Data-Driven Extract Method Recommendations: A Study at ING

van der Leij, David; Binda, J.R.; van Dalen, Robbert; Vallen, Pieter; Luo, Yaping; Aniche, Maurício

DOI
10.1145/3468264.3473927

Publication date
2021

Document Version
Final published version

Published in
ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering

Data-Driven Extract Method Recommendations: A Study at ING

David van der Leij
davidvanderleij@gmail.com
Delft University of Technology, ING
The Netherlands

Jasper Binda
Jasper.Binda@ing.com
ING
The Netherlands

Robbert van Dalen
Robbert.van.Dalen@ing.com
ING
The Netherlands

Pieter Vallen
Pieter.Vallen@ing.com
ING
The Netherlands

Yaping Luo
Yaping.Luo@ing.com
ING, Eindhoven University of Technology
The Netherlands

Maurício Aniche
M.F.Aniche@tudelft.nl
Delft University of Technology
The Netherlands

ABSTRACT

The sound identification of refactoring opportunities is still an open problem in software engineering. Recent studies have shown the effectiveness of machine learning models in recommending methods that should undergo different refactoring operations. In this work, we experiment with such approaches to identify methods that should undergo an Extract Method refactoring, in the context of ING, a large financial organization. More specifically, we (i) compare the code metrics distributions, which are used as features by the models, between open-source and ING systems, (ii) measure the accuracy of different machine learning models in recommending Extract Method refactorings, (iii) compare the recommendations given by the models with the opinions of ING experts. Our results show that the feature distributions of ING systems and open-source systems are somewhat different, that machine learning models can recommend Extract Method refactorings with high accuracy, and that experts tend to agree with most of the recommendations of the model.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**;

KEYWORDS

Software Engineering, Software Refactoring, Machine Learning for Software Engineering.

ACM Reference Format:
David van der Leij, Jasper Binda, Robbert van Dalen, Pieter Vallen, Yaping Luo, and Maurício Aniche. 2021. Data-Driven Extract Method Recommendations: A Study at ING. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3468264.3473927

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3473927

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 code degradation. As a project evolves, the scope and complexity increase and the original design decisions tend to fade. Refactoring, as defined by Fowler, is the process of changing a software system in such a way that does not alter the external behaviour of the code yet improves its internal structure. This process has been shown to improve code quality [10, 16, 17], and its benefits are perceived by developers [16].

Nonetheless, software refactoring does not come without its challenges. Any refactoring operation incurs costs [16], and its benefits may not be immediately clear [3]. It is also challenging to identify sound refactoring opportunities, i.e., pieces of code that would undoubtedly benefit from refactoring. A wide variety of approaches have been proposed to tackle these challenges, such as rule-based approaches, where refactoring opportunities are detected through static rules (e.g., [6, 25, 26]), search-based approaches, where the problem is modeled as a search problem (e.g., [1, 14, 21]) and, finally, machine-learning approaches, where models learn to predict future refactorings (e.g., [4]). In particular, Aniche et al. [4] experimented with six different machine learning algorithms. Authors modelled the problem as binary classification, and used code metrics as features to predict whether a piece of code should be refactored. When using a Random Forest model, with open-source systems as training and test data, authors achieved a precision rate of over 90%.

Interested in such results, we set out our goal to replicate the study of Aniche et al. and explore the effectiveness of their approach within ING[1], a large financial organization. We set our initial goal to explore recommendations of the Extract Method refactoring, which ING software experts deem to be an important refactoring operation and believe that such recommendations would improve the state of their code base.

Our work can be divided into three parts:

- First, we explore the differences in the distributions of the code metrics, which are used as features of the model, between open-source and ING systems (Section 3 of this paper). This goal of this initial parts is to shed some light on how different ING code base is from open source, and whether

[1] https://www.ing.com

we should expect a model that is trained on top of thousands of open-source systems to work well at ING.
- Second, we explore the effectiveness of machine learning models in predicting refactoring operations that have happened at ING (Section 4 of this paper). We explore different machine learning models, the feature importances of the best performing models, and how the models behave when trained and tested on different sets of internal projects.
- Third, we collect the perceptions of five ING experts on the recommendations that are provided by the models by means of a user study (Section 5 of this paper). More specifically, we show a set of recommendations of the model (of methods that should and should not undergo an Extract Method) and ask the opinion of experts who do not know what was the recommendation made by the model.

Our results show that (i) the feature distributions of ING systems and open-source systems are somewhat different, (ii) that machine learning models can recommend Extract Method refactorings with high accuracy, that different models may work better in different systems, and (iii) that experts tend to agree with many of the recommendations of the model.

2 RELATED WORK

The software engineering community has been studying software refactoring for a long time and from many different angles. When it comes to identify and suggest refactorings, we observe mainly three techniques in the field: (i) rule-based approaches, (ii) search-based approaches, and (iii) machine learning-based approaches.

