Data-Driven Extract Method Recommendations: An Initial Study at ING

Master's Thesis

David van der Leij

Data-Driven Extract Method Recommendations: An Initial Study at ING

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

David van der Leij
born in Assen, the Netherlands

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

IT Innovation
Omnichannel
ING Bank N.V.
Bijlmerdreef 106
Amsterdam, The Netherlands
www.ing.nl

This thesis project is executed under the supervision of ING as part of the Industrial-grade Verification and Validation of Evolving Systems (IVVES) project, which is a part of EU ITEA 3 [1].

[1] https://itea3.org/project/ivves.html

Data-Driven Extract Method Recommendations: An Initial Study at ING

Abstract

Refactoring is the process of improving the structure of code without changing its functionality. The process is beneficial for software quality but challenges remain for identifying refactoring opportunities. This work employs machine learning to predict the application of the refactoring type Extract Method in an industry setting with the use of code quality metrics.

We detect 919 examples in industry code of Extract Method and 986 examples where Extract Method was not applied and compare this to open-source code. We find that feature distributions between industry and open-source code differ, especially in class-level metrics.

We train models to predict Extract Method in industry code and find that Random Forests perform best. We find that class-level metrics are most important for the performance of these models. We then investigate whether models trained on an open-source set generalize to an industry setting. We find that, although less performant than a custom fit model, a Logistic Regression type model performs admirably. Afterward, we examine whether these models perform on unseen industry projects by validating on projects excluded from the training set. We find that average performance is decent but lower than when using the whole industry dataset or an open-source dataset for training.

Lastly, we conduct a blind user study in which we ask experts to judge predictions made by our best model. We find that experts generally agree with the model's predictions. In the case that experts agree with the model's prediction to apply Extract Method, they do so because of high code complexity. When they agree with the model's prediction not to refactor they most frequently give the reason that the respective methods are already sufficiently understandable.

Author: David van der Leij
Student id: 4701151
Email: davidvanderleij@gmail.com

Thesis Committee:

Chair:	Dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. Aniche, Faculty EEMCS, TU Delft
Company supervisor:	Dr. Y. Luo., ING, Omnichannel, ING / TU Eindhoven

Preface

I want to thank my supervisor Maurício Aniche for the great supervision he has given me during this unusual time. The ease and style of comunications have helped me make this project possible during a time where meeting in-person was impossible.

I would also like to thank my colleagues at ING for their feedback and suggestions. Their insights and willingness to spend time to improve my work helped me to stay motivated and improved the end result. In particular, I want to thank Luna Yaping and Robbert van Dalen for their advice and help. In addition, I would like to thank Jasper Binda and Pieter Vallen for their insights and help in performing the user study.

Finally, I would like to thank prof. dr. Eelco Visser for his interest in my work and for joining the thesis committee.

David van der Leij
Amsterdam, the Netherlands
April 23, 2021

Contents

List of Figures

Chapter 1

Introduction

Software projects are ever-evolving due to the advent of new functionality, bug fixes, and performance optimizations. With this evolution, however, comes the problem of degradation of code quality. As a project evolves, the scope and complexity increase and the original design ideologies fade. Refactoring is defined by Fowler as the process of changing a software system in such a way that does not alter the external behavior of the code yet improves its internal structure [14]. This process has been shown to improve code quality in terms of code quality metrics and other aspects [23, 21, 15] In addition to this, developers perceive it to have a wide array of benefits [21].

Nonetheless, there are challenges for developers applying refactoring operations. For example, developers worry about behavior preservation when applying refactoring operations [23, 6, 21, 30]. Next to this, refactoring might incur certain costs for developers [21]. Also, the benefits of refactoring might not immediately be apparent [3]. Finally, developers might misclassify refactoring opportunities, or miss them altogether [30]. A wide variety of approaches have been applied to circumvent the above problems. One of which is facilitating the detection of suitable refactoring opportunities on which this thesis will focus.

Detecting refactoring opportunities is an active field of research. An example of a well-known tool for recommending refactoring opportunities is JDeodorant [10]. This Eclipse plugin implements results in the field to recommend several refactoring operations within the IDE. Some examples of operations supported are Extract Method [33, 32] and Extract Class [11, 9, 10].

Another approach using machine learning was proposed by Aniche et al. in 2020. Their paper applies a total of six different machine learning algorithms to the problem [4]. They model the problem as binary classification and use code metrics as features to predict whether code should be refactored or not. Aniche et al. use RefactoringMiner [34], a tool that detects whether a refactor occurred. They define code as non-refactored when it was changed but not refactored for 50 consecutive changes. When using a

Random Forest type model with open-source code as input, they achieve a precision rate of over 90%. The authors also show that the resulting models generalize well, as precision levels are high for unseen code.

Aniche et al. pose several research questions to be further investigated. Many of these were examined by Gerling, who expanded on this approach [16]. He did so by ensuring data quality, increasing the scale of the dataset, and investigating the reason behind the high importance of process-and ownership metrics. He also achieved an understanding of refactoring operations and source code metrics, and further explored the machine learning algorithms and their potential for this problem.

Some issues in the original paper [4] are still open, however. In Section 6.3, the authors note that the current models are trained on open-source projects only and, to show the effectiveness of the approach on industry projects as well, a replication study on a large-scale industry dataset is necessary. In the same section, the authors mention that they did not include any large-scale industrial systems in their research and can therefore not make any strong assumptions about the generalizability of their models in such industry settings. They suggest further case studies are necessary to investigate this. Finally, in Section 6.2, the authors mention that they did not take into account any reasons a developer might have for applying any particular refactoring operation.

To address all of these issues, we define the following three focuses for this thesis.

1. **Data analysis:** Collecting and analyzing a refactoring dataset based on industry code, and subsequently comparing it to the open-source counterpart.

2. **Machine learning:** Analyzing the results of applying supervised machine learning on the industry dataset, where models are trained on both open-source and industry datasets.

3. **User study:** Conducting a user study that measures industry experts' opinions on predictions generated by the best-performing models.

In this thesis, we dedicate a chapter to each of these focuses. This work will focus on the refactoring of type "Extract Method" only. The reasons for this choice can be found in Section 3.1. We will now go into more details on each focus.

Data Analysis is explored in Chapter 3 and serves to understand the behavior of our models better by investigating their respective feature distributions. Doing this gives us more insight into the following aspects:

1. **Performance measures:** Why does a certain model perform better than another? Can we explain this with the feature distribution?

2. **Feature importance:** Can we recognize any patterns confirming or contradicting the reported feature importances in Chapter 4?

Section 3.2 investigates differences between open-source and industry feature distributions. This is done to better understand the reasons behind the generalizability of models on industry code when they were trained on open-source code. Section 3.3 investigates whether there are differences between individual industry projects. This has a similar goal as before, but this time it gives us information on why models might generalize within industry code.

Machine learning is explored in Chapter 4 and will examine how well classification algorithms perform on the industry dataset as described in Chapter 3. First, we investigate whether the approach described by Aniche et al. performs well on industry code. We do this in Section 4.2. Next, we investigate whether a model trained on open-source code generalizes to an industry setting. We do this in Section 4.3. To better understand the way these models operate and how they differ from each other we also investigate which features are important for their performance. Finally, we are interested in how these models perform on unseen projects in industrial systems. We achieve this in Section 4.4.

The results in Chapter 4 only give insight into the theoretical performance of these models. In Chapter 5 we analyze whether our models generate predictions that are deemed useful to industry experts. We investigate their qualitative and quantitative opinions on suggestions generated by the best-performing model.

We summarize these goals by the main research question:

MRQ: What is the effectiveness of ML-based Extract Method recommendation models in industry code?

We summarize the contributions this thesis provides as follows:

- An empirical study of code metrics in code that underwent an Extract Method refactoring in an industry setting.
- An analysis in the performance of machine learning models for recommending Extract Method in industry code.
- Industry expert quantitative and qualitative assessment on Extract Method predictions generated by machine learning models.
- A pipeline that allows for the automatic recommendation of Extract Method predictions generated by machine learning models in production code.

Chapter 2

Related work

Refactoring was popularized by Fowler when he wrote his book on the subject in 1999 [14]. Since the book's release, many literature studies on the subject have been performed.

One of these was by Mens and Tourwe and their study was primarily concerned with the following criteria [26]: They examine the different types of refactoring activities and the kind of software that is being refactored. They also investigate which aspects are important in systems that recommend refactoring operations and what their effects are on the software development process. They find that identifying code smells is one of the most widespread techniques to detect code that requires refactoring. They also observe that the rate of success in identifying refactoring opportunities can highly depend on the software's domain. Mens and Tourwe's survey explores papers in a more exploratory manner and does not systematically compare them.

Baqais and Alshayeb, introduce a systematic literature survey on refactoring [5]. They analyze 148 papers and answer five qualitative questions regarding refactoring for these papers. Baqais and Alshayeb find that a majority of papers describe refactoring techniques (71%), rather than providing tools (29%). Out of these tools, they find the majority of them use a search-based genetic algorithm variant (69%).

A tertiary literature study by Lacerda et al. identifies 40 secondary studies [22]. The study focuses primarily on code smells and refactoring and their relationship in literature. The authors find that the techniques that appear most are extraction techniques such as Extract Method. They also observe that there is a gap between refactoring execution and identifying refactoring opportunities. Moreover, Lacerda et al. argue that there is a lack of validation using experts in the field. They also find that the majority of projects described by current research utilize open-source code and more research into industry code is necessary.

Lastly, a survey by Dallal about identifying refactoring opportunities analyzed 47 papers and found that the majority of these used open-source as

their source for data [8]. They strongly advise researchers to incorporate industry experts to participate in future studies.

From these studies, we identify that code smell detection and refactoring are often strongly related. We also identify the following three techniques prevalent in this field of research:

1. Rule-based approaches (RB)
2. Search-based approaches (SB)
3. Machine learning-based approaches (ML)

We summarize the works described in this chapter in Table 2.1.

2.1 Rule-based approaches

Rule-based approaches apply rules on code metrics or other aspects of code to detect code smells or refactoring opportunities. Marinescu proposes a metric-based code smell detection technique [25]. They use logic rules in combination with code quality metrics to detect code smells. They report an average accuracy rate of 67% when analyzing 9 different types of smells.

Silva et al. use a similarity-based approach to detect Extract Method refactoring opportunities. They define a heuristic that scores candidates based on code dependencies. Using this in combination with selecting only the top recommendation per method, they achieve precision and recall rates of both 0.87.

Another approach was proposed by Moha et al. [27]. They created a framework named DECOR, which they later implemented in a tool called DETEX. Their tool uses DSLs in the form of rules which generate smell detection algorithms. These are then applied to the systems that were used to build these DSLs.

Tsantalis and Chatzigeorgiou propose a method to detect Extract Method type refactoring opportunities using code slicing. In a small case study, they report a developer agreement ratio of 5/9 methods [32].

2.2 Search-based approaches

Harman and Jones describes search-based software engineering as redefining software engineering problems as search-based problems and solving them using search techniques [18]. In addition to a search-based representation, he discusses the need for a fitness function and mutation operators for the representation as requirements for reformulating software engineering problems as search-based ones.

Later, Harman and Tratt proposed a search-based method that applies to Java code [19]. Their approach is fit for general purpose and allows for multiple fitness functions to present different Pareto optimal metrics.

	Approach	Results
RB		
[25]	Logic rules with code metrics to detect code smells.	Average accuracy of 69% on 9 code smells.
[31]	Predicting Extract Method using similarity.	Precision and recall of 87%.
[27]	Rule base DSL's that generate smell detection algorithms.	Precision and recall of up to 88.6% and 100% when detecting Blob code smell
[32]	Predicting Extract Method with code slicing.	5 out of 9 correct predictions in a small case study.
SB		
[18]	Proposal of search-based approaches for SE.	Definitions for facets required for search-based SE.
[19]	Finding Pareto optimal frontier to detect optimal metrics.	General purpose technique that allows multiple fitness functions.
[28]	Improvement of code maintainability testing 4 types of SB algorithms.	Ascent Hill climbing performs best with their tool.
[1]	Genetic algorithms with input from developer interactions to serve refactoring opportunities.	Precision and recall of 82% and 86% respectively on 10 different systems.
ML		
[13]	ML techniques to predict smells.	Accuracy up to 99% with Random forests when detecting Long Method.
[12]	32 different ML algorithms to detect 4 types of smells with code metrics.	Tree-based and naive Bayes perform best.
[24]	Deep learning to detect Feature Envy and subsequently Move Method.	53% and 75% accuracies on Feature Envy and Move Method respectively.
[36]	Combining static analysis and ML for recommending Extract Method for clones	83% and 76% accuracies within project and cross-project respectively.

Table 2.1: Summary of the related work discussed in this chapter

O'Keeffe and Cinnéide describe CODe-Imp, a search-based approach that allows for automatic improvement of code maintainability [28]. They test four different types of algorithms in conjunction with their tool and find that multiple-ascent hill climbing is the best-performing algorithm.

More recently, Alizadeh et al. propose an interactive method that defines an offline and an online phase [1]. The offline phase collects refactoring solutions using a genetic algorithm to serve the developer. In the online phase, the developer ranks these suggestions and these rankings are used in the next iteration of the offline phase to constrain the set of refactoring solutions.

2.3 Machine learning-based approaches

Fontana et al. use a machine learning-based approach to predict several types of code smells, including class-level smells Large Class and Data Class and method-level smells Long Method and Feature Envy [13]. They apply several machine learning algorithms including but not limited to SVM's, Random forests, and Naïve Bayes. Using code metrics as input, they achieve up to an accuracy of 0.990 when predicting Long Method using a Random forest type model.

Fontana et al. conducted a study comparing several machine learning techniques for code smells [12]. They evaluated 32 different machine learning algorithms to detect four different types of code smells. They used code metrics as input for their algorithm. They found that tree-based and naive Bayes algorithms resulted in the best performance in classifying code smells.

Another study by Liu et al. shows the application of deep learning to detect the Feature Envy code smell [24]. As input, they use code metrics and code transformed into vectors. They then, based on the outcome, predict the destination of a Move Method refactoring operation. They present an average f1 score of 52.98% in detecting feature envy and an accuracy score of 74.94% in recommending Move Method destinations.

Yue et al. combine static analysis and machine learning to recommend Extract Method to software clones [36]. They report an average f1 score of 83% when testing within projects. An average f1 cross-project score of 76% is reported.

2.3.1 The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring

In Chapter 1, we briefly discussed previous work done by Aniche et al. on refactoring. The work in this thesis, for a large part, expands on that approach. More implementation details can be found in Sections 3.1 and 4.1. Aniche et al. collected examples of refactoring and non-refactoring code

from three datasets, namely: Apache, GitHub, and F-Droid [4]. They collected refactoring examples, which include 21 different types of refactoring operations on class, method, and variable level.

Aniche et al. recorded many different types of code and process metrics with these examples to be used as feature vectors. Out of six types of models tested, they found that Random forest performed best with accuracy up to 0.99 on some types of refactoring operations. When analyzing which features are important, they found process metrics to be most important for predicting class-level refactoring operations, and class-level metrics to be important for predicting method-level refactoring operations. They also found that Random forest type models generalize well to unknown datasets. In addition, models trained on heterogeneous projects increase generalizability to unknown data.

Chapter 3

An Empirical Study of Extracted Methods in Industry

In this chapter we answer the following research questions:

RQ1 **How does code that underwent and did not undergo Extract Method in industry and open-source compare, in terms of code metrics?**

RQ2 **How does code that did and did not undergo Extract Method differ in different industrial projects?**

3.1 Methodology

To build a dataset consisting of code that underwent an Extract Method refactoring and code that did not undergo an Extract Method refactoring, we make use of the Git history. We walk through the Git history in reverse, all the while analyzing each commit. For every commit, we check whether it can be classified as an Extract Method commit or a non-Extract Method commit.

We explain these classifications steps in Sections 3.1.1 and 3.1.2. If we classify a commit, we save the metrics for the classified unit by using CK. CK collects code metrics by using static analysis[1]. This chapter analyzes a few key metrics but the complete list of collected features can be found in Appendix B. The approach is the same as the one used by Aniche et al. and replicated by Gerling [4, 16].

The scope of this thesis will be limited to the refactoring of type Extract Method for the following reasons:

1. This operation is relatively simple to understand, which will facilitate the interpretability of results. This is also advantageous for the user

[1] `https://github.com/mauricioaniche/ck`

study in Chapter 5, in which it is preferable that the operation is easily executable and comprehensible for all types of developers.

2. The current version of the data collection tool does not provide a sufficient degree of granularity for a large proportion of refactoring types detected. Take, for example, the refactoring of the type Rename Parameter which occurs when a method parameter is renamed. Our current data collection tool will tell us in which method this occurred but, if the relevant method has more than one parameter, it does not tell us which of these was renamed. Since we want to use this data to predict new refactoring opportunities it is important to choose a type that is usable in a practical situation. Although Extract Method does not tell us what part of the method to extract, it is an abstract enough operation to be more applicable than other refactoring types.

