: TCC and LCC, RFC and WMC.

4.4 Process- and Ownership Metrics

In this section, we analyze the process- and ownership metrics in detail in order to answer ***RQ 3: What are implications of the distribution of process- and ownership metrics?***. We analyze in-depth the distribution of process- and ownership metrics for the various Refactoring- and Stable-Levels (Fig. 4.12), individual refactoring types within each level (Fig. 4.14, Fig. 4.15) and feature distribution for the various K's (Fig. 4.16).

Observation 1: Process- and Ownership metrics are non-normally distributed and are right skewed. The statistical analysis of the overall distri-

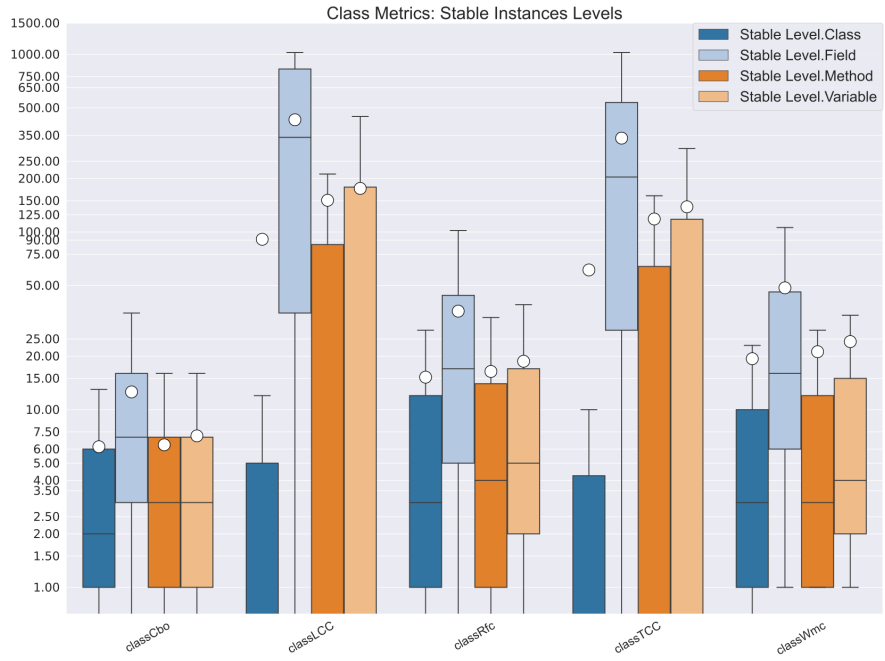


Figure 4.10: Distribution of the class metrics without LCOM for the Stable-Levels with `K=15`, y-axis with log-scale

bution of the process- and ownership metrics shows, they are all non-normally distributed and are right skewed for both Refactoring- and Stable-Instances, except for the Author Ownership metric. The mean for the six metrics (except Author Ownership) is at the 75% quartile or even above it. The Author Ownership metric is slightly left skewed (-0.78 to -1.41), and not fat or long tailed. Thus, the mean describes the distribution of the Author Ownership fairly accurate.

Observation 2: Only minor differences between Stable-Levels. As both the Stable-Instances and the process- and ownership metrics were collected for class files, the distribution of the metrics is highly similar among the four Stable-Levels. Thus, an analysis of the process- and ownership metrics for the differences between the individual Stable-Levels did not yield any results.

Observation 3: Class- and Other-Level refactorings are predominantly applied to young classes with few authors. For the Refactoring-Instances, the process metrics differ strongly for the five Refactoring-Levels, see Fig. 4.12. Especially, the Class- and Other-Level refactorings have significantly lower process- and ownership metrics. The only exception is the Author Ownership metric, it is very comparable among the Refactoring-Levels with an average value around 0.79 to 0.84, the median of the Refactoring-Levels ranges from

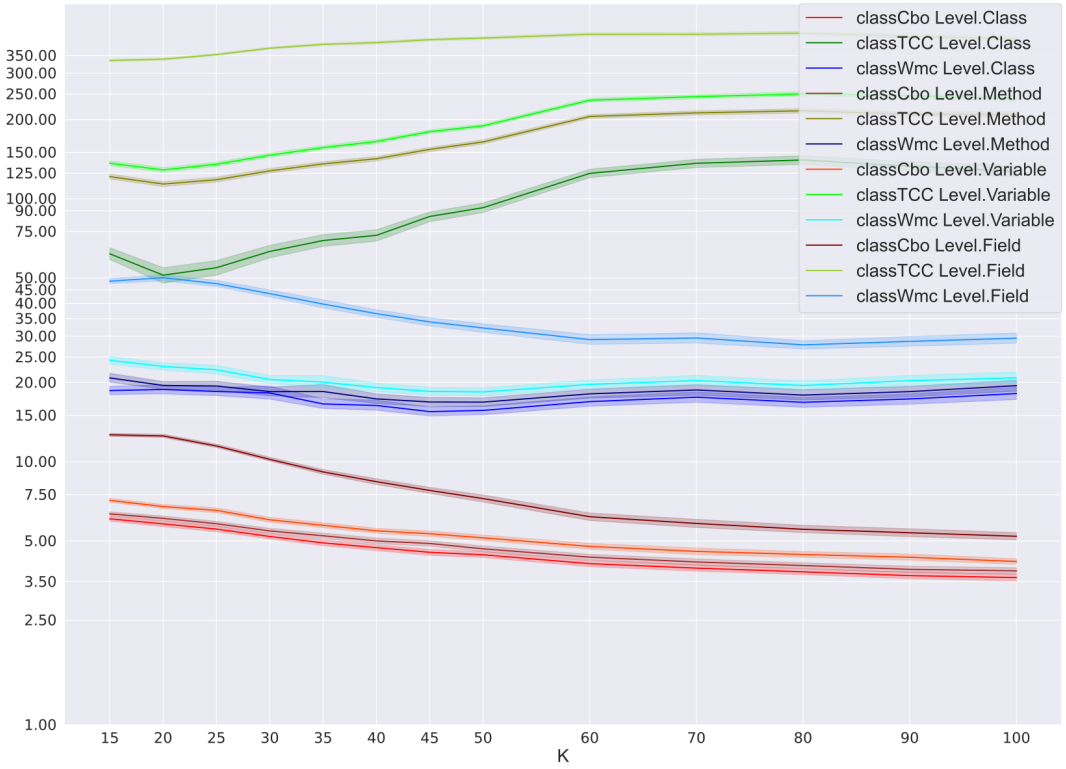


Figure 4.11: Class Metrics for all Stable-Levels and all K's, without LCC and RFC, y-axis log-scale

0.92 to 1. The Quantity of Minor Authors is almost zero for Other-Level refactorings (mean 0.08) and very small for the Class-Level refactorings (mean 0.23). Field-Level refactorings, for example, have an average of 0.60. Smaller variations of the distribution can be observed for the other ownership metrics. For the Class- and Other-Level refactorings, the Quantity of Major Authors and Quantity of Authors are only marginally lower. Based on these observations, we conclude that Class- and especially Other-Level refactorings are predominantly applied to classes that are maintained by a single developer.

Observation 4: Process- and ownership metrics are very different for Stable- and Refactoring-Instances. Comparing the Refactoring-Instances with the Stable-Instances with `K=15`, shows that for all process metrics the Refactoring-Instances are considerably different from the Stable-Instances. For example, the Bugfix Count the Variable-Level Refactoring-Instances (highest distribution) has a median of 2 and a mean of 6 compared to a mean greater 21 and a median of 6 and for the Stable-Instances with `K=15`. The analysis of the ownership metrics showed that both Author Ownership and Quantity of Au-

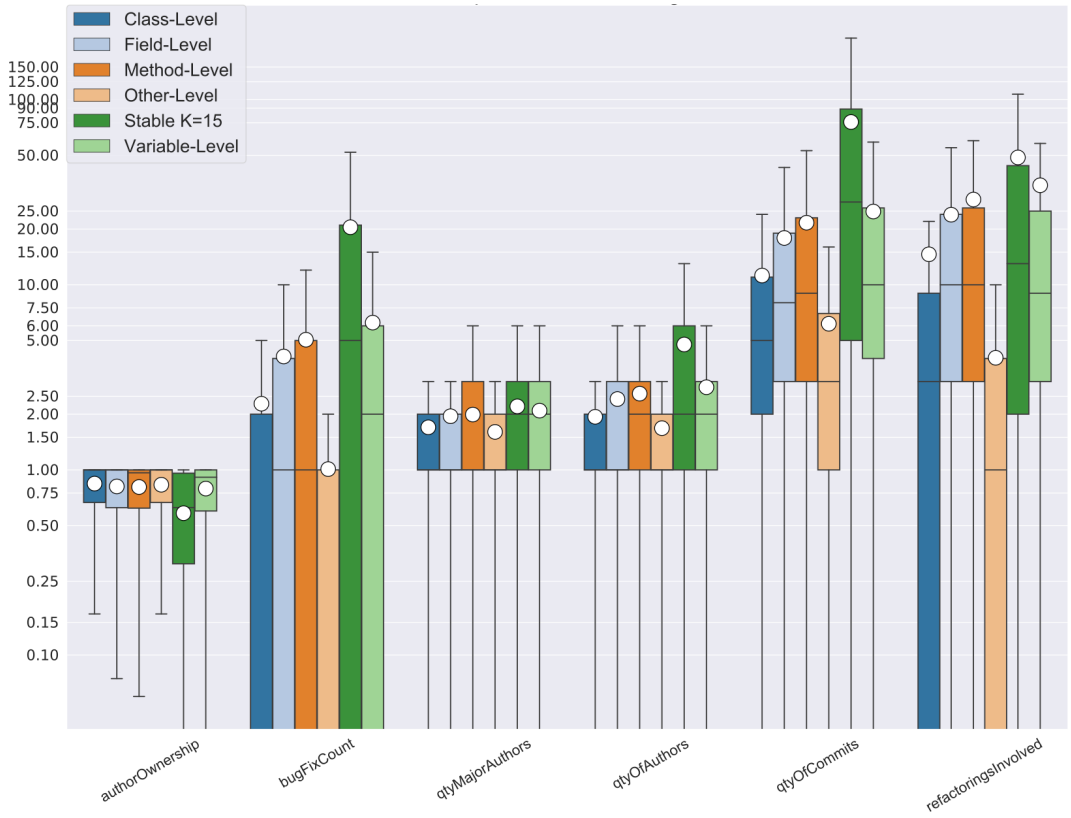


Figure 4.12: Distribution of process- and ownership metrics for all Refactoring-Levels vs Stable-Instances with `K=15`

thors differ strongly between Refactoring- and Stable-Instances. The Quantity of Authors ranges on average from 1.5 to 3 for Refactoring-Instances and is for the Stable-Instances above 5. Both process- and ownership metrics display a clear separation between the Refactoring- and Stable-Instances, see Fig. 4.12.

Observation 5: Process- and ownership metrics form similar clusters as class metrics and attributes. The results of the clustering of refactoring types with process- and ownership metrics is shown in Fig. 4.13. The cluster analysis for the process and ownership metrics gave fairly similar results as for the class metric and attributes, see Section 4.2 and Section 4.3. The most outstanding difference is that the *Replace Variable with Attribute* refactoring is not part of cluster 2 but 3, see Fig. 4.13. The cluster stability across different metrics indicates that the refactorings are indeed used in very similar circumstances. Furthermore, this indicates that the class metrics and process- and ownership metrics are correlated.

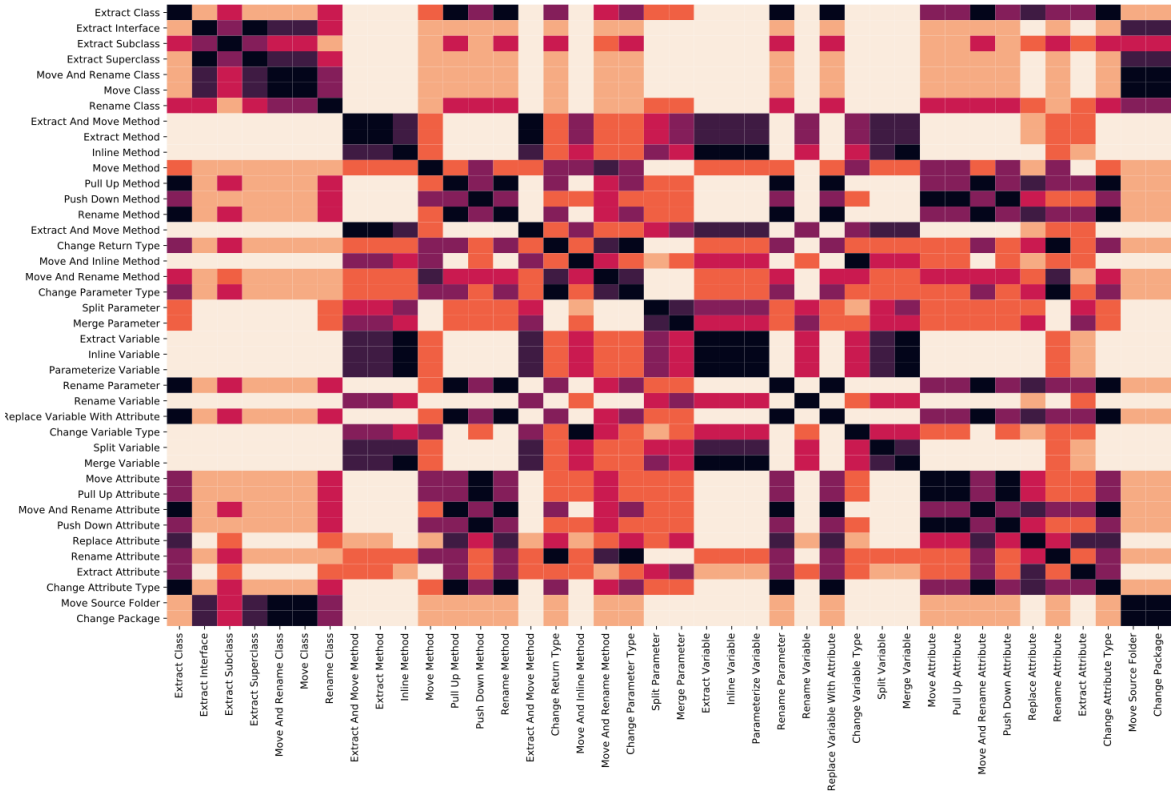


Figure 4.13: Heatmap of the likelihood of co-occurrence in a cluster for all refactorings clustered with KMeans with the mean and median of the process and ownership metrics

Observation 6: Process- and ownership metrics are similar within the Refactoring-Levels. In Fig. 4.14 and Fig. 4.15 the distribution of the process- and ownership metrics for the Class- and Method-Level refactorings is given. An in-depth analysis of the different refactoring types of each Refactoring-Level shows that the ownership metrics are highly similar for most refactoring types within their respective level, except for Quantity of Authors which differs slightly for the refactorings within each level, see Fig. 4.14 and Fig. 4.15. For the individual refactorings within each Refactoring-Level, the process metrics differ widely, except for Other-Level. This is especially pronounced for the Class- and Method-Level. For the Variable- and Field-Level, these differences are considerably lower. The *Move Class*, *Rename Class* and *Move and Rename Class* refactoring have a very low Bugfix Count and Quantity of Commits compared to all other refactoring types. These refactorings strongly affect the overall distribution of process- and ownership metrics of the Class-Level as they are by far the most common refactorings for the Class-Level, see Table A.1. Furthermore, the number of Refactorings Involved is in most cases (75% quantile) signifi-

cantly lower compared to the other Refactoring-Levels. Method-Level refactorings vary strongly for the Bugfix Count and the Quantity of Commits. Especially *Change Parameter Type, Change Return Type, Pull Up Method* and *Push Down Method* refactorings have significantly lower Bugfix Counts with a 75% quantile of 3 compared to 7 for the *Extract Method* refactoring.

