Measuring Trivial and Non-Trivial Refactoring: A Predictive Analysis and Index Proposal

Darwin Pinheiro, Carla Bezerra (advisor), Anderson Uchôa (co-advisor)

Federal University of Ceará, Brazil

darwinfederal@alu.ufc.br, carlailane@ufc.br, andersonuchoa@ufc.br

Abstract. *This study investigates the relationship between trivial and non-trivial refactorings and proposes a metric to evaluate refactoring triviality. We analyzed 1.9M refactorings from 1,291 open-source projects with 45 code metrics using supervised learning. We evaluate 5 classification models and 7 regression models under various configurations. Based on these results, we propose a metric based on complexity, speed and risk, with insights from 15 developers on 58 selected features. The results show that separating the refactorings by triviality improves the predictions and that the use of all features outperforms the prioritization of the developer. Ensemble models outperformed linear ones, and expert perceptions aligned with model results. These findings support refactoring decisions and highlight future research opportunities.*

1. Introduction

Software maintenance is one of the most expensive activities in software engineering effort [Zarnekow and Brenner 2005]. According to [Bertrand 1994], maintenance represents a total cost of 70% of a system. The main factor for this high cost is poor software quality [Dehaghani and Hajrahimi 2013]. During software maintenance, it is possible that developers introduce code with poor structural quality in an unintentional or intentional way [Ouni et al. 2015]. One solution that can solve this problem is applying transformations to the source code, a very common type of transformation that meets this objective is software refactoring [Silva et al. 2016].

[Opdyke 1992] introduced the term refactoring, but this term only became popular with the book by [Fowler 2018]. Refactoring is defined as a transformation that changes the internal structure of the source code without changing the external behavior [Fowler 2018]. Maintaining external behavior means that after applying the refactoring activity, the software should produce the same output as before.

Some benefits of using software refactoring techniques are: (i) improved overall software quality; (ii) lower maintenance costs; and (iii) increased developer productivity [Fowler 2018, Moser et al. 2007]. In this way, the benefits extend to internal quality attributes such as coupling, cohesion and external software quality attributes. They can significantly improve the reusability and readability of systems [Mens and Tourwé 2004, Bavota et al. 2015, Malhotra and Chug 2016].

Researchers have investigated different perspectives for the use of refactoring [Mens and Tourwé 2004, Azeem et al. 2019, de Paulo Sobrinho et al. 2018, Bibiano et al. 2023], such as: (i) solutions that recommend refactorings to developers [Bavota et al. 2015, Tsantalis et al. 2018, Almogahed et al. 2023,

Bibiano et al. 2024, Nikolaidis et al. 2024]; (ii) machine learning-based refactoring detection [Aniche et al. 2020, AlOmar et al. 2021, Nyamawe 2022, Tan et al. 2024]; (iii) developer motivation to refactor code [Silva et al. 2016, Palomba et al. 2017], and (iv) obstacles to refactoring [Kim et al. 2014, Sharma et al. 2015, Liu et al. 2024]. However, even though refactoring has been investigated as a provider of benefits for software quality, whether by improving internal or external attributes, some studies indicate that in some cases, refactorings can negatively affect software maintenance.

[Azeem et al. 2019] and [Baqais and Alshayeb 2020] emphasize the need for further studies on how ML can identify refactoring opportunities, as well as the development of automated solutions involving refactoring. [Kaur and Singh 2019] highlights that many commonly practiced refactorings in the industry are still unexplored. Moreover, the lack of a clear assessment of the triviality of refactorings creates challenges in selecting the ideal technique and ensuring that the system's behavior is not affected [Akhtar et al. 2022]. In addition,developers tend to prefer manual, ttime-consumingnd risky refactorings, avoiding automated tools to minimize the risk of introducing new bugs [Abid et al. 2022, Silva et al. 2016]. Consequently, our motivation is based on the identified needs, proposing a solution based on ML that can be automated and provides developers with reliable information to decide whether to apply automated refactoring.

In this work, the main objective is **to identify the relationship between trivial and non-trivial refactorings and propose a metric that evaluates the triviality of implementing a refactoring**. We also propose a metric that evaluates the triviality of implementing refactoring operations, considering simplicity, speed and risk. The main research questions are presented below:

> **RQ_1:** How effective are trivial refactorings in predicting non-trivial refactorings?
> **RQ_2:** How can an effective refactoring triviality index be developed using an ML approach from a developer's perspective?

To answer these questions, we conducted two separate studies. The methodology adopted involved the collaboration of developers and refactoring experts. Until the completion of this work, two articles on the subject were published. The first study [Pinheiro et al. 2022] was awarded the best paper at SBCARS'22. The second [Pinheiro et al. 2024] was published in JSERD in 2024. The most recent article, detailed in Study 2, is in the submission phase for the Journal of Systems and Software.

2. Related Work

[Agnihotri and Chug 2020] conducted a literature review to investigate studies based on applying refactoring operations to remove code smells and their effect on software quality. The authors selected 68 studies that were published between 2001 and 2019. Based on the results, the authors pointed out the need for researchers to conduct studies proposing automatic tools for detecting and applying refactorings to improve software quality. Similarly, [Naik et al. 2023] conducted a detailed review on the use of deep learning (DL) techniques in code refactoring, analyzing 17 studies published between 2016 and 2022. The objective was to discover which DL techniques are applied in refactoring, evaluate the performance of these approaches, and identify the limitations and challenges faced in the area.

