Journal of Software Engineering Research and Development, 2024, 12:5, doi: 10.5753/jserd.2024.3324
This work is licensed under a Creative Commons Attribution 4.0 International License.

On the Effectiveness of Trivial Refactorings in Predicting Non-trivial Refactorings

Darwin Pinheiro [**Federal University of Ceara** | ***darwinfederal@alu.ufc.br***]

Carla Bezerra [**Federal University of Ceara** | ***carlailane@ufc.br***]

Anderson Uchôa [**Federal University of Ceara** | ***andersonuchoa@ufc.br***]

Abstract

Refactoring is the process of restructuring source code without changing the external behavior of the software. Refactoring can bring many benefits, such as removing code with poor structural quality, avoiding or reducing technical debt, and improving maintainability, reuse, or code readability. Although there is research on how to predict refactorings, there is still a clear lack of studies that assess the impact of operations considered less complex (trivial) to more complex (non-trivial). In addition, the literature suggests conducting studies that invest in improving automated solutions through detecting and correcting refactoring. This study aims to identify refactoring activity in non-trivial operations through trivial operations accurately. For this, we use classifier models of supervised learning, considering the influence of trivial refactorings and evaluating performance in other data domains. To achieve this goal, we assembled 3 datasets totaling 1,291 open-source projects, extracted approximately 1.9M refactoring operations, collected 45 attributes and code metrics from each file involved in the refactoring and used the algorithms Decision Tree, Random Forest, Logistic Regression, Naive Bayes and Neural Network of supervised learning to investigate the impact of trivial refactorings on the prediction of non-trivial refactorings. For this study, we contextualize the data and call context each experiment configuration in which it combines trivial and non-trivial refactorings. Our results indicate that: (i) Tree-based models such as Random Forest, Decision Tree, and Neural Networks performed very well when trained with code metrics to detect refactoring opportunities. However, only the first two were able to demonstrate good generalization in other data domain contexts of refactoring; (ii) Separating trivial and non-trivial refactorings into different classes resulted in a more efficient model. This approach still resulted in a more efficient model even when tested on different datasets; (iii) Using balancing techniques that increase or decrease samples may not be the best strategy to improve models trained on datasets composed of code metrics and configured according to our study.

Keywords: *Refactoring, Machine Learning, Software Quality.*

1 Introduction

During software maintenance, developers can introduce low-quality code intentionally or unintentionally (Ouni et al., 2015; de Mello et al., 2022). Over time, this low-quality code can deteriorate the overall code quality and lead to crashes in the future (Yamashita and Moonen, 2012). Refactoring is a solution that can be used to address this problem by applying transformations to the source code (Silva et al., 2016a). Refactoring is a term introduced by Opdyke (1992) but only became widely known after the publication of Martin Fowler's book (Martin Fowler, 2000). Refactoring refers to a transformation that changes the internal structure of the source code without changing its external behavior (Martin Fowler, 2000). In other words, the software should produce the same output after the refactoring activity as it did before.

Researchers have investigated different perspectives for the use of refactoring (Mens and Tourwé, 2004; Azeem et al., 2019; de Paulo Sobrinho et al., 2018; Du Bois et al., 2004; Cassell et al., 2011; Bavota et al., 2010; Alkhalid et al., 2011; Al Dallal, 2012; Bibiano et al., 2023). Among them: (i) solutions that recommend refactorings for developers (Bavota et al., 2015; Tsantalis et al., 2018); (ii) challenges in applying refactoring (Sharma et al., 2015; Kim et al., 2014); (iii) developers' motivation to refactor the code (Silva et al., 2016a; Palomba et al., 2017; Paixão et al., 2020); and (iv) machine learning-based refactoring detection (Aniche et al., 2020; Nyamawe, 2022; AlOmar et al., 2021). The utilization of machine learning predictive models (ML) to assist developers in identifying refactoring opportunities to improve design is a relatively new area. (Azeem et al., 2019). Some studies use ML to detect refactoring opportunities through supervised learning (Aniche et al., 2020; AlOmar et al., 2021; Nyamawe, 2022; AlOmar et al., 2022; Rish et al., 2001). Others investigate refactoring opportunities using unsupervised learning (Alkhalid et al., 2010; Bryksin et al., 2018).

Despite many studies investigating how ML can be leveraged as a way to improve refactoring techniques (Aniche et al., 2020; Nyamawe, 2022; AlOmar et al., 2021; Panigrahi et al., 2020; Rish et al., 2001; Bryksin et al., 2018; Alkhalid et al., 2010), few studies investigate strategies on how to improve the prediction of refactorings by these models. Kumar et al. (2019a) states that software metrics are the most important factors in helping to estimate the propensity for refactoring at the class level among the main possible approaches. Azeem et al. (2019) conducts a literature review and points out that there is room for studies to investigate how ML can detect refactoring opportunities. Therefore, our motivation is based on the scarcity of studies investigating strategies to enhance refactoring prediction. Moreover, in our review of existing literature, we noted the classification of refactoring is a strategy frequently used in the literature.