3. For industry code, there are fewer refactoring samples available compared to the large open-source dataset. For some refactoring types, such as Extract Interface, there are as few as 95 samples available in our dataset. Choosing Extract Method, gives us a combination of the above-mentioned advantages and a large number of samples to analyze.

3.1.1 Methods that underwent an Extract Method refactoring

We use RefactoringMiner version 2.0 [34, 35] to detect Extract Method occurrences in Git commits. RefactoringMiner is a refactoring detection tool that allows as input Git commits and subsequently detects any refactoring operations that occurred in those commits. Tsantalis et al. report to achieve precision and recall rates of 99.8% and 95.8% respectively for detecting an Extract Method refactoring operations [34, 35].

We collect the metrics of a class and method from the unit before the refactoring has occurred rather than after it has been completed. This is because we want to investigate the method's state for when it was a candidate for refactoring.

3.1.2 Methods that did not undergo an Extract Method refactoring

For this type of code, we could simply look at every commit and detect whether the class's methods did not undergo an Extract Method refactoring. However, a coherent change to a project often does not encompass only a single commit. For example, a change request that implements a particular feature often consists of multiple commits. To incorporate this phenomenon, we use a heuristic that defines a sensitivity parameter s. We

classify a class as one that did not contain methods that underwent an Extract Method refactoring if it was changed but not refactored for s consecutive times. In the context of the Git history this means the following: If we take s steps (commits) in the history without encountering a commit where any Extract Method occurred, we classify that commit and class as one which does not contain methods that underwent an Extract Method refactoring.

If we increase s, the tool gets less sensitive about what to class as a non-refactored class as the class needs to not be refactored for more consecutive commits. Conversely, if we lower the sensitivity, the collector will require fewer steps and confidence before classing a class as non-refactored.

In this report, we analyze a single value of s. We choose this value by using the following heuristic: The idea is that a change request consists of one comprehensive change in the project. Therefore, if we take the median amount of commits in change requests in all projects this might be a good approximation of the sensitivity level. This approach results in $s = 20$ for industry code and this will be the value used throughout this report.

A summary of the data collection process can be found in Figure 3.1 The

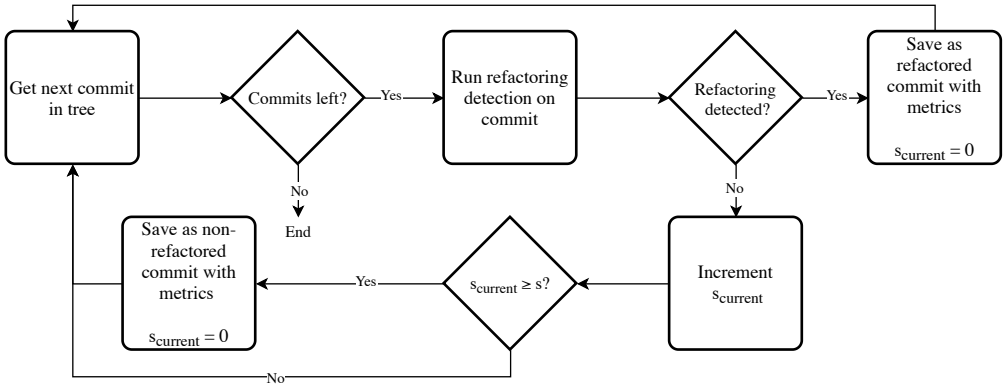


Figure 3.1: Simplified control flow for the data collection pipeline.

code for the data collection process can be found on GitHub [2].

3.1.3 Metrics to analyze

Metrics are captured on class and method-level. We only use metrics that are available in CK. For the metrics TCC and LCC, no method-level metric is available.

We primarily analyze metrics that were deemed highly important in our best-performing machine learning models as described in Sections 4.2.2 and 4.3.2. This leads us to analyze the following metrics:

- **Lines of code (LOC)** $[0, \infty]$ A longer class or method might indicate more complexity than a short one.

[2] `https://github.com/refactoring-ai/Data-Collection`

- **Response for class (RFC)** $[0, \infty]$ This is the sum of all distinct method calls plus the number of methods in a class/method. A higher value indicates more potential interactions and could indicate a higher complexity.
- **Cyclomatic complexity (WMC for classes, CC for methods)** $[0, \infty]$ Indicates branching complexity. For classes, we use the sum of the cyclomatic complexity of the methods in that class.
- **Quantity of unique words (UW)** $[0, \infty]$ A higher value might indicate more responsibilities or interactions with different domains for a certain class/method.
- **Coupling between objects (CBO)** $[0, \infty]$ Represents the number of connections to a respecting class/method.
- **Tight class cohesion (TCC)** $[0, 1]$ Measures cohesion between visible methods. This is calculated by dividing the number of direct connections between a class by the number of possible connections.
- **Loose class cohesion (LCC)** $[0, 1]$ The same as TCC, but this metric also takes into account indirect connections.

3.1.4 Duplication removal

Our datasets contain a significant amount of duplicates for both Extract Method and non-refactored instances. We class two instances as duplicates when their complete feature vector is equal to each other. For both cases, we remove these duplicates. We now go over why these duplicates occur in both cases and why we remove them.

We record an instance as Extract Method refactored when RefactoringMiner detects it in a commit. The problem arises when multiple Extract Method operations occur for the same method in the same commit. We give an example instance that was found during a manual investigation of the duplicates to elaborate. The instance in question was a method of a few hundred lines that was split up into five different methods. Therefore the data collection tool classed this as five Extract Methods, which is correct behavior. However, since this all occurred in the same commit, the same metrics were recorded for all five occurrences of this Extract Method. This behavior is useful when we want to know exactly what a certain Extract Method operation entailed, as with this approach, we can, for example, see to which new method part of the source method was extracted. However, since we are only interested in binary classification (i.e. did an Extract Method occur in a commit or not) it does not make sense to include these duplicates in our analysis.

For non-refactored instances, most duplicates occur because the detection of these types of instances is performed at class-level. We do not know whether a specific method was not changed for s consecutive times. When a class is classified as non-refactored for a commit, the tool classes all methods in that class as stable. It can happen, however, that a class is classified

as stable twice without the metrics associated with the methods in that class changing, causing a duplicate.

For both instance types, we also ran into the issue of code duplication, especially on project-level. Some projects had partially been duplicated into other projects. In some cases, we even saw complete duplication of commit history. This can lead to misleading results in our data analysis, as certain classes and/or methods can be overrepresented, while in reality, they are duplicates.

These duplicated instances can have adverse effects on our classification pipeline. All of the above reasons can lead us to end up with duplicates in our training and test set when splitting train-test data. This can cause our training algorithms to have access to the test set, overfit and inflate performance metrics. More details on the adverse effects of duplicated code in machine learning models are further elaborated upon in a paper by Allamanis [2].

3.1.5 Datasets

As explained prior, we analyze and compare two datasets. The first dataset consists of open-source code from GitHub. This dataset was mined by Gerling [16]. The projects were sourced from GHTorrent [17]. From this dataset, Gerling selected the top 100000 watched projects, and after removing faulty projects they were left with 92280 projects to analyze.

This resulted in 616088 Extract Method and 503393 non Extract Method instances. After removing duplicates, we were left with 449949 Extract Method and 460974 non-Extract Method instances.

The second dataset consists of proprietary code from an industry partner. This dataset contains metrics from 18 industry projects which were chosen with the help of industry experts. The data collection resulted in 2083 Extract Method and 1483 non-Extract Method instances. After removing duplicates for this dataset, we were left with 919 Extract Method and 986 non-Extract Method instances. These amounts are summarized in Table 3.1.

Dataset	Extract Method	Non Extract Method
Industry	919	986
Open-source	449949	460974

Table 3.1: Amount of samples in each dataset for each type of instance.

When analyzing individual projects we only analyze projects that have at least 30 Extract Method and non-refactored samples. We do this to ensure a certain level of confidence in our observations. After removing duplicates and applying the above rule we are left with six projects to analyze on an

individual level. Table 3.2 displays these projects and their Extract Method and non-refactored amount of samples.

Project	Extract Method	Non Extract Method
#1	58	37
#2	273	450
#3	152	84
#4	135	212
#5	49	46
#6	52	32

Table 3.2: Amount of instances for individual industry projects.

3.2 How does code that underwent and did not undergo Extract Method in industry and open-source compare, in terms of code metrics?

We plot each metric in violin plots, which gives the advantage of including the distribution of the data. The left side of the violin plot summarizes the metric for the open-source dataset while the right side summarizes the metric for the industry set. Since much of the data is widely distributed, using just the median as a summary of the data is not sufficient. We, therefore, use both the median and the interquartile range in our analysis and display them in the plots. We also summarize our findings in Table 3.3. The plots of some metrics are included in Appendix A to increase readability.

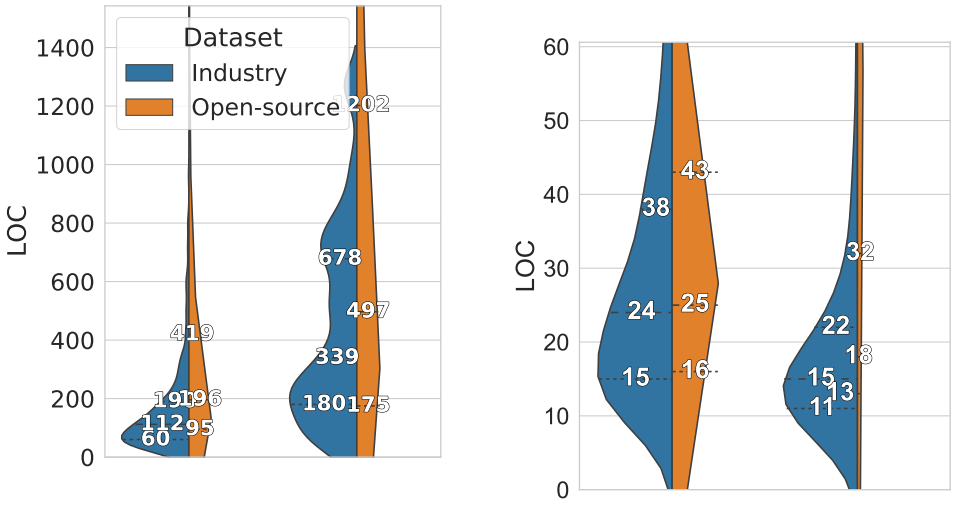
We plot both Extract Method refactored and non-refactored instances for each dataset in the same figure to allow for comparison between the two types.

Metric	$Class_{EM}$	$Class_{NR}$	$Method_{EM}$	$Method_{NR}$
LOC	↓	↓	≈	↓
RFC	↓	↓	↑	↓
WMC/CC	↓	↓	≈	↓
UW	↓	≈	↑	↑
CBO	≈	↑	≈	≈
TCC	↑	↑	—	—
LCC	↑	↑	—	—

Table 3.3: Summary of comparing industry to open-source feature distributions.
$Class_{EM}$ indicates classes that contain methods that underwent an Extract Method refactoring.
$Class_{NR}$ indicates classes that did not undergo an Extract Method refactoring.
$Method_{EM}$ indicates methods that underwent an Extract Method refactoring.
$Method_{NR}$ indicates methods that did not undergo an Extract Method refactoring.
↑ Indicates that the industry values of the metric are higher than the open-source one, ↓ indicates they are lower and, ≈ indicates they are approximately the same.

Observation 1: Classes that contain methods that underwent an Extract Method refactoring in industry code are smaller and less complex than classes in OSS. From Figure 3.2a, we see that industry classes that contain methods that underwent an Extract Method refactoring ($MED = 112, IQR = [60–190]$) are shorter than the same type of open-source classes ($MED = 196, IQR = [95–419]$). Similarly, for classes that did not contain methods that underwent an Extract Method refactoring, industry classes are also shorter ($MED = 339, IQR = [180–678]$) as their open-source counterparts ($MED = 497, IQR = [175–1202]$). From Figure A.1a we see that for classes that contain methods that underwent an Extract Method refactoring, the industry RFC ($MED = 38, IQR = [20–60]$) is slightly lower than RFC in open-source classes of this variant ($MED = 43, IQR = [23–79]$). We also observe that RFC for industry classes that do not contain methods that underwent an Extract Method refactoring ($MED = 80, IQR = [33–163]$) is lower than the RFC of the same type of open-source classes ($MED = 69, IQR = [31–134]$).

We see from Figure A.2a that the industry WMC for classes that contain methods that underwent an Extract Method refactoring ($MED = 23, IQR = [12–42]$) is almost half that of its open-source counterpart ($MED = 41, IQR = [19–92]$). We see the same pattern in the classes that do not contain methods that underwent an Extract Method refactoring, where industry WMC



(a) Class level: The left violin plot indicates classes that contain methods that underwent an Extract Method refactoring. The right violin plot indicates classes that do not contain methods that underwent an Extract Method refactoring.

(b) Method level: The left violin plot indicates methods that underwent an Extract Method refactoring. The right violin plot indicates methods that did not undergo an Extract Method refactoring.

Figure 3.2: LOC distributions for open-source and industry code on both class and method-level

($MED = 57, IQR = [22–142]$) is again almost half that of open-source ($MED = 105, IQR = [35–269]$). We also note that WMC for open-source classes is much more widely spread than its industry counterpart. This can be seen by the extremely thin violin for both types of instances for this dataset.

Figure A.3a We observe that for classes that contain methods that underwent an Extract Method refactoring, the unique word quantity is higher for open-source ($MED = 116, IQR = [68–199]$) as it is for industry ($MED = 90, IQR = [61–124]$). For classes that do not contain methods that underwent an Extract Method refactoring, we see that the industry unique word quantity is similar ($MED = 225, IQR = [134–407]$) to the open-source unique word quantity ($MED = 217, IQR = [103–394]$).

Observation 2: Industry method-level complexity metrics are generally more similar to open-source method-level complexity metrics than class-level complexity metrics are. Although there are differences between industry and open-source method-level complexity metrics, generally, they are less pronounced than the differences between industry and open-source class-level metrics.

Figure 3.2b shows us that industry methods that did not undergo an Extract Method refactoring ($MED = 25, IQR = [16–43]$) are similar to industry methods that did not undergo an Extract Method refactoring ($MED =$

$24, IQR = [15–38]$).

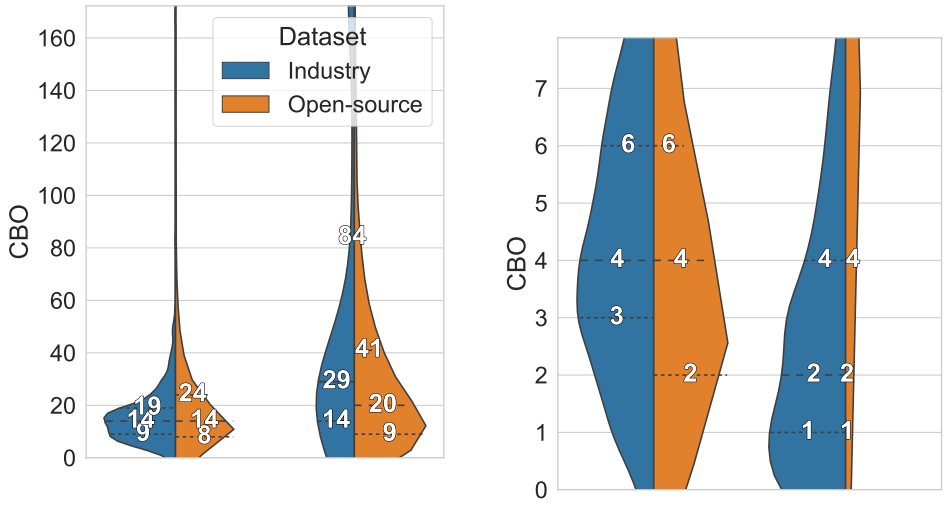
Figure A.1b shows that for methods that underwent an Extract Method refactoring, industry RFC ($MED = 13, IQR = [8–22]$) is slightly higher than open-source RFC ($MED = 10, IQR = [6–17]$). The opposite happens in methods that did not get refactored with Extract Method, where we see that industry methods ($MED = 5, IQR = [3–11]$) have a marginally lower RFC as the open-source ones ($MED = 7, IQR = [4–12]$).

In Figure A.2b We see similar industry cyclomatic complexity for methods that did not undergo an Extract Method refactoring ($MED = 3, IQR = [2–4]$) as the same type of methods in open-source ($MED = 4, IQR = [3–8]$).