Rule-based approaches. Such methods apply rules on code metrics, or other aspects of code, to detect code smells or refactoring opportunities. For example, Marinescu proposes a metric-based code smell detection technique [19]. Authors use logic rules in combination with code quality metrics to detect code smells. They report an average accuracy rate of 67% when analyzing nine different types of smells.

Silva et al. [24] use a similarity-based approach to detect Extract Method refactoring opportunities. Authors 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. [20]. The authors 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 [25] 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.

Search-based approaches. Such approaches model software engineering problems as search-based problems and solve them using search techniques [13]. Harman and Tratt [14] proposed a search-based method that applies to Java code. Their approach is fit for general purpose and allows for multiple fitness functions to present different Pareto optimal metrics.

O'Keeffe and Cinnéide [21] describe CODe-Imp, a search-based approach that allows for automatic improvement of code maintainability. Authors test four different algorithms in conjunction with their tool and find that multiple-ascent hill climbing is the best-performing algorithm.

Schröder et al. [23] explores the effectiveness of search-based approaches in improving the modularization of a large-scale system. Authors show that the approach is able to find important Move Class refactorings that reduce coupling and increase the cohesion of the industry partner's codebase. Moreover, the developers of the system under study agreed that such classes were in the wrong modules, although they disagreed with the model's suggestion on the new module to move the class to.

More recently, Alizadeh et al. [1] propose an interactive method that defines an offline and an online phase. 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.

Machine learning-based approaches. Such approaches leverage machine learning algorithms that learn how to predict refactoring operations.

Fontana et al. [8] 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. 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. [7] conducted a study comparing several machine learning techniques for code smells. The authors 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. [18] shows the application of deep learning to detect the Feature Envy code smell. As input, the approach uses code metrics and code transformed into vectors. Then, based on the outcome, the approach predicts the destination of a Move Method refactoring operation. The paper reports 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. [30] combine static analysis and machine learning to recommend Extract Method to software clones. They report an average f1 score of 83% when testing within projects. An average f1 cross-project score of 76% is reported.

Finally, Aniche et al. [4] experiments with different machine learning algorithms to recommend different types of refactoring, with Extract Method being one of them. The approach relies on different code, process, and ownership metrics of methods that underwent an Extract Method and methods that did not need an Extract Method. Results show that such models can learn and predict Extract Methods with high accuracy.

3 AN EMPIRICAL STUDY OF EXTRACT METHOD REFACTORINGS AT ING

The goal of this section is to empirically understand the characteristics of methods that underwent an Extract Method refactoring, and methods that did not need such a refactoring. Moreover, we also compare the distributions of features between ING's code and open-source systems (used in the related work), as a way to better understand their similarities and differences. To that aim, we answer the following research question:

RQ1 How does code that underwent an Extract Method refactoring in ING and open-source systems compare in terms of code metrics?

3.1 Methodology

Similar to the work of Aniche et al. [4], we make use of the Git history of the ING projects to build a dataset consisting of code that underwent an Extract Method refactoring and code that did not need an Extract Method refactoring.

We identify instances of methods that underwent an Extract Method via RefactoringMiner version 2.0 [27, 28]. RefactoringMiner is a refactoring detection tool that analyses Git commits and detects any refactoring operations that have occurred. Tsantalis et al. report to achieve precision and recall rates of 99.8% and 95.8% respectively for detecting an Extract Method refactorings [27, 28].

To identify methods that did not need an Extract Method, we use the heuristic proposed by Aniche et al. [4]. We classify a method as one that did not need an Extract Method if its class did not undergo any refactoring for s consecutive commits (i.e., the class was changed s times without any refactoring operation happening). Deciding the sensitive parameter s is still an open problem. If we increase s, the tool gets less sensitive about what to class as a non-refactored class as the class needs not to 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. Aniche et al. [4] reports different accuracies when picking different parameters. After some exploration, we opt for $s = 20$ for ING.

From both refactored and non-refactored methods, we collect a large set of code metrics. Similarly to Aniche et al. [4], metrics are collected before the refactoring had occurred rather than after it has been completed, as we want to investigate the method's state for when it was a candidate for refactoring.

We remove any duplicated data points that may exist in our dataset. Duplicated data is a well-known problem for machine learning models, as they can cause 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].

Given that analysing all the 61 metrics that Aniche et al. [4] use in their models is a daunting and manually-impossible task, we focused on a subset of metrics. In particular, we focus on three types of metrics considered relevant by the ING experts, complexity, coupling, and cohesion:

Table 1: Number of data points, per ING project.

Project	Underwent an Extract Method	Did not need an Extract Method
ING #1	58	37
ING #2	273	450
ING #3	152	84
ING #4	135	212
ING #5	49	46
ING #6	52	32
Others	200	125
Total	919	986

- **Complexity metrics:**
  - **Lines of code (LOC)** $[0, \infty]$ A longer class or method might indicate more complexity than a short one.
  - **Response for class (RFC)** $[0, \infty]$ 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 metric:**
  - **Coupling between objects (CBO)** $[0, \infty]$ Represents the number of connections to a respecting class/method.
- **Cohesion metrics:**
  - **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.