Table 1. Group of Trival and Non-trivial Refactorings used in this research

Group	Refactoring	Problem	Solution
Trivial	Add Class Annotation	When an annotation is needed	Add an annotation
	Add Class Modifier	When it is necessary to use modifiers	Add the final modifier, static or abstract
	Change Access Modifier	When it is necessary to change the access modifier	Change access to default, private, protected or public
	Modify Class Annotation	When you need to change an annotation	Change the annotation
	Remove Class Annotation	When no longer need to use annotation	Remove annotation
	Remove Class Modifier	When no longer need to use modifiers	Remove final modifier, static or abstract
	Rename Class	When the class name is inappropriate	Rename the class
Non-trivial	Extract Class	When one class does the work of two	Create a new class to be responsible for fields and methods
	Extract Subclass	When a class uses resources in specific cases	Create a subclass to use these specific cases
	Extract Superclass	When two classes have common features	Create a superclass and move similar attributes and methods
	Merge Class	When a class does nothing or has no responsibilities	Merge data from two classes into one
	Move Class	When a class is in an inappropriate package	Move the class to a suitable package
	Move and Rename Class	Combination of the two aforementioned refactorings	Combination of the two aforementioned solutions

Thus, we chose to incorporate this perspective into our research, adopting a classification based on the triviality of refactoring.

In our previous study (Pinheiro et al., 2022), we investigated the impact of trivial refactorings on classification model prediction. Non-trivial refactorings are operations that generate changes in the design of system, while trivial refactorings do not significantly change the system design. The models were trained using the algorithm: Decision Tree, Random Forest, Logistic Regression, Naive Bayes, and Neural Network in 884 open-source systems. We identified contexts in which trivial refactorings can positively impact the prediction of non-trivial refactorings. We analyzed: (i) the performance of ML algorithms to predict refactorings; (ii) the effect of trivial operations on the prediction of non-trivial ones; and (iii) the use of balancing techniques to improve the predictions.

This article is an extension of our previous study (Pinheiro et al., 2022), in which we investigated the effectiveness of trivial refactorings in predicting non-trivial ones. Furthermore, we used classifier models of supervised learning, taking into account the influence of trivial refactorings. We also evaluated the performance of these models in other datasets. For this study, we: (i) added a new research question (RQ_4) to assess whether the ML models trained with the code metrics and attributes of the dataset used in our previous study (Pinheiro et al., 2022) can generalize to two other datasets selected in this new study; (ii) increased the number of projects used to compose each dataset, totaling 407 new projects (207 from the Apache community and 200 from the Eclipse community) in comparison to the previous study (Pinheiro et al., 2022); (iii) expanded the data extraction process to include refactorings, files, commits, and code metrics, to save all the necessary data for training the machine learning models; (iv) implemented a balancing technique called Synthetic Minority Oversampling Technique (SMOTE), which uses an approach to deal with unbalanced datasets through oversampling of minority classes (Chawla et al., 2002); and, (v) used the Area Under the ROC (AUC) metric in all models of this new study, a widely used metric to measure the classification of ML models (Hanley and McNeil, 1982).

As additional contributions to this article, we claim that ML with tree-based models such as Random Forest and Decision Tree performed extremely well and demonstrated good generalization in other data domains related to refactoring. Additionally, separating trivial and non-trivial refactorings into distinct classes resulted in a more effective model, even when tested on different datasets. However, altering the data balancing technique may lead to a comparable or worse outcome compared to the unbalanced model. This extended version of our study makes the following contributions:

- Our results show that tree-based machine learning models, such as Random Forest and Decision Tree, have shown excellent performance when trained with code metrics to detect refactoring opportunities.
- We identified that separating trivial and non-trivial refactorings into different classes resulted in a more efficient model, suggesting that this approach may improve the accuracy of automated solutions based on ML.
- We observed that sampling balancing techniques might not be the best strategy to improve models trained on datasets composed of code metrics and configured according to the study at hand.
- Finally, we observed that models trained with code attributes and metrics demonstrate good generalization in other data domain contexts.

The remainder of this article is organized as follows. Section 2 introduces the key definitions of this study. Section 3 presents our study settings. Section 4 presents our main findings, followed by a discussion. Section 5 presents an overview of the related work. Section 6 discusses the main threats to validity. Finally, Section 7 concludes the article and suggests future work.