[Aniche et al. 2020] conducted a large-scale empirical study on 11,149 projects to verify the effectiveness of machine learning algorithms in predicting refactoring recommendations. The authors found that supervised algorithms are effective in predicting refactorings, and the results indicate that the resulting models achieved accuracy above 90%. Similarly, [Nyamawe 2022] used machine learning with commit history to predict refactorings. The authors implemented a binary classifier to predict the need for refactorings and a multi-label classifier to recommend refactoring. The results suggest that leveraging commit messages significantly improved the accuracy of refactoring recommendations.

[Ferreira et al. 2023] proposed a definition to order dependencies between refactorings and developed an algorithm to detect these relationships, using a vast dataset with 1,457,873 refactorings recommended in 9,595 open source projects. The authors classified refactorings into trivial and non-trivial graphs, representing collections of operations with their respective application dependencies

Based on the results found in the literature review on refactoring, it is possible to identify some deficiencies and needs for this area, such as: (i) conducting studies that develop new automated solutions involving refactoring activities; (ii) conducting more studies with refactoring activities commonly used in the industry; and (iii) conducting studies addressing machine learning as a solution to code design problems. Therefore, and knowing the importance of filling these gaps, this work proposes to explore a semi-automated solution to classify the refactorings most used by developers, covering those involved in the industry and based on models trained by supervised learning algorithms.

Overall, the papers make significant contributions to understanding the refactoring activity and its impact on software quality, in addition to emphasizing the development of automated solutions to detect and apply these operations. One of the approaches that has received considerable attention from researchers is the use of machine learning to support refactoring. In view of this, this paper proposes a solution for refactoring activity based on machine learning to identify the triviality of refactorings. This approach also includes validation by experts and developers in the area of software engineering, allowing a combination of automatic predictions and specialized human knowledge.

3. Methodology

This work aims to identify the relationship between trivial and non-trivial refactorings and propose a metric that evaluates the triviality of implementing a refactoring. To this end, we conducted two secondary studies with the purpose of answering the proposed research questions. The following are the sub-questions formulated for each study, as well as the methodology used in their conduct:

3.1. Study 1: On the effectiveness of trivial refactorings in predicting non-trivial refactorings

The study aims to investigate how trivial refactorings affect the prediction of non-trivial refactorings. In this step, we combined different types of refactorings in different contexts to investigate the influence caused by supervised learning algorithms on the classification problem. We describe our research questions (RQs) as follows:

- **RQ$_1$:** *What is the performance of ML algorithms to predict trivial and non-trivial refactorings?*
- **RQ$_2$:** *How effective is the inclusion of trivial refactorings to predict non-trivial refactorings?*
- **RQ$_3$:** *How effective are data balancing techniques in the prediction of trivial and non-trivial refactorings?*
- **RQ$_4$:** *Can the best models be carried over to different contexts?*

To answer our research questions, we designed and followed a structured methodology composed of five sequential steps.

Step 1: Selection and analysis of open source systems. The first step consisted of selecting a set of open-source software systems. For this, we have built three sets of data. For the first dataset, we selected 884 engineering software projects from different authors. The second dataset (D2) is composed of 207 projects from the Apache ecosystem. Finally, the third dataset (D3) is composed of 200 projects from the Eclipse ecosystem.

Step 2: Detect refactoring opportunities and features mining. In this step, we have extracted the data about refactorings and code metrics (used as features) for all selected projects. To this end, we have performed three key activities: (1) extracting code refactorings; (2) tracking the modified files before and after refactorings, and (3) extracting code metrics to be used as features. We detailed each step as follows.

Step 3: Context Selection. In this step, with the datasets defined (**D1**, **D2** and **D3**), we separate and combine each dataset by type of refactoring (trivial and non-trivial) and state of refactoring (before and after the activity occurred). Each separation and combination in this study we call *context*. Furthermore, we subdivide each context into two classes depending on the type and state of the refactoring. In this study, we call one class **0** and the other **1**.

Step 4: Training and testing the models; In this step, we used the datasets constructed by the combinations **C1**, **C2** and **C3** created in the previous step to predict refactorings. All contexts have been tested, with some changes to the processing pipeline. Thus, the data from each dataset was split into two datasets: 80% for the training set (used to train the model) and 20% for the test set (used to validate and test the model). The trained models formed binary classifiers based on supervised ML algorithms: Random Forest, Decision Tree, Logistic Regression, Naive Bayes, and Neural Network.

Step 5: Evaluation Results. We describe each step as follows. Finally, we calculate accuracy, precision, recall, F1 score, and area under the curve metrics to evaluate trained models and compare the results by context. We decided to use the mean as we needed a value to represent the data.

This five-step methodology provided a systematic and reproducible framework for how trivial refactorings affect the prediction of non-trivial refactorings across different software ecosystems.

3.2. Study 2: Towards an effective refactoring triviality index: a machine learning approach from a developer's perspective

The second study aim to: **analyze** and propose an index that evaluates the triviality of refactoring; **with the purpose of** identify the degree of difficulty of its implementation; **in relation to** the aspects of complexity, speed, and risk; **from the point of view** of software developers; **in the context of** refactoring operations. We detail each RQs as follows:

- **RQ$_1$:** *Which code metrics are considered most relevant by developers to determine the triviality of a refactoring operation?*
- **RQ$_2$:** *How do different ML techniques behave in predicting the code refactorings triviality index?*
- **RQ$_3$:** *What is the impact of prioritizing features ranked by developers on the effectiveness of triviality index prediction models?*
- **RQ$_4$:** *To what extent is the proposed triviality index aligned with the developers' perception regarding the triviality of applying refactorings?*

To answer our research questions, we designed and followed a structured methodology composed of five sequential steps.

Step 1: Selection and analysis of open-source systems. The first step of this study consisted of the selection and analysis of open-source systems. We selected datasets that met the following criteria: (i) systems implemented in Java and open source since they are supported by the tool adopted for the extraction of refactorings; (ii) systems that Git must version, necessary for the feature extraction tool; and, (iii) have a significant volume of refactoring operations to facilitate the training of ML algorithms.