From Figure A.3b, we see more unique words for methods that underwent an Extract Method refactoring in industry code ($MED = 34, IQR = [23–47]$) as the same type of open-source methods ($MED = 29, IQR = [19–44]$). We see the same pattern but more pronounced when we compare industry and open-source methods that did not undergo an Extract Method refactoring (($MED = 30, IQR = [18–43]$) for industry vs ($MED = 23, IQR = [15–36]$) in open-source).

Observation 3: Coupling is similar at class and method-level except for in classes that do not contain methods that underwent an Extract Method refactoring. In these cases it is higher for industry code. Figure 3.3a shows that for classes that did not undergo an Extract Method refactoring, the open-source class CBO ($MED = 20, IQR = [9–41]$) is much lower than the same industry class CBO ($MED = 29, IQR = [14–84]$). Most interestingly, however, is the upper quartile, where the open-source has a normal distance from the median, but the industry CBO is very high in comparison.

Figure 3.3b illustrates that for methods that underwent Extract Method, method CBO levels are very similar between industry ($MED = 4, IQR = [2–6]$) and open-source ($MED = 4, IQR = [3–6]$). The same holds for methods that did not undergo Extract Method, where the industry ($MED = 2, IQR = [1–4]$) and open-source ($MED = 2, IQR = [1–4]$) median and interquartile ranges are equal.

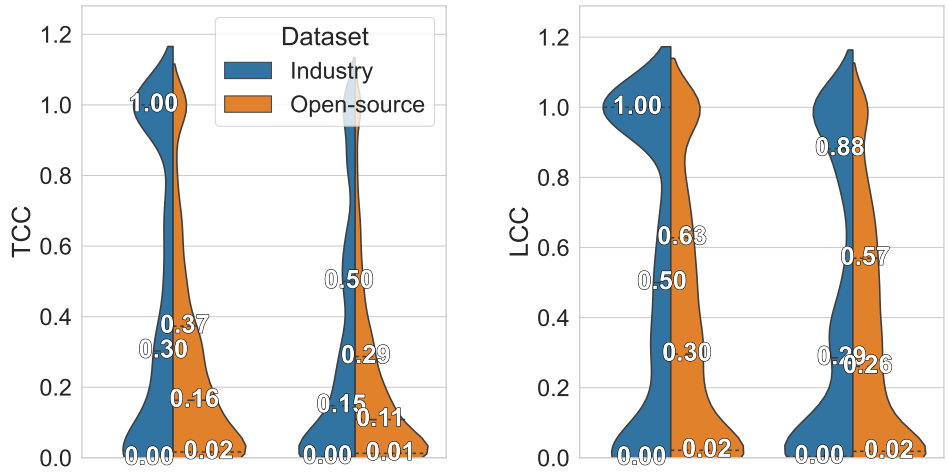


(a) Class level: The left violin plot indicates classes that contain methods that underwent an Extract Method refactoring. The right violin plot indicates classes that do not contain methods that underwent an Extract Method refactoring.

(b) Method level: The left violin plot indicates methods that underwent an Extract Method refactoring. The right violin plot indicates methods that did not undergo an Extract Method refactoring.

Figure 3.3: CBO distributions for open-source and industry code on both class and method-level.

Observation 4: Cohesion is higher in industry classes. From Figure 3.4a we see that for industry classes that contain methods that underwent an Extract Method refactoring, the interquartile range ($MED = 0.30, IQR = [0.00–1.00]$) spans all possible values of the TCC metric. This is probably due to the high density around 1.00 as seen by the upper part of the violin plot. This range is much wider as in the same open-source case ($MED = 0.16, IQR = [0.02–0.37]$). We see the same pattern, but more pronounced in LCC from Figure 3.4b, supporting our observation.



(a) TCC distributions for open-source and industry code.

(b) LCC distributions for open-source and industry code.

Figure 3.4: The left violin plot in each subfigure indicates classes that contain methods that underwent an Extract Method refactoring. The right violin plot in each subfigure indicates classes that do not contain methods that underwent an Extract Method refactoring.

Key takeaways:

- Classes that contain methods that underwent an Extract Method refactoring in industry code are smaller and less complex than classes in OSS.
- Industry method-level complexity metrics are generally more similar to open-source method-level complexity metrics than class-level complexity metrics are.
- Coupling is similar at class and method-level except for in classes that do not contain methods that underwent an Extract Method refactoring. In these cases it is higher for industry code.
- Cohesion is higher in industry classes.

3.3 How does code that did and did not undergo Extract Method differ in different industrial projects?

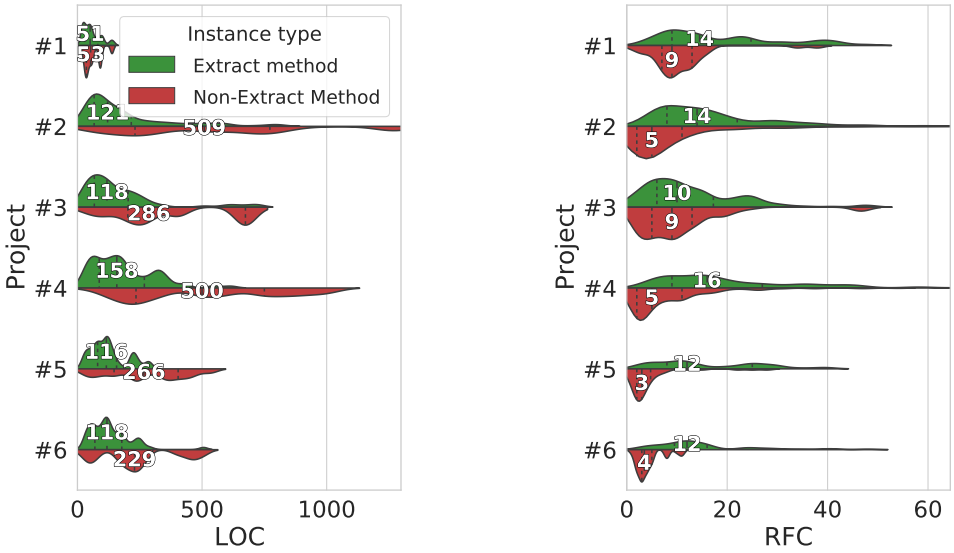
To analyze the differences in metrics in individual projects we create a violin plot per project in one figure. This allows for comparison between the projects. The top of the violin plot indicates code that underwent an Extract Method refactoring while the bottom indicates code that did not. For these figures, we only plot the median values and omit the interquartile range of the metrics to increase readability.

Observation 5: We observe clusters for class complexity metrics in the form of projects #2 and #4 and #3, #5 and #6.. Figure 3.5a illustrates that, for classes that do not contain methods that underwent an Extract Method refactoring, projects #2($MED = 509, IQR = [231–772]$) and #4 ($MED = 500, IQR = [235–750]$) form a group with similar class lengths and wide interquartile ranges. They are different from the non-refactored cases in #3 ($MED = 286, IQR = [202–674]$), #5 ($MED = 266, IQR = [147–404]$) and #6 ($MED = 229, IQR = [170–302]$) which also seem similar to each other, although the IQR's of these projects differ slightly.

Observation 6: Project #1 has very different complexity feature distributions when compared to the other projects. In Figure 3.5a we see that project #1 has much shorter classes than the other projects for both Extract Method ($MED = 51, IQR = [51–82]$) and non-refactored ($MED = 53, IQR = [36–88]$) code. For this project, we also see a very small IQR for both classes that contain methods that underwent an Extract Method refactoring and classes that do not contain such methods. The next project in line with the shortest classes is project #6 where classes that contain methods that underwent an Extract Method refactoring ($MED = 118, IQR = [70–178]$) are more than double as long. We also see that classes that do not contain methods that underwent an Extract Method refactoring ($MED = 229, IQR = [176–302]$) are more than quadruple as long.

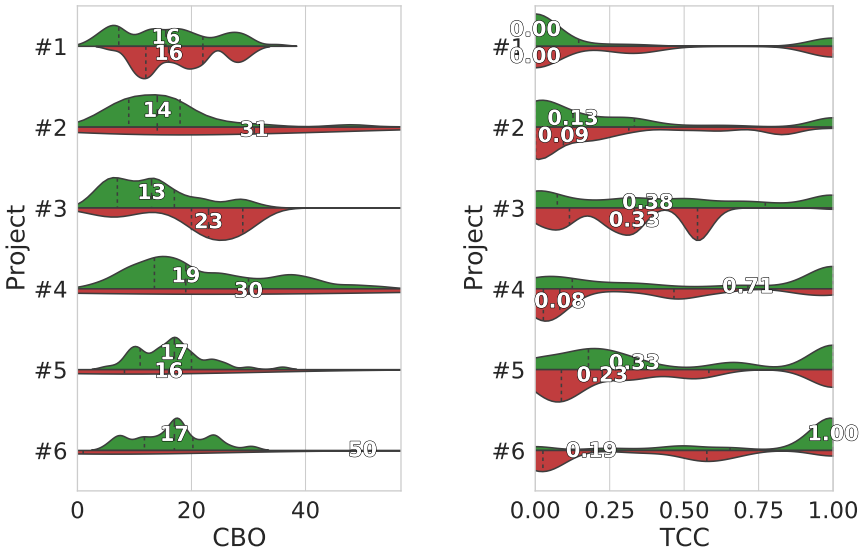
For RFC, quantity of unique words, and WMC we see the same patterns as described in observation 6. That is, for project #1 these metrics are much lower in both classes that contain methods that underwent an Extract Method refactoring and classes for which this is not the case in comparison with these metrics in the other projects and their range is also much less wide. The plots for RFC, quantity of unique words, and WMC can be found in Appendix A, in figures A.4a, A.4b and A.4c respectively.

Observation 7: Method level metrics are very similar between different projects. Figure 3.5b illustrates method-level CBO across industry projects. We see that for all projects, and for both types of instances, ranges



(a) Class level LOC

(b) Method level RFC



(c) Class level CBO

(d) TCC

Figure 3.5: Metrics per project. The y-axis indicates the project while the x-axis shows the distribution for the metric in question. The top violin indicates the distribution for code that underwent an Extract Method refactoring while the lower violin indicates the distribution for code that did not undergo an Extract Method refactoring. The number indicates the median for the metric in question.

of values do not differ from project to project to a large degree. For example, if we compare the largest difference between two projects for methods that underwent an Extract Method refactoring, which can be found between projects #4 ($MED = 16, IQR = [9–27]$) and #6 ($MED = 12, IQR = [10–16]$), we see that they are relatively insignificant. Appendix A contains the plots for the other method-level metrics which show similar patterns.

Observation 8: Class coupling varies widely in some projects for classes that do not contain methods that underwent an Extract Method refactoring. Figure 3.5c shows that the interquartile ranges for all projects but #1 ($MED = 16, IQR = [12–22]$) and #3 ($MED = 23, IQR = [22–29]$) are very large. The upper quartiles of some projects, such as #4 ($MED = 30, IQR = [19–320]$) are even larger, as evidenced by the fact they do not fit in the plot.

Observation 9: Code from project #2 has less widely distributed TCC values as other projects. Figure 3.5d illustrates that TCC rates for code in project #2 are much lower and are distributed less widely for both classes that contain methods that underwent an Extract Method refactoring ($MED = 0.13, IQR = [0.00–0.33]$) and classes that did not ($MED = 0.09, IQR = [0.00–0.31]$).

Key takeaways:

- There are two clusters of projects with similar feature distributions.
- Project #1 has very different complexity feature distributions when compared to the other projects.
- Method level metrics are very similar between different projects.
- Class coupling varies widely in some projects for classes that do not contain methods that underwent an Extract Method refactoring.
- Code from project #2 has less widely distributed TCC values as other projects.

Chapter 4

The Effectiveness of Data-Driven Extract Method Models in Industry

In this chapter we answer the following research questions:

RQ3 **How effective are supervised machine learning models at predicting Extract Method refactoring opportunities for industry code?**

RQ4 **How well do Extract Method recommendation models generalize?**

RQ5 **How well do Extract Method models generalize across different industrial projects?**

4.1 Methodology

We model the problem of classifying whether an Extract Method refactoring should be applied as binary classification. We attach true labels to code that is refactored with Extract Method. False labels are attached to code that did not undergo an Extract Method refactoring. Our feature vectors consist of the collected code metrics.

We train our models using scikit-learn [29]. We make use of the same algorithms as used in the paper of Aniche et al. except neural networks [4]. From an exploratory investigation, we found that neural networks did not achieve better performance when compared to our best-performing model. Because of this, the large number of hyperparameters, and the long training times we decide not to investigate this algorithm. To keep our figures readable we will refer to the algorithms by their abbreviation. The algorithms used and their corresponding abbreviations are as follows:

- Random forest (RF)

- Decision Tree (DT)
- Logistic Regression (LR)
- Linear SVM (SVM)
- Gaussian Naive Bayes (NB)

On a high level, the training pipeline consists of the following steps for each algorithm:

1. Pre-process data:
   a) Query Extract Method and non Extract Method instances and their corresponding metrics.
   b) Apply the associated labels.
   c) Shuffle the data.
   d) Split the data into a train and test set in a stratified manner.
   e) Scale the features.
   f) Apply feature reduction (LR only).
2. Train the model for every hyperparameter combination and investigate their performance.
3. Record the hyperparameter combination of the model with the highest performance.
4. Calculate the performance of this model on the test set.
5. Train a production model with the above parameters using both the training and test set and persist it.

The next sections will go into more detail on each of these steps. The code for the machine learning pipeline can be found on GitHub[1].

4.1.1 Data pre-processing

We use the data as described in Chapter 3. Our tool allows for the selection of training and test datasets. When we train and validate on the same dataset, for example, in the case of training and validating on the industry set, we split the dataset into a train and test set. If our training and test sets differ, for example, in the case of training on open-source, there is no need for a split, as the model is trained on open-source data and validated on industry data.

We next apply a label of “true” to the Extract Method instances and a label of “false” to instances that did not undergo an Extract Method refactoring. Depending on the datasets used, the aforementioned splitting is applied. We use stratification during the split to ensure classes are not overrepresented in the test or training set by chance. Afterward, we scale all our features using a MinMaxScaler since this benefits most machine

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

learning algorithms[2]. Finally, we apply feature reduction only for the logistic regression model.

4.1.2 Training approach

Each of these models has associated hyperparameters that must be tuned. To improve our models' performance, we optimize these hyperparameters by defining a hyperparameter space and exhaustively searching for the combination of parameters that results in the best performance. We use f1 as our performance metric because we are working with a slightly imbalanced dataset. We execute sklearn's grid search, which fits a model using all combinations in a pre-defined hyperparameter search space. For every combination of hyperparameters, we calculate the performance of the resulting model with the help of K-fold validation on the training set where $k = 10$. This will give us the combination of hyperparameters that produces the model with the highest performance on this training set. Appendix D contains the hyperparameter combinations of the best-performing models. With the resulting model, we now calculate the confusion matrix using the kept out test set. The raw confusion matrices can be found in Appendix E. These confusion matrices are then used to calculate the following metrics:

Accuracy	$=$	$\frac{TP+TN}{n}$	The proportion of correct predictions out of all predictions.
Precision	$=$	$\frac{TP}{TP+FP}$	The proportion of correct positive predictions.
Recall	$=$	$\frac{TP}{TP+FN}$	The proportion of correct positive predictions while taking into account the number of wrong negative predictions.
f1	$=$	$2 \cdot \frac{Precision \cdot Recall}{Precision+Recall}$	Designed to mitigate class imbalance. A high f1 score indicates a good performance for predicting both classes.

To analyze which features are most important for a model's performance we make use of permutation importance. This measures the reduction of performance in a particular model on a validation set if we randomly permute a certain feature [7]. If the performance drops significantly, we know that the feature is important for the model's performance. Conversely, if it drops only a small amount or not at all we know the feature to be unimportant for the model's performance on that set. We compute this on the validation set to measure the performance on unseen data. We permute

[2]Based on sklearn's manual: `https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler`

each feature 50 times to increase our confidence in the performance reduction.

We do not analyze coefficients for linear models (SVM and LR) and the feature importances available in impurity-based models (DT and RF). We opt for the permutation importance instead, since the linear coefficients and impurity-based feature importances illustrate features' importance on the training set rather than its importance for a model to perform well on unseen data. In addition to this, impurity-based feature importances are biased towards high cardinality numerical features[3]. For reference, plots of the above coefficients and feature importances can be found in Appendix C.

Finally, for the best-performing model only, we create a so-called "production model". This model is trained with the previously found best-performing hyperparameter set and uses all data, including the test set. It is not used for analysis as the test set is used to build it. This model and its associated scaler are then saved in ONNX[4] format. This format is useful in the experiment in Chapter 5 since it allows prediction in other programming languages.