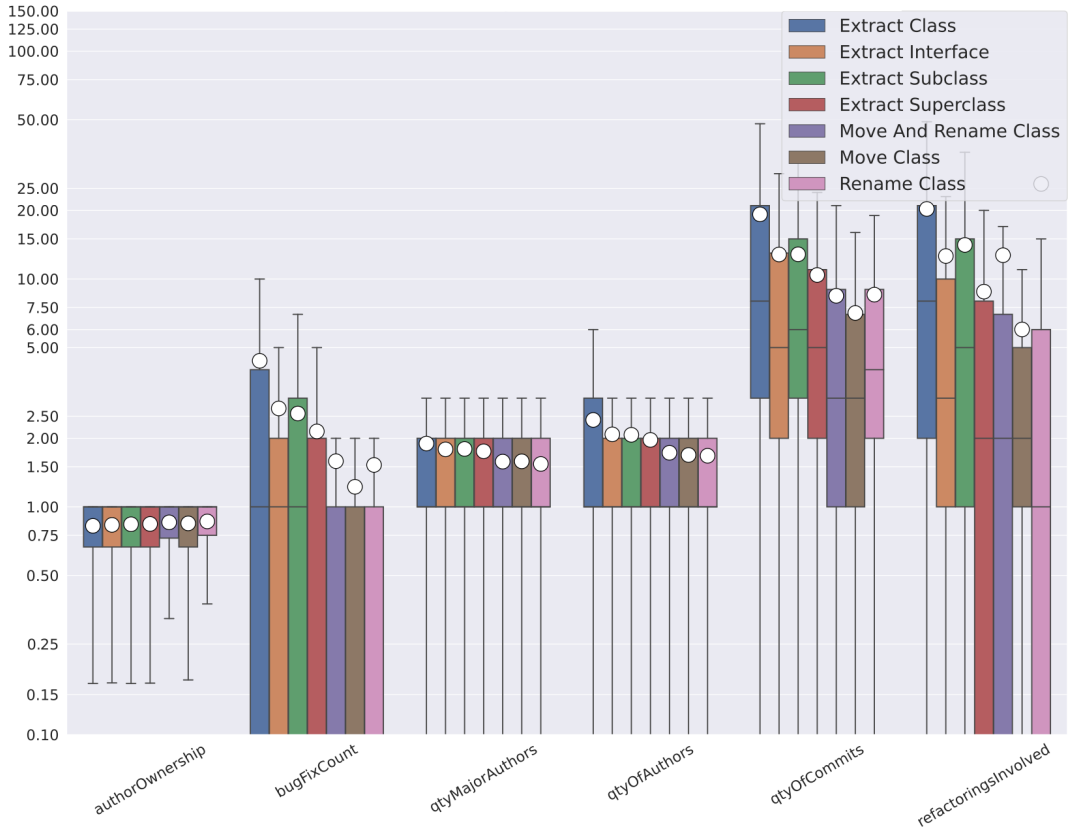


Figure 4.14: Distribution of Process- and Ownership metrics for all refactorings of Class-Level

Observation 7: *Rename* refactorings are mostly applied to classes controlled by a single author and few commits. Across all Refactoring-Levels *Rename* refactorings have often the lowest process- and ownership metrics within their respective level, except for Author Ownership for which the metrics are highest. This is particularly pronounced for the Class- and Method-Level. In contrast to the *Rename* refactorings, *Extract* refactorings consistently have the highest ownership metrics within each Refactoring-Level. These refactorings are not the most dominant refactorings within their levels, see Table A.1. Thus, they do not effect the overall level considerably. The visual analysis did

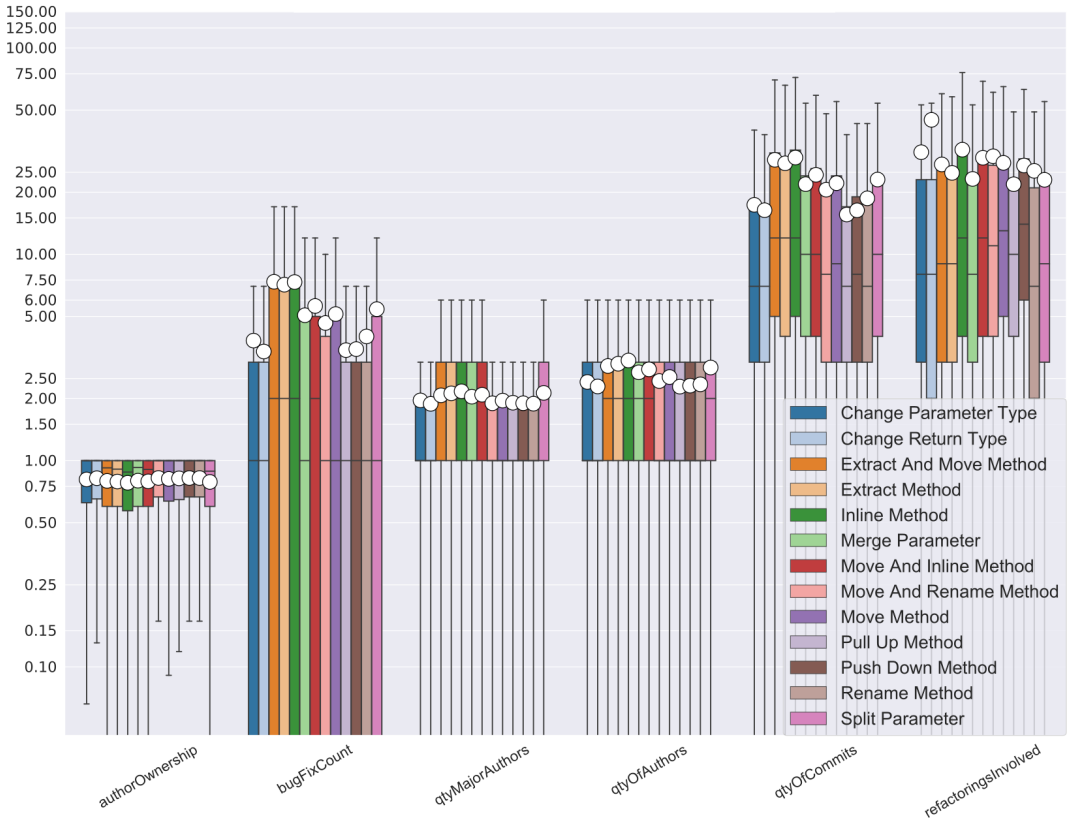


Figure 4.15: Distribution of Process- and Ownership Metrics for all refactorings of Method-Level

not identify another "category" of refactoring types which showed a cohesive distribution across the five Refactoring-Levels.

Observation 8: Author Ownership and Refactorings Involved decline significantly with increasing K. Fig. 4.16 displays the mean of the 6 process- and ownership metrics for all the K's. The analysis of the different K's reveals that only the two metrics Author Ownership and Refactorings Involved decline significantly with increasing K, see Fig. 4.16. The Author Ownership metric strongly declines from a mean of 0.6 at `K=15` to 0.2 at `K=60` (mean at 0.85 for Refactoring-Instances), for higher K's it stabilizes. The Quantity of Authors in contrast is very stable with a mean of 5 to 6 across all K's. Also, the quantity of major authors is only slowly declining with increasing K's from a mean of 2.3 at `K=15` to 1.5 at `K=60`. Stable-Instances with a low K have a very high count Refactorings Involved, with a mean above 40, but it declines quickly for K's larger than 25. Stable-Instances with `K=30` have a similar count of Refactorings Involved as Field-, Method- and Variable-Level Refactoring-Instances and

at `K=50` it is similar to Class-Level Refactoring-Instances. The Quantity of Commits (mean 91) is very coherent among different K's, see Fig. 4.16. And finally the mean Bugfix Count per instance slowly declines from 24 at `K=20` to 20 at `K=50` to 15 at `K=100`.

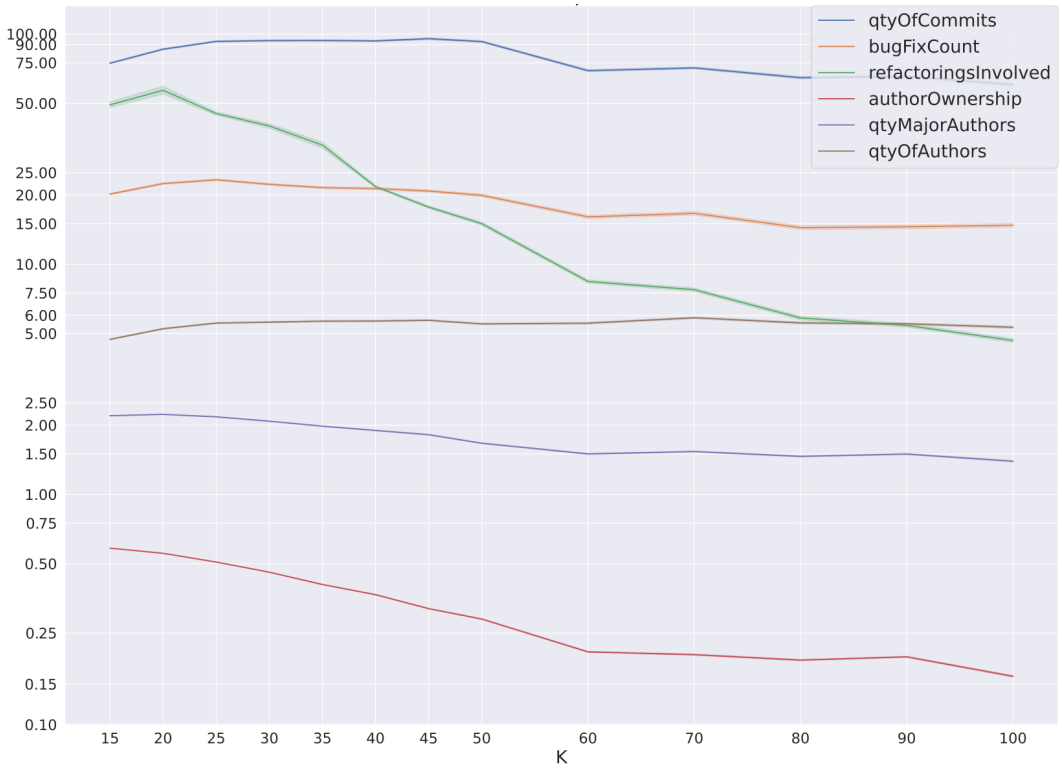


Figure 4.16: Distribution of process- and ownership Metrics for all K's

Observation 9: Stable-Instances are highly maintained classes. As only two of the metrics are strongly affected by the K of Stable-Instances, it is likely that the Stable-Instances are a subset of classes with very specific characteristics. Stable-Instances with a lower K are highly maintained and utilized classes, as indicated by the high Quantity of Commits, the high counts of Refactorings Involved and low Author Ownership. With increasing K's these classes were considered structurally sufficient early but still extended by their developers.

4.5 Conclusion

The refactoring data set presented in this thesis allows an in-depth analysis of the refactorings and stable classes in open source projects. The collected metrics can be used to analyze individual refactorings in detail. For both

Refactoring- and Stable-Instances the analyzed features are non-normally distributed and right skewed, except for the Author Ownership metrics which is not skewed.

RQ 1: How are features distributed among refactoring types and levels? The analyzed features are similarly distributed within the Refactoring-Levels and Class- and Other-Level refactorings show very similar feature distributions. The refactorings cover 94.85% of all collected classes. In general, refactorings are mostly applied to classes that are controlled by a single author. Class- and Other-Level refactorings occur most frequently in classes with few commits and few other refactorings. These classes are comparably simple and not very complex. Method-, Variable- and Field-Level refactorings occur in all development stages.

RQ 2: How are features distributed among Stable-Instances? Stable-Instances are a strong minority class with unique characteristics that can be clearly separated from the Refactoring-Instances by class level features, except for the Field-Level Stable-Instances. The class metrics are in general better and the classes are less complex compared to refactored classes. Additionally, the Stable-Instances cover only a maximum of 7.33% of the collected classes and with increasing K this fraction declines strongly. Furthermore, the analysis showed that selection of K only affects two class metrics and two class attributes significantly, with `K=25` and `K=60` being the most relevant thresholds in regard to the feature distribution.

RQ 3: What are implications of the distribution of process- and ownership metrics? The process- and ownership metrics can be used to distinguish between Refactoring-Instances and Stable-Instances. The effect of an increased K on the ownership metrics is minor, but are significant for the process metrics Refactorings Involved and Author Ownership. With increasing K the Stable-Instances have a significantly lower Author Ownership and number of Refactorings Involved.

Chapter 5

Recommending Refactorings via Machine Learning

In this chapter, we explore the use of machine learning for recommending software refactorings. We conducted three experiments to answer the Research Questions 4 to 7. The machine learning pipeline, classifier selection and model evaluation are explained in detail. The experimental set up and the results are layed out in this chapter. We finish this chapter with a conclusion answering the Research Questions.

5.1 Classifier Training

One of the main objectives of this thesis is the further exploration of machine learning algorithms for refactoring prediction, see Section 1.2 for more details. In order to achieve this objective, various binary classifiers have to be trained and the models evaluated. The training and data-preprocessing procedure is explained in detail in the upcoming section, see Fig. 3.1 for the entire procedure. The final phase, classifier evaluation, is described in Section 5.2. The created models are supposed to predict refactorings at all four considered levels: class, method, variable and field. The entire classifier training and data-preprocessing pipeline was implemented with Python 3.6+ and the source code can be found on GitHub.

5.1.1 Data Preprocessing

Our data preprocessing pipeline consists of eight steps, a simplified overview is shown in Fig. 3.1. The individual steps are explained here in more detail:

- **(i) Data Selection:** Selection of `k` (the commit threshold) for Stable-Instances and the refactoring type.

- **(ii) Data Split**: The data is split into training, validation and test (optional) data sets. The data is either split randomly by using 80% of the entire data set `d` for training and 20% for validation `v` or by using predefined splits, see Section 5.2.
- **(iii) Labeled Data:** All labeled positive- (refactorings) `r` and negative (stable) `s` instances are retrieved from the database and stored. To speed-up data retrieval the data is pre-cached on the local machine. Afterwards, the data is merged into `X` and the labels are moved to `Y`.
- **(iv) Balancing (optional):** Both `r` and `s` are balanced equally, this is optional and might be utilized differently in various training setups.
- **(v) Shuffle:** `d` is shuffled randomly with the pandas `sample()` function [1].
- **(vi) Feature Scaling:** All features are scaled with a Min-Max[2] scaler provided by the scikit-learn framework which scales all (numerical) features in the range `[0, 1]` . Scaling the features before training can improve the training speed of ML-algorithms [32, 31].
- **(vii) Feature Reduction:** Features are reduced with the selected classifier to maximize the results and speed of the classifier training.

5.1.2 Training

In this research we only considered supervised machine learning algorithms for binary classification. The input for each model is a feature vector `X` representing the entity, e.g. a class or method, at the current time and the model will predict if this entity should be refactored by a specific refactoring. Each model will only predict a single refactoring, thus to cover a variety of refactorings multiple models have to be combined. As scoring strategies, which are functions evaluating the overall quality of a model during hyperparameter tuning and training, `accuracy` and `f1-score` are used.

Classifier We utilize the following two binary classifiers in this work:

1. **Logistic Regression (LR):** In their research, Aniche et al. [4] found that Logistic regression yields competitive results for predicting refactorings. Furthermore, it is very efficient and fast to train [45].
2. **Random Forest (RF)[10]:** This was found to be the best performing classifier to refactoring prediction with similar data [4].

[1] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sample.html
[2] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html.sample.html

The implemented training pipeline uses the classifier from version 0.23.2 of the scikit-learn framework [55]. All of the selected classifiers also support parallelization, which drastically increases the training further, as we can utilize the full potential of the training clusters.

Feature Reduction Most classifiers' performance benefits from feature reduction [11, 12, 45]. Feature reduction, also referred to as dimensionality reduction or feature selection, reduces the number of features before training, e.g. instead of using all 47 features for a Class-Level refactorings only 25 relevant features are selected for training. This reduces the impact of the *Curse of Dimensionality* and thereby, can drastically reduce the necessary training time. Additionally, reducing the number of features also improves feature understanding and generalization of the model [12]. We chose recursive feature reduction (RFE) with cross validation (CV) to reduce the dimensionality. The algorithm recursively attempts to reduce the feature space by recursively removing the least important features from the feature set [26]. It was found to be one of the feature reduction algorithms for supervised machine learning with the highest accuracy [11].

Parameter Selection The two classifiers have a variety of relevant parameters that need to the configured to optimize results of the training. The parameters are hyper-tuned for the estimators with random search and cross validation. We utilize the function `RandomizedSearchCV` [3] from scikit-learn for this purpose. The algorithm tries 100 randomly sampled parameter sets and selects the best performing set in regard to the specified scoring strategy. The advantage over grid search is that a larger search space can be explored (to some extend) without increasing the exploration time by Randomized search. This is relevant for random forest as a variety of different parameters and values per parameter create a huge search space, see Table 5.1.

The parameter distribution considered in this research is shown in Table 5.1. The utilized parameters are explained hereafter:

1. **Logistic Regression:** *Max-iter* is the maximum number of iterations for the solver to converge. *Solver* is the algorithm of choice solving the optimization problem. We selected the SAGA solver, because it performs well on very large data sets and can handle non-convergence. *C* is the inverse of regularization strength.

2. **Random Forest:** *Max-depth* is the maximum depth of the generated decision trees. *Max-features* defines the number of considered features for the best split. *Min-samples-split* is the minimum number of samples to

[3] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

Classifier	Parameter	Search Space
Logistic Regression	max-iter: [100, 500, 1000, 2000, 5000, 10000] C: [uniform(0.01, 100) for i in range(0, 10)]	**60**
Random Forest	max_depth: [3, 6, 12, 24, 48, 96, None] max_features: [auto, log2, None] min_samples_split: [2, 3, 4, 5, 10] bootstrap: [True, False] criterion: [gini, entropy] n_estimators: [10, 50, 100, 150, 200, 250, 300]	**2940**

Table 5.1: Parameter distribution for the hyperparameter search for each classifier with the size of the search space

split a node. *Bootstrap* defines if all samples are used to build the trees. *Criterion* measures the quality of a split and *number of estimators* is the number of trees in the forest.