3.2 Datasets

We analyze and compare two datasets. The first dataset consists of open-source systems (OSS) from GitHub. This dataset was mined by Gerling [11]. The projects were sourced from GHTorrent [12]. From this dataset, Gerling selected the top 100, 000 watched projects, and after removing faulty projects, they were left with 92, 280 projects to analyze. This resulted in 616, 088 Extract Method and 503, 393 non Extract Method instances. After removing duplicates, we were left with 449, 949 Extract Method and 460, 974 non-Extract Method instances.

The second dataset consists of proprietary code from ING. This dataset initially contained 18 ING projects which were chosen with the help of experts. The data collection resulted in 2, 083 Extract Method and 1, 483 non-Extract Method instances. After removing duplicates for this dataset, we were left with 919 Extract Method and

Table 2: Differences between ING and open-source feature distributions, per metric. ↑ indicates that the ING values of the metric are higher than the open-source values, ↓ indicates that ING values are lower than open-source values, and ≈ indicates that values in ING and open-source systems are similar.

	Class-level metrics		Method-level metrics	
Metric	Re-factored	Not refactored	Re-factored	Not refactored
Complexity metrics				
LOC	↓	↓	≈	↓
RFC	↓	↓	↑	↓
WMC/CC	↓	↓	≈	↓
UW	↓	≈	↑	↑
Coupling metrics				
CBO	≈	↑	≈	≈
Cohesion metrics				
TCC	↑	↑	–	–
LCC	↑	↑	–	–

986 non-Extract Method instances. Table 1 describes the number of data points per project.

When analyzing individual projects, we only analyze projects that have at least 30 methods that underwent an Extract Method refactoring and 30 methods that did not need an Extract Method. This restriction ensures a higher level of confidence in the observations of individual projects. We identify six projects to analyze on an individual level. Note that we still keep data points of the other software systems for when analyzing all projects together.

3.3 RQ1: How Does Code That Underwent an Extract Method Refactoring in ING and Open-Source Systems Compare in Terms of Code Metrics?

We summarize our findings in Table 2. We show an example violin plot of the LOC metric in Figure 1. Due to space constraints, all the other violin plots we used to infer the observations are available only in our appendix [29]. We nevertheless report medians and interquartile ranges near all our observations.

Observation 1: Classes that contain methods that underwent an Extract Method in ING tend to be smaller and less complex than classes in open-source systems; on the other hand, methods that underwent an Extract Method, in ING and in open-source, are generally similar in terms of size and complexity. We see that ING classes that contain methods that underwent an Extract Method refactoring are shorter in terms of lines of code ($MED = 112, IQR = [60–190]$) than the open-source classes whose methods underwent the refactoring ($MED = 196, IQR = [95–419]$). Similarly, for classes that did not contain methods that underwent an Extract Method refactoring, ING classes are also shorter ($MED = 339, IQR = [180–678]$) as their open-source counterparts ($MED = 497, IQR = [175–1202]$).

In terms of complexity, we observe that the WMC of classes that contain methods that underwent an Extract Method refactoring in the ING systems is almost half that of its open-source counterpart (($MED = 23, IQR = [12–42]$) vs ($MED = 41, IQR = [19–92]$)). We see the same pattern in the class-level metrics of methods that did not need an Extract Method refactoring, where the WMC in ING systems ($MED = 57, IQR = [22–142]$) is again almost half that of open-source ($MED = 105, IQR = [35–269]$). We also note that WMC in the open-source classes is much more widely spread than in ING's classes.

At method-level, differences between methods that underwent an Extract Method in ING and in open-source system are generally less pronounced. ING methods that did not need an Extract Method refactoring ($MED = 25, IQR = [16–43]$) are similar in length to open-source methods that did not need an Extract Method refactoring ($MED = 24, IQR = [15–38]$). We also see similar cyclomatic complexity for methods that did not need an Extract Method refactoring in the ING systems ($MED = 3, IQR = [2–4]$) and open-source ($MED = 4, IQR = [3–8]$).

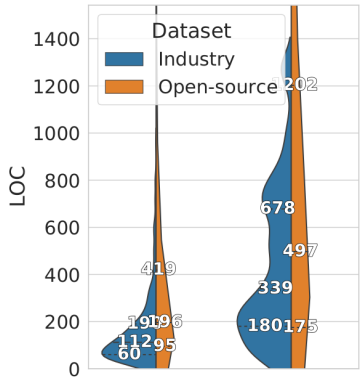
Observation 2: Coupling is generally similar among classes and methods that underwent an Extract Method in ING and in open-source. We see that, for methods that underwent an Extract Method, method-level CBO is similar between ING systems ($MED = 4, IQR = [2–6]$) and open-source systems ($MED = 4, IQR = [3–6]$). The same holds for methods that did not need an Extract Method, where the median and inter-quartile ranges are equal for both ING and open-source systems($MED = 2, IQR = [1–4]$) .