Step 2: Detecting refactoring opportunities and mining features. In this step, we extracted data on refactorings and code metrics (used in this study as features) for all selected projects. To do so, we performed three main activities: (1) extracting code refactorings; (2) tracking modified files before and after refactoring; and (3) extracting code metrics to be used as features.

Step 3: Developers' perception of the most relevant metrics. In conducting this study, we sought to understand which metrics developers consider most relevant in code refactoring activities.

Step 4: Applying supervised learning to predict refactoring triviality. Next, to better explain the machine learning process, we present the activities of data preprocessing, feature engineering, data normalization, data balancing, model training and evaluation. We must highlight that we added the refactoring operations that each commit received in the dataset.

Step 5: Evaluation of agreement between experts and predictive models. In this step, we calculated the MSE, RMSE, MAE, MAPE and adjusted R^2 metrics to evaluate the trained models [Kuhn and Johnson 2013, James et al. 2023], comparing the results between algorithms and databases, both with and without prioritization of features by developers. In addition, we sought to identify the agreement of the solution with developers in the code refactoring activity.

These five steps allowed us to achieve our goal of proposing and evaluating a refactoring triviality index from a developer's perspective. By combining feature mining, developer insights, and machine learning techniques, we were able to analyze the relevance of code metrics, evaluate predictive performance, and examine the alignment between model outputs and experts.

4. Results of Study 1: On the effectiveness of trivial refactorings in predicting non-trivial refactorings

In this section, we describe our results. We present an overview of calculating metrics for the contexts mentioned in Table 1. The choice of AUC and F1-score metrics is supported by the need to comprehensively assess the performance of machine learning models in classification tasks.

We combined several datasets and evaluated the performance of the ML algorithm in predicting trivial refactorings (present in all contexts). Table 1 presents the performance of each ML algorithm by ML metric and context specified in this study. In summary, our findings indicate that the **Random Forest** and **Decision Tree** algorithms are the most effective. However, regarding the AUC metric, the **Neural Network** created the best classifier. Furthermore, using a balancing technique, either through undersampling or oversampling of the data, did not significantly improve the results.

Tabela 1. Results of the different ML models after trained and tested

		None					Under					Over			
Alg	M	C0	C1	C2	C3	M	C0	C1	C2	C3	M	C0	C1	C2	C3
Decision	**acc**	0.52	0.52	0.58	0.88	**acc**	0.52	0.84	0.58	0.86	**acc**	0.53	0.85	0.58	0.88
	pre	0.53	0.53	0.59	0.90	**pre**	0.53	0.86	0.60	0.87	**pre**	0.53	0.86	0.60	0.89
	rec	0.51	0.48	0.58	0.93	**rec**	0.45	0.83	0.50	0.85	**rec**	0.44	0.83	0.50	0.87
	f1	0.52	0.51	0.58	0.91	**f1**	0.48	0.84	0.54	0.86	**f1**	0.48	0.84	0.55	0.88
	auc	0.52	0.52	0.58	0.85	**auc**	0.52	0.84	0.58	0.86	**auc**	0.53	0.85	0.58	0.88
Logistic	**acc**	0.50	0.52	0.51	0.81	**acc**	0.50	0.61	0.50	0.60	**acc**	0.50	0.61	0.50	0.61
	pre	0.50	0.60	0.51	0.81	**pre**	0.50	0.59	0.50	0.58	**pre**	0.50	0.59	0.50	0.59
	rec	0.93	0.82	0.92	0.97	**rec**	0.49	0.71	0.36	0.71	**rec**	0.48	0.67	0.44	0.74
	f1	0.65	0.69	0.65	0.82	**f1**	0.49	0.65	0.42	0.64	**f1**	0.49	0.63	0.47	0.65
	auc	0.50	0.61	0.50	0.54	**auc**	0.50	0.61	0.50	0.60	**auc**	0.50	0.61	0.50	0.61
Navie	**acc**	0.49	0.49	0.49	0.68	**acc**	0.50	0.52	0.50	0.52	**acc**	0.49	0.52	0.50	0.52
	pre	0.50	0.50	0.52	0.70	**pre**	0.50	0.51	0.50	0.51	**pre**	0.49	0.51	0.50	0.51
	rec	0.07	0.12	0.06	0.95	**rec**	0.06	0.93	0.08	0.92	**rec**	0.06	0.93	0.08	0.92
	f1	0.12	0.20	0.11	0.80	**f1**	0.11	0.66	0.14	0.65	**f1**	0.11	0.66	0.14	0.66
	auc	0.49	0.50	0.50	0.51	**auc**	0.50	0.52	0.50	0.52	**auc**	0.49	0.52	0.50	0.52
Neural	**acc**	0.50	0.76	0.50	0.82	**acc**	0.50	0.76	0.51	0.75	**acc**	0.49	0.76	0.52	0.82
	pre	0.50	0.74	0.74	0.81	**pre**	0.50	0.74	0.52	0.73	**pre**	0.49	0.73	0.78	0.80
	rec	0.97	0.82	0.50	0.95	**rec**	0.97	0.78	0.28	0.80	**rec**	0.95	0.82	0.06	0.81
	f1	0.66	0.78	0.60	0.87	**f1**	0.66	0.76	0.36	0.76	**f1**	0.65	0.78	0.11	0.82
	auc	0.50	0.85	0.51	0.86	**auc**	0.50	0.85	0.51	0.84	**auc**	0.51	0.86	0.51	0.91
Random	**acc**	0.53	0.84	0.58	0.88	**acc**	0.53	0.84	0.58	0.85	**acc**	0.53	0.84	0.58	0.88
	pre	0.54	0.86	0.59	0.89	**pre**	0.54	0.86	0.60	0.86	**pre**	0.54	0.86	0.59	0.89
	rec	0.55	0.84	0.61	0.94	**rec**	0.50	0.82	0.52	0.84	**rec**	0.49	0.83	0.55	0.86
	f1	0.54	0.85	0.60	0.91	**f1**	0.52	0.84	0.56	0.85	**f1**	0.51	0.84	0.57	0.87
	auc	0.53	0.84	0.58	0.84	**auc**	0.53	0.84	0.58	0.85	**auc**	0.53	0.84	0.58	0.88