5.2 Classifier Evaluation

Evaluating the performance and quality of the different models comprises the third and last main objective of this thesis. This section lays out the details of the evaluation of the various classifiers and the specifications of the experiments, designed to answer Research Questions 4 - 7, see Section 1.3. In detail the following components are described: (i) the creation of the test data set, (ii) the computation and selection of metrics and (iii) the experiments.

5.2.1 Test Set

Each trained model is evaluated with a test set consisting of unseen projects. The test set contains 500 randomly selected projects which meet the following criteria in order to ensure that these projects are properly represented in the test set and to exclude potentially faulty samples:

- **Commit count >100**
- **Number Of production files >25**
- **Production LoC >2500**
- **Test LoC >500**
- **Data extraction finished**

The test and training sets are already separated in the database, thus we ensure all projects are unseen by the classifier and the evaluation is always comparable,

because the test samples remain the same. This has the advantage that we can later investigate the results in more detail and we can draw conclusions on how the model might perform in practice. For all predictions the instance id and prediction results are stored.

5.2.2 Evaluation Scores

For the evaluation, a multitude of metrics is computed. These metrics allow an in-depth analysis of the quality and behavior of a model to answer the Research Questions 4 to 7. The computed metrics are listed and explained in the following:

- **Accuracy [0,1]:** a frequently used metric, which shows the fraction of correct predictions. The accuracy score performs well in balanced data sets, as all samples have an equal weight.
- **F1-Score [0,1]:** the weighted average of precision and recall, its calculation is given in Eq. (5.1). This function attempts to balance both precision and recall and thus, also performs well in unbalanced data sets.
- **Precision [0,1]:** describes the ability of a classifier to predict positive samples correct.
- **Recall [0,1]:** describes the ability of a classifier to predict negative samples correct.
- **Confusion Matrix:** Consisting of true negative(TN), false negatives (FN), true positives (TP) and false positives (FP) the confusion matrix can be used to compute the previous metrics and allows better understanding of the behavior of the evaluated model, e.g. if a classifier always predicts true.
- **Classifier Feature Importance [0,1]:** The random forest classifier has an internal representation for the importance of each feature (`feature_importance`), the sum of which is 1. This can be used to analyse the results of the model and understand certain characteristics of the data set. The LR does not have a representation of features similar to the random forest classifier, but `coef_` stores all features in the decision function with their coefficients.
- **Permutation Importance [0,1]:** Estimates the importance of each feature by permutating a feature vector and then evaluating the quality of predictions with the permutated feature compared to a baseline metric. The permutation importance is the difference between the baseline metric and the permutated one.

$$F1 = 2 * \frac{precision * recall}{precision + recall} \tag{5.1}$$

5.2.3 Setup of the Experiments

This section describes the experiments conducted to answer the Research Questions 5 to 7. It is important to make some remarks before explaining the details of each experiment. The amount of experiments and different setups would result in a too large number of models to be trained. Thus, we reduced the number of considered refactoring types from 40 to 11. The selection consists of 11 structural refactorings, from all 4 levels, with sufficient samples in the data and different feature distributions, see Table 5.2. The training was done on two machines with these specifications:

- Ubuntu 18.04.2 LTS VM with 40 CPU cores and 396GB of RAM
- Ubuntu 18.04.2 LTS VM with 14 CPUs and 50 GB of RAM

The parameter sets were also analyzed for the different experiments.

Level	**Class**	**Method**	**Variable**	**Field**
Refactorings	Extract Class	Inline Method		Move Attribute
	Move Class	Pull-Up Method	Parameterize Variable	Pull Up Attribute
	/	Push-Down Method	Replace Variable With Attribute	Replace Attribute
	/	Merge Parameter	/	/
Count	2	4	3	3

Table 5.2: Selected refactorings for the classifier training per level with total count

5.3 Reproduction Experiment

In this section we tackle ***RQ 4: How does the prior approach perform with the new data set?***. We use the original design of the machine learning pipeline, as presented by Aniche et al. [4] on the newly created data set.

5.3.1 Experimental Setup

The original design of the machine learning pipeline was applied to the new data set. The main differences to the machine learning pipeline previously presented (see Section 5.1) are as follows:

- `K=50`
- *80%/20% train/test split* used for training and initial evaluation

- *No feature reduction*: the features are not reduced before the classifier training.
- *Refactoring types*: for this experiment all 20 refactoring types of the prior experiment are considered.

We made some additions to the initial approach to analyze the results of the experiment in more detail:

- *Improved hyperparameter set*: The parameter set was significantly extended and adjusted to this task, especially for the RF.
- *Permutation feature importance*: We store the permutation importance of every feature.
- *Predictions are traceable*: We store all results including the sample identifier for every prediction to explore the prediction results in detail.

Similar to the authors of the original paper, we reported how often a feature appears in the Top-1, Top-5 and Top-10 of the most important features as reported by each model and the permutation importance.

5.3.2 Results

We display the performance of the new models in and and compare it to the results reported by Aniche et al. [4] in Table 5.3. Furthermore, we identify the most important features for the performance of the models. We analyze the feature distributions of mis-classified samples in order to identify thresholds and patterns in the models.

Table 5.3 displays the accuracy, F1-score, recall and precision for the RF and LR models from the original paper and the reproduction experiment with a random train/ test split of 80%. More detailed statistics of the reproduction experiment can be found in the online appendix [20].

Observation 1: All classifiers perform better. Across both classifier types (LR and RF) and all Refactoring-Levels and types, the newly trained models perform significantly better in regard to all metrics compared to the prior results reported by Aniche et al. [4]. The new RF models have a mean **accuracy of 99%** exceeding the mean accuracy of the original ones by 6%. For the LR, the mean accuracy improves by 13% to 92%. Only for the *Move Class* refactoring RF performance slightly decreases with an accuracy of 97% instead of 98%. For the RF, both precision and recall were improved by an average of 6% and 7%. This is very different for the LR, because on one hand precision only improves for the Method- (2%) and Variable-Level (3%) but on the other hand drops for the Class-Level by 5%.

	Random Forest				Random Forest Original			Logistic Regression				Logistic Regression Original		
	Acc	**F1**	**Pr**	**Re**	**Acc**	**Pr**	**Re**	**Acc**	**F1**	**Pr**	**Re**	**Acc**	**Pr**	**Re**
Class-Level														
Extract Class	0.97	0.97	0.96	0.98	0.85	0.93	0.89	0.92	0.92	0.89	0.96	0.78	0.91	0.82
Extract Interface	0.97	0.97	0.96	0.99	0.93	0.92	0.92	0.91	0.91	0.87	0.96	0.83	0.93	0.87
Extract Subclass	0.97	0.97	0.96	0.99	0.92	0.94	0.93	0.92	0.92	0.89	0.95	0.85	0.94	0.89
Extract Superclass	0.97	0.97	0.95	0.98	0.91	0.93	0.92	0.91	0.92	0.88	0.96	0.84	0.94	0.88
Move And Rename Class	0.96	0.96	0.95	0.97	0.95	0.95	0.95	0.91	0.92	0.88	0.95	0.89	0.93	0.91
Move Class	0.97	0.97	0.95	0.98	0.98	0.97	0.98	0.92	0.92	0.89	0.95	0.92	0.96	0.94
Rename Class	0.96	0.96	0.96	0.97	0.95	0.94	0.94	0.91	0.91	0.87	0.95	0.87	0.94	0.90
Method-Level														
Extract And Move Method	0.99	0.99	0.98	1.00	0.90	0.81	0.86	0.92	0.92	0.90	0.95	0.72	0.86	0.77
Extract Method	0.99	0.99	0.99	1.00	0.80	0.92	0.84	0.93	0.93	0.91	0.95	0.80	0.87	0.82
Inline Method	1.00	1.00	1.00	1.00	0.97	0.97	0.97	0.92	0.92	0.89	0.96	0.72	0.88	0.77
Move Method	1.00	1.00	1.00	1.00	0.99	0.98	0.99	0.92	0.92	0.89	0.96	0.72	0.87	0.76
Pull Up Method	1.00	1.00	1.00	1.00	0.99	0.94	0.96	0.94	0.94	0.91	0.97	0.78	0.90	0.82
Push Down Method	1.00	1.00	1.00	1.00	0.97	0.83	0.90	0.94	0.94	0.92	0.97	0.75	0.89	0.80
Rename Method	1.00	1.00	1.00	1.00	0.79	0.85	0.81	0.92	0.92	0.89	0.96	0.77	0.89	0.80
Variable-Level														
Extract Variable	1.00	1.00	1.00	1.00	0.90	0.83	0.87	0.95	0.95	0.93	0.97	0.80	0.83	0.82
Inline Variable	1.00	1.00	0.99	1.00	0.94	0.96	0.95	0.91	0.91	0.88	0.95	0.76	0.86	0.79
Parameterize Variable	0.99	0.99	0.99	1.00	0.93	0.92	0.92	0.92	0.92	0.88	0.96	0.75	0.85	0.79
Rename Parameter	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.92	0.93	0.90	0.96	0.79	0.88	0.83
Rename Variable	1.00	1.00	1.00	1.00	1.00	0.99	0.99	0.91	0.92	0.88	0.95	0.77	0.85	0.80
Replace Variable With Attribute	0.99	0.99	0.99	1.00	0.94	0.92	0.93	0.92	0.92	0.89	0.96	0.79	0.88	0.82
Averages														
Class-Level	0.97	0.97	0.96	0.98	0.93	0.94	0.93	0.91	0.92	0.88	0.95	0.85	0.94	0.89
Method-Level	1.00	1.00	1.00	1.00	0.92	0.90	0.90	0.93	0.93	0.90	0.96	0.75	0.88	0.79
Variable-Level	1.00	1.00	1.00	1.00	0.95	0.94	0.94	0.92	0.93	0.89	0.96	0.78	0.86	0.81
Total	0.99	0.99	0.98	0.99	0.93	0.92	0.93	0.92	0.92	0.89	0.96	0.80	0.89	0.83

Table 5.3: Comparison of the quality metrics for the RF and LR classifier from both the reproduction and original experiment, evaluated with the randomly split data set

Overall, the most significant improvements for the random forest classifier were made on three refactorings. The first is *Extract Class* of the Class-Level with an accuracy increase of 12%, and the two Method-Level refactorings *Extract Method* and *Rename Method* with an accuracy improvement of 19% and 21%.

Observation 2: RF generalizes better than LR. The RF generalizes better to unseen projects than LR. The RF models achieve an average accuracy of 89% and an average F1-score of 90% for all refactoring types compared to a mean accuracy of 84% and a mean F1-score of 85% for the LR. The reduced performance compared to the evaluation on the random split set can be explained by the strongly reduced precision of the models. Precision is drastically lower for the evaluation of the models on unseen projects, on average 15% lower for RF and 12% for LR. Recall, on the other hand, did not change at all. It is also observed that the precision for the Class-Level refactorings is 6% higher than for the Method- and Variable-Level refactorings. Due to the very high recall on all three levels the difference in the accuracy and F1-score is only 3%. These

findings are in line with the results reported by Aniche et al. [4].

Observation 3: *Move* and *Rename* refactorings perform significantly worse in generalized contexts. The performance of the newly trained classifiers on unseen projects is displayed in Table 5.3. A more detailed analysis of

	Random Forest Test-Set				**Logistic Regression Test-Set**			
	Acc	**F1**	**Pr**	**Re**	**Acc**	**F1**	**Pr**	**Re**
Class-Level								
Extract Class	0.95	0.95	0.92	0.98	0.82	0.84	0.76	0.94
Extract Interface	0.95	0.95	0.92	0.98	0.84	0.86	0.77	0.96
Extract Subclass	0.95	0.96	0.93	0.99	0.85	0.86	0.78	0.96
Extract Superclass	0.91	0.92	0.87	0.97	0.79	0.82	0.72	0.95
Move And Rename Class	0.9	0.9	0.85	0.97	0.78	0.81	0.71	0.95
Move Class	0.9	0.91	0.85	0.98	0.83	0.84	0.76	0.94
Rename Class	0.84	0.86	0.77	0.97	0.88	0.89	0.82	0.97
Method-Level								
Extract And Move Method	0.89	0.9	0.82	1	0.87	0.88	0.81	0.95
Extract Method	0.9	0.91	0.83	0.99	0.87	0.88	0.82	0.96
Inline Method	0.9	0.91	0.83	1	0.84	0.86	0.78	0.96
Move Method	0.87	0.88	0.79	1	0.84	0.86	0.77	0.96
Pull Up Method	0.89	0.9	0.82	1	0.85	0.87	0.78	0.97
Push Down Method	0.9	0.91	0.83	1	0.86	0.88	0.79	0.98
Rename Method	0.85	0.87	0.77	1	0.85	0.86	0.78	0.97
Variable-Level								
Extract Variable	0.93	0.94	0.88	1	0.9	0.9	0.85	0.97
Inline Variable	0.89	0.9	0.82	1	0.78	0.81	0.71	0.96
Parameterize Variable	0.89	0.9	0.83	0.99	0.81	0.84	0.74	0.97
Rename Parameter	0.82	0.84	0.73	1	0.82	0.84	0.74	0.97
Rename Variable	0.86	0.87	0.78	1	0.78	0.81	0.71	0.95
Replace Variable With Attribute	0.86	0.88	0.79	0.99	0.8	0.83	0.73	0.96
Averages								
Class-Level	0.91	0.92	0.87	0.98	0.83	0.85	0.76	0.95
Method-Level	0.89	0.90	0.81	1.00	0.83	0.85	0.76	0.96
Variable-Level	0.88	0.89	0.81	1.00	0.83	0.85	0.77	0.95
Total	0.89	0.90	0.83	0.99	0.84	0.85	0.77	0.95

Table 5.4: Comparison of the quality metrics for the Random Forest and Logistic Regression classifier evaluated with the test set

the precision scores shows that the change in precision varies strongly for the different refactoring types. For three of the Class-Level refactorings, *Extract Class*, *Extract Interface* and *Extract Subclass*, the precision was only reduced by 4% for the RF instead of 8% for the *Move Class* refactoring or 19% for *Rename Class* refactoring. A similar pattern can be observed for LR. Overall, for *Rename* and *Move* refactoring types, the precision was reduced significantly by an average of 18.86% for RF and by 14.57% for LR. This result is in contrast to the findings by Aniche et al. [4].

Observation 4: The models struggle to separate Stable-Instances from Refactoring-Instances. False positives (FP) are the main reason for the re-

Fraction	Random Forest				Logistic-Regression			
	TN	FP	FN	TP	TN	FP	FN	TP
Class-Level								
Extract Class	91.40%	8.60%	1.72%	98.28%	70.75%	29.25%	5.81%	94.19%
Extract Interface	91.40%	8.60%	1.51%	98.49%	71.61%	28.39%	3.87%	96.13%
Extract Subclass	92.26%	7.74%	1.29%	98.71%	72.90%	27.10%	3.87%	96.13%
Extract Superclass	84.95%	15.05%	2.58%	97.42%	63.44%	36.56%	5.16%	94.84%
Move And Rename Class	82.80%	17.20%	3.23%	96.77%	62.15%	37.85%	5.38%	94.62%
Move Class	82.58%	17.42%	1.94%	98.06%	70.97%	29.03%	5.81%	94.19%
Rename Class	71.40%	28.60%	3.01%	96.99%	78.49%	21.51%	2.80%	97.20%
Method-Level								
Extract And Move Method	78.29%	21.71%	0.18%	99.82%	77.93%	22.07%	4.90%	95.10%
Extract Method	80.09%	19.91%	0.53%	99.47%	78.29%	21.71%	3.75%	96.25%
Inline Method	79.56%	20.44%	0.15%	99.85%	72.11%	27.89%	3.78%	96.22%
Move Method	73.29%	26.71%	0.21%	99.79%	71.64%	28.36%	3.81%	96.19%
Pull Up Method	78.61%	21.39%	0.32%	99.68%	72.58%	27.42%	2.60%	97.40%
Push Down Method	79.41%	20.59%	0.21%	99.79%	74.24%	25.76%	2.07%	97.93%
Rename Method	69.45%	30.55%	0.03%	99.97%	71.91%	28.09%	2.60%	97.40%
Variable-Level								
Extract Variable	86.73%	13.27%	0.17%	99.83%	82.65%	17.35%	3.22%	96.78%
Inline Variable	78.66%	21.34%	0.19%	99.81%	60.05%	39.95%	3.91%	96.09%
Parameterize Variable	78.99%	21.01%	0.50%	99.50%	65.36%	34.64%	3.45%	96.55%
Rename Parameter	63.41%	36.59%	0.08%	99.92%	66.35%	33.65%	3.22%	96.78%
Rename Variable	71.42%	28.58%	0.03%	99.97%	60.21%	39.79%	4.65%	95.35%
Replace Variable With Attribute	73.37%	26.63%	0.89%	99.11%	63.98%	36.02%	4.44%	95.56%
Averages								
Class-Level	85.25%	14.75%	2.18%	97.82%	70.05%	29.95%	4.67%	95.33%
Method-Level	76.96%	23.04%	0.23%	99.77%	74.10%	25.90%	3.36%	96.64%
Variable-Level	75.33%	24.67%	0.29%	99.71%	66.54%	33.46%	3.82%	96.18%

Table 5.5: Confusion-matrix for evaluation of the newly trained classifier for the reproduction experiment, fraction of total, red indicates poor performance and green good

duced precision in the classification task in the unseen projects. In Table 5.5 we show the confusion matrices for all refactoring types and both classifiers. We can observe clearly that false positives are the main contributor for the reduced precision, the average FP rate using RF for the Class-Level is 14.75%, for the Method-Level 23.04% and 24.67% for the Variable-Level. Interestingly, for the Class-Level refactorings the LR has a significantly higher average of FP rate of 29.95% compared to the Method-Level refactorings with 25.90%, this is in contrast to the results for the RF.