On the other hand, the class-level coupling of methods that did not need an Extract Method refactoring i much lower in open-source systems ($MED = 20, IQR = [9–41]$) than in ING systems ($MED = 29, IQR = [14–84]$). Most interestingly, however, is the upper quartile, where the open-source has a normal distance from the median, but the CBO of ING systems is significantly high in comparison.

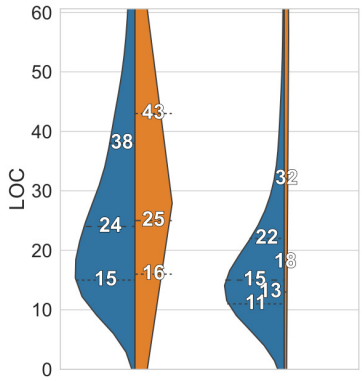
Observation 3: Cohesion is higher in ING classes. For ING classes that contain methods that underwent an Extract Method refactoring, the inter-quartile range ($MED = 0.30, IQR = [0.00–1.00]$) spans all possible values of the TCC metric. This is probably due to the high density around 1.00, as seen in 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 the LCC metric.

Observation 4: Within the six ING systems, we observe different similarities and differences. When it comes to the class-level complexity, we observe clusters of similar projects. More specifically, #2 and #4 are somewhat similar to each other, while #3, #5 and #6 are similar among each other. Interestingly, #1 is different from all others.

We also observe that class-level coupling varies widely in some projects for the class-level metrics of methods that did not need an Extract Method refactoring. More specifically, 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



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



(b) Method-level LOC: The left violin plot indicates methods that underwent an Extract Method refactoring. The right violin plot indicates methods that do not need to undergo an Extract Method refactoring.

Figure 1: LOC distributions for open-source and ING code on both class- and method-level.

some projects, such as #4 ($MED = 30, IQR = [19–320]$), are even larger.

On the other hand, at method-level, metrics are very similar between different projects. We see that for all projects, and for both types of instances, ranges 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. Our appendix [29] contains the plots for the other method-level metrics which show similar patterns.

Key takeaway: The differences in metrics between ING and open source systems tend to happen more significantly at class-level, and are less pronounced at method-level. We, therefore, conjecture that a model trained on open-source data will have just reasonable performance.

4 THE EFFECTIVENESS OF MACHINE LEARNING MODELS IN RECOMMENDING EXTRACT METHOD REFACTORINGS AT ING

The goal of this section is to explore the effectiveness of machine learning in predicting Extract Method refactorings in the ING code base. Moreover, we explore whether models trained on top of open-source systems are also effective in predicting refactorings in the ING code base, and how much ING models generalise across different ING software systems.

To that aim, we answer the following research questions:

RQ2 How effective are supervised machine learning models at predicting Extract Method refactoring opportunities in ING?

RQ3 How well do models trained on open-source systems perform in predicting refactoring in ING systems?

RQ4 How well do Extract Method models generalise across different ING systems?

4.1 Methodology

We follow a similar method to the one proposed by Aniche et al. [4]. We model the problem of classifying whether an Extract Method refactoring should be applied as binary classification. We attach true labels to methods that underwent an Extract Method refactoring, and false labels to methods that did not need an Extract Method refactoring. Our feature vectors consist of the collected class- and method-level code metrics. More specifically, we collect 41 class-level metrics, and 20 method-level metrics. These metrics are extracted using the CK tool[2] and are listed in our appendix [29].

We train our models using scikit-learn [22]. We make use of the same algorithms as used in the paper of Aniche et al. [4], except neural networks. From an exploratory investigation, we found that neural networks did not achieve better performance when compared to our best-performing model. Because of the large number of hyper-parameters and the long training times, we decide not to investigate this algorithm. The algorithms used and their corresponding abbreviations are as follows: Random forest (RF), Decision Tree (DT), Logistic Regression (LR), Linear SVM (SVM), and Gaussian Naive Bayes (NB).

In a nutshell, the training pipeline consists of the following steps for each algorithm:

(1) Pre-process data:

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

(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.

In their paper, Aniche et al. [4] choose to balance their dataset such that the amount of positive samples is equal to the number of negative samples. Authors do this because, for most refactoring types, the classes are highly unbalanced, and imbalanced data can lead to problems in machine learning [15]. In our experiments, we choose to not balance our dataset because of two factors: First, 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 1. The ratio is 919 for Extract Method (48%) to 986 for non-Extract Method (52%) for industry code and 449,949 for Extract (49%) to 460,974 for non-Extract Method (51%) for open-source code. Second, 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.