In addition, we performed several combinations of trivial and non-trivial refactorings, in which each combination corresponds to a context in our study. In total, four different contexts were created. Additionally, two balancing techniques were applied to each of them. The first context (**C0**) is the unique context in which trivial refactorings are not present in any of the classes. We compared the results obtained in other contexts

with **C0** to evaluate the effectiveness of including trivial refactorings. By looking at Table 1, we can see that the values obtained in **C0** are recurrently smaller than **C1**, **C2**, and **C3** even in the set of unbalanced data. To compare the performance of the classification models, we used the values of the metrics F1-Score and AUC.

Our findings indicate that trivial refactoring operations can impact the result of predicting new refactorings, which can be positive or negative. In the first case, an increase in accuracy was observed when partially combining the trivial refactorings in the **C1** and **C3** contexts, compared to the context without trivial refactorings (**C0**). In the second case, when combining all trivial and non-trivial refactorings and separating them into before and after refactoring, some cases did not show significant values and even worsened the indices. Therefore, trivial refactorings can improve the models' prediction by choosing the appropriate configuration.

Furthermore, to improve the performance between the models and reduce the outliers between classes, we have evaluated the Effectiveness of two well-known data balancing techniques: *Random Under Sampler* and *Oversampling with SMOTE*. In summary, the data balancing techniques' results varied in the different models, both by context and algorithm. In some cases, a complete rejection of the technique was observed since the use of the technique did not result in improvements or at least maintained the original results. All outliers are presented with a gray highlight in Table 2.

Tabela 2. Performance of algorithms in the contexts with balancing techniques

	C0		C1		C2		C3	
Alg	under	over	under	over	under	over	under	over
decision	0.00	0.01	0.32	0.33	0.00	-0.02	-0.02	0.00
	0.00	0.00	0.33	0.33	0.01	-0.03	-0.03	-0.01
	-0.06	-0.07	0.35	0.35	-0.08	-0.08	-0.08	-0.06
	-0.04	-0.04	0.33	0.33	-0.04	-0.03	-0.05	-0.03
	0.00	0.01	0.32	0.33	0.00	0.00	0.01	0.03
logistic	0.00	0.00	0.09	0.09	-0.01	-0.01	-0.21	-0.20
	0.00	0.00	-0.01	-0.01	-0.01	-0.01	-0.23	-0.22
	-0.44	-0.45	-0.11	-0.15	-0.56	-0.48	-0.26	-0.23
	-0.16	-0.16	-0.04	-0.06	-0.23	-0.18	-0.18	-0.17
	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.07
navie	0.01	0.00	0.03	0.03	0.01	0.01	-0.16	-0.16
	0.00	-0.01	0.01	0.01	-0.02	-0.02	-0.19	-0.19
	-0.01	-0.01	0.81	0.81	0.02	0.02	-0.03	-0.03
	-0.01	-0.01	0.46	0.46	0.03	0.03	-0.15	-0.14
	0.01	0.00	0.02	0.02	0.00	0.00	0.01	0.01
neural	0.00	-0.01	0.00	0.00	0.01	0.02	-0.07	0.00
	0.00	-0.01	0.00	-0.01	-0.22	0.04	-0.08	-0.01
	0.00	-0.02	-0.04	0.00	-0.22	-0.44	-0.15	-0.14
	0.00	-0.01	-0.02	0.00	-0.24	-0.49	-0.11	-0.05
	0.00	0.01	0.00	0.01	0.00	0.00	-0.02	0.05
random	0.00	0.00	0.00	0.00	0.00	0.00	-0.03	0.00
	0.00	0.00	0.00	0.00	0.01	0.00	-0.03	0.00
	-0.05	-0.06	-0.02	-0.01	-0.09	-0.06	-0.10	-0.08
	-0.02	-0.03	-0.01	-0.01	-0.04	-0.03	-0.06	-0.04
	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.04

We evaluated the best models obtained in C3 context with the base dataset with respect to datasets with different named data domains (D2 and D3). These other datasets were configured in the same C3 context, trained, and evaluated. Next, we evaluate the performance of the model in terms of predicting refactorings, we also compared it with the models trained using the base dataset. In general, the generalization of the models trained with the base dataset was positive. Although refactoring data was extracted from

multiple projects, the models were able to identify refactorings based on code attributes and metrics, regardless of the data domain. Results can be seen in the Table 3.