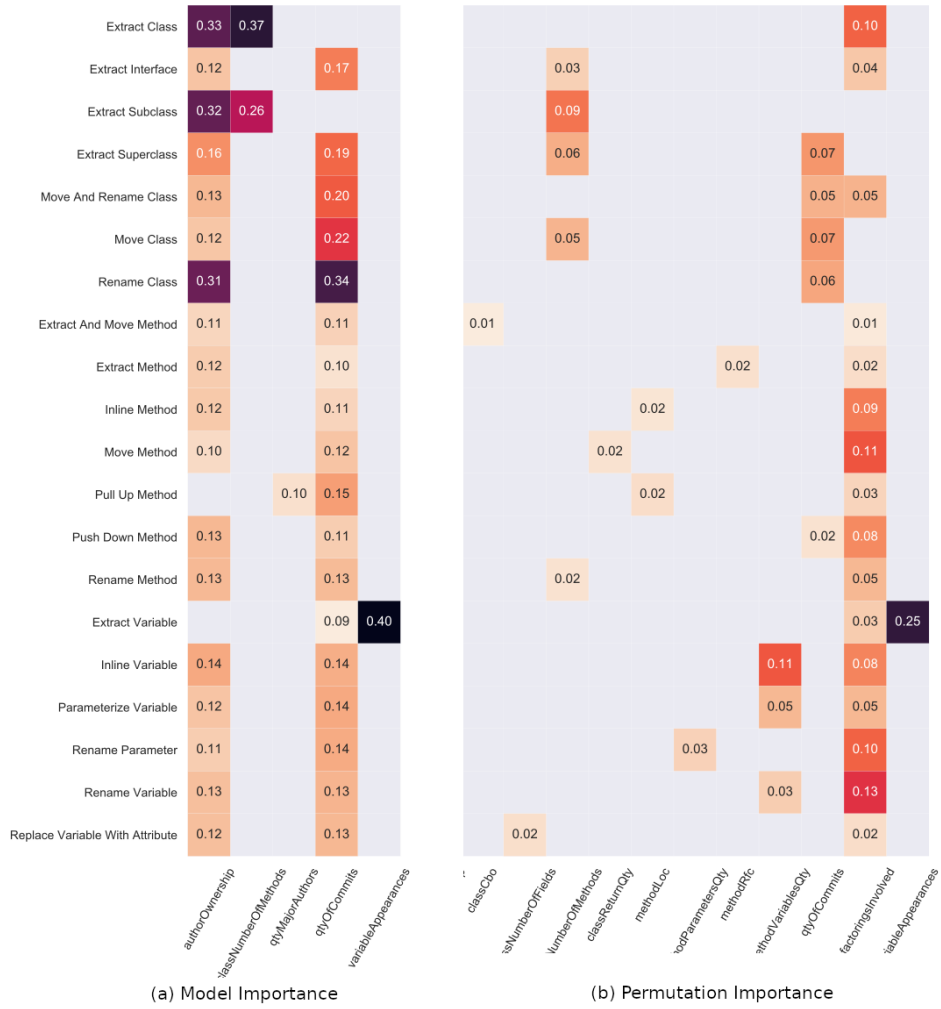


Figure 5.1: Top-2 feature importances for RF as reported by the model and computed with permutation importance

Observation 5: Permutation importance and model feature importance differ strongly. The importances of the features are displayed in Fig. 5.1 for RF and in Fig. 5.2 for LR. Furthermore, Table 5.7 and Table 5.6 show the Top-N features for both classifiers for all three Refactoring-Levels. The features identified as important by the two importance measures disagree on a variety of features. The RF models value Quantity of Commits and Author Ownership highly, in contrast to the permutation importance applied to RF, which only identified Quantity of Commits as a Top-2 feature for four of the refactoring types and the Author Ownership for three Class-Level refactorings. For the LR models, Author Ownership was identified as the feature with highest performance

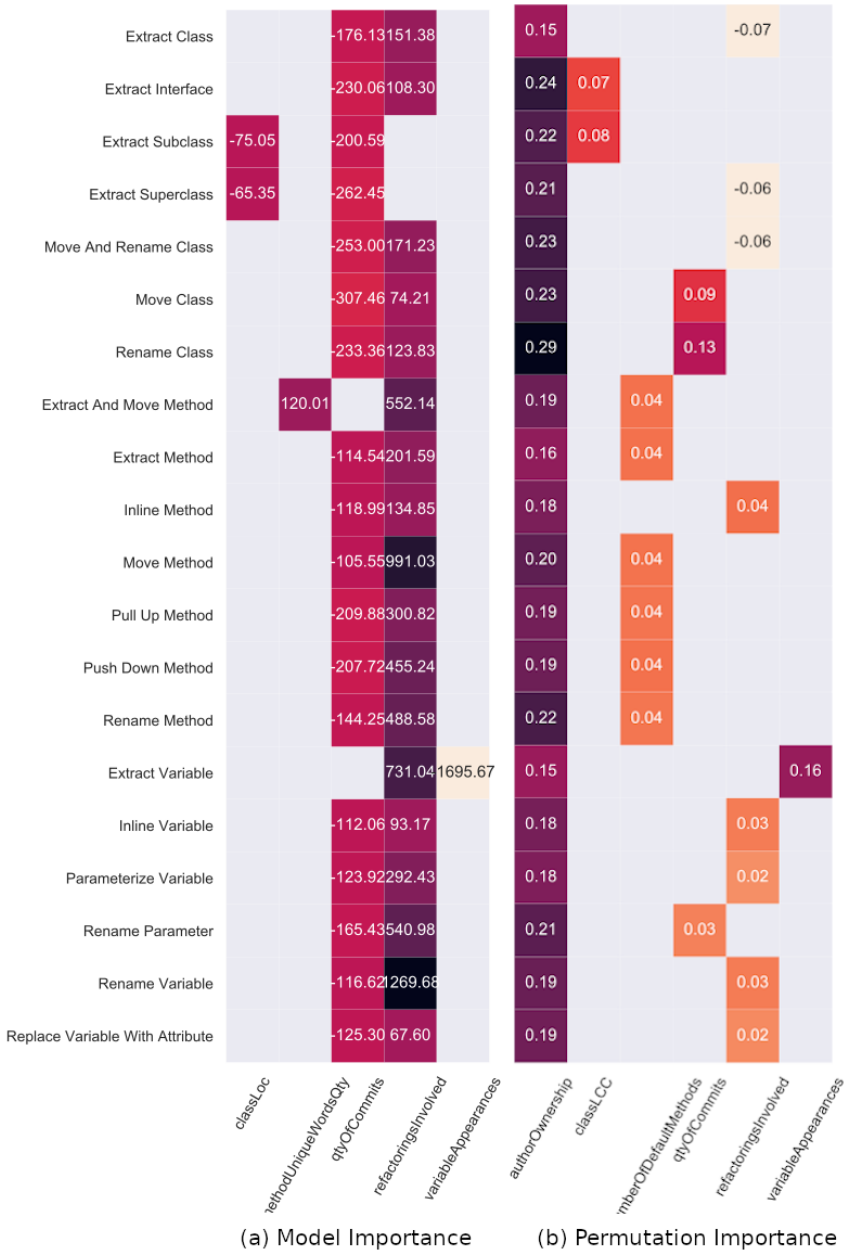


Figure 5.2: Top-2 feature importances for LR as reported by the model and computed with permutation importance

impact by the permutation importance measure, compared to Quantity of Commits and Refactorings Involved by the coefficients extracted from the models. The weights assigned by the LR models for Author Ownership and Refactorings Involved are not even in the Top-10 for a single refactoring type.

Observation 6: LR models highly rely on few process- and ownership metrics. The Top-5 of the most important features for LR models are strongly dominated by process- and ownership metrics. The metrics Author Ownership is for all models the dominant metric, followed by Quantity of Commits and Refactorings Involved. The metrics Quantity of Major Authors is only relevant for five models, Bugfix Count for two and Quantity of Minor Authors for none. Other highly relevant metrics are LCC for the Class-Level and Number of Default Methods for Method-Level refactoring models based on LR classifier.

Observation 7: RF models rely on a variety of metrics. Permutation importance identified a large variety of features relevant for the classification performance for the different RF models. In the Top-2 of the most important features are 12 and in the Top-5 are 21 different feature, compared to the model importance, which identified only 5 different features in the Top-2 and 12 different features for the model importance extracted from the RF. Additionally, for the *Extract Variable* refactoring, Variable Appearances is the most important feature. For the RF models, ownership metrics are only relevant for Class-Level refactorings, only three models for the Class-Level refactorings *Extract Class*, *Extract Subclass* and *Rename Class* highly rely on the Author Ownership metric.

Observation 8: The most relevant features are of Class-Level. In addition to the class file based process- and ownership metrics, the models for all three Refactoring-Levels predominantly use Class-Level features such as Number of Methods or Number of Fields. In the Top-5 of feature importances extracted from the models, Method- and Variable-Level features occur only 4 times and in the Top-5 of the permutation importances they occur 7 times out of a total of 65 each. No Method- and Variable-Level feature is in the Top-3 or higher. The reported Method- and Variable-Level features are almost all unique: Method RFC, Method CbO, Number of Unique Words in a Method, Method RFC, Method SLoC and Maximum Number of Nested Blocks in a Method. In their work, Aniche et al. [4] reported as well that Class-Level features are also very important for Method- and Variable-Level refactorings.

Observation 9: Clear classification thresholds for the relevant process- and ownership metrics are observable. An analysis of the feature distribution of the classified samples shows clear thresholds for the most relevant process- and ownership metrics for the classification, namely Author Ownership and Quantity of Major Authors. These thresholds are soft boundaries covering almost all of the samples, but some outliers still exist. The RF models classify samples with an authorship below 0.5 as Stable-Instances and above 0.6 as refactorings. This can be clearly observed for the FP, as their Author Ownership

metrics are predominantly above this threshold. For Quantity of Major Authors the threshold for a positive classification is 1. For the LR models, the threshold for Author Ownership is around 0.5, but for Quantity of Major Authors no distinct threshold could be identified.

For the metric Refactorings Involved, most of the samples can be separated by a threshold of 11, with samples with more than 11 refactorings predominantly being classified as positive by the RF models. For the Quantity of Commits, it can be clearly separated among . For the other process- and ownership metrics, feature distributions in the classification results vary stronger and patterns were not observed. This is particularly interesting for the metric Refactorings Involved, as it is for all Method- and Variable-Level refactorings in the Top-2 of the most important features. The results for the (few) False Negative classifications are not clear, feature distribution varies strongly for all refactoring types across all levels and covers the entire range of features for both Stable- and Refactoring-Instances.

	Random Forest	**Logistic Regression**
Class-Level		
Top-1:	authorOwnership (3), qtyOfCommits (3), refactoringsInvolved (1)	authorOwnership (7)
Top-5:	qtyOfCommits (7), refactoringsInvolved (6), classNumberOfMethods (6), classCbo (4), authorOwnership (3)	authorOwnership (7), classLCC (7), qtyOfCommits (7), refactoringsInvolved (6), qtyMajorAuthors (3)
Top-10	qtyOfCommits (7), classNumberOfMethods (7), classCbo (7), refactoringsInvolved (6), classNumberOfFields (5)	authorOwnership (7), classLCC (7), qtyOfCommits (7), refactoringsInvolved (6), classNumberOfPrivateFields (6)
Method-Level		
Top-1:	refactoringsInvolved (7)	authorOwnership (7)
Top-5:	refactoringsInvolved (7), methodLoc (5), classCbo (4), classNumberOfMethods (4), qtyOfCommits (3)	authorOwnership (7), classNumberOfDefaultMethods (7), qtyOfCommits (7), refactoringsInvolved (5), classNumberOfMethods (4)
Top-10	refactoringsInvolved (7), classCbo (6), classNumberOfMethods (6), methodLoc (5), qtyOfCommits (5)	authorOwnership (7), classNumberOfDefaultMethods (7), qtyOfCommits (7), refactoringsInvolved (6), qtyMajorAuthors (5)
Variable-Level		
Top-1:	refactoringsInvolved (4), variableAppearances (1), methodVariablesQty (1)	authorOwnership (5), variableAppearances (1)
Top-5:	refactoringsInvolved (6), classNumberOfFields (4), methodVariablesQty (4), startLine (3), qtyOfCommits (3)	authorOwnership (6), qtyOfCommits (6), refactoringsInvolved (5), classVariablesQty (3), qtyMajorAuthors (2)
Top-10	refactoringsInvolved (6), classNumberOfFields (6), startLine (5), classNumberOfPrivateFields (5), classUniqueWordsQty (5)	authorOwnership (6), qtyOfCommits (6), bugFixCount (5), refactoringsInvolved (5), classVariablesQty (4)

Table 5.6: Most relevant features as measured by permutation importance for all three Refactoring-Levels. Only the top 5 features for the Top-N are displayed here.

5.3.3 Discussion

The reproduction experiment can reproduce the main findings of the initial work by Aniche et al. [4], namely (i) the great performance of the models, (ii) the high importance of process- and ownership metrics for the refactoring prediction and (iii) the good performance of the models on unseen projects. Furthermore, we can verify that RF models perform better than LR models, which nonetheless perform well with an average accuracy of 92%. Moreover, features collected at class level such as Number of Methods outweigh all other levels.

In the following subsections, the results of the experiment are discussed in more detail. As RF models perform significantly better than LR models and the LR models only served as a baseline, only RF is discussed in more detail.

	Random Forest	Logistic Regression
Class-Level		
Top-1:	qtyOfCommits (5), classNumberOfMethods (1), authorOwnership (1)	qtyOfCommits (7)
Top-5:	authorOwnership (7), qtyOfCommits (7), qtyOfAuthors (4), qtyMajorAuthors (4), bugFixCount (4)	qtyOfCommits (7), classStringLiteralsQty (7), refactoringsInvolved (6), classLoc (5), bugFixCount (3)
Top-10	classNumberOfMethods (7), authorOwnership (7), qtyOfCommits (7), refactoringsInvolved (7), classCbo (7)	qtyOfCommits (7), refactoringsInvolved (7), classStringLiteralsQty (7), classWmc (6), classNumberOfPrivateFields (6)
Method-Level		
Top-1:	authorOwnership (4), qtyOfCommits (3)	refactoringsInvolved (7)
Top-5:	qtyOfCommits (7), authorOwnership (7), qtyOfAuthors (7), qtyMajorAuthors (7), refactoringsInvolved (6)	refactoringsInvolved (7), qtyOfCommits (7), classLcom (5), startLine (3), classNumberOfFinalFields (3)
Top-10	qtyOfCommits (7), authorOwnership (7), qtyOfAuthors (7), qtyMajorAuthors (7), refactoringsInvolved (7)	refactoringsInvolved (7), qtyOfCommits (7), classStringLiteralsQty (6), classNumberOfFinalFields (6), classLcom (5)
Variable-Level		
Top-1:	qtyOfCommits (4), variableAppearances (1), authorOwnership (1)	refactoringsInvolved (3), qtyOfCommits (2), variableAppearances (1)
Top-5:	qtyOfCommits (6), authorOwnership (6), qtyMajorAuthors (6), qtyOfAuthors (6), refactoringsInvolved (4)	refactoringsInvolved (6), qtyOfCommits (6), classStringLiteralsQty (4), classNumberOfStaticMethods (3), classLcom (2)
Top-10	qtyOfCommits (6), authorOwnership (6), qtyMajorAuthors (6), qtyOfAuthors (6), refactoringsInvolved (6)	refactoringsInvolved (6), qtyOfCommits (6), classStringLiteralsQty (5), classNumberOfFinalFields (5), classNumberOfStaticMethods (5)

Table 5.7: Most relevant features as extracted from the models for all three Refactoring-Levels. Only the 5 most frequent features for the Top-N are displayed here.