4.2 Data Collection, Pre-Processing, and Balancing

We extract the data from ING systems as described in Section 3.1. In a nutshell, we visit the Git's history of the project and extract instances of methods that underwent an Extract Method and methods that did not need an Extract Method. We then apply a true label to the Extract Method instances and a false label to instances that did not need an Extract Method refactoring. We remove duplicated data points.

We use stratification during the split to ensure classes are not over-represented in the test or training set by chance. We scale all our features using a MinMaxScaler since this benefits most machine learning algorithms[3]. Finally, we apply feature reduction only for the logistic regression model.

When answering RQ2, we split the entire ING dataset, without any specific separation among projects, into train (80%) and test (20%). When analysing the performance of open-source models in industry code (RQ3), we use all industry data as test set. Finally, when analysing the performance of models in unseen ING systems (RQ4), we train the model with the data available for all projects but the one we test on, and we repeat the procedure for all six projects.

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

4.3 Training

To improve our models' performance, we optimise 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 $k = 10$. Our appendix [29] contains the hyperparameter combinations of the best-performing models. With the resulting model, we calculate the confusion matrix using the test set. The raw confusion matrices can be found in the appendix [29]. These confusion matrices are then used to calculate well-known accuracy, precision, recall, and F1 metrics.

To analyse 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 [5]. 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 each feature 50 times to increase our confidence in the performance reduction.

We do not analyse 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[4]. For reference, plots of the above coefficients and feature importances can be found in our appendix [29].

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[5] format.

4.4 RQ2: How Effective Are Supervised Machine Learning Models at Predicting Extract Method Refactoring Opportunities in ING?

We summarise the achieved performance of each model in Table 3.

Observation 5: All the different machine learning models perform relatively well. From Table 3, we see that there is no precision rate below 0.721 (Naive Bayes). For recall, the lowest rate is 0.832 for Decision Trees. Lastly, if we look at the aggregated

[4]Based on sklearn's manual: https://scikit-learn.org/stable/auto_examples/inspection/plot_permutation_importance.html
[5]https://github.com/onnx/onnx

Table 3: Performance metrics for models trained and tested on ING code.

Model	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: Performance metrics for models trained on open-source and tested on ING code.

Model	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

metrics of accuracy and F1, we see that they do not drop below 0.782 (NB) and 0.799 (NB), respectively.

Observation 6: Random Forest has the highest performance. Table 3 shows that Random Forest 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. We note that Random Forests were also the most accurate model in the work of Aniche et al. [4].

Observation 7: Class-level features are most important for model performance. For all different machine learning algorithms, the top five features that reduce performance the most when permuted are all class-level features. For the Random Forest model, for example, the top five features are quantity of unique words (UW), CBO, LCOM, LOC, and RFC.

Observation 8: Random Forests are more stable with regard to performance when permuting features in comparison with other types of models. Out of all models, when we permute features for the Random Forest, the performance drops at most 0.0139. For all the other models, the maximum drop of 0.0233 occurs for Naive Bayes and happens when permuting CBO. This amount is much much higher as the maximum drop in performance for Random Forest. We also see performance drops as high as 0.1504 for the SVM model when permuting UniqueWordsQty. This is almost 15 times as high as the maximum performance drop for the Random Forest.

4.5 RQ3: How Well Do Models Trained on Open-Source Systems Perform in Predicting Refactoring in ING Systems?

We summarise the achieved performance of each model in Table 4.

Observation 9: Open-source-trained models perform reasonably well on ING code, although not as good as models trained on ING's code.. Table 4 shows that the best-performing model (LR) achieves an accuracy and F1 score of 0.761 and 0.794 respectively. These are relatively high values, but much lower than the best-performing ING-trained model (RF), which achieves accuracy and F1 scores of 0.934 and 0.935, respectively.

Observation 10: Models trained on open-source code are much better at predicting non-Extract method instances as they are at predicting Extract Method instances in ING code. Table 4 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 ING-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 ING-trained model (RF) achieves a precision score of 0.899.

Table 5: The average of the performance metrics, when training on all ING projects but one.

Model	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

4.6 RQ4: How Well Do Extract Method Models Generalise Across Different ING Systems?

Table 5 shows the mean of the performance metrics when training on all but one ING project. We show more detailed numbers in our appendix [29].

Observation 11: Average performance is still reasonable, but much lower than when using the whole dataset, and slightly lower than when training on open-source. From Table 5, we see reasonable aggregated performance metrics, with the best model (RF) having mean accuracy and F1 rates of 0.744 and 0.771 However, we note that performance is much lower than when testing on the whole dataset (RQ2) and slightly lower than the open-source-trained model (RQ3).

Observation 12: For each project, there exists a type of model that performs well, except for project #1. For all projects, except project #1, there is always one type of model that achieves a score of 0.75 or higher. Project #1 is an exception, as the highest F1 score is only 0.66 for the RF type model. This is somewhat expected as we observed in RQ1 that Project #1 has a feature distribution completely different from all other projects.