Tabela 3. Performance of generalization in the best context

	None		Under		Over	
Alg	**D1_D2**	**D1_D3**	**D1_D2**	**D1_D3**	**D1_D2**	**D1_D3**
decision	-0.03	-0.03	-0.04	-0.06	-0.02	-0.03
	-0.03	-0.02	-0.04	-0.06	-0.02	-0.03
	-0.01	-0.01	-0.03	-0.06	-0.02	-0.03
	-0.02	-0.02	-0.03	-0.06	-0.02	-0.03
	-0.03	-0.05	-0.04	-0.06	-0.02	-0.03
logistic	0.00	0.00	0.00	-0.01	-0.01	0.00
	0.00	0.00	0.00	0.01	-0.02	-0.01
	0.01	0.00	-0.01	-0.05	0.00	0.02
	0.01	0.00	-0.01	-0.02	0.03	0.00
	0.00	0.00	0.00	-0.01	-0.01	0.00
navie	0.01	0.00	0.00	0.00	0.00	0.00
	0.00	-0.01	0.00	0.00	0.00	0.00
	0.00	0.00	0.00	0.00	-0.01	-0.01
	0.00	0.00	0.00	0.00	0.00	0.00
	0.00	0.00	0.00	0.00	0.00	0.00
neural	-0.26	-0.15	-0.42	-0.28	-0.33	-0.31
	-0.22	-0.10	-0.42	-0.28	-0.31	-0.31
	-0.08	-0.06	-0.20	-0.18	-0.36	-0.31
	-0.17	-0.08	-0.27	-0.23	-0.33	-0.31
	-0.41	-0.37	-0.40	-0.36	-0.41	-0.40
random	-0.03	-0.03	0.07	-0.05	-0.02	-0.02
	-0.03	-0.03	-0.04	-0.06	-0.02	-0.03
	-0.01	0.00	-0.02	-0.04	-0.02	-0.02
	-0.02	-0.02	-0.03	-0.05	-0.02	-0.03
	-0.02	-0.05	-0.03	-0.05	-0.02	-0.02

Main Findings

- ML with tree-based models such as Random Forest, Decision Tree, and Neural network performed very well when trained with code metrics to detect refactoring opportunities.
- Separating trivial and non-trivial refactorings into different classes still results in a more efficient model, even on different datasets.
- Using balancing techniques that increase or decrease samples may not be the best strategy to improve models trained by datasets composed of code metrics and configured according to our study.
- We understand that a possible explanation for the performance improvements when ”trivial refactorings”are included is that the ML models have increased knowledge of what is not non-trivial refactoring, thus improving their prediction.

5. Results of Study 2

We considered the developers' opinions on code metrics. We also evaluated the key results by analyzing the average ratings for each metric, as well as the significant differences between the two groups of developers. Table 4 presents an overview of the evaluations of different code metrics conducted by both internal and external developers, with scores ranging from 1 to 5 on the Likert scale [Likert 1932] of perceived importance.

Tabela 4. Developer perspectives on the importance of code metrics

METRICS	Internal Dev	External Dev	AVERAGE
Issues of Security	4.00	4.80	4.40
CBO	4.00	4.60	4.30
Issues of Error Proneness	4.00	4.00	4.00
Quantity of Fields	4.00	4.00	4.00
LCOM	3.60	4.30	3.95
Max Nested Blocks	3.80	4.10	3.95
Issues of Design	3.20	4.40	3.80
Quantity of Loops	3.00	4.60	3.80
Issues of Performance	3.20	4.30	3.75
Usage of Each Field	3.40	4.10	3.75
Usage of Each Variable	3.40	4.10	3.75
Issues of Best Pratices	3.20	4.20	3.70
Quantity of Comparisons	3.20	4.20	3.70
Issues of Multithreading	3.20	4.10	3.65
FAN-IN	3.00	4.20	3.60
TCC	3.40	3.80	3.60
NOC	3.20	3.90	3.55
Number of Visible Method	3.40	3.70	3.55
FAN-OUT	3.00	4.00	3.50
DIT	2.80	4.20	3.50
WMC	3.40	3.60	3.50
LCC	3.00	4.00	3.50
FOC7	3.60	3.40	3.50
Quantity of Try/Catches	3.00	4.00	3.50
Number of Method	3.00	3.80	3.40
FOC14	3.40	3.20	3.30
Quantity of Variables	3.00	3.60	3.30
Quantity Method Invocations	2.40	4.10	3.25
FOC21	3.20	3.20	3.20
Quantity of Anonymous Classes	2.60	3.70	3.15
LOCC	2.80	3.40	3.10
FOC28	3.00	3.10	3.05
Lambda Expressions	2.40	3.60	3.00
Quantity of Returns	2.20	3.60	2.90
Quantity of Parenthesized Expressions	2.20	3.60	2.90
Path of Class	2.40	3.40	2.90
RFC	2.40	3.30	2.85
Inner Classes	1.80	3.80	2.80
Number of Log Statements	2.00	3.50	2.75
String Literals	2.00	3.40	2.70
File Type	2.20	3.10	2.65
Quantity of Modifiers	2.20	3.10	2.65
Quantity of Math Operations	1.80	3.50	2.65
Number of Unique Words	2.00	3.10	2.55
Quantity of Number	1.80	3.20	2.50
NOSI	1.60	3.00	2.30

In summary, our findings indicate that external developers tend to place higher value on code metrics related to structural complexity, best practices, performance, and security. They generally provided more positive ratings across most metrics, which may reflect a heightened level of caution or a more critical perspective on code quality. In contrast, internal developers appeared more conservative in their evaluations, possibly due to their familiarity with the code assessed in the study. These differences suggest that perceptions of the need for refactoring can vary significantly based on a developer's experience and context. Additionally, developers emphasize the importance of agile practices, an adaptable and continuous approach to refactoring, and the interdependence between testing, code complexity, and risk management. In addition, we analyzed all datasets and assessed the performance of various ML algorithms in predicting the triviality index. Table 5 provides important insights into the average performance of different ML algo-

rithms across different metrics and configurations used in this study. The values in the table are highlighted on a grayscale, where the intensity of the color increases as values approach the highest and lowest extremes. The average analysis of performance metrics offers an overview of the strengths and limitations of each model, highlighting those that are effective for predicting the triviality index of code refactorings in the context of our study.