4.1.3 Balancing

In their paper, Aniche et al. choose to balance their dataset such that the amount of positive samples is equal to the amount of negative samples. They do this because, for most refactoring types, the classes are highly unbalanced, and imbalanced data can lead to problems in machine learning [20].

In our experiments, we choose to not balance our dataset because of the following factors:

1. For the refactoring type Extract Method, this imbalance between classes is not as severe. We see this by the number of samples in each class displayed in Table 3.1. The ratio is 919 for Extract Method (48%) to 986 for non-Extract Method (52%) for industry code and 449949 for Extract (49%) to 460974 for non-Extract Method (51%) for open-source code.

2. During exploratory runs of our models, we did not observe significant changes in f1 performance when comparing a model trained on balanced vs imbalanced data.

4.1.4 Testing performance on unseen projects

In Section 4.1.1 we mention that we use a train-test split ratio of 0.8/0.2. We want to know, however, whether models perform well on completely unseen

[3] Based on sklearn's manual: `https://scikit-learn.org/stable/auto_examples/inspection/plot_permutation_importance.html`

[4] `https://github.com/onnx/onnx`

projects. That is if a model has not encountered any samples from a certain project, is it still able to perform on that project. To examine whether models trained on industry code generalize well to unseen projects, we train the model on all industry projects except for a single project. We then validate on that excluded project. We continue this until all projects have been used for validation once. An example follows to elaborate on this process.

Assume we have three projects: #1, #2, and #3. The first step would be to train on instances from #2 and #3 and validate on instances from #1. The next step would be to train on projects #1 and #3 and validate on #2. Finally, we train on #1 and #2 and validate on #3. For every excluded project we collect the performance metrics and analyze the performance on each project for every model type.

4.2 How effective are supervised machine learning models at predicting Extract Method refactoring opportunities for industry code?

First, we examine the performance metrics of the best-performing models. Then we investigate which features are important for those models.

4.2.1 Performance

We summarize our findings by compiling the achieved performance of each model into Table 4.1.

Observation 10: All performance metrics are relatively high in all model types. From Table 4.1 we see that there is no precision rate below 0.721 (NB). For recall, the lowest rate is 0.832 for a DT type model. Lastly, if we look at the aggregated metrics of accuracy and f1, we see that they do not drop below 0.782 (NB) and 0.799 (NB) respectively.

Observation 11: Out of all types of models tested, RF has the highest performance for all types of performance metrics. Table 4.1 shows that RF type models achieve the highest values for all performance metrics. For accuracy, f1, precision and recall we observe values of 0.934, 0.935, 0,899 and 0.973 respectively.

4.2.2 Feature importances

First, we analyze which features are most important for each type of model. We do this by sorting the permutation importance for each type of model from high to low. We then pick the top five out of this list and display them in Table 4.2 for comparison between model types.

Model type	Accuracy	F1	Precision	Recall
RF	0.934	0.935	0.899	0.973
DT	0.850	0.843	0.855	0.832
LR	0.824	0.825	0.794	0.859
SVM	0.829	0.832	0.793	0.875
NB	0.782	0.799	0.721	0.897

Table 4.1: Performance metrics for models trained and validated on industry code

Next, we investigate how each feature affects the performance of a model quantitatively. We achieve this by plotting the permutation importances in Figure 4.1.

Observation 12: The RF type model is more stable with regard to performance when permuting features in comparison with other types of models. Figure 4.1 shows that out of all models, when we permute features for the RF type model, the performance drops at most 0.0139. Out of all other models, the minimum max drop of 0.0233 occurs for NB when permuting CBO which is much higher as for RF. We also see performance drops as high as 0.1504 for the SVM type model when permuting UW, which is almost 15 times as high as the max performance drop for RF.

Observation 13: Class level features are most important for model performance. In Table 4.2 we see that for all model types, the top five features that reduce performance the most when permuted are all class-level. We do not see a single method-level feature in the top five for all types of models.

Observation 14: UW and CBO are the most important for most models. CBO is in the top five most important features for all types of models while UW is in all but one top five. Table 4.2 shows that UniqueWordsQty shows up as the number one feature for permutation importance in four out of five types of models. Only NB differs from this pattern and has NumberOfPublicMethods as the number one model. CBO shows up in the top five for all types of models and is the second most important for 3 out of models.

Rank	DT	LR	RF
1	UniqueWordsQty	UniqueWordsQty	UniqueWordsQty
2	Cbo	Loc	Cbo
3	methodRfc	Cbo	Lcom
4	Lcom	Wmc	Loc
5	Private	AssignmentsQty	Rfc

	NB	SVM
1	NumberOfPublicMethods	UniqueWordsQty
2	NumberOfMethods	Cbo
3	Cbo	ReturnQty
4	Lcom	methodRfc
5	ReturnQty	VariablesQty

Table 4.2: Top five highest feature importances for each model type both trained and validated on industry code. **All features are class-level except for method RFC.**

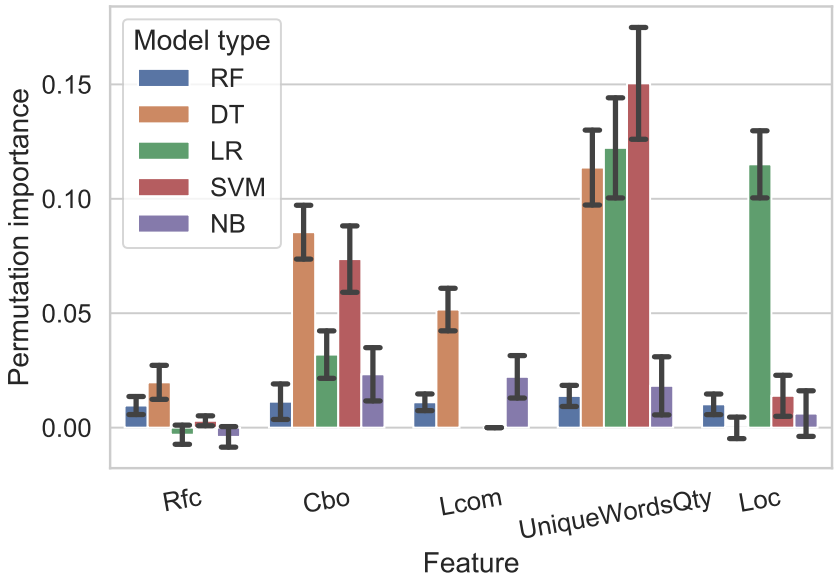


Figure 4.1: Permutation importance for models trained and validated on industry data. All features are class-level. The y-axis shows the mean decrease in model accuracy on the validation set when we permute the corresponding feature displayed on the x-axis 50 times. The error bars are the length of the standard deviations. The color of each bar indicates a type of model type. All metrics are class-level.

Key takeaways:

- All performance metrics are relatively high in all model types.
- Out of all types of models tested, RF has the highest performance for all types of performance metrics.
- Class level features are most important for model performance.
- UW and CBO are the most important for most models. CBO is in the top five most important features for all types of models while UW is in all but one top five.

4.3 How well do Extract Method recommendation models generalize?

We do this using the same approach as in Section 4.2 but instead of using the industry-trained model, we use the open-source-trained model.

4.3.1 Performance

Observation 15: Open-source-trained models, in particular linear ones, perform reasonably well on industry code. They do not perform nearly as well on industry code as models trained on industry code. Table 4.3 shows that the best-performing model (LR) achieves an accuracy and f1 score of 0.761 and 0.794 respectively. This is relatively high but much lower than the best-performing industry-trained model (RF) which achieves accuracy and f1 scores of 0.934 and 0.935 respectively.

Observation 16: Models trained on open-source code are much better at predicting non-Extract method instances as they are at predicting Extract Method instances in industry code. Table 4.3 illustrates that recall scores, which indicate the ability to perform well on non-Extract Method instances, are high in all models including the best-performing type (LR). For non-tree type models (SVM, NB, and LR), recall is even higher than their industry-trained counterparts. The same table shows that precision, a metric that indicates the ability to predict refactoring instances correctly, is much lower for all types of models where the best-performing type (LR) only achieves a score of 0.678 while the best-performing industry-trained model (RF) achieves a precision score of 0.899.

Model type	Accuracy	F1	Precision	Recall
RF	0.687	0.748	0.611	0.963
DT	0.606	0.664	0.564	0.808
LR	0.761	0.794	0.678	0.958
SVM	0.759	0.794	0.676	0.963
NB	0.614	0.713	0.556	0.995

Table 4.3: Performance metrics for models trained on open-source and validated on industry code.

4.3.2 Feature importances

Observation 17: Features that are important for open-source-trained models are similar to features important to industry-trained models. For models trained on open-source code, Table 4.4 shows that UniqueWordsQty occurs in the top five most important features for all but one (NB) type of model. CBO appears in the top five for three out of five types of models, including the best-performing types (SVM and LR). These are the same features important to the models trained on industry code as illustrated in observation 14.

Rank	DT	LR	RF
1	methodParametersQty	UniqueWordsQty	UniqueWordsQty
2	UniqueWordsQty	Cbo	Lcom
3	methodRfc	methodRfc	methodRfc
4	NumberOfFields	StringLiteralsQty	NumberOfPublicMethods
5	NumberOfStaticFields	Rfc	TCC

	NB	SVM
1	Cbo	UniqueWordsQty
2	StringLiteralsQty	Cbo
3	NumberOfPublicMethods	methodRfc
4	ReturnQty	StringLiteralsQty
5	NumberOfMethods	Rfc

Table 4.4: Top five highest feature importances for each model type for a model trained on open-source and validated on industry code. **All features are class-level except for methodRfc and methodParametersQty.**

Observation 18: Unlike industry-trained models, a method-level metric is important to the best-performing open-source-trained models. In observation 13 we noted that, for industry-trained models, there are

no method-level metrics important for all types of models, including the best-performing type (RF). From Table 4.4 we see that method RFC is the third most important feature for the best-performing models (SVM and LR). Method RFC is also important for tree-based models (DT and RF). Lastly, we see that one other method-level metric is important, namely methodParametersQty for the DT type model, where it is the most important feature for performance.

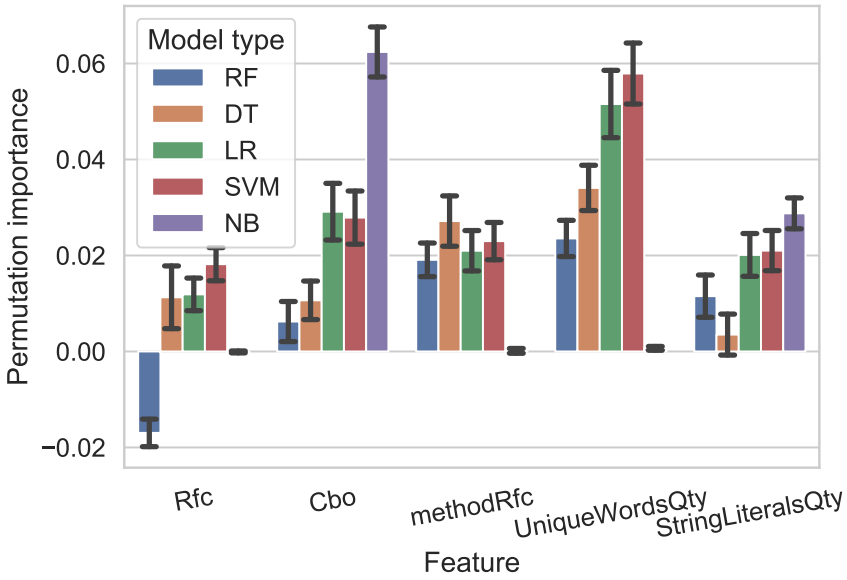


Figure 4.2: Permutation importance for models trained on open-source data and validated on open-source. **All features are class-level except for method RFC**. The y-axis shows the mean decrease in model accuracy on the validation set when we permute the corresponding feature displayed on the x-axis 50 times. The error bars are the length of the standard deviations. The color of each bar indicates a type of model type.

Key takeaways:

- Open-source-trained models, in particular linear ones, perform reasonably well on industry code. They do not perform nearly as well on industry code as models trained on industry code.
- Models trained on open-source code are much better at predicting non-Extract method instances as they are at predicting Extract Method instances in industry code.

- Features that are important for open-source-trained models are similar to features important to industry-trained models.
- Unlike industry-trained models, a method-level metric is important to the best-performing open-source-trained models.

4.4 How well do Extract Method models generalize across different industrial projects?

Table 4.5 shows the means of the performance metrics for the experiment explained in 4.1.4. Figure 4.3 summarizes the performance of all types of models on each individual project.

Observation 19: Average performance is still decent, but much lower than when using the whole dataset and slightly lower than when training on open-source. From Table 4.5 we see decent aggregated performance metrics with the best model for aggregated performance (RF) having mean accuracy and f1 rates of 0.744 and 0.771 However, performance is much lower as when testing on the whole dataset (RF, 0.934 and 0.935) and slightly lower as the open-source-trained model (SVM, 0.759 and 0.794). We do see, with the help of the precision metric, that performance on Extract Method instances (LR, 0.787) is higher compared to the open-source-trained model (LR, 0.678).

Model type	Accuracy	f1	Precision	Recall
DT	0.612	0.653	0.609	0.724
NB	0.652	0.601	0.758	0.525
SVM	0.711	0.671	0.762	0.641
LR	0.720	0.692	0.787	0.668
RF	0.744	0.771	0.715	0.860

Table 4.5: Means of performance metrics when training on all industry projects but one.

Observation 20: For each project, there exists a type of model that performs well, except for project #1. In Figure 4.3 we see that for all projects, except project #1, there is always one type of model that achieves a score of 0.75 or higher. For project #1 this is not the case, as the highest f1 score is only 0.66 for the RF type model. This is due to a low recall score, as the aforementioned RF type model only achieves a recall score of 0.58 on the project.

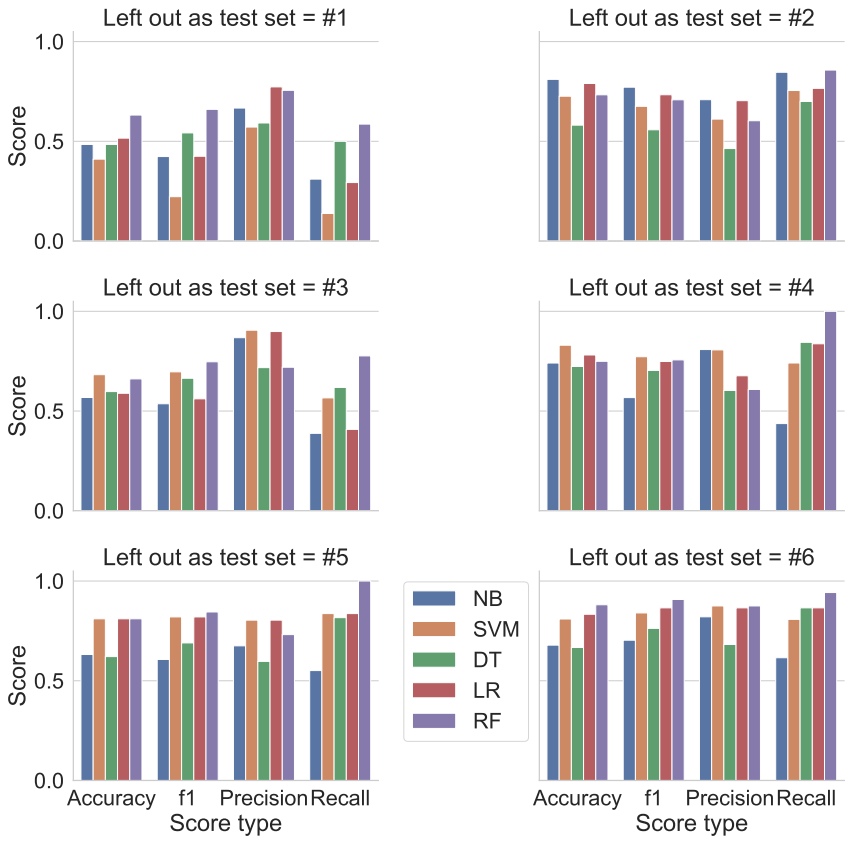


Figure 4.3: Performance metrics when training on all but one project and validating on the left out project. There is a sub-plot for each excluded and validated on project. Every bar in the plot signifies a certain model type. The groups on the x-axis of bars indicate the type of scoring metric.

Observation 21: The project with the worst performance (project #1) is the same project that showed a different feature distribution for class complexity metrics in Section 3.3. In observation 6 we showed that project #1 differs in complexity metrics from the other projects for classes that do not contain methods that underwent an Extract Method refactoring. This is the same project on which our models do not perform well while predicting that type of code.

Observation 22: Different types of models perform well on different projects. From Figure 4.3 we see that there is not one type of model that performs the best on one project. From the f1 score, we see that the RF type model performs best on projects #1, #3, #5, and #6. We observe that the SVM type model performs best on project #4. Finally, we see that the LR type model performs best on project #2.