Observation 13: Different types of models perform well on different projects. We observe that there is no 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 takeaway: Machine learning models seem to accurately predict Extract Method refactorings in ING systems (accuracy of around 94%). When models are trained on top of open-source data, the accuracy drops when compared to ING-trained models, but their performance is still high (around 76%). When trained on ING-systems and tested on unseen ING systems, performance is smaller than when trained with the entire dataset together, but still high (around 74%). Different models perform better for different ING systems.

5 THE PERCEPTIONS OF ING EXPERTS ON THE DATA-DRIVEN RECOMMENDATIONS

The goal of this section is to compare the perceptions of ING experts with the recommendations provided by the machine learning models. To that aim, we answer the following research questions:

RQ5 **Do ING experts deem recommended Extract Method refactorings useful/not useful?**
RQ6 **Why do ING experts deem recommended Extract Method refactorings useful/not useful?**

5.1 Methodology

In a nutshell, we show a selection of methods that our best-performing model recommended to (and not to) undergo an Extract Method to the expert. We then ask the expert to decide whether or not s/he would refactor that method, without knowing the prediction of the model.

All predictions served to experts in this section were generated by the best-performing model in Section 4, the Random Forest model that was trained on ING code. We choose code quality experts at ING and invite them to fill in a questionnaire. The choice was based on convenience and availability. All participants have substantial experience with programming. Two participants work with the code displayed in the survey daily, while the other three offer an outside perspective.

The survey consists of 30 questions, each displaying a method originating from ING code. The set of methods to display were randomly chosen from methods that were added or changed in the ING code base between January 20th 2021 and February 20nd 2021. The participants do not know whether the model recommended the refactoring or not.

For every method, the ING expert answers two questions. 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; a score of 1 or 2 is interpreted as a sign that the expert thinks the method should not undergo an Extract Method refactoring. The second question, an open question, asks the expert to elaborate on their choice. A concrete example of a question can be found in our appendix [29].

We settle on a total of 30 methods, as this limits the time spent on the survey by each expert and gives us a reasonable sample size. The average time for an expert to complete the survey was approximately 42 minutes. Every expert receives a survey containing the same methods. Given that 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, we select 20 methods where the model attached a true label (i.e., the model recommends the refactoring) to the method and ten where it attached a false label (i.e., the model does not recommend the refactoring).

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 RQ5: Do ING Experts Deem Recommended Extract Method Refactorings Useful/Not Useful?

We measure how often the experts agree with the model's prediction to apply Extract Method or to not 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.
- **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 agreement (N_A)** The model classifies the method as to not be refactored. The expert agrees that no Extract Method refactoring is 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:

- **Accuracy**: Agreement ratio of the experts with the model's predictions: $Acc = \frac{RA+NA}{RA+NA+RD+ND}$.
- **Precision**: Agreement ratio of experts for methods where the model predicts Extract Method: $Precision = \frac{RA}{RA+RD}$.
- **Recall**: Agreement ratio of experts for methods where the model predicts to not apply an Extract Method refactoring: $Recall = \frac{RA}{RA+ND}$.
- **F1**: Agreement ratio while taking into account imbalance of classes such as is the case in our experiment: $F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision+Recall}$.

These definitions allow for one-to-one comparison with the theoretical performance. The raw occurrences and calculations can be found in the appendix [29].

Table 6: The agreement between ING experts and the recommendation of the models.

Expert	Accuracy	F1	Precision	Recall
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

Observation 14: The opinions of ING experts align with the model's predictions to a certain extent. From Table 6, we see that the average accuracy and F1 are 0.720 and 0.756, respectively. At an individual level, we see accuracy numbers ranging from 0.567 (the smallest agreement we observed, from expert #3) up to 0.9 (the most significant agreement we observed, from expert #5).

Observation 15: Experts agree more with the model when it does not recommend the Extract Method. Table 6 shows that recall rates are much higher than precision 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 an Extract Method refactoring. The average recall is 0.903, while the average precision of 0.650 is much lower. Also, if we look at the minimum precision and recall, we see that the minimum precision of 0.5 is much lower than the recall counterpart of 0.769. The same is true for the maximum, where the precision of 0.850 is quite a bit lower than the maximum recall of 1.00.

Observation 16: Experts that work with projects where the model was trained on seem to agree with the model's predictions more. Table 6 shows that for participants #2, #5, and who use the analyzed code daily, performance rates are higher on average than for people who do not use the code daily. The average accuracy, F1, precision, and recall 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 is the expert with the highest agreement with the model.

5.3 RQ6: Why Do ING Experts Deem Recommended Extract Method Refactorings Useful/Not useful?

We observe seven different reasons why experts agree or disagree with the model:

(1) **Understandable:** Experts describe the method as understandable and comprehensible enough.
(2) **Specific:** Experts describe the method as being too domain-specific.
(3) **Complexity:** Experts mention the (high) complexity of the method. This includes matters such as long methods, too many try-catch blocks and, other complexity-related issues.
(4) **Anti-Patterns:** Experts mention that the method contains an anti-pattern that should be removed.
(5) **Potential for refactoring:** Experts mention that, although an Extract Method is not fully crucial, the method would benefit from refactoring.
(6) **Repetition:** Experts mention the existence of duplicated code in the method.
(7) **Lack of Readability:** The participant mentions that readability of the code can be improved with the refactoring.