Tabela 5. Table of machine learning model results

		APACHE		ECLIPSE		RANDOM		
Model	**Metric**	**DEV**	**ALL**	**DEV**	**ALL**	**DEV**	**ALL**	**AVG**
LinearRegression	mse	0.0033	0.0023	0.0032	0.0020	0.0032	0.0019	0.0027
	rmse	0.0577	0.0475	0.0566	0.0451	0.0564	0.0436	0.0512
	mae	0.0297	0.0187	0.0255	0.0182	0.0273	0.0209	0.0234
	mape	5.4001	3.3257	4.4645	3.4268	4.3312	3.7205	4.1115
	r2	0.8369	0.8861	0.8528	0.8999	0.8792	0.9202	0.8792
	r2a	0.8359	0.8810	0.8547	0.8918	0.8596	0.8745	0.8663
Ridge	mse	0.0033	0.0023	0.0032	0.0020	0.0032	0.0019	0.0027
	rmse	0.0577	0.0475	0.0566	0.0451	0.0565	0.0434	0.0511
	mae	0.0296	0.0187	0.0254	0.0181	0.0265	0.0203	0.0231
	mape	5.3791	3.3214	4.4495	3.4193	4.1952	3.5737	4.0564
	r2	0.8373	0.8861	0.8529	0.8999	0.8785	0.9207	0.8792
	r2a	0.8360	0.8810	0.8547	0.8917	0.8592	0.8741	0.8661
ElasticNet	mse	0.0047	0.0037	0.0044	0.0033	0.0047	0.0030	0.0040
	rmse	0.0685	0.0611	0.0666	0.0571	0.0688	0.0545	0.0628
	mae	0.0409	0.0377	0.0381	0.0365	0.0318	0.0271	0.0354
	mape	7.7201	6.9908	7.0111	6.9241	5.3722	5.1912	6.5349
	r2	0.7704	0.8111	0.7964	0.8397	0.8199	0.8753	0.8188
	r2a	0.7673	0.8041	0.8003	0.8323	0.7860	0.7974	0.7979
DecisionTree	mse	0.0025	0.0019	0.0029	0.0019	0.0012	0.0007	0.0019
	rmse	0.0502	0.0434	0.0537	0.0432	0.0350	0.0270	0.0421
	mae	0.0167	0.0140	0.0179	0.0129	0.0084	0.0061	0.0127
	mape	2.7332	2.4028	2.8854	2.1549	1.2251	1.1033	2.0841
	r2	0.8768	0.9050	0.8670	0.9082	0.9535	0.9694	0.9133
	r2a	0.8825	0.9057	0.8716	0.9098	0.9526	0.9481	0.9117
RandomForest	mse	0.0022	0.0015	0.0024	0.0017	0.0008	0.0007	0.0016
	rmse	0.0467	0.0393	0.0494	0.0411	0.0285	0.0268	0.0386
	mae	0.0170	0.0124	0.0183	0.0135	0.0075	0.0064	0.0125
	mape	2.7654	2.0721	3.0298	2.2836	1.0637	1.0695	2.0474
	r2	0.8934	0.9219	0.8873	0.9168	0.9691	0.9698	0.9264
	r2a	0.8983	0.9222	0.8896	0.9190	0.9610	0.9577	0.9246
GradientBoosting	mse	0.0021	0.0016	0.0024	0.0017	0.0008	0.0007	0.0016
	rmse	0.0462	0.0394	0.0490	0.0409	0.0275	0.0273	0.0384
	mae	0.0180	0.0130	0.0191	0.0144	0.0085	0.0064	0.0132
	mape	3.0005	2.2104	3.1916	2.4536	1.3346	0.9938	2.1974
	r2	0.8955	0.9216	0.8893	0.9177	0.9713	0.9686	0.9273
	r2a	0.8988	0.9221	0.8902	0.9183	0.9590	0.9584	0.9245
XGBoost	mse	0.0022	0.0015	0.0024	0.0017	0.0008	0.0008	0.0016
	rmse	0.0464	0.0390	0.0490	0.0411	0.0281	0.0279	0.0386
	mae	0.0188	0.0135	0.0199	0.0145	0.0087	0.0076	0.0138
	mape	3.1591	2.3081	3.1945	2.4680	1.2676	1.1461	2.2572
	r2	0.8946	0.9231	0.8901	0.9169	0.9700	0.9672	0.9270
	r2a	0.8984	0.9236	0.8950	0.9172	0.9599	0.9605	0.9258

In summary, our results indicate that these models achieved the lowest mean squared error (MSE) values, averaging around 0.0016 for Random Forest, Gradient Boosting, and XGBoost, compared to 0.0019 for Decision Tree, indicating high prediction accuracy. The root mean squared error (RMSE) was similarly low, with average values of 0.0384 for Gradient Boosting and 0.0386 for Random Forest and XGBoost. Moreover, the mean absolute errors (MAE) were lower for these models, approximately 0.0125 for Random Forest and 0.0132 for Gradient Boosting. Regarding the mean absolute percentage error (MAPE), these models recorded average values of 2.0474% for the Random Forest and

2.0841% for the Decision Tree, underscoring their superior performance with minimal percentage deviation.