Key takeaways:

- Average performance is still decent, but much lower than when using the whole dataset and slightly lower than when training on open-source.
- For each project, there exists a type of model that performs well, except for project #1.
- The project with the worst performance (project #1) is the same project that showed a different feature distribution for class complexity metrics in Section 3.3.
- Different types of models perform well on different projects.

Chapter 5

A User Study on Extract Method Recommendations in the Wild

In this chapter we answer the following research questions:

RQ6 **Do industry experts deem recommended Extract Method refactorings useful/not useful?**

RQ7 **Why do industry experts deem recommended Extract Method refactorings useful/not useful?**

5.1 Methodology

All predictions served to experts in this chapter were generated by the best-performing model in Chapter 4. This is the Random Forest type model that was trained on industry code. We choose code quality experts at ING and invite them to fill in a questionnaire. All participants have substantial experience with the Java programming language. Two participants work with the code displayed in the survey daily, while three offer an outside perspective.

The survey consists of 30 questions, each displaying a method originating from industry code. The set of methods to display was randomly chosen from the predictions generated during the experiment described in Section 6.1. For every method, the industry expert will answer two questions, one qualitative and one quantitative. The study is blind with participants not knowing what the model's assigned label for a given method is.

The first question consists of a scale with four levels where the expert indicates to what extent they find an Extract Method should be applied to the method shown. A score of 1 indicates that they think it should not be applied at all, while a score of 4 indicates that the operation surely should be applied. A score of 3 or 4 is interpreted as a sign that the expert thinks an Extract Method should be applied to the method. Vice-versa, a score of

1 or 2 is interpreted as a sign that the expert thinks the method should not be refactored with Extract Method. The second question asks the expert to elaborate on their choice. An example of a question about a method can be found in Appendix F, in Figure F.1.

We settle on a total of 30 methods because this limits the time spent on the survey by each expert and gives us a decent statistical sample. The average time for an expert to complete the survey was approximately 42 minutes. We select one set of methods for the questionnaire, that is, every expert receives a survey containing the same methods. We are more interested in evaluating methods for which an Extract Method refactoring is suggested as opposed to when no Extract Method refactoring is recommended. Therefore, we select 20 methods where the model attached a true label to the method and ten where it attached a false label.

We manually check the answers to the qualitative questions. We do this by examining and attaching characteristics to each answer. We then identify patterns in answers and summarize the reasons into characteristics for the quantitative answers. We only characterize an answer a certain way if the participant explicitly mentions that specific characteristic in their answer.

5.2 Do industry experts deem recommended Extract Method refactorings useful/not useful?

We analyze the agreement of the experts with the model by measuring how often the experts agree with the model's prediction to apply Extract Method or not to refactor. We define the following four situations:

- **Extract Method agreement (R_A)** The model classifies the method as to be refactored. The expert agrees that an Extract Method refactoring is necessary.
- **Non-refactor agreement (N_A)** The model classifies the method as to not be refactored. The expert agrees that no Extract Method refactoring is necessary.
- **Extract Method disagreement (R_D)** The model classifies the method as to be refactored. The expert disagrees and thinks an Extract Method refactoring is not necessary.
- **Non-refactor disagreement (N_D)** The model classifies the method as to be refactored. The expert disagrees and thinks an Extract Method refactoring is necessary.

With these situations, we define four ratios similar to accuracy, precision, recall, and f1 used in Chapter 4.

- AR: Agreement ratio of the experts with the model's predictions (similar to accuracy).
- AR_{EM}: Agreement ratio of experts for methods where the model predicts Extract Method (similar to precision).
- AR_{NR}: Agreement ratio of experts for methods where the model predicts to not apply an Extract Method refactoring (similar to recall).
- AR_{f1}: Agreement ratio while taking into account imbalance of classes such as is the case in our experiment (similar to f1).

These definitions allow for one-to-one comparison with the theoretical performance. The raw occurrences and calculations can be found in Appendix G.

Expert	AR	AR_{f1}	AR_{EM}	AR_{NR}
1	0.767	0.788	0.650	1.000
2	0.700	0.757	0.700	0.824
3	0.567	0.606	0.500	0.769
4	0.667	0.687	0.550	0.917
5	0.900	0.919	0.850	1.000
All/Mean	0.720	0.756	0.650	0.903

Table 5.1: Each row shows the scores for one participant.

Observation 23: Industry experts' opinions on model predictions seem to align with the model's predictions to a certain extent, but do not align as well with the theoretical score. From Table 5.1 we see that, for each participant, no aggregated scores (AR, AR_{f1}) are below 0.567. The scores are generally much lower than in the best theoretical case, however. For that case, we see in Table 4.1 that accuracy and f1 scores are 0.934 and 0.935 respectively.

Observation 24: The agreement of experts with the model where the model predicted to apply an Extract Method is lower than the agreement with the model's predictions to not apply Extract Method. Table 5.1 shows that AR_{NR} rates are much higher as AR_{EM} rates, which indicates higher agreement on predictions where the model predicted to not apply Extract Method in comparison with cases where the model did recommend to apply an Extract Method refactoring. The average AR_{NR} is 0.903 while the average AR_{EM} of 0.650 is much lower. Also, if we look at the minimum AR_{EM} and AR_{NR}, we see that the minimum AR_{EM} of 0.5 is much lower than the AR_{NR} counterpart of 0.769. The same is true for the maximum, where the AR_{EM} of 0.850 is quite a bit lower than the maximum AR_{NR} of 1.00.

Observation 25: Experts who work with projects where the model was trained on seem to agree with the model's predictions more. Table 5.1 shows that for participants #2, #5, and who use the analyzed code daily, performance rates are higher on average as for people who do not use the code daily. The average AR, AR_{f1}, AR_{EM} and AR_{NR} rates are 0.720, 0.756, 0.903 and 0.650 respectively. Expert #2's outperforms this, with metrics of 0,700, 0.757, 0.824, and 0.700. The same is true to an even greater extent for expert #5's metrics of 0.900, 0.919, 1.000, and 0.850 which are the best out of all experts.

Key takeaways:

- Industry experts' opinions on model predictions seem to align with the model's predictions to a certain extent, but do not align as well with the theoretical score.
- The agreement of experts with the model where the model predicted to apply an Extract Method is lower than the agreement with the model's predictions to not apply Extract Method.
- Experts who work with projects where the model was trained on seem to agree with the model's predictions more.

5.3 Why do industry experts deem recommended Extract Method refactorings useful/not useful?

Using the process explained in Section 5.1, the following characteristics were identified:

1. **Understandable:** The method is described to be understandable.
2. **Specific:** The method is described as specific in its goal. Often these answers contained remarks on how the method's purpose was domain /functionality-specific.
3. **High complexity:** The method is marked as too complex. This includes matters such as long methods, too many try-catch blocks and, other complexity-related issues.
4. **Pattern:** The participant mentions a design pattern or anti-pattern. The specific pattern that was meant was not mentioned in most cases.
5. **Potential:** The participant mentions that an Extract Method operation does not necessarily have to be applied but that it could improve code quality.
6. **Repetition:** Code repetition is mentioned.

7. **Readability:** The participant mentions that understandability can be improved.

These include the reasons for all situations, including agreement and disagreement for the models.

We summarize the results of this experiment by plotting the frequency of characteristics in a bar chart. We present two figures each with two bar charts, one figure displays the frequencies of reasons given where the participants agreed with the model (Figure 5.1), and the other plots the reason frequencies where the experts did not agree with the model (Figure 5.2).

Observation 26: When experts agree with the model's decision to apply Extract Method, they most often cite the need to apply Extract Method because of high code complexity, followed by patterns or code smells. We see from Figure 5.1 that the most commonly given reason for apply Extract Method to a method with 25 occurrences is that it is too complex. The second most given reason with 21 occurrences is that the method contains an anti-pattern or does not adhere to a pattern. Other reasons are not given very often as can be seen by the number of occurrences of only four for the next most common reason.

Observation 27: When experts agree with the model's prediction to not apply Extract Method, they give as a reasons that the code is specific enough most frequently. The second most frequently given reason in this situation is that the method in question is sufficiently understandable. We observe from Figure 5.1 that when participants agree with the model's prediction not to refactor, they most commonly, with 22 occurrences, mention that the method is specific enough. The next most common reason, with 17 appearances, is that the method is understandable enough. The next reason ("potential") is much less common but still appears four times.

Observation 28: Experts tend to give similar reasons for their choices whether they agree with the model or not. Figure 5.2 shows the distribution of characteristics in answers for instances where the study participant did not agree with the model's prediction. We see from the left plot that, when a participant thinks an Extract Method operation is necessary, but the model proposes it is not, the top reasons given by the participant stay the same as when they do agree. In the situation the participant says an Extract Method is necessary, they mention design patterns/code smells and high complexity. The same is true when the participant does not think an Extract Method operation is necessary, where we see that understandability and a method being specific enough are the top reasons, which is the same as when the participant agrees with the model.

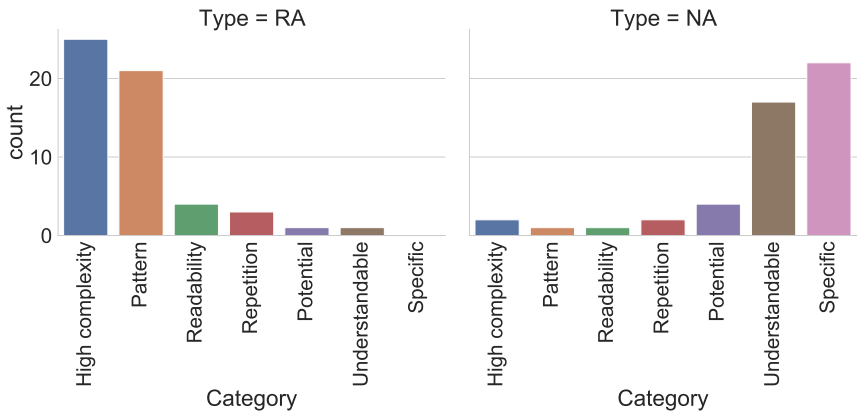


Figure 5.1: The number of times a characteristic appeared in a participant's reasoning for cases where they agreed with the model's prediction. The plot on the left shows the case where the model suggested to apply an Extract Method operation on the method. The plot on the right shows when the model suggested to not refactor the method.

Observation 29: When experts disagree with the model's prediction to apply Extract Method, they often do propose a potential refactoring operation different from "Extract Method". Figure 5.2 shows that the third most common characteristic of "potential" is mentioned when the user does not agree with the model's prediction to apply Extract Method. We see this occur eight times. These answers were often also accompanied by an explanation on how the method could be Extract Method, but the corresponding refactoring operation mentioned was not "Extract Method".

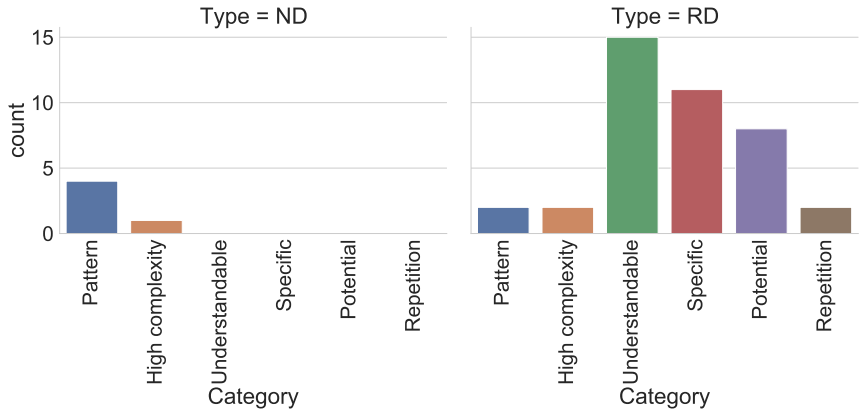


Figure 5.2: The number of times a characteristic appeared in a participant's reasoning for cases where they did not agree with the model's prediction. The plot on the left shows the case where the model suggested to apply an Extract Method refactor to the method but the user did want to refactor. The plot on the right shows when the model suggested applying Extract Method to the method but the expert did not think an Extract Method refactoring was necessary.

Key takeaways:

- When experts agree with the model's decision to apply Extract Method, they most often cite the need to apply Extract Method because of high code complexity, followed by patterns or code smells.
- When experts agree with the model's prediction to not apply Extract Method, they give as a reasons that the code is specific enough most frequently. The second most frequently given reason in this situation is that the method in question is sufficiently understandable.
- Experts tend to give similar reasons for their choices whether they agree with the model or not.
- When experts disagree with the model's prediction to apply Extract Method, they often do propose a potential refactoring operation different from "Extract Method".

Chapter 6

A Proposal for a Tool To Recommend Data-Driven Extract Methods

This chapter elaborates on how the different software artifacts of the prediction pipeline function and interact with each other. Figure 6.1 shows an overview of every separate software artifact used to achieve the results in this thesis. We refer to each component with the number displayed in

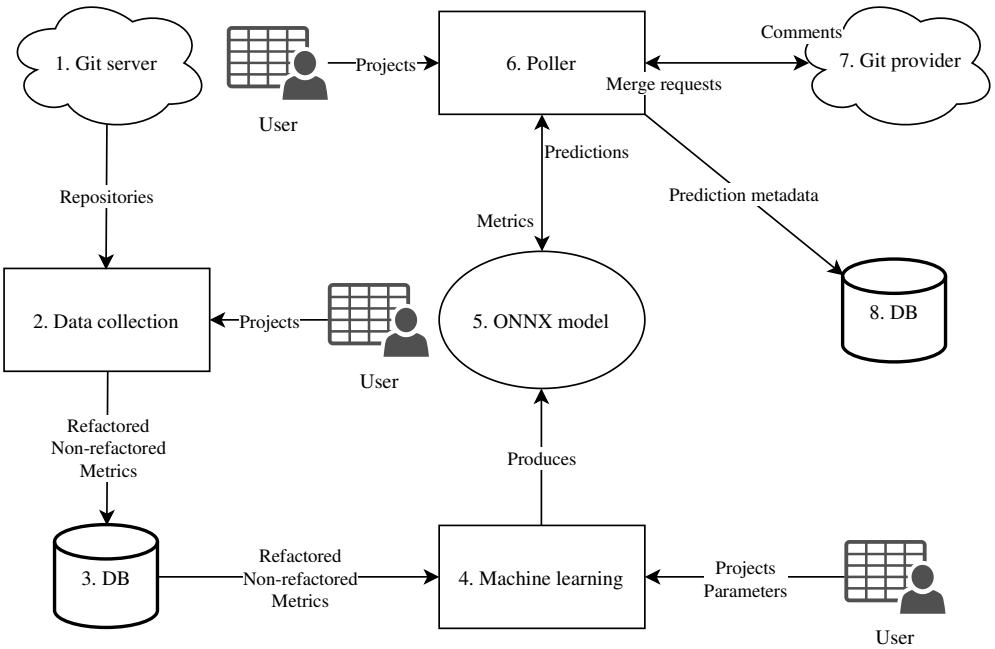


Figure 6.1: The architecture of all artifacts and their relationships with each other.

Figure 6.1. A normal workflow would be as follows:

1. The user starts the data collection process (box 2. in Figure 6.1) with

a CSV containing the projects they wish to collect data of as input.

2. For each project, the data collection process (box 2.) fetches the projects from a Git server (box 1.) and collects data as specified in Section 3.1. It then stores this in a database (box 3.).

3. The user then starts the machine learning (box 4.) pipeline with the list of training and test projects and the required machine learning parameters. The Machine learning (box 4.) process fetches the relevant data from the DB (box 3.) and builds the model (box 5.) as specified in Section 4.1.

4. Next, the user starts the poller (box 6.) with the projects they want to recommend Extract Method on.

The poller does what its name suggests: it polls for any change requests that might have been opened since it last polled. A more elegant solution would be to implement this in a continuous integration (CI) pipeline, that is, the recommendation of refactoring opportunities would be a step in the CI pipeline. Such a prototype was built, but it required too many privileges to implement into a production system. The approach we took allows for any user with access to the VCS to place comments and minimizes risk as only the privileges of the user running the tool are exposed. It also allows for iterative experimentation, as we can easily tweak the process and models without having to go through the steps of approval again.

The tool goes through all projects that were specified by the user, and for each of these projects, it uses the Git repository provider API to fetch a list of open change requests. Then, it checks whether that particular change request is already processed, and if not it starts processing it. It clones the project if necessary, and checks out the commit hash of the change request. Then, for every changed method, it collects the code metrics and feeds them into the ONNX model. The model then returns a label and a probability. For every method where the model predicted an Extract Method refactoring, the model attempts to place a comment on the appropriate line in the corresponding change request. The details of the following entities used in the process are saved to a DB (box 8.) for further analysis.