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 3), and in the other plot the frequencies of reasons where the experts did not agree with the model (Figure 2).

Observation 17: When experts agree with the model's decision to apply Extract Method, they most often cite the method's high complexity. We see, from Figure 3, that the most commonly given reason for apply Extract Method to a method (25 occurrences) is that the method is too complex. The second most given reason is that the method contains an anti-pattern or does not adhere to a pattern (21 occurrences).

Observation 18: When experts agree with the model's prediction to not apply an Extract Method, it is because the code is specific enough or already sufficiently understandable. We observe from Figure 3 that when participants agree with the model's prediction not to refactor, they most commonly mention that the method is is a domain-specific (with 22 occurrences). The next most common reason, with 17 appearances, is that the method is understandable enough without further explanation. The next reason (the method has "potential to be refactored") is much less common but still appears four times.

Observation 19: When experts disagree with the model's prediction to apply Extract Method, they often propose a potential refactoring operation different from "Extract Method". On eight occasions (Figure 2, "potential"), experts did not agree with the recommendation for an Extract Method refactoring per se, but did mention that the method would benefit from refactoring. Interestingly, while experts mentioned that a refactoring needed to happen in such methods, they were not specific about which one.

Key takeaway: The opinions of experts on whether methods should undergo an Extract Method seem to match the recommendations of the model (with an average accuracy of 72%). Experts that are closer to the software systems agree even more with the models. A common reason for the agreement is that methods are complex; a common reason for disagreement is that the method is already quite understandable.

6 THREATS TO VALIDITY

6.1 Internal Validity

Partially duplicated class-level metrics. In this paper, we study Extract Method refactoring recommendations. Given that methods are contained in a class, we train not only on method-level metrics, but also on class-level metrics, as a way to give the classification

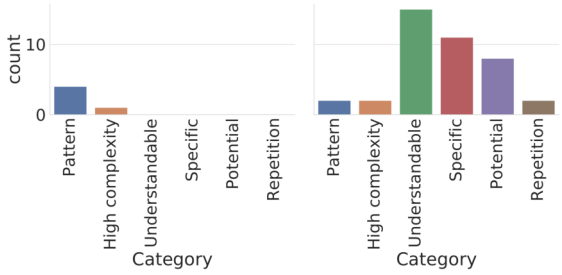


Figure 2: The experts' reasoning for cases where they did not agree with the model's prediction. The plot on the left shows the case where the model suggested to not apply an Extract Method refactor to the method and the user did want to apply an Extract Method refactoring. The plot on the right shows when the model suggested applying an Extract Method, but the expert did not think the refactoring was necessary.

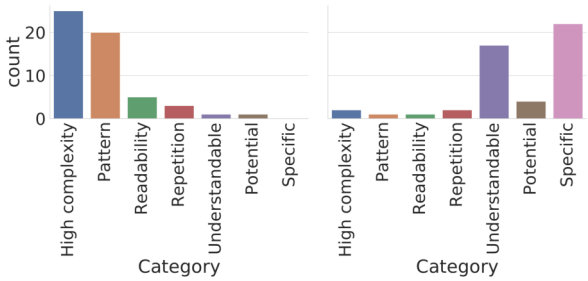


Figure 3: The experts' reasoning for cases where they agreed with the model's prediction. The plot on the left shows the cases where the model suggested to apply an Extract Method operation on the method. The plot on the right shows when the model suggested to not refactor the method.

algorithms some "context". This means, however, that these class metrics are often duplicated in our dataset, as multiple methods exist in one class, and they all share the same class-level metrics. Such partially duplicated data may inflate performance, which may explain why the performance obtained in RQ1 is much higher than the ones obtained in other RQs. More research is needed into the role of partially duplicated feature vectors in such models.

Difference between refactored and non-refactored instances. We feed the learners with two types of data points, methods that underwent an Extract Method (the true labels) and methods that did not undergo an Extract Method (the false labels). While we identify Extract Method instances through the sound strategies implemented by RefactoringMiner, the detection of methods that do not need an Extract Method is the heuristic proposed by Aniche et al [4].

We used $s = 20$ as threshold. This threshold was selected after non-systematic exploration in the ING dataset. Different thresholds might yield different results, and more exploration is needed to define what the best threshold for ING is.

The selection of experts. In our survey, we chose three ING experts that did not know any of studied projects, and two that were part of their development teams. It could be that the outside perspective was not an accurate assessment of what would be appropriate refactoring opportunities for the relevant code base, since one could argue that doing so would take experience with the code base. We, however, make no guarantees about whether results would be the same if evaluated by other experts at ING. Future work should expand on the human evaluation of the refactoring recommendations provided by the models.