Accuracy

The results of the first experiment show a very high mean accuracy of 99% for all models, which is 6% higher compared to the original results. A likely explanation for the great performance is that the very unique characteristics of the Stable-Instances are more pronounced. The analysis in Chapter 4 showed that the Stable-Instances are a unique subset of classes. Due to the addition of 19 refactoring types in the data collection, the unique characteristics of the Stable-Instances are likely to be more pronounced than in the first work. A class is less likely to be considered stable if more refactoring types are collected, because more refactorings might be detected. Furthermore, the models are likely to have seen some of the Refactoring- and Stable-Instances during the training as they data set was split randomly. This makes the classification problem considerably easier for the models, as they already know a part of the data. Thus, the significant performance improvements of the newly trained models can be explained. The increased number of samples is not a likely explanation for the improved performance, as all refactoring types with very few samples in the initial approach (e.g. *Extract Subclass* (6436) or *Extract Variable* (6709)) did not under-perform. Additionally, the drastic performance improvements for the three Refactoring-Types *Extract Class*, *Extract Method* and *Rename Method* were possible due to the poor performance of the prior models on these specific refactorings.

In unseen projects, the models show still a very high recall (99%), but the mean precision (83%) is 4% lower than in the original work. Furthermore, it can be observed that Class-Level refactorings generalize better than other Refactoring-Levels with a mean precision of 87% compared to 81% for Method-

and Variable-Level refactorings. The higher performance of the Class-Level models is caused by the great performance of the models for *Extract Class*, *Extract Interface* and *Extract Subclass* refactorings. An analysis of their distribution of process- and ownership metrics reveals that the metric Refactorings Involved is significantly higher (median 8) compared to the other refactoring types of the Class-Level (median 3). These major improvements were possible, because the initial models were performing particularly bad on these refactoring types, with low recall and precision scores.

Important Features

For both classifier types and the two importance measures, four of the six process- and ownership metrics (Quantity of Major Authors, Author Ownership, Refactorings Involved and Quantity of Commits) were identified as highly relevant. The high relevance of the two process metrics Author Ownership and Quantity of Major Authors matches with the findings of the data analysis (see Chapter 4). Classes that are maintained by a high number of different authors are very likely to be considered stable and classes with a high Author Ownership (>0.6) are likely to be refactored, especially Class-Level refactorings. Furthermore, almost all of the FP's for all models have a high Author Ownership and more than one major author. A more likely refactoring of classes maintained by a single author can be attributed to the lower integration and coordination efforts. Unfortunately, we are missing a metric measuring the number of external references to a class, in order to further explore this topic.

The analysis of the process metrics Refactorings Involved and Quantity of Commits revealed no clear conclusion. Classes with a higher number of commits (>10) and more than 10 refactorings involved make up the majority of the false positives. Nonetheless, these samples have metrics that are well within the range of the samples (75% quartile) the models correctly classified as negative. The two process metrics Quantity of Commits and Refactorings Involved were classified as highly relevant by both important measures. The mismatch of the two importance measures for the ownership metrics Author Ownership and Quantity of Major Authors is likely due to their strong correlation. Permutation importance suffers from misleading values and suggestions on features with strong correlation. In case two features are strongly correlated, the model still has access to the permuted feature through the correlated feature(s) [65]. The metric Quantity of Minor Authors was not identified as relevant and, as it is not strongly correlated to any other metric, it can be concluded that Quantity of Minor Authors is not relevant for the prediction.

Multiple factors likely contribute to the high importance of Class-Level features for the prediction performance of all models. First, 47 out of total 69 features are Class-Level features. Second, due to the design of the classifi-

cation experiment, for Method- and Variable-Level refactorings, the model is shown many Stable-Instances sharing the same class-level features. Therefore, it is most likely the "best" option for the classifier to improve the classification of these features as a mistake is more costly. Third, the Class-Level features already capture a large variety of characteristics.

5.4 Influence of Different Commit Thresholds

In this section, we answer ***RQ 5: How does the selection of the K effect classifier performance?*** and ***RQ 6: How is the performance of the classifier on Field-Level refactorings?***. First, we describe the setup of the experiment and display the results. Afterwards we discuss the results and answer the two aforementioned RQs.

5.4.1 Experimental Setup

For this experiment, we use the ML pipeline as described in Section 5.1. Also, all results were stored and analyzed as in the previous experiment, see Section 5.3. For the evaluation of the models only the unseen test set (see Section 5.2.1) was used.

To evaluate the different K's, we selected the following K's based on the results of the data analysis (see Chapter 4):

- **K=15:** It is the lowest K in the data set with the most Stable-Instances collected (total 26.99 million). Stable-Instances with this K have the highest similarity to the Refactoring-Instances (see Section 4.3).
- **K=25:** The data analysis showed that `K=25` is an interesting K. Furthermore, with total of 6.98 million Stable-Instances this threshold still contains many training samples.
- **K=50:** This is the baseline for comparison with the results of the reproduction experiment and the earlier experiment conducted by Aniche et al. [4]. Furthermore, the data analysis showed that for Stable-Instances with `K>50` the metrics are very stable, see Chapter 4.

5.4.2 Results

Table 5.8 displays the performance of the models using the three K's for all 11 refactoring types. All results for this experiment can be found in the online appendix [20]. Please note, if not specified otherwise the results in the following section always refer to the RF models and do not include the LR models, as the RF models perform significantly better and are therefore of greater interest.

	K=15			K=25			K=50		
Refactoring	**Acc**	**Pr**	**Re**	**Acc**	**Pr**	**Re**	**Acc**	**Pr**	**Re**
Class-Level									
Extract Class	0.88	0.83	0.96	0.92	0.88	0.97	0.95	0.92	0.98
Move Class	0.87	0.85	0.91	0.91	0.88	0.94	0.9	0.85	0.98
Method-Level									
Inline Method	0.81	0.81	0.79	0.8	0.72	0.98	0.9	0.83	1
Pull Up Method	0.83	0.85	0.8	0.86	0.81	0.96	0.89	0.82	1
Push Down Method	0.84	0.89	0.78	0.9	0.88	0.92	0.9	0.83	1
Merge Parameter	0.95	0.94	0.96	0.97	0.96	0.97	0.97	0.95	1
Variable-Level									
Parameterize Variable	0.82	0.81	0.82	0.85	0.8	0.92	0.89	0.83	0.99
Replace Variable With Attribute	0.82	0.83	0.81	0.83	0.79	0.9	0.86	0.79	0.99
Field-Level									
Move Attribute	0.77	0.71	0.91	0.66	0.6	0.99	0.71	0.63	1
Pull Up Attribute	0.84	0.86	0.8	0.84	0.77	0.96	0.73	0.65	1
Replace Attribute	0.79	0.79	0.79	0.83	0.85	0.8	0.82	0.77	0.9
Averages									
Class-Level	0.88	0.84	0.94	0.92	0.88	0.96	0.93	0.89	0.98
Method-Level	0.86	0.87	0.83	0.88	0.84	0.96	0.92	0.86	1.00
Variable-Level	0.82	0.82	0.82	0.84	0.80	0.91	0.88	0.81	0.99
Field-Level	0.80	0.79	0.83	0.78	0.74	0.92	0.75	0.68	0.97
Total	0.84	0.83	0.85	0.85	0.81	0.94	0.87	0.81	0.99

Table 5.8: Comparison of the performance of the RF models for all K's.

Observation 1: The RF models perform good for all different thresholds. The overall accuracy for the RF models is high and fairly similar with 84% for `K=15`, 85% for `K=25` and 87% for `K=50`. All models show the highest performance for Class-Level refactorings with a mean accuracy of 88% for `K=15`, 92% for `K=25` and 93% for `K=50`, and the lowest for Field-Level refactorings with a mean accuracy of 80% for `K=15`, 78% for `K=25` and 75% for `K=50`, see Table 5.8. The LR models perform significantly worse with an average accuracy of 75% for `K=15` and 83% for `K=15`.

Observation 2: LR models cannot adapt Variable-Level refactorings to `K=15` The Table 5.9 shows the performance of the LR models for `K=15` and `K=25`. The LR models show a significantly worse performance for `K=15` compared to `K=25` (-8% average accuracy) and the RF models (-9% average accuracy). This significant loss of performance can be attributed to the very low recall on Variable-Level refactorings (25%), which drastically reduces the overall recall and accuracy of the LR models. In regard to precision, the LR models achieve comparable results to the RF models.

	K=15			K=25		
Refactoring	Acc	Pr	Re	Acc	Pr	Re
Class-Level						
Extract Class	0.83	0.81	0.86	0.87	0.84	0.9
Move Class	0.82	0.76	0.92	0.86	0.81	0.94
Method-Level						
Inline Method	0.72	0.77	0.63	0.79	0.76	0.85
Pull Up Method	0.8	0.77	0.84	0.84	0.79	0.92
Push Down Method	0.81	0.82	0.78	0.86	0.83	0.92
Merge Parameter	0.83	0.98	0.69	0.94	0.96	0.92
Variable-Level						
Parameterize Variable	0.58	0.9	0.18	0.8	0.83	0.77
Replace Variable With Attribute	0.64	0.89	0.32	0.8	0.82	0.76
Field-Level						
Move Attribute	0.73	0.7	0.81	0.76	0.71	0.88
Pull Up Attribute	0.77	0.74	0.84	0.8	0.75	0.93
Averages						
Class-Level	0.83	0.79	0.89	0.87	0.83	0.92
Method-Level	0.79	0.84	0.74	0.86	0.84	0.90
Variable-Level	0.61	0.90	0.25	0.80	0.83	0.77
Field-Level	0.75	0.72	0.83	0.78	0.73	0.91
Total	0.75	0.81	0.69	0.83	0.81	0.88

Table 5.9: Comparison of the performance of the LR models for `K=15` and `K=25`.

Observation 3: Increasing K improves only recall `K=15` has the highest total precision with 87% compared to 84% for `K=25` and 83% for `K=50`, but `K=15` has also the lowest recall with 85% compared to 94% for `K=25` and 99% for `K=50`. A similar picture can be observed for the LR models with an average recall of 69% for `K=15` and 88% `K=25`, see Table 5.8. The precision is significantly higher for `K=15` for all levels (86% to 88%), except the Class-Level (84%). Especially for the Field-Level refactorings, the average precision is 7% higher compared to `K=25` and 12% compared to `K=50`. On the other hand, the overall recall is significantly lower for `K=15`, the average of the total recall is 8% lower compared to `K=25` and 12% compared to `K=50`, see Table 5.8.

Observation 4: Field-Level refactorings have the lowest prediction performance Overall, all models show the lowest accuracy for Field-Level refactorings, 4% below the total average accuracy for `K=15`, 7% lower for `K=25` and 12% lower for `K=50`. Models with `K=15` show the highest performance for the Field-Level refactorings with an average accuracy of 80% compared to 78% for `K=25` and 75% for `K=50`. For the RF models trained with `K=25` and `K=50` this loss of accuracy can be attributed to the low precision. For `K=15`

both precision and recall are balanced. The analysis of the confusion metrics (see online appendix) reveals that *Move Attribute* refactorings have only a TN rate of 33.17%, *Pull Up Attribute* refactorings of 71.66% and *Replace Attribute* of 75.8% for `K=25`. For `K=15` the TN rates for the *Move Attribute* refactoring are also low with 63.23%, for the *Pull Up Attribute* refactorings of 87.16% and for the *Replace Attribute* refactoring of 78.38%.

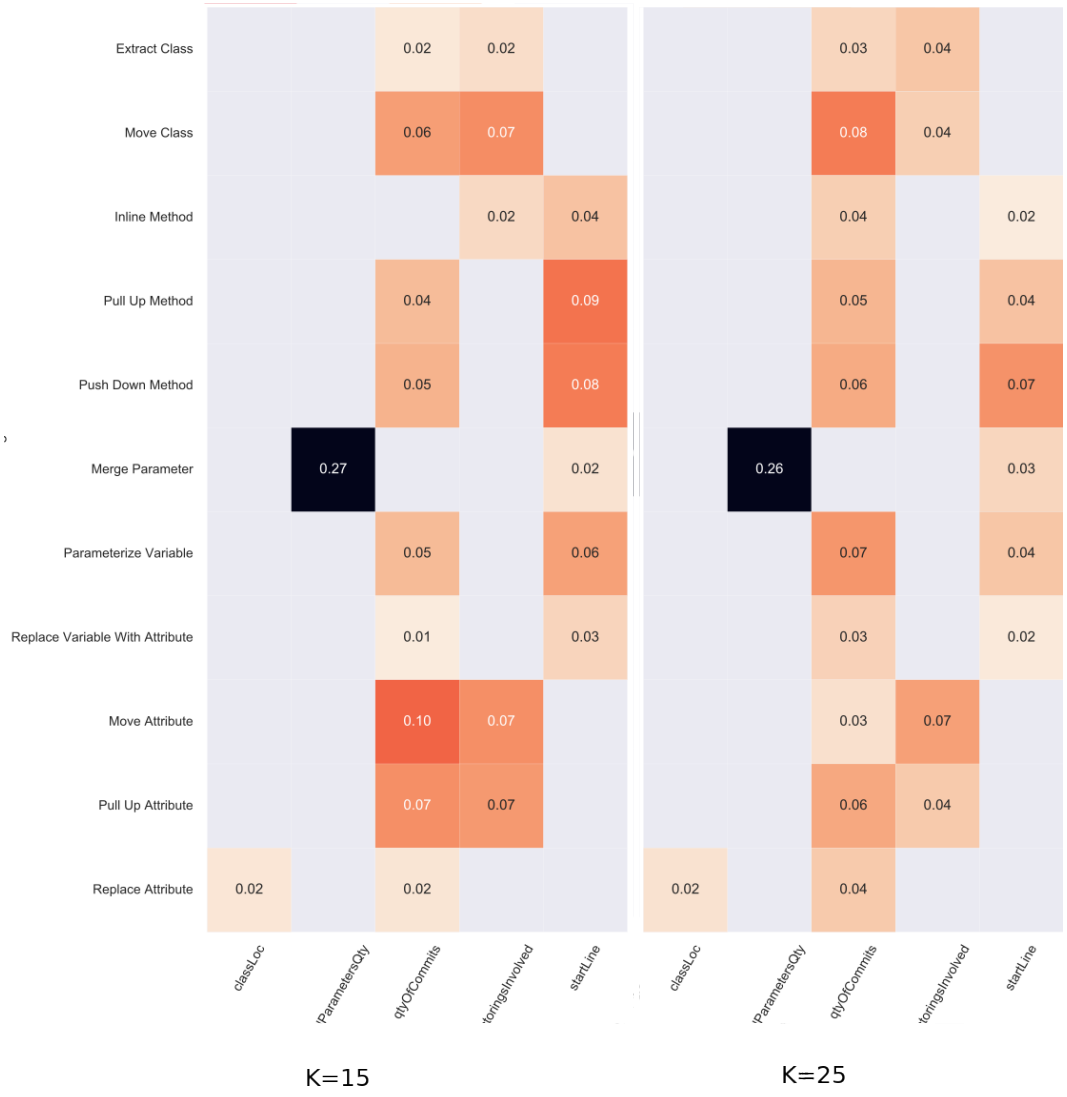


Figure 5.3: Top-2 features measured by permutation importance for `K=15` and `K=25` for the RF models.

Observation 6: Ownership metrics are not highly relevant for lower K's
In Fig. 5.3 and Fig. 5.4 the Top 2 features as measured by permutation impor-

tance for all 11 refactoring types for `K=15`, `K=25` and `K=50` are displayed. The other Top-N features for RF and LR can be found in the online appendix. The metric Author Ownership occurs only once and Quantity of Major Authors never occurs in the Top-10 of highest permutation importance for the RF models. Also, Quantity of Major Authors occurs only twice in the Top-5 of the most important model features. Author Ownership is frequently within the Top-5, 8 times for `K=15` and 12 times for `K=25`, but never in the Top-3. For comparison, Quantity of Major Authors appears in the Top-2 of the model importance measure for all RF models for `K=50` and Quantity of Major Authors is in the Top-5 for all models, see online appendix for more details. Both metrics Quantity of Authors and Quantity of Minor Authors do not appear in the Top-10 for `K=15` and `K=25`.