- The methods that were analyzed.
- Their assigned labels and probabilities.
- The model used to predict these.
- The change request in which the methods were encountered.
- The project in which the change request was located.

In addition to this, the link to the survey in the comment on the change request also contains the id of the prediction so the feedback can easily be linked to the corresponding prediction. An example of such a comment can be found in Appendix F in Figure F.2.

We also allow for the specification of a probability threshold and a maximum amount of comments per change request. For our experiment, we set the probability threshold to 60% and the number of comments to three. We also sort the predictions so that the highest confidence predictions are recommended first. We added this step so as to avoid the potential to overwhelm developers, as when testing, we saw some change requests with over 20 generated comments.

The code for this tool can be found on GitHub[1].

6.1 The tool in practice

We ran this tool on 16 different projects for the duration of two months. Before deploying the tool, we sent out an email informing the developers who worked on these projects that they might expect comments and asked them to interact with the comments.

We analyzed a total of 147 change requests with 1641 total methods. Out of these, 57 predictions were successfully recommended on open change requests.

The tool worked well technically, but as alluded to in Chapter 5, the tool's initially set goal was not achieved due to the following reasons:

1. Although a large number of methods were analyzed, we were only able to place comments on a small fraction of them. This is because the method has to be edited in the change request in order to place a comment on the corresponding line.

2. The conversion rate for people clicking on the feedback link was low. Out of the 57 comments placed, only 11 people clicked on a link of which seven actually finished filling in the survey. Of these people, only two gave a reason for their answer. Some indications for why this was the case were given by developers who received the comments on their change requests. First, the refactoring suggestion was not specific enough in what lines to extract from the respective method. Secondly, they mentioned that when a comment is placed on a line in a change request, they can only see the title of the corresponding method and not the actual contents of the method without extra manual steps. This makes it difficult to see the contents of the method as one has to manually inspect the diff and find the corresponding method.

Nevertheless, this tool can be adjusted and used in the future to further research the use of machine learning generated refactoring predictions in industry code. In particular, it might be interesting to investigate whether making predictions more specific increases conversion.

[1] `https://github.com/refactoring-ai/Refactory`

Chapter 7

Threats to validity

7.1 Internal validity

This section illustrates potential threats to validity originating from design decisions.

The disparity between models After already having finished the execution of the user survey, we found a few imperfections in the pre-processing step for training our models. Since the user study was already completed when these were found, we were unable to deploy the most up-to-date model. We did see, however no large differences in theoretical results after implementing these improvements. We also did not see any significant change in model behavior regarding feature importances. The following list outlines the changes in the pipeline and their potential impact on the user study:

1. **The lack of stratification when splitting into a train and test set.** Our dataset consists of slightly unbalanced data that is shuffled before the split. This could cause one class to be over or under-represented in either the train or test set due to random chance. However, we did not see a large change of performance metrics or feature importances after implementing this stratification. In addition to this, the dataset was only slightly unbalanced in the first place.

2. **The discovery of duplicates in the training set.** In Section 3.1.4, we explain the removal of two types of duplicates in the training set. These were only discovered after the completion of the user study. It resulted in the removal of around 500 samples from the industry dataset. Again, we saw no significant change in feature importance. We did see a slight decrease in performance which is a known phenomenon for duplicated code in machine learning for software engineering [2].

Since the improved model more accurately follows the real distribution of data and we, therefore, decrease areas where it could overfit, we hypothesize that the results shown in the user study can only improve when deploying such an improved model in the future. Nevertheless, it cannot be guaranteed that these factors did not have an influence on the predictions the model generated for use in the user study and therefore also on the user's responses to them. A replication of the user study with these improved models could be performed in order to investigate the impact of the issue.

Use of class metrics in method-level refactoring This report analyzes the refactoring of type Extract Method, and since every method is contained in a class, we train not only on method-level metrics but also on class-level metrics. We do this to give the classification algorithms "context". This does mean, however, that these class metrics are often duplicated in our dataset, as multiple methods exist in one class. We saw that class-level metrics are most important for our model's performances. This can cause the same problems as described in 3.1.4, as class metrics from the test set "leak" into the test set, inflating performance. More research is needed into the role of partially duplicated feature vectors in such models.

Correlation between features In both this and previous work, feature performance is analyzed by permuting single features and seeing the decrease in performance in the model. When two features are correlated, however, the model could still have access to that feature by proxy, making it appear that the feature is not important when in reality it is[1]. Further research needs to be done into the role of potentially correlated features.

Difference between refactored and non-refactored instances In our problem definition, we define two types of instances, Extract Method and non-Extract Method instances. While we classify Extract Method instances by detecting them, non-Extract Method instances are classified using a heuristic. This heuristic includes the step of checking whether no refactoring operations occurred. When the tool does this, it checks for all types of refactoring operations detected by RefactoringMiner 2.0 and not only for Extract Method.

This means there is a disparity between the way Extract Method and non-Extract Method instances are detected as we are, in theory, building a model that classifies Extract Method as one class and no refactoring of any type as the other class. Future research needs to be done to measure the effectiveness of a truly binary model. This entails detecting Extract Method

[1] According to the sklearn manual: `https://scikit-learn.org/stable/auto_examples/inspection/plot_permutation_importance_multicollinear.html`

in the same way as is done in this work but detecting non-Extract Method instances by using only Extract Method and no other refactoring types.

Software faults As seen by Figure 6.1, building a complete pipeline that executes all steps for generating Extract Method predictions is a complicated process. Even though automated tests were implemented for most of these steps, bugs, and design flaws were still found during the execution of this project. Most of the individual tools build for the pipeline are not production quality, and faults in any single part of them could affect our results. That being said, extensive manual checking was done at every step of the process. It would, however, be advisable for future research to combine all steps of this pipeline into a single software artifact. This minimizes the risk of mismatches between the individual artifacts or misalignment between configurations.

Finally, we use non-mature external libraries and tools such as RefactoringMiner 2.0, ONNX, CK, sklearn. Any bugs, inconsistencies, or faults in these libraries might render our results less reliable.

7.2 External validity

This section illustrates possible threats to validity originating from a lack of generalizability of our approach.

Past refactoring operations as an oracle for future operations Our models are based upon the belief that decisions made in the past regarding refactoring are correct. It could be, however, that code that is a valid candidate for refactoring, might not be refactored as explained in Chapter 1. Furthermore, refactoring could decrease code quality, introduce bugs or cause a change of behavior in some instances.

We believe this not to be a significant problem, as it is shown that refactoring generally has many advantages [23, 21, 15], and we believe these will be captured into our models as a result.

Survey participants In our survey, we chose three experts who gave an outsider's perspective and two who worked with the relevant code on a daily basis. It could be that the outside perspective is not an accurate assessment of what would be appropriate refactoring opportunities for the respective project. This could be because, in order to know what to refactor, experience with the project is needed.

Next to this, it could be the case that the assessment of the experts familiar with the respective projects is inaccurate. The model might have learned from refactoring in the past that these two experts executed. Since the model is trained on this data, a subsequent suggestion would therefore be biased towards these developers, creating a self-fulfilling prophecy.

However, refactoring is often subjective, and code quality is in the eye of the beholder. The quality of a project often not only consists of metrics and adherence to patterns but also out of how well it is regarded by the team working on it. In addition to this, we have no reason to believe that the refactoring operations executed by these experts reduce code quality for these projects in particular. We can, however, make no strong guarantees about their generalizability of the approach to experts other than the ones in this study.

Language specificity The current tool focuses only on the Java programming language. It is assumed that many of the observations made in this thesis would translate to other object-oriented programming languages but there is no guarantee for this. Next to this, we can make no assumptions on how well the approach would work on different programming paradigms such as imperative ones. It would therefore be useful to try a similar approach in other popular languages, both object-oriented and imperative. Another domain of interest is the application of this approach for a more high-level multi-language approach.

Domain specificity For this report, we compare the performance of the approach on industry code. This is code from a single company in the financial industry. It could be that the results shown here do not translate to other industries in the same manner. This approach has to be applied in different companies to investigate this. In addition to this, it is interesting to investigate whether a model trained for one company would translate well to another.

The role of process metrics In previous work, it was noted that process metrics were classed as important for model performance [4, 16]. We decided not to use these types of metrics because of the complexity of calculation in our recommendation engine for the user study. Therefore, performance might be lower than it could potentially be. However, due to their importance in previous work, it might be interesting to test the approach in an industry setting including these metrics.

Chapter 8

Conclusions and future work

8.1 Conclusions

This work has shown that using supervised machine learning models to predict Extract Method refactoring opportunities in industry code is an effective approach. These results could lead to better, easier and finer tailored automated refactoring suggestions in large-scale industrial systems.

The following paragraphs will summarize the approach and results for each research question.

How does code that underwent and did not undergo Extract Method in industry and open-source compare, in terms of code metrics? We analyzed and compared code metric distributions between industry and open-source code for both code that underwent an Extract Method refactoring and code that did not undergo an Extract Method refactoring.

From our observations, we can conclude that **code that underwent Extract Method and code that did not undergo Extract Method in an industry setting does differ from the same type of open-source code.** We see differences especially at class-level, but less so at method-level, as illustrated by observation 2. Observation 1 illustrates longer and more complex classes in open-source code as opposed to industry code. Next to this, observation 3 shows us that coupling levels are similar between industry and open-source code, except for classes that contain methods that underwent an Extract Method refactoring. Lastly, in observation 4, we observe that cohesion is higher in industry code than it is in open-source code.

How does code that did and did not undergo Extract Method differ in different industrial projects? We compared the code metric distributions of six different industry projects.

From our observations, we can conclude that **code that underwent Extract Method and code that did not undergo Extract Method does**

differ between individual industry projects. Observation 7 illustrates that this is most prominent in class-level metrics as opposed to method-level metrics. Observation 6 shows us that project #1 has a different distribution for class-complexity metrics when compared to other projects. We show in observation 5 that there seem to be two groups of projects that are similar to each other. Observation 8 shows us that, for many projects, class coupling varies widely for classes that do not contain methods that underwent an Extract Method refactoring. We also see from observation 9 that project #2 has a lower and a smaller distribution for TCC for both classes that contain methods that underwent an Extract Method refactoring and classes that do not contain such methods.

How effective are supervised machine learning models at predicting Extract Method refactoring opportunities for industry code? We trained and validated six different types of models on industry code for the purpose of predicting Extract Method opportunities and subsequently measured their performance.

We conclude that **the approach is effective in predicting refactoring opportunities for industry code with models trained on industry code.** From observation 10 we see that all types of model, accuracy, f1, precision are high. The Random Forest type model performs the best out of all types as illustrated by observation 11. Observation 12 explains that this type of model is very stable and does not depend on any single feature for its high level of performance. Observation 14 illustrates that UW and CBO, in particular, are most important to four out of five types of models, including the best-performing Random Forest type. Lastly, we explain in observation 13 that all metrics that are most important to any type of model are class-level rather than method-level.

How well do Extract Method recommendation models generalize? We repeated the experiment but instead of training on industry code, we trained on open-source code and validated on industry code.

We conclude that **open-source-trained models generalize reasonably well to an industry setting but perform worse than industry-trained models.** We show in observation 15 that linear models (LR and SVM) have the highest performance on industry code but still not nearly as high as the best-performing industry-trained models. We show in observation 16 that these models seem to be better at classifying instances where Extract Method was not applied as opposed to classifying instances of where Extract Method was applied. Feature-wise, many of the features that are important to industry-trained models, such as CBO and UW, are also important to open-source models as shown in observation 17. Observation 18 illustrates that for open-source trained models, some method-level metrics are also important as opposed to industry-trained models, where

only class-level metrics are.

How well do Extract Method models generalize across different industrial projects? We trained models where we excluded one project at a time out of the training set and then validated on that project. We find that **models generalize reasonably on most unseen industry projects.** Observation 19 shows that these models, on average, have much lower performance rates as models trained on the whole dataset and slightly lower performance than the ones trained on open-source data except for on instances where Extract Method was applied. We see from observation 20 that for all but one project there exists a type of model that performs well on that project. Also, in observation 21 we note that models perform poorly when predicting instances where no Extract Method was applied on the project that differs in class complexity from the other projects. Finally, observation 22 illustrates that different types of models do well on different projects and there is not a single type that does well on all projects.

Do industry experts deem recommended Extract Method refactorings useful/not useful? We surveyed with 30 predictions generated by our best-performing model and asked industry experts to evaluate whether these methods should be refactored or not.

We conclude that **industry experts do find predictions generated by supervised machine learning models accurate.** Observation 23 shows that, although not as accurate as in theory, developers agree with the model more often than not. In observation 24, we observe a lower rate of accuracy in cases where an Extract Method refactoring should be applied vs cases where an Extract Method should not be applied. Lastly, as shown observation 25, we see that experts who work with the projects that the model was trained on, tend to agree with the model's prediction to a higher extent than experts who do not work with these projects.

Why do industry experts deem recommended Extract Method refactorings useful/not useful? We asked experts to explain their reasoning behind their choice to apply or not apply Extract Method to methods. The following are the situations and their reasons of industry experts regarding Extract Method predictions: The most brought up characteristic when experts agree with the model's prediction to apply Extract Method is high code complexity followed by a design pattern/code smell as seen in observation 26. We see in observation 27 that when experts agree with the model's decision to not apply Extract Method, they most often mention that the code is specific enough. The second most cited reason is that the code is understandable. We see from observations 28 and 29 that experts give the same reasons for their choices in cases they disagree with the model's predictions.

8.2 Future work

This section elaborates on future work in this field of research.

Binary classification In this thesis, we class the problem of predicting Extract Method as binary classification. We see advantages to model the problem as a multi-class problem instead. One approach would be to introduce one output class for each type of refactoring and one additional output class for non-refactored cases. This could give more accuracy in classifying refactored instances, as not only one type of refactoring is considered. This would also be more accurate to how the current data collection tool classifies non-refactored instances, as it examines whether any type of refactoring was not found in a commit, rather than a specific type.

Choice of input and classification algorithms Our choice of algorithms is based on the choices made in Aniche et al.'s original paper. We also only use quantitative input for our training data. We see, however, that text-based metrics, like the number of unique words, often appear as important features for a model's performance. Further research is needed into the use of natural language for predicting refactoring opportunities.

Relationship between feature distributions and models In our work, we see strong indications that the described models are better at predicting Extract Method if metric distributions are similar to the training set. For example, increasing the amount of test data, as we did when training with the open-source set, does not increase accuracy the same way as when training on such similar code does.

We see this by the fact that the performance of industry-trained models is much higher than that of open-source models as described in Sections 4.2 and 4.3. We see from Section 3.2 that industry code does differ from open-source code which supports this claim. Another indicator of this phenomenon is observed from the experiment described in Section 4.4. Here we see lower performance on project #1, which is the same project that is identified as having a very different feature distribution from other projects in Section 3.3.

We do see that for predicting instances where no Extract Method is applied, increasing the number of samples does increase accuracy, as the open-source trained model performs very well for these types of instances. This is supported by the fact that in Section 4.4, where the amount of examples is not as high, models perform poorly such instances originating from project #1.

The need for complex models In the user study, we find that for many questions, experts do not explain why they find an Extract Method refactor-

ing is necessary or not. In these cases, especially when the participant finds that an Extract Method is necessary, the participant elaborates on how they would execute a refactor on the method shown. When a method is marked as not to be refactored with Extract Method, experts often just mention the method is “understandable” without mentioning the reasons why it is understandable. We find that over a third of all answers (54/150) consists of these cases. It seems that participants often do not consciously think about why a method should be extracted, but rather have a more complex, unconscious decision-making process.

This could indicate that complex models, like the ones explored in this report, are necessary for recommending Extract Method, as developers cannot put their reasoning into words easily and, in many cases, rely on intuition and experience.

Bibliography

[1] Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ocinneide, Ali Ouni, and Yuanfang Cai. An Interactive and Dynamic Search-Based Approach to Software Refactoring Recommendations. *IEEE Transactions on Software Engineering*, 46(9):932–961, sep 2020. ISSN 19393520. doi: 10.1109/TSE.2018.2872711.

[2] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Onward! 2019 - Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, co-located with SPLASH 2019*, volume 11, pages 143–153, New York, NY, USA, oct 2019. Association for Computing Machinery, Inc. ISBN 9781450369954. doi: 10.1145/3359591.3359735. URL https://dl.acm.org/doi/10.1145/3359591.3359735.

[3] Erik Ammerlaan, Wim Veninga, and Andy Zaidman. Old habits die hard: Why refactoring for understandability does not give immediate benefits. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, pages 504–507. Institute of Electrical and Electronics Engineers Inc., 2015. ISBN 9781479984695. doi: 10.1109/SANER.2015.7081865.