The results indicate that tree-based and ensemble models, particularly Random Forest, Gradient Boosting, and XGBoost, are the most effective for predicting the triviality index of code refactorings in the context of our study. These models not only achieved the lowest error metrics but also exhibited strong explanatory power, effectively capturing data variability. In contrast, Elastic Net was less effective, suggesting that it may not be the optimal choice for the data types and issues addressed in this study. Furthermore, we compared the performance of ML models configured with features prioritized by developers (DEV) against those using all collected features (ALL). We assessed how these approaches influence the prediction of the triviality index for code refactorings. The analysis of the data reveals that the difference between using all features (ALL) and only the prioritized features (DEV) provides insights into the effectiveness of feature prioritization. Overall, the results show that using all features (ALL) often resulted in better model performance, as evidenced by reduced error metrics such as MSE, RMSE, MAE, and MAPE, as well as an increase in the R^2 and adjusted R^2 coefficients.

Finally, we collected feedback from developers with experience in refactoring by asking questions that covered various areas: developers' profiles (such as age, gender, country, and education level), their experiences with software refactoring, and their evaluations of complexity, speed, and risk in various refactoring scenarios. We rely on the Likert scale to measure the perceived difficulty of performing operations based on these aspects, ranging from *very difficult* (1) to *very easy* (5).

The results indicate that 43.7% of developers who participated in the survey have over 7 years of experience in software development, indicating a high level of seniority. Many hold positions as backend or full-stack developers, equivalent to 56.2%. Additionally, a significant portion of the participants(93.7%) have experience in the software development process related to refactoring and have worked directly with it. This is supported by responses indicating that many developers regularly engage in refactoring as a common practice in their work. Regarding age distribution, more than 62% of the developers are between 25 and 34 years old, followed by a smaller group of younger developers aged 18 to 24. This indicates that the sample primarily consists of professionals in an intermediate stage of their careers.

These findings suggest that continuous refactoring practice may not be critically important for perceptions of complexity and speed, but it does have a subtle impact on the perception of risk. Additionally, there is alignment between the perceptions of the expert developer groups and the predictive models presented in this study. This indicates that the triviality index is effectively aligned with developers' perceptions, positioning it as a reliable proposal for implementation in automated refactoring solutions for developers.

Main Findings

- The most important aspects, based on our analysis of developers' opinions and the distribution of metrics, were risk and complexity, followed by speed. This suggests that simpler and faster refactorings with lower risk are more likely to be implemented;

- Our machine learning models configured with all available features performed significantly better than those using only the features prioritized by developers;
- Unlike linear models, ensemble models such as Random Forest and Gradient Boosting demonstrated higher performance and efficiency, regardless of whether they used all features or just those prioritized by developers. These models were less impacted by feature selection, maintaining stable and effective performance even with a reduced set of variables; and
- The proposed triviality index metric proved to be effective and aligns well with developers' perceptions, making it a reliable solution for implementation in automated refactoring tools.

6. Discussion

The main objective of this work was to identify the relationship between trivial and non-trivial refactorings, in addition to proposing a metric that evaluates the triviality of refactoring implementation. To this end, a study was initially conducted to investigate the impact of trivial refactorings on the prediction of non-trivial refactorings. Next, a study was conducted that proposes an index that evaluates refactoring triviality to identify the degree of difficulty of its implementation concerning the aspects of complexity, speed and risk from the point of view of software developers.

The results revealed that algorithms such as Random Forest, Decision Tree, and Neural Network performed better when trained with code metrics to identify refactoring opportunities. In addition, we realize that separating trivial and non-trivial refactorings into different classes results in more efficient models, even when applied to different datasets. One of the main contributions of this work is the proposal of a metric called the Triviality Index. We designed this index to evaluate the triviality of implementing refactorings, which can assess the degree of difficulty of implementation based on complexity, speed, and risk.

The practical implications of our study are significant for software engineering. Using predictive models that are both highly effective and aligned with developers' perceptions makes refactoring decisions more reliable and accurate. Furthermore, this study is useful for software engineering tool developers who can use these models to improve refactoring suggestions. It also allows researchers to optimize code refactoring by increasing the efficiency of the technique through the methodology used.

7. Conclusion and Future Work

In this work, we presented two empirical studies. The first investigated how trivial refactorings can affect the prediction of non-trivial refactorings, considering code attributes and metrics. The second study proposed a metric based on ML through the regression problem to evaluate the triviality of implementing a refactoring based on the aspects of complexity, speed and risk. The study investigated how the prioritization of the features considered most important by developers affects the effectiveness of ML models in predicting the triviality index. In addition, we verified the alignment of perceptions between groups of developers with experience in refactoring and predictive models.

We identify the relationship between trivial and non-trivial refactorings and propose a metric that evaluates the triviality of refactoring implementation. The triviality index is an important metric for developers, as it can influence the decision of whether or not to implement a refactoring. In particular, we conduct empirical studies to explore the triviality of refactoring implementation. In fact, there is a growing need for reliable automated tools that can help developers evaluate the implementation of refactoring.

For future research, we recommend: (i) to develop an automated tool that calculates this metric during software development; (ii) investigation of triviality aspects in different software companies and domains; (iii) exploring differentiated weights in the triviality index; and (iv) integrating with Large Language Models for Enhanced Refactoring Prediction can improve the accuracy and interpretability of refactoring triviality indices.

Referências

Abid, C., Gaaloul, K., Kessentini, M., and Alizadeh, V. (2022). What refactoring topics do developers discuss? A large scale empirical study using stack overflow. *IEEE Access*, 10:56362–56374.

Agnihotri, M. and Chug, A. (2020). A systematic literature survey of software metrics, code smells and refactoring techniques. *Journal of Information Processing Systems*, 16(4):915–934.

Akhtar, S. M., Nazir, M., Ali, A., Khan, A. S., Atif, M., and Naseer, M. (2022). A systematic literature review on software-refactoring techniques, challenges, and practices. *VFAST Transactions on Software Engineering*, 10(4):93–103.