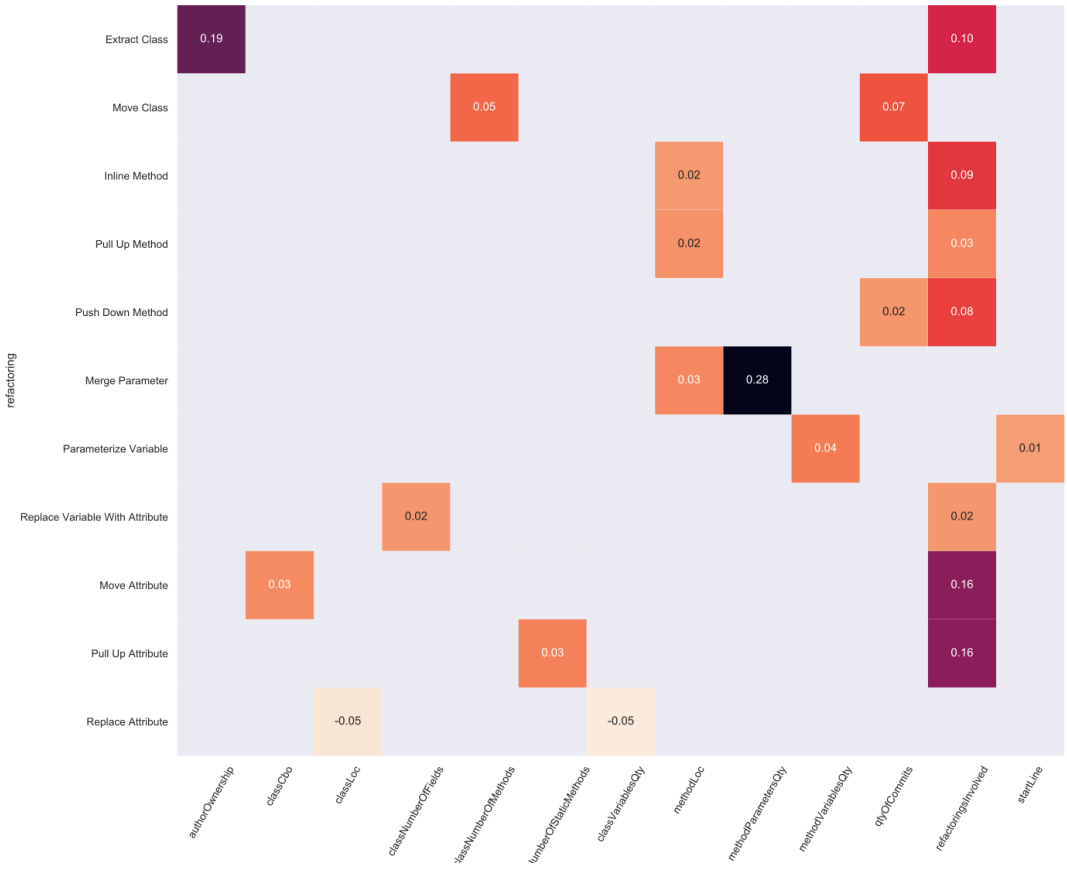


Figure 5.4: Top-2 features measured by permutation importance for `K=50` for the RF models.

Observation 7: Process Metrics are highly relevant for all K's Similar to the results of the reproduction experiment (see Section 5.3.2) the two process metrics Quantity of Commits and Refactorings Involved are the two most impor-

	Feature Importance	Permutation Importance
K=15		
Top-1:	qtyOfCommits (8), classNumberOfMethods (1), methodParametersQty (1), classLoc (1)	startLine (5), qtyOfCommits (3), refactoringsInvolved (2), methodParametersQty (1)
Top-5:	qtyOfCommits (11), authorOwnership (8), classLoc (7), refactoringsInvolved (6), startLine (6)	qtyOfCommits (10), refactoringsInvolved (9), startLine (6), classLoc (5), classNumberOfFields (3)
Top-10:	qtyOfCommits (11), authorOwnership (11), classLoc (10), refactoringsInvolved (10), classUniqueWordsQty (8)	qtyOfCommits (10), refactoringsInvolved (9), classNumberOfMethods (7), startLine (6), classLoc (6)
K=25		
Top-1:	qtyOfCommits (9), classNumberOfMethods (1), methodParametersQty (1)	qtyOfCommits (7), refactoringsInvolved (2), startLine (1), methodParametersQty (1)
Top-5:	qtyOfCommits (11), authorOwnership (11), refactoringsInvolved (7), startLine (6), classLoc (5)	qtyOfCommits (11), refactoringsInvolved (9), startLine (6), classLoc (4), classNumberOfMethods (3)
Top-10:	qtyOfCommits (11), authorOwnership (11), qtyOfAuthors (11), qtyMajorAuthors (10), classLoc (9)	qtyOfCommits (11), refactoringsInvolved (9), classStringLiteralsQty (9), startLine (6), classLoc (6)
K=50		
Top-1:	qtyOfCommits (7), authorOwnership (2), classNumberOfMethods (1), methodParametersQty (1)	refactoringsInvolved (6), authorOwnership (1), qtyOfCommits (1), methodParametersQty (1), methodVariablesQty (1)
Top-5:	authorOwnership (11), qtyOfCommits (11), qtyOfAuthors (10), qtyMajorAuthors (10), refactoringsInvolved (9)	refactoringsInvolved (10), qtyOfCommits (7), classNumberOfMethods (5), startLine (4), classNumberOfFields (3)
Top-10:	authorOwnership (11), qtyOfCommits (11), refactoringsInvolved (11), qtyOfAuthors (11), qtyMajorAuthors (10)	refactoringsInvolved (10), classNumberOfFields (8), qtyOfCommits (8), classCbo (8), classNumberOfMethods (5)

Table 5.10: Most relevant features as measured by permutation and model importance for all refactorings. Only the 5 most frequent features of the Top-N are displayed here.

tant metrics for the model performance. Quantity of Commits has the highest model importance for 9 out of the 11 models for `K=15` and `K=25`, each with an importance value ranging from 0.06 to 0.16. For the permutation importance it is not scoring as high but it is 9 times in the Top-2 for `K=15` and 10 times in the Top-2 for `K=25`, see Fig. 5.3. Despite, the top importance of the two process metrics Quantity of Commits and Refactorings Involved their importance is reduced for lower K. The total average permutation importance for `K=15` of Quantity of Commits is 4.0 and of Refactorings Involved is 3.18, for `K=25` it is 4.54 and 2.64, and for `K=50` it is 1.90 and 6.36.

Observation 8: `K=15` and `K=25` models for Method- and Variable-Level refactorings rely on Start Line The models for the Method- and Variable-Level refactorings trained with `K=15` strongly make use of the Start Line metric. Start Line is the most important feature for six of the seven Method- and Variable-Level refactorings measured by permutation importance for `K=15`, see Fig. 5.3. Also, Start Line is in in the Top-3 model feature importance for all of the seven refactorings, see Table 5.10. Models for Method- and Variable-Level refactorings trained with `K=50` do not consider this metric important and strongly rely on the process- and ownership metrics instead.

Observation 9: RF models for low K's make frequent use of SLoC for Field-Level refactorings For the RF models with `K=15` SLoC is in the Top-3 of permutation and model importance for the Field-Level refactorings. In contrast, the LR models for `K=15` do not utilize the SLoC metric at all, they rather use Author Ownership. For models trained with `K=25` SLoC is a Top-5 feature for the Field-Level refactorings, except for *Move Attribute*. `K=50` on the

other hand uses the metric CbO. In general, the Field-Level models for `K=15` and `K=25` both share many features of high relevance.

5.4.3 Discussion

The main results of the experiment with multiple K's and Field-Level refactorings are the following: (i) a higher K strongly increases the recall, (ii) RF models adapt well to different K's, but the LR models struggle, (iii) for lower K, ownership metrics are not of high relevance, (iv) a large variety of features is used by the different models and (v) Field-Level refactorings are more difficult to predict.

In the following subsections, the results of this experiment are discussed in more detail. Similar to the discussion for reproduction experiment, only RF is discussed in detail because the RF models perform significantly better than LR models.

Accuracy The selection of the K strongly affects the recall of the models, with increasing K the recall also increases to more than 98% for `K=50`. The precision of the models is only insignificantly altered, with `K=15` having a slightly higher (2%) precision when compared to the other thresholds. Already a low K of `K=25` is sufficient to achieve high recalls, as the average total recall is 94%. Previous experiments by Aniche et al. [4] showed comparable results regarding the recall, but not for precision: both precision (-5.0%) and recall (-9.3%) were significantly lower on average for `K=25`, but for four models the precision was higher. Furthermore, they found that with increasing K, the recall and overall accuracy of the models was improved, but precision was reduced.

The data analysis showed that the selection of K affects the feature distribution of the Stable-Instances for all Stable-Levels and that the Stable-Instances show overall very different characteristics compared to Refactoring-Instances except for the Field-Level Instances, see Chapter 4. The data analysis showed that Stable-Instances only cover a fraction of the collected classes (max. 7.33% for `K=15`) and this fraction rapidly decreases for higher K's (max. 0.56% for `K=50`). Additionally, with increasing K, Stable-Instances improve on multiple quality metrics and tend to be smaller and generally less complex. The results of the data analysis can explain the the very high accuracy reported by Aniche et al. [4] and in this experiment. The separation of the Refactoring- and Stable-Instances becomes increasingly easier with higher K, as only few Instances of the two classes have similar features and thus the number of False Negatives decreases.

The RF model and to some extend the LR are able to adapt their good performance to different K's, but show significantly lower recall. During training the model is optimized with the accuracy function, therefore the models are

likely to consider a sample as negative as it is the cheaper mistake it can make. This can also explain the slight increase in precision for `K=15`, as the models have to focus more on the characteristics of the refactorings instead of identifying Stable-Instances by their process- and ownership metrics. This can also be observed in the feature analysis. The ownership metrics Author Ownership and Quantity of Major Authors are of low importance for `K=15` and `K=25`, as they are less distinct for these K's, see Chapter 4. Furthermore, the models for the lower thresholds use a wide variety of the important features for the different refactoring types with the two process metrics Quantity of Commits and Refactorings Involved being of very high relevance. The models for `K=50` are centered around on the same features across the different refactoring types, they strongly depend on four process- and ownership metrics and a small variety of class metrics.

Field-Level refactorings The prediction of Field-Level refactorings was introduced in this experiment and they are thus especially interesting. The performance of the models for the Field-Level refactorings was considerably worse compared to all other Refactoring-Levels, the highest accuracy was achieved by the `K=15` models with 80%. Especially `K=25` and `K=50` models are suffering from low precision (average 74% and 68%) on predicting Field-Level refactorings. For `K=15` the models also achieve a considerably lower precision (-4%) with 79%. The lower precision across all K's can be explained by the higher similarity of Stable- and Refactoring-Instances for the Field-Level, see Chapter 4. The separation between these two classes is more difficult for the classifier and the number of False Positives increases. Interestingly, the models for `K=15` and `K=25` make both strong use of the SLoC metric. Nonetheless, also the Field-Level metrics rely strongly on the two process metrics Quantity of Commits and Refactorings Involved for which the Field-Level Stable-Instances show a different distribution when compared to the Refactoring-Instances.

5.5 Imbalanced Training

In this section, we answer ***RQ 7: What are the effects of imbalanced training on the prediction quality?***. We explain the setup of the experiment and make some important notes. Afterwards, we display and discuss the results, before we answer RQ 7.

5.5.1 Experimental Setup

This experiment explores the effects of imbalanced training data on the performance of the RF classifier. The training set contains **80% negative samples** (Stable-Instances) and **20% positive samples** (Refactoring-Instances) for each

refactoring type. We down-sampled the one of the two classes with random undersampling to fulfill the distribution criteria. The updated machine learning pipeline was utilized in this approach similar to the experiment with multiple K's, see Section 5.4. In contrast to the training, the **evaluation was done with the equally balanced test set**, see Section 5.2.1. To speed up the experiment, we disabled feature reduction and only run it with the RF, because it gives better results as LR. [4]

5.5.2 Results

Table 5.11 displays the performance of the seven models for which the RF was trained with the imbalanced training set. The confusion matrix for this experiment is shown in Fig. 5.5. In Fig. 5.6 the Top-3 features measured by permutation and model importance for the seven models created for this experiment are displayed. More detailed results can be found in the online appendix [20].

	Random Forest			
Refactoring	**Acc**	**F1**	**Pr**	**Re**
Class-Level				
Extract Class	0.80	0.77	0.90	0.67
Move Class	0.84	0.82	0.91	0.75
Method-Level				
Inline Method	0.73	0.66	0.91	0.51
Pull Up Method	0.80	0.77	0.90	0.67
Push Down Method	0.75	0.68	0.94	0.54
Merge Parameter	0.92	0.92	0.98	0.87
Variable-Level				
Parameterize Variable	0.73	0.66	0.91	0.52
Averages				
Class-Level	0.82	0.80	0.91	0.71
Method-Level	0.80	0.76	0.93	0.65
Variable-Level	0.73	0.66	0.91	0.52
Total	0.80	0.75	0.92	0.65

Table 5.11: The performance of the RF models for the imbalanced training set.

Observation 1: High precision, but low recall. The models in this experiment all achieve very high precision with a total average precision of 92% (see Table 5.11), which is 9% higher compared to the models trained with balanced data (see Section 5.4). The precision of the individual models ranges from 90% for the *Extract Class* refactoring to 98% for the *Merge Parameter* refactoring. On the other hand, the models have low recall with a total average of 65%, which is 18% lower compared to the ones generated with the balanced training

[4] Despite the efforts to speed up the experiment, the experiment did not finish in time. Therefore, models were created only for seven out of the eleven refactoring type.

data. Furthermore, the Class-Level refactorings have the highest average recall with 71%, but the difference is highest (-24%) when compared to the balanced models.

Observation 2: The *Merge Parameter* refactoring is exceptionally good. The *Merge Parameter* refactoring achieves the highest accuracy of all seven models with 92%, exceeding the total average accuracy by 12%. This can be explained by exceptionally high precision (98%) and recall (87%), see Table 5.11. The RF model trained on the balanced data set achieved a higher accuracy of 95% (+3%), but a lower precision of 94% (-4%) and a higher recall of 96% (+9%), see Section 5.4. The *Merge Parameter* refactoring is the only refactoring exceeding an accuracy of 90% in both experiments for `K=15`.

Observation 3: False Negatives classifications are very common. All models, except *Merge Parameter*, generated in this experiment suffer from high FN rates: the total average FN rate is 35.38%. The Class-Level refactorings have an average FN rate of 28.94%, the Method-Level refactorings of 35.39% and the only Variable-Level refactoring *Parameterize Variable* of 48.24%, see Fig. 5.5. The False Positive (FP) rates are low for all models, ranging from 4.42% for the Method-Level to 7.33% for the Class-Level refactorings. In the balanced training for `K=15`, the models had higher FP rates, ranging from an average of 11.89% for the Method-Level to 18.51% for the Class-Level. The FN rates on the other hand were significantly lower in the balanced training with an average of 6.38% for the Class-Level, 16.59% for the Method-Level refactorings and 4.36% for the *Merge Parameter* refactoring, see Section 5.4.

Observation 4: Ownership metrics are of low relevance for the models. None of the ownership metrics appears in the Top-5 features measured by permutation importance of all models and only Author Ownership appears three times in the Top-5 features measured with the model importance. In the Top-10 features measured with model importance Quantity of Authors appears twice, Quantity of Major Authors once and Author Ownership for all of them.

Observation 5: SLoC and Start Line are of great importance for the Method- and Variable-Level models. Fig. 5.6 shows that apart from the process metrics Quantity of Commits and Refactorings Involved, the class metric Start Line is highly relevant for Method- and Variable-Level models. Start Line is in the Top-3 features measured by permutation importance for all Method- and Variable-Level refactoring models and SLoC is in the Top-3 of three of the four, see Fig. 5.6. Start Line is also the most important metric for four of the seven models. For the Top-3 features by model importance, Start Line is also present

	Random Forest			
Refactoring	**TN**	**FP**	**FN**	**TP**
Class-Level				
Extract Class	92.49%	7.51%	32.88%	67.12%
Move Class	92.85%	7.15%	25.00%	75.00%
Method-Level				
Inline Method	95.09%	4.91%	48.84%	51.16%
Pull Up Method	92.65%	7.35%	33.01%	66.99%
Push Down Method	96.55%	3.45%	46.42%	53.58%
Merge Parameter	98.03%	1.97%	13.29%	86.71%
Variable-Level				
Parameterize Variable	94.82%	5.18%	48.24%	51.76%
Averages				
Class-Level	92.67%	7.33%	28.94%	71.06%
Method-Level	95.58%	4.42%	35.39%	64.61%
Variable-Level	94.82%	5.18%	48.24%	51.76%
Total	94.64%	5.36%	35.38%	64.62%

Figure 5.5: Confusion matrix for the seven RF models trained with the imbalanced training set.