[4] M Apa) Aniche, E ; Maziero, R ; Durelli, and V ; Durelli. The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring. *IEEE Transactions on Software Engineering Citation*, 2020. doi: 10.1109/TSE.2020.3021736. URL https://doi.org/10.1109/TSE.2020.3021736.

[5] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. Automatic software refactoring: a systematic literature review, jun 2020. ISSN 15731367. URL https://doi.org/10.1007/s11219-019-09477-y.

[6] G Bavota, B De Carluccio, A De Lucia, M Di Penta, R Oliveto, and O Strollo. When Does a Refactoring Induce Bugs? An Empirical Study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 104–113, 2012. doi: 10.1109/SCAM.2012.20.

[7] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, oct 2001. ISSN 08856125. doi: 10.1023/A:1010933404324. URL https://link-springer-com.tudelft.idm.oclc.org/article/10.1023/A:1010933404324.

[8] Jehad Al Dallal. Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and Software Technology*, 58:231–249, 2015. ISSN 09505849. doi: 10.1016/j.infsof.2014.08.002.

[9] Marios Fokaefs, Nikolaos Tsantalis, Alexander Chatzigeorgiou, and Jörg Sander. Decomposing object-oriented class modules using an agglomerative clustering technique. In *IEEE International Conference on Software Maintenance, ICSM*, pages 93–101, 2009. ISBN 9781424448289. doi: 10.1109/ICSM.2009.5306332.

[10] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. JDeodorant: Identification and application of extract class refactorings. In *Proceedings - International Conference on Software Engineering*, pages 1037–1039, 2011. ISBN 9781450304450. doi: 10.1145/1985793.1985989.

[11] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. The Journal of Systems and Software Identification and application of Extract Class refactorings in object-oriented systems. *The Journal of Systems and Software*, 85:2241–2260, 2012. doi: 10.1016/j.jss.2012.04.013. URL http://dx.doi.org/10.1016/j.jss.2012.04.013.

[12] Francesca 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, jun 2016. ISSN 15737616. doi: 10.1007/s10664-015-9378-4. URL https://link-springer-com.tudelft.idm.oclc.org/article/10.1007/s10664-015-9378-4.

[13] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V. Mäntylä. Code smell detection: Towards a machine learning-based approach. In *IEEE International Conference on Software Maintenance, ICSM*, pages 396–399, 2013. doi: 10.1109/ICSM.2013.56.

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

[15] M. Gatrell and S. Counsell. The effect of refactoring on change and fault-proneness in commercial C# software. *Science of Computer Programming*, 102:44–56, may 2015. ISSN 01676423. doi: 10.1016/j.scico.2014.12.002.

[16] Jan Gerling. Machine Learning for Software Engineering: a large-scale empirical study. Master's thesis, Delft University of Technology, 2020. URL http://resolver.tudelft.nl/uuid:bf649e9c-9d53-4e8c-a91b-f0a6b6aab733.

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

[18] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, dec 2001. ISSN 09505849. doi: 10.1016/S0950-5849(01)00189-6.

[19] Mark Harman and Laurence Tratt. Pareto optimal search based refactoring at the design level. In *Proceedings of GECCO 2007: Genetic and Evolutionary Computation Conference*, pages 1106–1113, New York, New York, USA, 2007. ACM Press. ISBN 1595936971. doi: 10.1145/1276958.1277176. URL http://portal.acm.org/citation.cfm?doid=1276958.1277176.

[20] Haibo He and Edwardo A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, sep 2009. ISSN 10414347. doi: 10.1109/TKDE.2008.239.

[21] Miryung Kim, Thomas Zimmermann, Nachiappan Nagappan, Nachi Nagappan, and Tom Zimmermann. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering*, 40(7), 2014. URL https://www.microsoft.com/en-us/research/publication/an-empirical-study-of-refactoring-challenges-and-benefits-at-microsoft/.

[22] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610, sep 2020. ISSN 01641212. doi: 10.1016/j.jss.2020.110610.

[23] R Leitch and E Stroulia. Assessing the maintainability benefits of design restructuring using dependency analysis. In *Proceedings - International Software Metrics Symposium*, volume 2003-Janua, pages

309–322. IEEE Computer Society, 2003. ISBN 0769519873. doi: 10.1109/METRIC.2003.1232477.

[24] Hui Liu, Zhifeng Xu, and Yanzhen Zou. Deep learning based feature envy detection. In *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 385–396. Association for Computing Machinery, Inc, sep 2018. ISBN 9781450359375. doi: 10.1145/3238147.3238166.

[25] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *IEEE International Conference on Software Maintenance, ICSM*, pages 350–359, 2004. doi: 10.1109/ICSM.2004.1357820.

[26] T Mens and T Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004. doi: 10.1109/TSE.2004.1265817.

[27] Naouel Moha, Yann Gaël Guéhéneuc, Laurence Duchien, and Anne Françoise Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010. ISSN 00985589. doi: 10.1109/TSE.2009.50.

[28] Mark O'Keeffe and Mel Ó Cinnéide. Search-based refactoring: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(5):n/a–n/a, sep 2008. ISSN 1532060X. doi: 10.1002/smr.378. URL http://doi.wiley.com/10.1002/smr.378.

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

[30] Tushar Sharma, Girish Suryanarayana, and Ganesh Samarthyam. Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective. *IEEE Software*, 32(6):44–51, nov 2015. ISSN 07407459. doi: 10.1109/MS.2015.105.

[31] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. Recommending automated Extract Method refactorings. In *22nd International Conference on Program Comprehension, ICPC 2014 - Proceedings*, pages 146–156, New York, New York, USA, jun 2014. Association for Computing Machinery, Inc. ISBN 9781450328791. doi: 10.1145/2597008.2597141. URL http://dl.acm.org/citation.cfm?doid=2597008.2597141.

[32] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of Extract Method refactoring opportunities. In *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 119–128, 2009. ISBN 9780769535890. doi: 10.1109/CSMR.2009.23.

[33] 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, oct 2011. ISSN 01641212. doi: 10.1016/j.jss.2011.05.016.

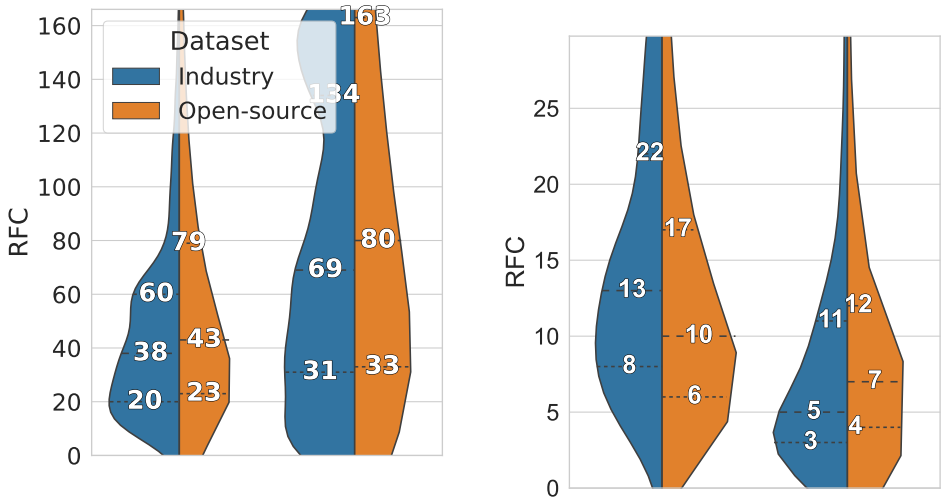
[34] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering*, pages 483–494. ACM, 2018. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180206. URL `http://doi.acm.org/10.1145/3180155.3180206`.

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

[36] Ruru Yue, Zhe Gao, Na Meng, Yingfei Xiong, Xiaoyin Wang, and J. David Morgenthaler. Automatic clone recommendation for refactoring based on the present and the past. In *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, pages 115–126. Institute of Electrical and Electronics Engineers Inc., nov 2018. ISBN 9781538678701. doi: 10.1109/ICSME.2018.00021.

Appendix A

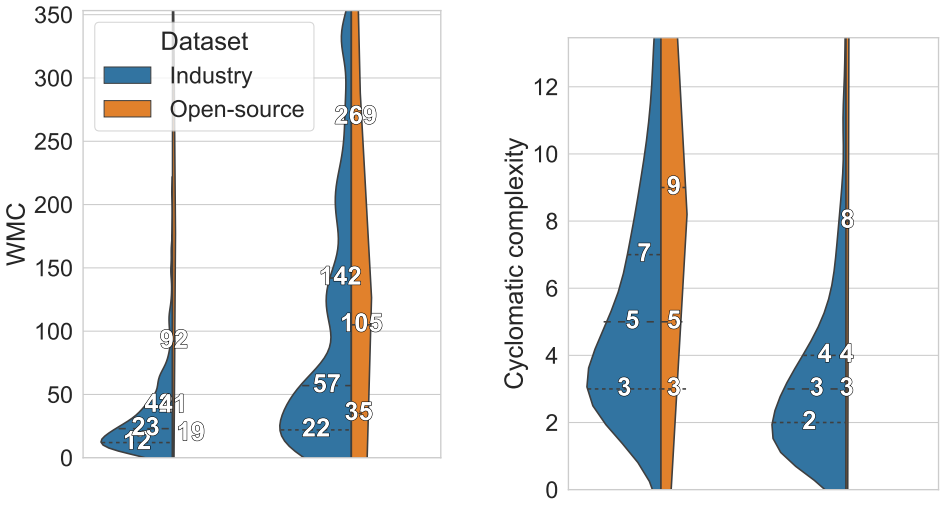
Violin Plots



(a) Class level: The left violin plot indicates classes that contain methods that underwent an Extract Method refactoring. The right violin plot indicates classes that do not contain methods that underwent an Extract Method refactoring.

(b) Method level: The left violin plot indicates methods that underwent an Extract Method refactoring. The right violin plot indicates methods that did not undergo an Extract Method refactoring.

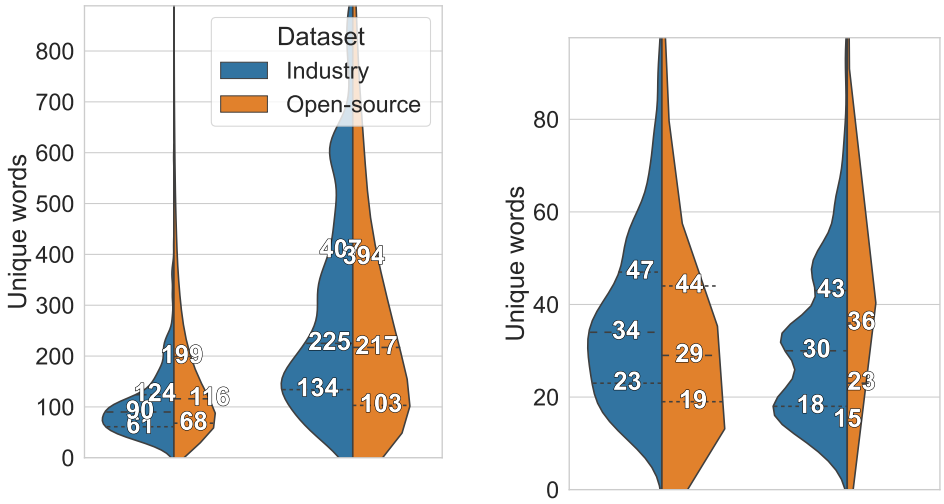
Figure A.1: RFC distributions for open-source and industry code on both class and method-level.



(a) Class level: The left violin plot indicates classes that contain methods that underwent an Extract Method refactoring. The right violin plot indicates classes that do not contain methods that underwent an Extract Method refactoring.

(b) Method level: The left violin plot indicates methods that underwent an Extract Method refactoring. The right violin plot indicates methods that did not undergo an Extract Method refactoring.

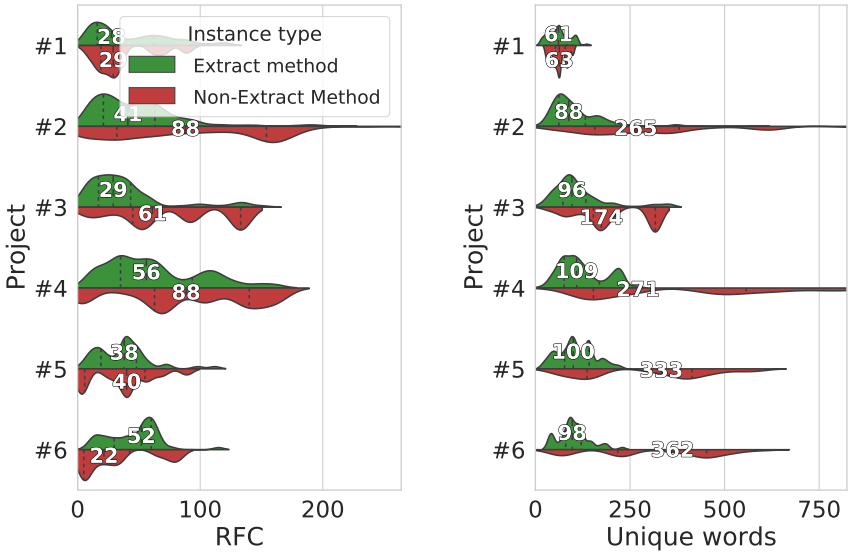
Figure A.2: Cyclomatic complexity distributions for open-source and industry code. On class-level weighted cyclomatic complexity is displayed.



(a) Class level: The left violin plot indicates classes that contain methods that underwent an Extract Method refactoring. The right violin plot indicates classes that do not contain methods that underwent an Extract Method refactoring.

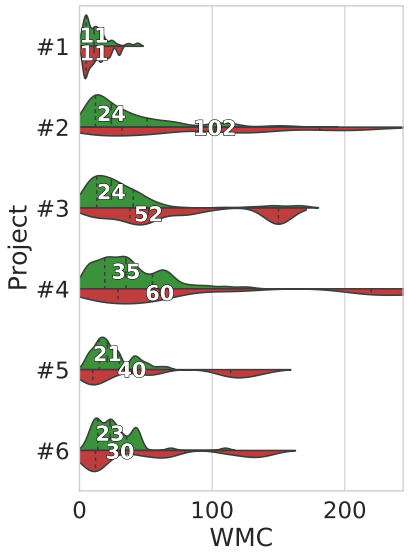
(b) Method level: The left violin plot indicates methods that underwent an Extract Method refactoring. The right violin plot indicates methods that did not undergo an Extract Method refactoring.

Figure A.3: Quantity of unique words distributions for open-source and industry code on both class and method-level.