Almogahed, A., Mahdin, H., Omar, M., Zakaria, N. H., Mostafa, S. A., AlQahtani, S. A., Pathak, P., Shaharudin, S. M., and Hidayat, R. (2023). A refactoring classification framework for efficient software maintenance. *IEEE Access*, 11:78904–78917.

AlOmar, E. A., Peruma, A., Mkaouer, M. W., Newman, C., Ouni, A., and Kessentini, M. (2021). How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications*, 167:114176.

Aniche, M., Maziero, E., Durelli, R., and Durelli, V. H. (2020). The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, 48(4):1432–1450.

Azeem, M. I., Palomba, F., Shi, L., and Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138.

Baqais, A. and Alshayeb, M. (2020). Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28(2):459–502.

Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., and Palomba, F. (2015). An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14.

Bertrand, G. (1994). Simple points, topological numbers and geodesic neighborhoods in cubic grids. *Pattern recognition letters*, 15(10):1003–1011.

Bibiano, A. C., Coutinho, D., Uchôa, A., Assunçao, W. K., Garcia, A., de Mello, R., Colanzi, T. E., Tenório, D., Vasconcelos, A., Fonseca, B., et al. (2024). Enhancing recommendations of composite refactorings based on the practice. In *24th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–12. IEEE.

Bibiano, A. C., Uchôa, A., Assunção, W. K., Tenório, D., Colanzi, T. E., Vergilio, S. R., and Garcia, A. (2023). Composite refactoring: Representations, characteristics and effects on software projects. *Information and Software Technology*, 156:107134.

de Paulo Sobrinho, E. V., De Lucia, A., and de Almeida Maia, M. (2018). A systematic literature review on bad smells–5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering*, 47(1):17–66.

Dehaghani, S. M. H. and Hajrahimi, N. (2013). Which factors affect software projects maintenance cost more? *Acta Informatica Medica*, 21(1):63.

Ferreira, T., Ivers, J., Yackley, J. J., Kessentini, M., Ozkaya, I., and Gaaloul, K. (2023). Dependent or Not: Detecting and Understanding Collections of Refactorings. *IEEE Transactions on Software Engineering*, 49(6):3344–3358.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional, Boston, MA, US, 2nd edition.

James, G., Witten, D., Hastie, T., Tibshirani, R., and Taylor, J. (2023). *An introduction to statistical learning: With applications in python*. Springer Nature, New York, NY, US, 3rd edition.

Kaur, S. and Singh, P. (2019). How does object-oriented code refactoring influence software quality? research landscape and challenges. *Journal of Systems and Software*, 157:110394.

Kim, M., Zimmermann, T., and Nagappan, N. (2014). An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649.

Kuhn, M. and Johnson, K. (2013). *Applied Predictive Modeling*. Springer, New York, NY, US.

Likert, R. (1932). A technique for the measurement of attitudes. *Archives of Psychology*, 22 140:55–55.

Liu, J., Jin, W., Zhou, J., Feng, Q., Fan, M., Wang, H., and Liu, T. (2024). 3erefactor: Effective, efficient and executable refactoring recommendation for software architectural consistency. *IEEE Transactions on Software Engineering*, pages 1–23.

Malhotra, R. and Chug, A. (2016). An empirical study to assess the effects of refactoring on software maintainability. In *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 110–117, Jaipur, India. IEEE.

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

Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., and Succi, G. (2007). A case study on the impact of refactoring on quality and productivity in an agile team. In *IFIP*

Central and East European Conference on Software Engineering Techniques, pages 252–266, Berlin, Germany. Springer.

Naik, P., Nelaballi, S., Pusuluri, V. S., and Kim, D.-K. (2023). Deep learning-based code refactoring: A review of current knowledge. *Journal of Computer Information Systems*, 64(2):314–328.

Nikolaidis, N., Mittas, N., Ampatzoglou, A., Feitosa, D., and Chatzigeorgiou, A. (2024). A metrics-based approach for selecting among various refactoring candidates. *Empirical Software Engineering*, 29(1):25.

Nyamawe, A. S. (2022). Mining commit messages to enhance software refactorings recommendation: A machine learning approach. *Machine Learning with Applications*, 9:100316.

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. Ph.d., University of Illinois at Urbana-Champaign, Urbana, IL, US.

Ouni, A., Kessentini, M., Bechikh, S., and Sahraoui, H. (2015). Prioritizing code-smells correction tasks using chemical reaction optimization. *Software Quality Journal*, 23(2):323–361.

Palomba, F., Zaidman, A., Oliveto, R., and De Lucia, A. (2017). An exploratory study on the relationship between changes and refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 176–185, Buenos Aires, Argentina. IEEE.

Pinheiro, D., Bezerra, C., and Uchôa, A. (2024). On the effectiveness of trivial refactorings in predicting non-trivial refactorings. *Journal of Software Engineering Research and Development*, 12(1):5–1.

Pinheiro, D., Bezerra, C. I. M., and Uchoa, A. (2022). How do trivial refactorings affect classification prediction models? In *Proceedings of the 16th Brazilian Symposium on Software Components, Architectures, and Reuse*, page 81–90, New York, NY, US. Association for Computing Machinery.

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

Silva, D., Tsantalis, N., and Valente, M. T. (2016). Why we refactor? Confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, pages 858–870, New York, NY, USA. Association for Computing Machinery.

Tan, A. J. J., Chong, C. Y., and Aleti, A. (2024). Rearrange: Effort estimation approach for software clustering-based remodularisation. *Information and Software Technology*, 176:107567.

Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2018). Ten years of jdeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*, pages 4–14, Campobasso, Italy. IEEE.

Zarnekow, R. and Brenner, W. (2005). Distribution of cost over the application lifecycle - A multi-case study. *ECIS 2005 Proceedings*, page 26.