6.2 External Validity

This paper is a case study within a single organization, ING, a large financial organization. It was not our goal to generalize our findings beyond it. That being said, the results we observed in this study are encouraging, and we suggest researchers to replicate this study in other industrial contexts and domains.

7 CONCLUSIONS AND FUTURE WORK

Identifying refactoring opportunities is a fundamental task for software development teams that aim to reduce their maintenance and evolution costs. Yet, it is still an open research problem.

Recent papers have been successful in building models that learn from previous refactoring operations in open-source systems. In this paper, we experiment with such models at ING, a large financial company that is always interested in improving the quality of its code bases. After a series of empirical studies and observations, we conclude that machine learning models are able to predict future refactorings, with interesting levels of accuracy, also in industry systems.

We see opportunities for future work. First, in the data collection process. The identification of methods that do not need refactoring is done through heuristics. ING (and we conjecture the same in other companies) does not have any labeled datasets of methods considered ideal from which models could learn from. Heuristics are therefore needed, and future work should explore different ones. Second, the models have much to improve. Our models achieve an accuracy of 74% in unseen ING projects. While 74% is an encouraging number, anecdotal evidence from industry suggests that tools should have at most a 15% false positive rate. This means these models still need to improve before developers do not find them problematic. Finally, presenting these results to developers is still a challenge. At ING, we experimented with showing the predictions within our code review tools, but the engagement was low. While we have our own assumptions of why this was the case, building a developer-friendly tool is surely an important step before deploying refactoring recommendation models.

ACKNOWLEDGMENTS

This project was partially funded by the EU ITEA3 Industrial-grade Verification and Validation of Evolving Systems (IVVES) project, under grant agreement "ITEA2019-18022-IVVES", and by the ICAI AI for Fintech Research. We also thank the five ING experts who took their time to support our research.

REFERENCES

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

[2] Miltiadis Allamanis. 2019. 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*, Vol. 11. Association for Computing Machinery, Inc, New York, NY, USA, 143–153. https://doi.org/10.1145/3359591.3359735 arXiv:1812.06469

[3] Erik Ammerlaan, Wim Veninga, and Andy Zaidman. 2015. 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*. Institute of Electrical and Electronics Engineers Inc., 504–507. https://doi.org/10.1109/SANER.2015.7081865

[4] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius Durelli. 2020. The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.3021736

[5] Leo Breiman. 2001. Random forests. *Machine Learning* 45, 1 (oct 2001), 5–32. https://doi.org/10.1023/A:1010933404324

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

[7] Francesca Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (jun 2016), 1143–1191. https://doi.org/10.1007/s10664-015-9378-4

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

[9] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

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

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

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

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

[14] Mark Harman and Laurence Tratt. 2007. Pareto optimal search based refactoring at the design level. In *Proceedings of GECCO 2007: Genetic and Evolutionary Computation Conference*. ACM Press, New York, New York, USA, 1106–1113. https://doi.org/10.1145/1276958.1277176

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

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

[17] R Leitch and E Stroulia. 2003. Assessing the maintainability benefits of design restructuring using dependency analysis. In *Proceedings - International Software Metrics Symposium*, Vol. 2003-Janua. IEEE Computer Society, 309–322. https://doi.org/10.1109/METRIC.2003.1232477

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

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

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

[21] Mark O'Keeffe and Mel Ó Cinnéide. 2008. Search-based refactoring: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice* 20, 5 (sep 2008), n/a–n/a. https://doi.org/10.1002/smr.378

[22] 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. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[23] Casper Schröder, Adriaan van der Feltz, Annibale Panichella, and Maurício Aniche. 2021. Search-Based Software Re-Modularization: A Case Study at Adyen. In *Proceedings of IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[24] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. 2014. Recommending automated Extract Method refactorings. In *22nd International Conference on Program Comprehension, ICPC 2014 - Proceedings*. Association for Computing Machinery, Inc, New York, New York, USA, 146–156. https://doi.org/10.1145/2597008.2597141

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

[26] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Identification of Extract Method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 84, 10 (oct 2011), 1757–1782. https://doi.org/10.1016/j.jss.2011.05.016

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

[28] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 483–494. https://doi.org/10.1145/3180155.3180206

[29] David van der Leij, Jasper Binda, Robbert van Dalen, Pieter Vallen, Yaping Luo, and Maurício Aniche. 2021. Data-Driven Extract Method Recommendations: A Study at ING: Appendix. https://doi.org/10.5281/zenodo.5083417.

[30] Ruru Yue, Zhe Gao, Na Meng, Yingfei Xiong, Xiaoyin Wang, and J. David Morgenthaler. 2018. 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*. Institute of Electrical and Electronics Engineers Inc., 115–126. https://doi.org/10.1109/ICSME.2018.00021 arXiv:1807.11184