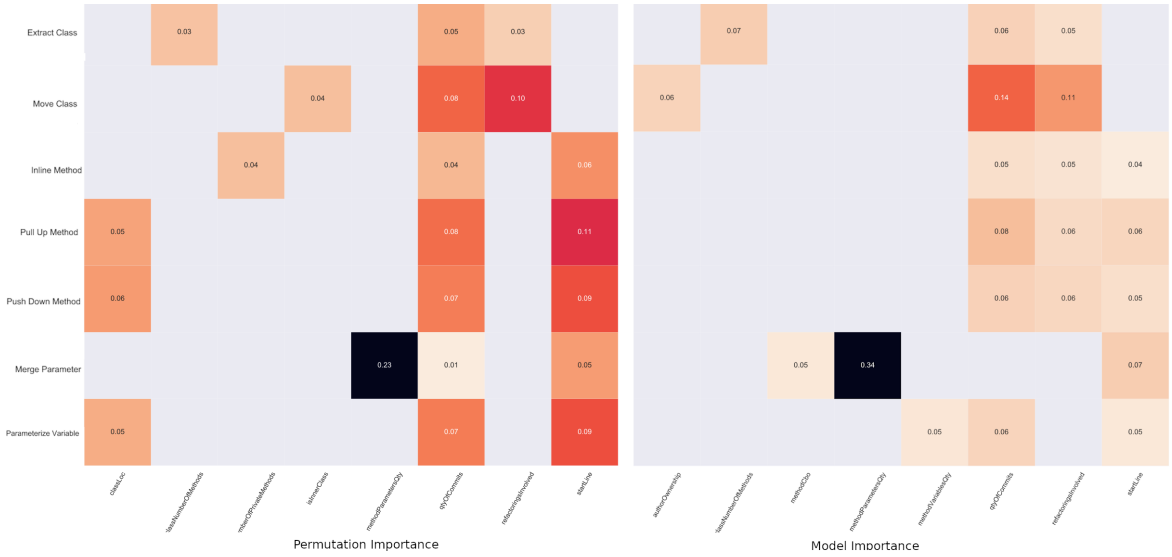
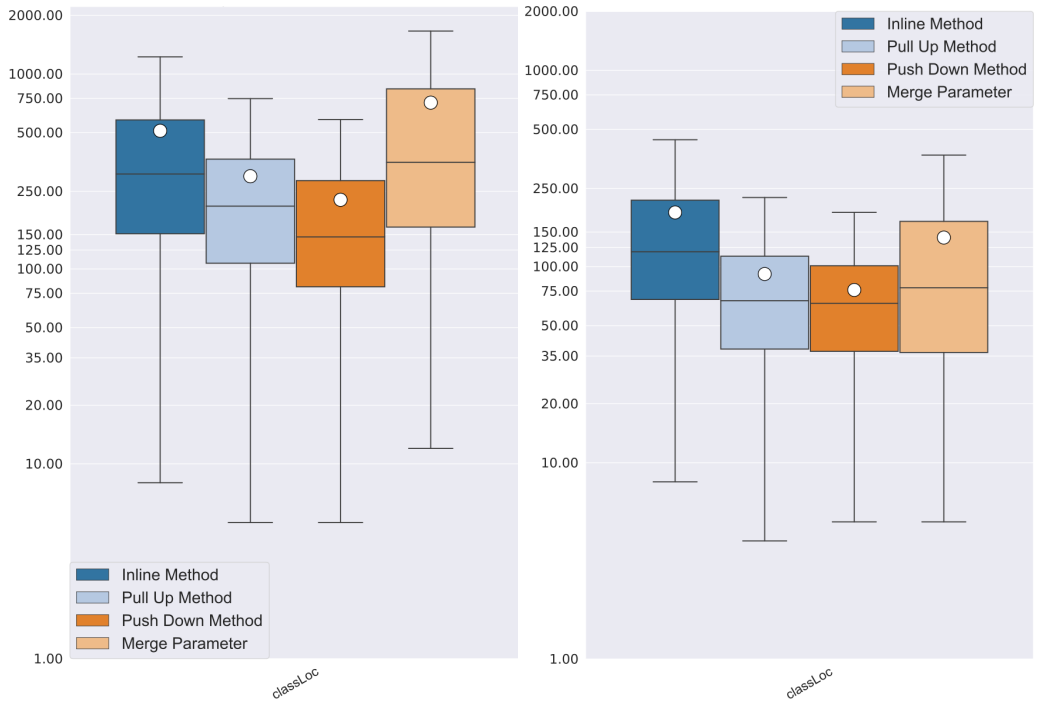


Figure 5.6: Top-3 features of the RF model measured with model and permutation importance.

for all the Method- and Variable-Level models, SLoC is not in the Top-3, but in the Top-5 for *Pull-Up Method* and *Push-Down Method*, see [20]. An analysis of the FN and TP samples shows that Method- and Variable-Level Refactoring-

Instances with high SLoC are likely to be misclassified as Stable-Instances, see Fig. 5.7.



(a) False Negatives (FN)

(b) True Positives (TP)

Figure 5.7: SLoC of False Negatives and True Positives of the Method-Level RF models for the imbalanced training experiment

Observation 6: Process metrics are the most important metrics. The two process metrics Quantity of Commits and Refactorings Involved are metrics with very high importance for the models. Quantity of Commits is a Top-3 permutation importance feature for all models and Refactorings Involved for the two Class-Level refactorings *Move Class* and *Extract Class*, see Fig. 5.6. For the model importance, Quantity of Commits is in the Top-3 for 6 models and Refactorings Involved is the Top-3 for 5 models. The other process metric Bug-fix Count appears in the Top-10 permutation importance features for 6 models and in the Top-10 model importance features 4 times, see [20].

5.5.3 Discussion

The main findings for the imbalanced training experiment are: (i) the models have high precision but very low recall, (ii) Merge Parameter performs well in

all configurations and (iii) process- and ownership metrics are becoming less important.

The overall performance of the RF models trained with a training set predominantly consisting of Stable-Instances (80%) does not reach the performance of the models trained in an equally balanced data set with the same K `K=15` . The total average accuracy is with 80%, is 6% lower than for the same refactorings of the balanced models. On the other hand, these models achieve a significantly higher (+7%) precision, but the recall is 21% lower. The main reasons for the low recall is the high FN rate of the models with a total average of 35.38%. A likely explanation for this behavior is the imbalance in the data set, only 20% of the samples in the training set were positive. Thus the model is incentivized to classify samples as negative, as it is more likely to be correct during training. The adaptability of the RF for imbalanced data is no surprise as it has been proven to perform well on imbalanced data [34].

Similar to the results of multiple K's experiment for the RF models for `K=15` (see Section 5.4), the ownership metrics are not important for the model performance. For the process metrics the results differ to some degree, the process metrics Quantity of Commits and Refactorings Involved are still highly important metrics, but are outweighed by Start Line for four of the five Method- and Variable-Level refactoring models. Also, the metric SLoC got more importance in the imbalanced training, than in any other model configuration. The Method- and Variable-Level models classified samples with high SLoC predominantly as Stable-Instances and thereby, made many FN classifications. For the Class-Level refactorings the models still consider Refactorings Involved highly relevant and achieved comparable results. For the identification of Class-Level Stable-instances, SLoC is not an appropriate feature, as it drastically lower for these Stable-Instances and more similar to the Refactoring-Instances. Within the Method- and Variable-Level Stable-Instances with classes containing many methods are over-represented (see Chapter 4), these classes are therefore also considerably longer and can thus easily be identified.

5.6 Conclusion

The evaluation of three conducted experiments combined with the results of the data analysis (see Chapter 4) give us insight into the created models. We can give profound answers to the Research Questions 4 to 7.

RQ 4: How does the initial approach perform with the new data set? The initial machine learning approach performs very well on the new data set and the changes it introduced. All performance measures were improved and the models still generalize well. Furthermore, all other main findings of the

original work could be reproduced. Furthermore, the performance improvements are likely explained by more pronounced unique characteristics of the Stable-Instances.

RQ 5: How does the selection of the K effect classifier performance? Increasing the K does increase the recall of the models for any refactoring types, but slightly decreases the precision. For `K=15`, the models have the highest total average precision with 83%. Furthermore, the models do not value ownership metrics highly for `K=15` and `K=25`. The most likely explanation for these findings is the increased difference for higher K's between Stable- and Refactoring-Instances, which is particularly pronounced for process- and ownership metrics.

RQ 6: How is the performance of the classifier for Field-Level refactorings? Both LR and RF achieve a decent performance in predicting Field-Level refactorings, LR has the highest accuracy for `K=15` with 80% and LR for `K=25` with 78%. The comparably lower accuracy for the Field-Level refactorings can be explained by the higher similarity of Refactoring- and Stable-Instances of the Field-Level compared to the other levels.

RQ 7: What are the effects of imbalanced training on the prediction quality? The RF models reach a total average accuracy of 80%. Having Stable-Instances as the majority class (80%) in the training data significantly improves the precision, but drastically reduces the recall of the refactoring prediction models. The models are more likely to classify a sample as Stable-Instance based on their ownership metrics. For the Method- and Variable-Level refactorings also the SLoC metric is of high relevance for the classification.

Chapter 6

Threads to Validity

In this chapter, we outline and discuss the limitations to the construct, internal and external validity of this thesis.

Construct Validity.

- **Stable-Instances**: The Stable-Instances in this work suffer from three main issues: (i) unique characteristics (see Chapter 4), (ii) selection of K and (iii) they are refactoring unspecific. The selected K has a significant impact on the performance of the models. For a class to be considered stable, all refactorings were considered. Thus, the question arises if the models identify refactorings or Stable-Instances. The variance in performance within the Refactoring-Levels suggests that the models can indeed detect different refactoring types and the great performance cannot be explained with the characteristics of the Stable-Instances alone. Nonetheless, repeating the experiments with a one-class classifier or a different heuristic for the generation of the Stable-Instances might answer this questions conclusively.
- **Bugs in metrics**: Aside from errors and exceptions during the data collection, other errors occurred. The two metrics Lines Added and Lines Deleted were faulty and thus removed. These metrics were identified as relevant for model performance in the prior work by Aniche et al. [4]. Missing these metrics might negatively affect the performance of the models. Furthermore, the implementation of the LCOM metric in CK is in its current state faulty and might yield incorrect results. Due to the large variety of collected metrics these issues should not be a larger thread to the validity of the results.
- **Balance**: The distribution of Refactoring-Instances and Stable-Instances across all classes was not considered in the data balancing for the classifier training. The two classes were balanced equally for both training

and evaluation. Therefore, when deploying these models in the wild, the perceived performance could vary strongly.

- **Few classifiers**: The evaluation of the models was focused on two classifiers (RF and LR), leaving out many other binary classifiers. Furthermore, this work did not evaluate the potential of one-class classifiers. Also, the potential of deep learning and natural language processing (NLP) were not considered. Therefore, this work cannot make any claims for the viability of these algorithms.

- **Data Analysis**: Only 18 of the 69 metrics were analyzed in full scope (see Chapter 4). These metrics were selected based on the results by Aniche et al. [4]. Therefore, the data analysis is biased towards machine learning purposes and other relevant findings were potentially missed. The data most likely contains plenty of other interesting insights into the motivation, the implementation and the general usage of refactorings in open-source projects, and high quality software examples.

- **RM**: The data collection process relies on RM to detect refactorings which it does very well (see Chapter 2). Nonetheless, RM misses various refactorings (mean recall 93%) and thus might introduce some minor bias into the data.

- ***Merge Parameter***: The models for *Merge Parameter* refactoring have shown exceptionally great performance in all experiments with an accuracy ranging from 92% in the imbalanced experiment (see Section 5.5) to 98% on the balanced training data with `K=50`, see Section 5.4.1. In all experiments the Method Parameters Quantity metric has been by far the most relevant metric for the models with a permutation and model importance > 0.21 in all configurations, see [20]. These results suggest that the Method Parameters Quantity metric can be used to easily distinguish between Stable-Instances of the Method-Level and *Merge Parameter* Refactoring-Instances and the models might not be suitable in predicting the refactoring.

Internal Validity.

- **High-Level Refactoring**: We considered the refactorings as atomical operations independent from other changes or operations on the source code. The data has shown that refactorings often do not occur in solitude and are a component of larger, more complex operations.

- **Motivation**: We did not consider the motivations of the developers to perform a specific refactoring. Furthermore, this research did not assess the

relevance of the performed refactorings, each refactoring was weighted equally without considering its impact on the software quality. As the training set for the models only consisted of refactorings developers considered worth implementing we expect to include the developers motivation in an abstract way. Thus, the aforementioned limitations do not pose a (major) threat to the validity of the results.

- **Data collection errors**: A small number of errors and exceptions occurred during the data collection, see Section 3.2.2, causing data collection not to finish for 1,36% of all projects. This small fraction of early fails should not affect the representation of the data significantly. It might slightly reduce the count of high K's Stable-Instances and the number of refactorings that are applied later in the development process, e.g. Variable-Level refactorings such as *Replace Variable*.

- **Code Smells**: Many approaches on refactoring recommendation use code smells to detect refactoring opportunities. The concept of code smells refers to a problematic fragment of the source code showing some clear characteristics, e.g. feature envy describes a case in which a class uses the methods of another class extensively. The data analysis on this work did not include code smells and thus, we cannot draw any conclusions about the relations of Refactoring- and Stable-Instances with code smells. Furthermore, we did not evaluate the results of the models with code smells, therefore, we do not know if the models identify code smells or are able to grasp their concept. Many approaches on detecting code smells rely on code metrics [60], thus with the computed data set it should be possible in future work to identify a variety of code smells in the data set and further investigate Refactoring- and Stable-Instances. Also, as our approach to solving the refactoring recommendation problem heavily relies on code metrics, it is quite likely that the models are able to identify code smells.

External Validity.

- **Open-source**: All considered projects were open-source projects, research has shown that industry code varies to some extent from open-source code. Nonetheless, many of the considered projects are widely used within the industry and or supported and maintained by leading players such as Facebook or Google.

- **Production-code**: This thesis only considered production code, thus generalizing the findings to test code is not possible. It is worth mentioning that the refactoring data set, a result of this work, allows to repeat the experiments and validate the findings for test-code.

- **Java**: Java was the only programming language considered in this research. Therefore, we cannot generalize the results to other object oriented programming (OOP) languages without reasonable doubt.

Chapter 7

Conclusions and Future Work

This chapter we draw a conclusion for the main findings of this thesis. Afterwards we draw the discuss ideas for future work.

7.1 Conclusions

This thesis presents, to our knowledge, the largest refactoring data set of open-source Java projects to date. In the data set, all refactorings are enriched with a multitude of code-, process- and ownership metrics describing the refactoring on multiple levels. Furthermore, this work collected non-refactored classes (Stable-Instances) that were introduced by Aniche et al. [4] as negative training samples for the refactoring prediction problem. The data set can be used by researchers to further explore the field of software refactoring and source code maintenance.

In an exploratory data analysis, we identified the Stable-Instances as classes with unique characteristics covering only a small fraction of all collected classes (7.33%). In general, Stable-Instances are frequently changed classes with various developers being involved and are less complex than refactored classes. Furthermore, the analysis showed that refactorings are predominantly applied to classes that are developed by a single author. Class- and Other-Level refactorings, like *Move Class* or *Move Package*, occur most frequently in early development stages of a class in contrast to Method- and Variable-Level refactorings, which occur uniformly over the lifetime of a class. Furthermore, Stable-Instances and Refactoring-Instances can be clearly separated by process- and ownership metrics.

Machine learning algorithms are effective in identifying Stable-Instances from metrics and show great potential in predicting refactoring opportunities. The best results were achieved with the Random Forest classifier, it proved to be adaptive for different configurations of the classification problem (varying Stable Commit Thresholds and imbalanced training data) and showed overall

outstanding results when evaluated on unseen projects, ranging from a total average accuracy of 80% up to 89%. The two process metrics Quantity of Commits and Refactorings Involved are consistently of very high importance for the model performance, also of high relevance are ownership metrics, and Class-Level metrics. Nonetheless, a large variety of metrics is considered by the models to classify the samples, these vary for the different refactoring types, K's and the data balance. The results of experiments show that using a multitude of metrics substantially improve the prediction quality and that using a small subset of metrics is insufficient to reliably predict refactorings from source code.

The answers to the Research Questions 1 to 3 can be found at the end of Chapter 4 and the answers for Research Questions 4 to 7 at the end of chapter Chapter 5. All results of this thesis can be found in the online appendix [21] and [20]. The online appendix includes (i) the complete refactoring data set, (ii) the lists of all mined projects, (iii) the charts and statistics generated for the data-analysis, (iv) the trained models for all three machine learning experiments and (iv) the evaluation results of the machine learning experiments. The source code for the data collection tool can be found online and can be used by other researchers. [1] Also, the source code for the machine learning experiment is publicly available.[2]

7.2 Future work

In this section, we discuss multiple promising avenues for future work based on the findings and observations from this thesis.

Specific Stable-Instances For a class to be stable, all refactorings were considered. This created most likely a broad anti class for refactorings which is not specific for a single refactoring. Selecting a single refactoring or small set of refactorings to define the stability of a class could resolve various of the issues of the current Stable-Instances. These specific Stable-Instances should be effective negative samples for model training and thereby, could be a crucial step towards a production ready model.

Analysis of test code Various research has suggested different refactoring practices for test code [74] and also shown that the refactoring practices for test code are different from production code [7, 8]. Test code was completely ignored during the visual analysis of the refactoring data and the evaluation of various machine learning algorithms. This is a missed opportunity as we

[1] https://github.com/refactoring-ai/Data-Collection
[2] https://github.com/refactoring-ai/Machine-Learning

collected more than 4 million Refactoring-Instances for test code which might yield interesting insights into refactoring practices for test code.