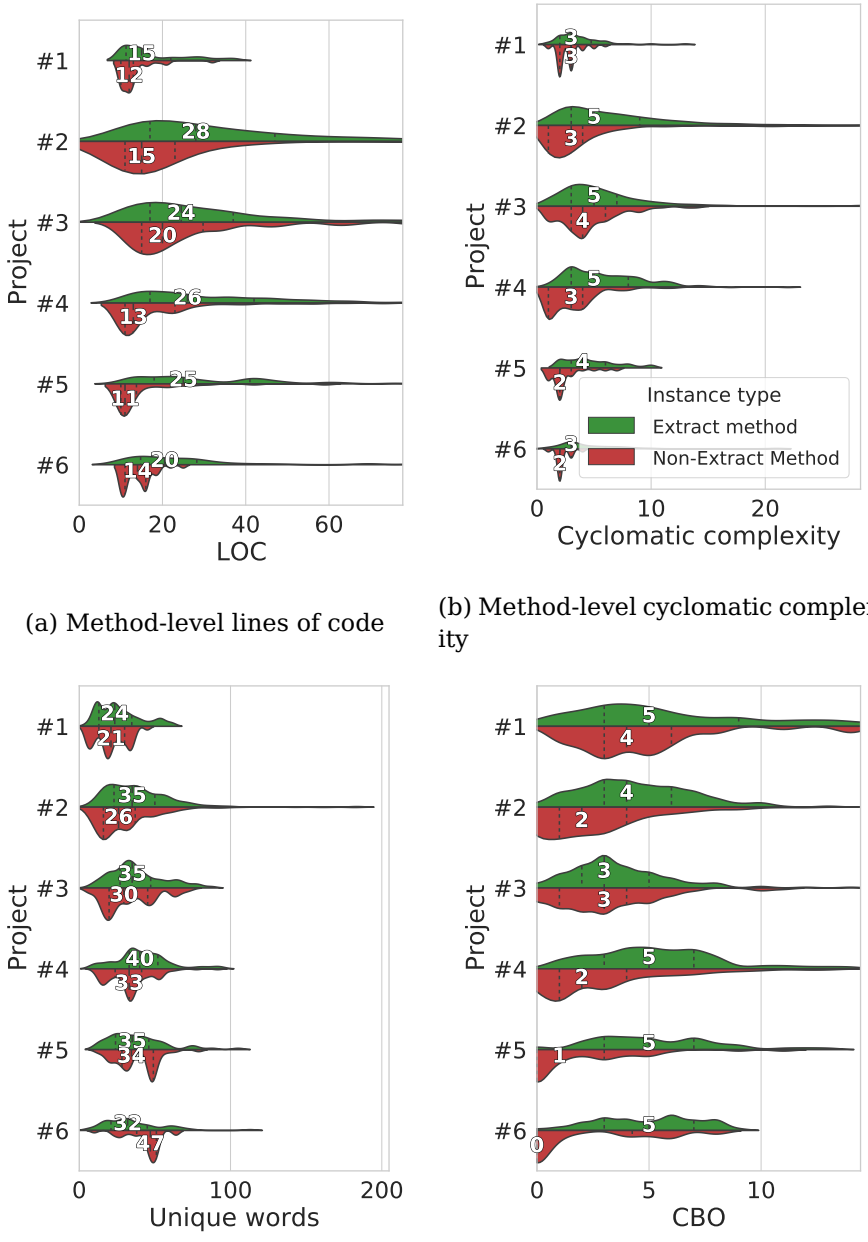
(a) Class-level RFC

(b) Class-level unique word quantity



(c) Weighted cyclomatic complexity

Figure A.4: Class-level metrics per project. The y-axis indicates the project while the x-axis shows the distribution for the metric in question. The top violin indicates the distribution for code that underwent an Extract Method refactoring while the lower violin indicates the distribution for code that did not undergo an Extract Method refactoring. The number indicates the median for the metric in question



(a) Method-level lines of code

(b) Method-level cyclomatic complexity

(c) Method-level unique word quantity

(d) Method-level coupling between objects

Figure A.5: Method-level metrics per industry project. The y-axis indicates the project while the x-axis shows the distribution for the metric in question. The top violin indicates the distribution for code that underwent an Extract Method refactoring while the lower violin indicates the distribution for code that did not undergo an Extract Method refactoring. The number indicates the median for the metric in question

Appendix B

Features Used

Explanations on features can be found in the CK documentation[1].

B.1 Class level metrics

- AnonymousClassesQty
- 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

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

B.2 Method level

- AnonymousClassesQty
- AssignmentsQty
- Cbo
- ComparisonsQty
- LambdasQty
- Loc
- LoopQty
- MathOperationsQty
- MaxNestedBlocks
- NumbersQty
- ParametersQty
- ParenthesizedExpsQty
- ReturnQty
- Rfc
- StringLiteralsQty
- SubClassesQty
- TryCatchQty
- UniqueWordsQty
- VariablesQty
- Wmc

Appendix C

Feature Importances and Linear Coefficients

All features are class-level except when prefixed with *method*.

C.1 Industry trained models

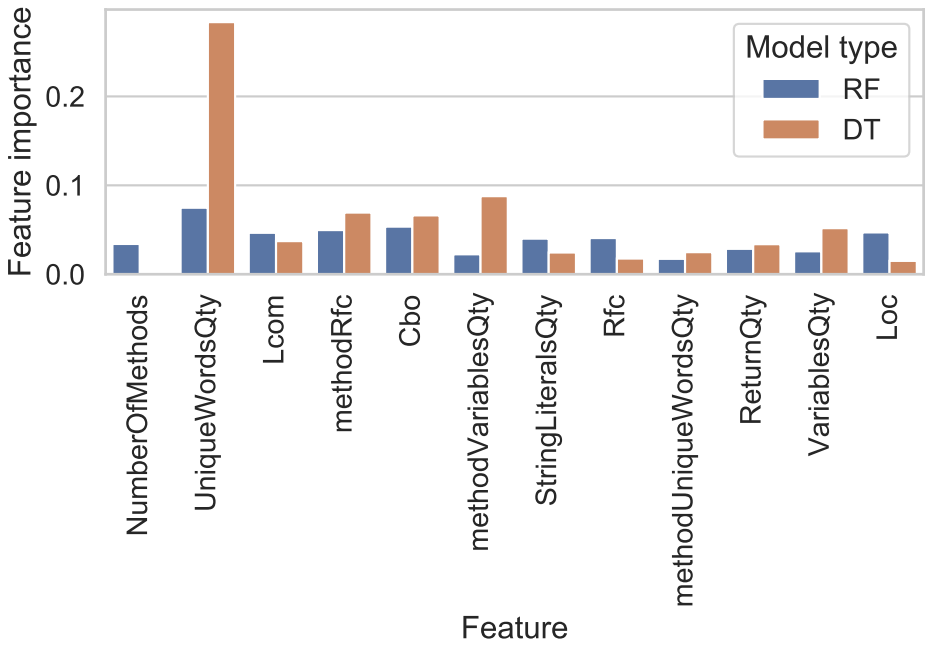


Figure C.1: Feature importance for models trained on industry data

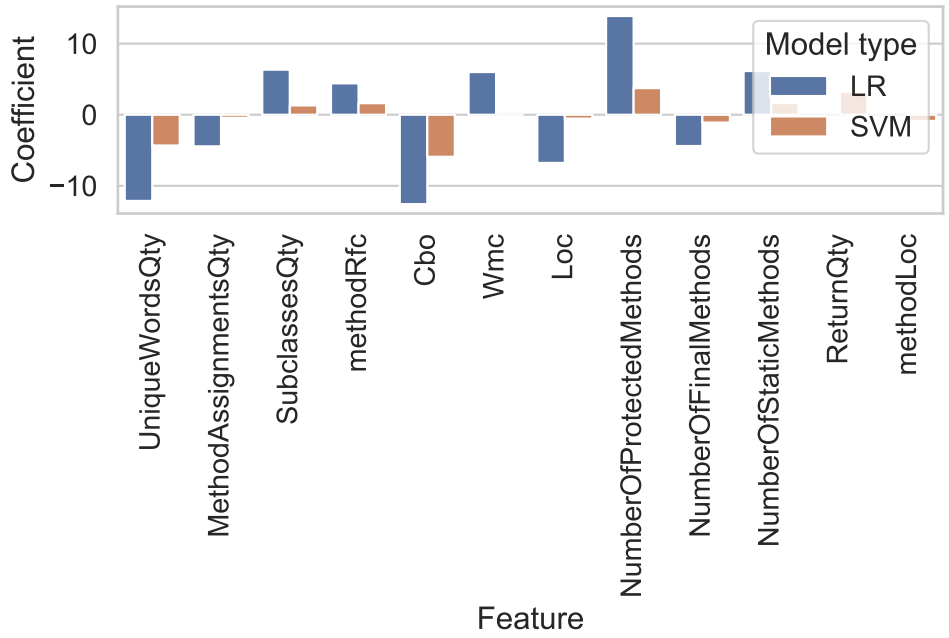


Figure C.2: Coefficients for models trained on industry data

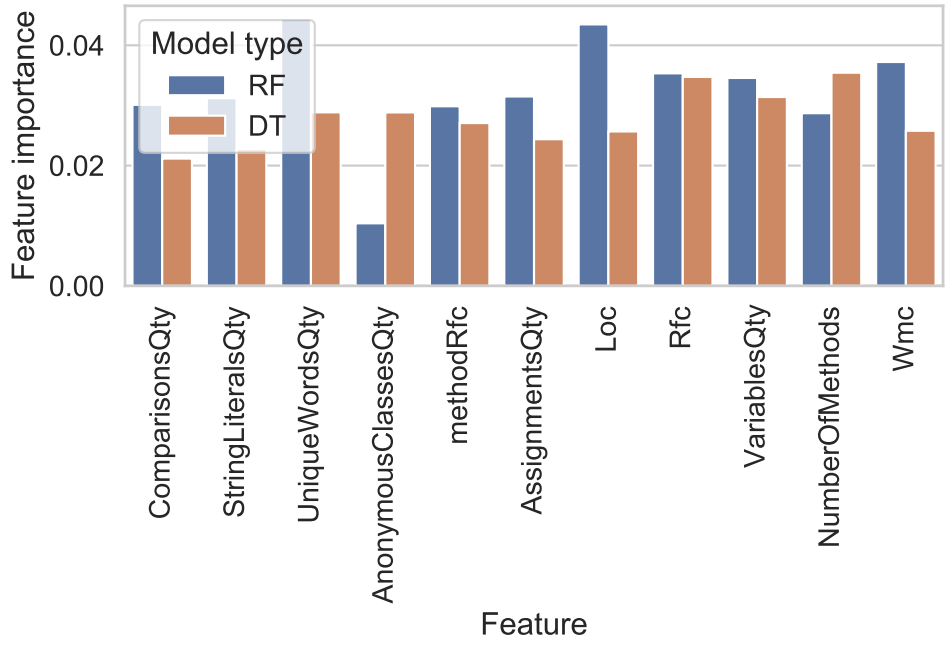


Figure C.3: Feature importance for models trained on open-source data

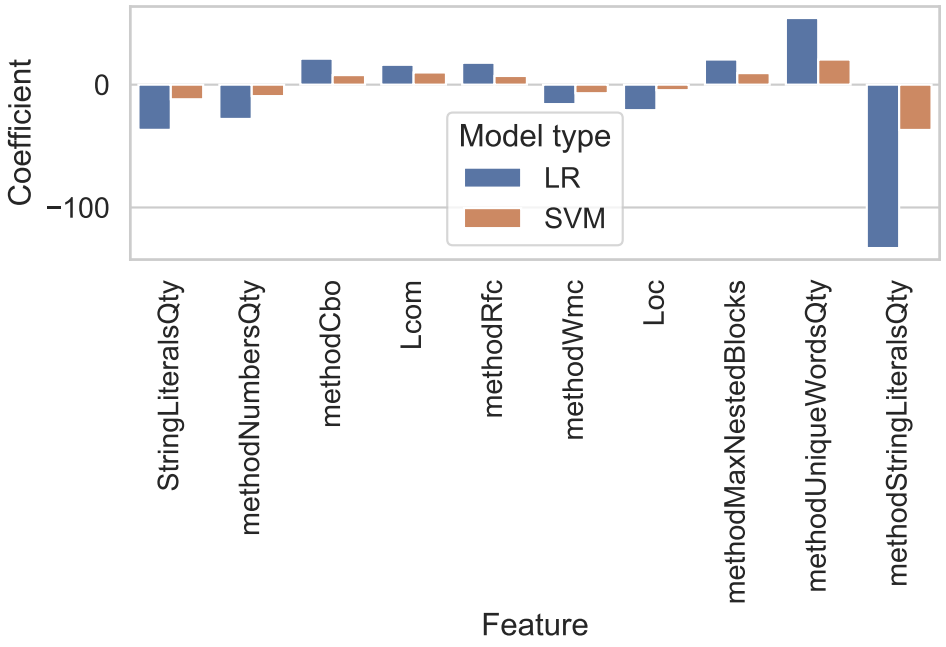


Figure C.4: Coefficients for models trained on open-source data

Appendix D

Hyperparameters Best-performing Models

Model type	RF
bootstrap	False
ccp_alpha	0.0
criterion	gini
max_depth	24
max_features	log2
min_impurity_decrease	0.0
min_samples_leaf	1
min_samples_split	2
min_weight_fraction_leaf	0.0
n_estimators	100

Model type	DT
ccp_alpha	0.0
criterion	entropy
max_depth	12
min_impurity_decrease	0.0
min_samples_leaf	1
min_samples_split	2
min_weight_fraction_leaf	0.0
splitter	best

Model type	LR
C	81.796
dual	False
fit_intercept	True
intercept_scaling	1
max_iter	100
multi_class	auto
penalty	l2
solver	saga
tol	0.0

Model type	SVM
C	0.718
dual	False
fit_intercept	True
intercept_scaling	1
loss	squared_hinge
max_iter	1000
multi_class	ovr
penalty	l1
tol	0.0

Model type	NB
var_smoothing	0.000

Table D.1: Parameters for the best-performing industry trained models

Model type	RF
bootstrap	False
ccp_alpha	0.0
criterion	gini
max_features	log2
min_impurity_decrease	0.0
min_samples_leaf	1
min_samples_split	2
min_weight_fraction_leaf	0.0
n_estimators	200

Model type	DT
ccp_alpha	0.0
criterion	entropy
max_features	log2
min_impurity_decrease	0.0
min_samples_leaf	1
min_samples_split	2
min_weight_fraction_leaf	0.0
splitter	random

Model type	LR
C	77.681
dual	False
fit_intercept	True
intercept_scaling	1
max_iter	500
multi_class	auto
penalty	l2
solver	saga
tol	0.0

Model type	SVM
C	2.504
dual	False
fit_intercept	True
intercept_scaling	1
loss	squared_hinge
max_iter	1000
multi_class	ovr
penalty	l2
tol	0.0

Model type	NB
var_smoothing	0.000

Table D.2: Parameters for the best-performing open-source trained models

Appendix E

Confusion Matrices

Model type	TP	TN	FP	FN
RF	179 (46%)	177 (46%)	20 (5%)	5 (1%)
DT	153 (40%)	171 (44%)	26 (6%)	31 (8%)
LR	158 (41%)	156 (40%)	41 (10%)	26 (6%)
SVM	161 (42%)	155 (40%)	42 (11%)	23 (6%)
NB	165 (43%)	133 (34%)	64 (16%)	19 (4%)

Table E.1: Confusion matrix for models trained and validated on industry code

Model type	TP	TN	FP	FN
RF	885 (46%)	423 (22%)	563 (29%)	34 (1%)
DT	743 (39%)	411 (21%)	575 (30%)	176 (9%)
LR	880 (46%)	569 (29%)	417 (21%)	39 (2%)
SVM	885 (46%)	561 (29%)	425 (22%)	34 (1%)
NB	914 (47%)	255 (13%)	731 (38%)	5 (0%)

Table E.2: Confusion matrix for models trained on open-source code and validated on industry code

Appendix F

User study and recommendation examples

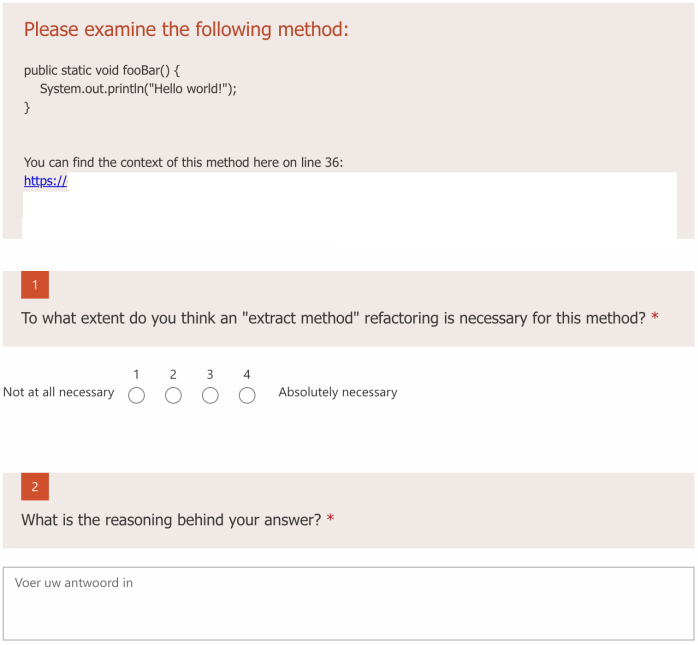


Figure F.1: Example of how a question would look like in the survey.

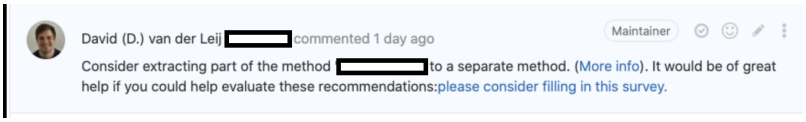


Figure F.2: Example of a refactoring recommendation on a merge made by our model. We include a link to give feedback and a link to an explanation in case the developer is unfamiliar with the refactor in question

Appendix G

User study — Agreement Metrics

G.1 Metric definitions

- $Agreement = \frac{RA+NA}{RA+NA+RD+ND}$
- $Agreement_{EM} = \frac{RA}{RA+RD}$
- $Agreement_{NR} = \frac{RA}{RA+ND}$
- $Agreement_{f1} = 2 \cdot \frac{Agreement_{EM} \cdot Agreement_{NR}}{Agreement_{EM}+Agreement_{NR}}$

G.2 Metric values

Expert	R_A	N_A	N_D	R_D
1	13	10	0	7
2	14	7	3	6
3	10	7	3	10
4	11	9	1	9
5	17	10	0	3
All/Mean	65	43	7	35

Table G.1: Agreement of the experts with the model's predictions.