Data Analysis The created refactoring data set was only analysed with certain goals, a multitude of future investigations is possible including:

- **Refactoring Documentation**: How do developers document refactoring activities in commit messages?
- **Code Smells**: Comparing refactoring activities with the existence of code smells. Code smells can be detected with code metrics, therefore it should be possible to detect code smells in the Refactoring-Instances.

One-class classifier A one-class classifier tries to identify samples of a single class from all samples not separating the sample space into two or more distinct classes. The crucial difference in training is the use of samples from only a single class and thus, one-class classifiers could be deployed to identify refactorings in source code based on metrics without considering Stable-Instances. Code metrics were already used to identify code smells in previous studies [44, 60], therefore the collected data in combination with a one-class classifier could be utilized to create models detecting refactorings.

Glossary

Author Ownership The proportion of commits achieved by the most active developer of a class file, e.g. developer A has 4 commits and developer B has 2 so the Author Ownership of the class is: 4 / (4 + 2) = 2/3. 17, 46, 47, 50–53, 65–68, 70, 75, 76, 78, 80

Bugfix Count Number of commits with bug fixes for a class file. 17, 47, 49, 50, 52, 67, 82

CbO Coupling between Object: A metric measuring the coupling of a class. Based on the number of dependencies of a class, it computes the relations between objects and thereby the metric. 35–39, 44, 45, 67, 77, 94

K Stable Commit Threshold: the minimum number of commits changing a class without refactoring it until it is considered stable. We collected these threshold [15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80, 90, 100]. i, vii–ix, 6, 20, 26, 27, 29, 40, 42–45, 47, 51–53, 60, 71–79, 83–85, 87, 90

LCC Loose Class Cohesion: Similar to Tight Class Cohesion but it further includes the number of indirect connections between visible classes for the cohesion calculation. Thus, the constraint LCC greater than TCC holds always. The metric ranges from 0 to 1024.. viii, 15, 35, 36, 45, 47, 67

LCOM Lack of Cohesion of Methods: Measures the correlation of local variables and methods in a class. vii, viii, 15, 35–38, 46, 85

Level The Level of a Refactoring-Instance or Stable-Instance describes its entity, e.g. *Extract Class* is a Class-Level refactoring as it is applied to an entire class in contrast to the *Replace Variable* refactoring which only affects a specific variable and therefore, is a Variable-Level refactoring. For a Stable-Instance it defines the level on which the metrics were collected.

Also, it defines for which refactoring types the Stable-Instance can be used as a negative sample in classifier training, e.g. a Method-Level Instance can be used for the training of *Extract Method* refactoring models. As an example, for the Method-Level Stable-Instances, class metrics, process- and ownership metrics and method metrics are collected. Thus, if a class has five methods, five Method-Level Stable-Instances are generated all sharing the same class, and process- and ownership metrics. 93, 95

Lines Added The number of lines added in a commit.. 23, 85

Lines Deleted The number of lines deleted in a commit.. 23, 85

LR Logistic Regression classifier. viii, ix, 6, 59, 61–68, 71–73, 75–77, 79, 84, 86

Maximum Number of Nested Blocks in a Method The maximum number of nested blocks (statements) for this method.. 67

Method CbO Similar to CbO but for methods. 67

Method SLoC This metric counts the number of lines of source code for a method skipping comments and empty lines. 67

Method RFC Total number of unique method invocations for a single method within a class. 67

Method Parameters Quantity The number of parameters for a method.. 86

Number of Default Methods Number of default methods in a class. 67

Number of Fields The total count of fields (also referred to as attributes) in a java class. 67

Number of Methods The total count of methods in a class including all access modifiers. 15, 35, 43, 67, 68

Number of Public Fields The count of public fields in a class. 15, 35, 44, 45

Number of Unique Words Total number of unique words in the source code of a class. 35, 37, 39

Number of Unique Words in a Method The total number of unique words, e.g. "John Doe is married to Mary Doe" has a unique word count of 6. 67

Number of Variables The total count of variables for all methods in a java class. 35, 39

Quantity of Authors Total count of unique authors making revisions to a class file. 17, 47–49, 51, 75, 80

Quantity of Commits Number of commits changing a class file. 17, 49, 50, 52, 65–68, 70, 75, 76, 78, 80, 82, 83, 90

Quantity of Major Authors Number of authors contributing more than 5% of the commits changing a class file. 17, 47, 67, 68, 70, 75, 78, 80

Quantity of Minor Authors Number of authors contributing less than 5% of the commits changing a class file. 17, 47, 67, 70, 75

Quantity of String Literals The number of string literals, e.g. for "John Doe" it is 2, in a class. 35, 39, 45

Refactoring-Instance A single sample of a refactoring in the database. It contains a reference to a project, a Level, class file and file path, commit meta data, the refactoring type and a summary of the refactoring, and all metrics for its Level. 93

Refactorings Involved Number of refactorings affecting a class file. 17, 49, 51–53, 66–68, 70, 75, 76, 78, 80, 82, 83, 90

RF Random Forest classifier. i, viii, ix, 6, 12, 61–65, 67, 68, 71–84, 86

RFC Response for a Class: Total number of unique method invocations within a class. viii, 15, 35, 37, 39, 45, 47, 67, 94

RM RefactoringMiner: A refactoring detection tool that can detect more than 55 individual refactoring types from source code [73]. 5, 10, 15, 86

SLoC (Source) Lines of Code: This metric counts the number of lines of source code in a class file skipping comments and empty lines. vii, viii, 35, 37, 39, 45, 67, 76, 78, 80–84, 94

Stable-Instance A single stable sample in the database. It contains a reference to a project, a Level, class file and file path, commit meta data, and all metrics for its Level. 6, 93, 94

Start Line The start line of a method in a class.. 76, 80, 83

TCC Tight Class Cohesion: Measures the cohesion of a class with a value range from 0 to 1. TCC measures the cohesion of a class via direct connections between visible methods. The metric ranges from 0 to 1024.. 15, 35, 36, 45

Variable Appearances Total count for a specific variable appearing in the local method. 67

WMC Weight Method Class: Also referred to as McCabe's complexity, the metric comprises the number of branch instructions in a class and thereby measures the complexity of all methods. 35, 37, 39, 44, 45

Bibliography

[1] Jehad Al Dallal. Predicting move method refactoring opportunities in object-oriented code. *Information and Software Technology*, 92:105–120, 2017.

[2] Mohammad Alshayeb. Empirical investigation of refactoring effect on software quality. *Information and software technology*, 51(9):1319–1326, 2009.

[3] Boukhdhir Amal, Marouane Kessentini, Slim Bechikh, Josselin Dea, and Lamjed Ben Said. On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring. In *International Symposium on Search Based Software Engineering*, pages 31–45. Springer, 2014.

[4] Maurício Aniche, Erick Maziero, Rafael Durelli, and Vinicius Durelli. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *arXiv preprint arXiv:2001.03338*, 2020.

[5] Maurício Aniche. *Java code metrics calculator (CK)*, 2015. Available in https://github.com/mauricioaniche/ck/.

[6] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138, 2019.

[7] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 56–65. IEEE, 2012.

[8] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[9] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011.

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

[11] Jie Cai, Jiawei Luo, Shulin Wang, and Sheng Yang. Feature selection in machine learning: A new perspective. *Neurocomputing*, 300:70–79, 2018.

[12] Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014.

[13] Lakshitha De Silva and Dharini Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1): 132–151, 2012.

[14] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. *ACM SIGPLAN Notices*, 35(10):166–177, 2000.

[15] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[16] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of feature envy bad smells. In *2007 ieee international conference on software maintenance*, pages 519–520. IEEE, 2007.

[17] 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*, pages 396–399. IEEE, 2013.

[18] 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.

[19] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[20] Jan Gerling and Mauricio Aniche. Appendix: Data Analysis and Machine Learning Experiments, November 2020. URL https://doi.org/10.5281/zenodo.4267824.

[21] Jan Gerling and Mauricio Aniche. Appendix: Refactoring data set, November 2020. URL https://doi.org/10.5281/zenodo.4267711.

[22] Michael W Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[23] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL http://dl.acm.org/citation.cfm?id=2487085.2487132.

[24] William G Griswold. *Program restructuring as an aid to software maintenance.* PhD thesis, University of Wisconsin - Madison, 1992.

[25] Richard A Groeneveld and Glen Meeden. Measuring skewness and kurtosis. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 33 (4):391–399, 1984.

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

[27] Mark Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE'07)*, pages 342–357. IEEE, 2007.

[28] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[29] Mark Harman and Laurence Tratt. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1106–1113, 2007.

[30] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):1–61, 2012.

[31] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[32] Piotr Juszczak, D Tax, and Robert PW Duin. Feature scaling in support vector data description. In *Proc. asci*, pages 95–102. Citeseer, 2002.

[33] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 576–585. IEEE, 2002.

[34] Taghi M Khoshgoftaar, Moiz Golawala, and Jason Van Hulse. An empirical study of learning from imbalanced data using random forest. In *19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007)*, volume 2, pages 310–317. IEEE, 2007.

[35] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 371–372, 2010.

[36] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[37] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014.

[38] Giri Panamoottil Krishnan and Nikolaos Tsantalis. Unification and refactoring of clones. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 104–113. IEEE, 2014.

[39] L. Kumar and A. Sureka. Application of lssvm and smote on seven open source projects for predicting refactoring at class level. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 90–99, 2017.

[40] Robert Leitch and Eleni Stroulia. Assessing the maintainability benefits of design restructuring using dependency analysis. In *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717)*, pages 309–322. IEEE, 2004.

[41] Hubert W Lilliefors. On the kolmogorov-smirnov test for normality with mean and variance unknown. *Journal of the American statistical Association*, 62(318):399–402, 1967.

[42] Hui Liu, Zhifeng Xu, and Yanzhen Zou. Deep learning based feature envy detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 385–396, 2018.

[43] Thainá Mariani and Silvia Regina Vergilio. A systematic review on search-based refactoring. *Information and Software Technology*, 83:14–34, 2017.

[44] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359. IEEE, 2004.

[45] Scott Menard. *Applied logistic regression analysis*, volume 106. Sage, 2002.

[46] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[47] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 331–336, 2014.

[48] Naouel Moha, Yann-Gael Gueheneuc, 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.

[49] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 181–190, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1368088.1368114. URL https://doi-org.tudelft.idm.oclc.org/10.1145/1368088.1368114.

[50] Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017.

[51] Mark O'Keeffe and Mel Ó Cinnéide. Search-based refactoring: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(5):345–364, 2008.

[52] William F Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign Champaign, IL, USA, 1992.

[53] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. An exploratory study on the relationship between changes and refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 176–185. IEEE, 2017.

[54] 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.

[55] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[56] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8):1397–1418, 2013.

[57] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 432–441, 2013.

[58] Santonu Sarkar, Shubha Ramachandran, G Sathish Kumar, Madhu K Iyengar, K Rangarajan, and Saravanan Sivagnanam. Modularization of a large-scale business application: A case study. *IEEE software*, 26(2):28–35, 2009.

[59] T. Sharma, G. Suryanarayana, and G. Samarthyam. Challenges to and solutions for refactoring adoption: An industrial perspective. *IEEE Software*, 32(6):44–51, 2015.

[60] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, 2018.

[61] Danilo Silva and Marco Tulio Valente. Refdiff: detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 269–279. IEEE, 2017.

[62] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 858–870, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950305. URL https://doi-org.tudelft.idm.oclc.org/10.1145/2950290.2950305.

[63] Chris Simons, Jeremy Singer, and David R White. Search-based refactoring: Metrics are not enough. In *International Symposium on Search Based Software Engineering*, pages 47–61. Springer, 2015.

[64] Michael A Stephens. Edf statistics for goodness of fit and some comparisons. *Journal of the American statistical Association*, 69(347):730–737, 1974.

[65] Carolin Strobl and Achim Zeileis. Danger: High power!? exploring the statistical properties of a test for random forest variable importance, 2008. URL http://nbn-resolving.de/urn/resolver.pl?urn=nbn:de:bvb:19-epub-2111-8.

[66] Liang Tan and Christoph Bockisch. A survey of refactoring detection tools. In *Software Engineering (Workshops)*, pages 100–105, 2019.

[67] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S Bigonha. Recommending refactorings to reverse software architecture erosion. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 335–340. IEEE, 2012.

[68] T. Tourwe and T. Mens. Identifying refactoring opportunities using logic meta programming. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pages 91–100, 2003.

[69] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

[70] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 329–331. IEEE, 2008.

[71] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. A multidimensional empirical study on refactoring activity. In *CASCON*, pages 132–146, 2013.

[72] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180206. URL http://doi.acm.org/10.1145/3180155.3180206.

[73] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 2020. doi: 10.1109/TSE.2020.3007722.

[74] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP)*, pages 92–95, 2001.

[75] Filip Van Rysselberghe and Serge Demeyer. Reconstruction of successful software evolution using clone detection. In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 126–130. IEEE, 2003.

[76] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, 2005.

[77] Zhenchang Xing and Eleni Stroulia. Refactoring detection based on umldiff change-facts queries. In *2006 13th Working Conference on Reverse Engineering*, pages 263–274. IEEE, 2006.

[78] Qiang Yang and Xindong Wu. 10 challenging problems in data mining research. *International Journal of Information Technology & Decision Making*, 5(04):597–604, 2006.

Appendix A

Appendix

Refactoring	All	Production-Code			Test-Code
		All	Training-Set	Test-Set	
Class-Level					
Extract Class	181821	160501	154786	4832	21320
Extract Interface	90520	86400	82798	3106	4120
Extract Subclass	24851	23471	22519	766	1380
Extract Superclass	293950	239044	230452	7626	54906
Move And Rename Class	247993	208684	200708	7249	39309
Move Class	3072283	2551857	2467925	76803	520426
Rename Class	646341	509885	490167	17702	136456
Method-Level					
Change Parameter Type	3264206	3081053	2964120	102386	183153
Change Return Type	2595738	2422332	2332327	79530	173406
Extract And Move Method	435262	360766	348505	10686	74496
Extract Method	1354614	1186251	1140640	38653	168363
Inline Method	237480	216787	208620	6722	20693
Merge Parameter	32278	31197	30102	963	1081
Move And Inline Method	103732	97324	94052	2805	6408
Move And Rename Method	171946	143908	138888	4260	28038
Move Method	910979	782996	754434	24992	127983
Pull Up Method	552954	488880	468945	17466	64074
Push Down Method	195903	185540	177561	6431	10363
Rename Method	2220587	1794512	1728182	58146	426075
Split Parameter	7538	7305	7007	265	233
Variable-Level					
Change Variable Type	3216005	2353287	2267038	76220	862718
Extract Variable	763190	674940	649238	22181	88250
Inline Variable	178533	159118	153498	4778	19415
Merge Variable	15660	14119	13588	452	1541
Parameterize Variable	96467	86164	82877	2780	10303
Rename Parameter	1850048	1762542	1697677	55793	87506
Rename Variable	1557697	1286164	1237836	41651	271533
Replace Variable With Attribute	129576	107116	103592	3132	22460
Split Variable	2988	2354	2270	75	634
Field-Level					
Change Attribute Type	1955035	1740420	1677111	54985	214615
Extract Attribute	143353	124440	120436	3477	18913
Move And Rename Attribute	6175	5291	5127	154	884
Move Attribute	756245	697333	672853	20800	58912
Pull Up Attribute	298290	255414	246335	7890	42876
Push Down Attribute	89295	85420	81803	2857	3875
Rename Attribute	1172867	1069841	1034477	30555	103026
Replace Attribute	4409	3612	3491	111	797
Other					
Change Package	1402976	1156192	1123266	30745	246784
Move Source Folder	3399720	2766923	2663027	93874	632797

Table A.1: Total count of refactoring instances in the database for production and test code, and the training and test sets

1	2	3	4	5
Move Class (1)	Change Variable Type (3)	Change Attribute Type (4)	Change Parameter Type (2)	Extract Interface (1)
Rename Class (1)	Extract Variable (3)	Move And Rename Attribute (4)	Change Return Type (2)	Extract Subclass (1)
Move and Rename Class (1)	Inline Variable (3)	Move Attribute (4)	Extract Method (2)	Extract Superclass (1)
Move Source Folder (5)	Merge Variable (3)	Pull Up Attribute (4)	Inline Method (2)	
Change Package (5)	Parameterize Variable (3)	Push Down Attribute (4)	Merge Parameter (2)	
	Rename Variable (3)	Replace Attribute (4)	Move And Rename Method (2)	
(Extract Interface (1))	Replace Variable With Attribute (3)	Push Down Method (2)	Move Method (2)	
(Extract Subclass] (1))	Split Variable (3)	Split Parameter (2)	Pull Up Method (2)	
(Extract Superclass (1))	Extract and Move Method (2)	Extract Class (1)	Rename Method (2)	

Table A.2: Refactoring types clustered by their class metrics and attributes, their level is in brackets, without Rename Parameter refactoring