2 Background

2.1 Code Refactorings

Code refactoring is a current practice of software development (Kim et al., 2014; Murphy-Hill et al., 2011; Silva et al., 2016b). Code refactoring was defined by Martin Fowler (2000) as a disciplined technique for structuring an existing source code, changing its internal structure without changing the system's functional behavior. In other words, code

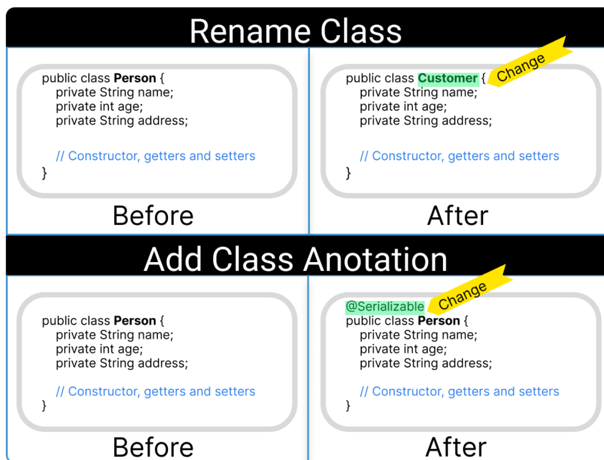

Figure 1. Example of Trivial refactoring

refactoring can be understood as applying transformations to the code structures to enhance software maintainability (Kim et al., 2014; Murphy-Hill et al., 2011). Each transformation type defines how developers should modify certain code elements, such as methods and classes. Extract Method is an example of a transformation type popularly adopted by developers (Silva et al., 2016b; Tsantalis and Chatzigeorgiou, 2011). This transformation type consists of extracting particular code statements of a method to create a new method. Extract Method can be used to separate the features across methods of a project (Tsantalis et al., 2018).

Some studies classify or group refactorings according to their general purpose (Sellitto et al., 2021; Smiari et al., 2022; Fernandes et al., 2020). Other studies prefer to simplify the binary classification by separating features into two classes, features referring to files before refactoring (not refactored) and after refactoring (refactored). (Eposhi et al., 2019; Nyamawe, 2022). AlOmar et al. (2021) classify their refactorings into: internal, external, fix bug, and fix the smell. Other than that, Sellitto et al. (2021) group refactorings into composing methods, moving resources, organizing data, simplifying method calls, and others.

Currently, in the literature, we have not identified any classification taken as a rule for refactorings. Martin Fowler (2000) describes cases where the same refactoring can be trivial or not, usually involving changes in the code scope. However, it does not describe or determine rules for classifying refactoring as trivial or not. Thus, we expanded the focus on this theme for this work and classified the refactorings into trivial and non-trivial. We consider trivial refactorings those operations that can change only one line, but not limited, of source code. Furthermore, trivial refactorings must be indivisible operations. Two examples can be observed in Figure 1, the Rename Class and Add Class Annotation.

Conversely, non-trivial refactorings generate a more significant change in code design, modifying several lines greater than one. Furthermore, they can be composed of other refactorings. An example of non-trivial refactoring is the Extract Class operation which aims to separate the responsibilities of a class, another example is the Move Class which aims to move a class to a more suitable package (see Figure 2). The procedure consists of: (i) creating a new class, and (ii) moving responsible attributes and methods. We can perform this operation when a class does not have a clear responsibility and when a subset of attributes and methods appear to form a new set (Martin Fowler, 2000). This refactoring is considered non-trivial because it significantly changes the code design.

This operation can also use other refactoring operations cataloged by Martin Fowler (2000), such as: Move Method, Move Field, and Change Reference to Value. Trivial refactorings are easier to identify because the operation changes the code design little. For example, the Rename Class operation might just change the name of the refactored class, changing only a single line. Also, it is not possible to split it into other refactoring operations.

Our study analyzed 13 types of transformations applied at the class level. Our choice was based on the need for more studies involving the main refactorings used in the industry (Khanam, 2018) and more studies of refactorings that prioritize the class level (Agnihotri and Chug, 2020). Additionally, we split these 13 types of transformations into trivial refactorings and non-trivial ones. Table 1 lists the set of trivial and non-trivial refactorings considered in this study.

2.2 Machine Learning Techniques

To investigate how trivial refactorings affect the prediction of non-trivial refactorings, we analyzed five ML techniques frequently used in literature (Aniche et al., 2020; Nyamawe, 2022; AlOmar et al., 2021, 2022; Panigrahi et al., 2020). These techniques involve different data analysis approaches, such as decision trees and regression analysis, responsible for creating the classifier models. We overview each ML technique as follows.

- **Decision Tree (DT)**: a technique used for both classification and regression. DT aims to learn decision rules inferred from the data to predict the value of a target variable. Represented by a binary tree model, where each node will be represented by an input variable and the leaves will represent an output variable used to make the prediction. Its characteristics are the speed to make predictions and accuracy for most problems (Quinlan, 2014).
- **Logistic Regression (LR)**: a technique that uses concepts of statistics and probability for binary classification. It analyzes different aspects or variables of an object to determine which class best fits. It can be divided into three models: binomial logistic regression, ordinal logistic regression and multinomial logistic regression (Bishop and Nasrabadi, 2006).
- **Naive Bayes (NB)**: a probabilistic classifier based on the application of Bayes' theorem (Jordan and Mitchell, 2015). This highly scalable technique disregards the correlation between the variables in the training set. It is considered a simple technique to train and fast (Rish et al., 2001).
- **Neural Network (NN)**: a technique commonly used for deep learning and designed to solve problems that other techniques cannot solve. NNs comprise many interconnected layers, so when data passes through these layers

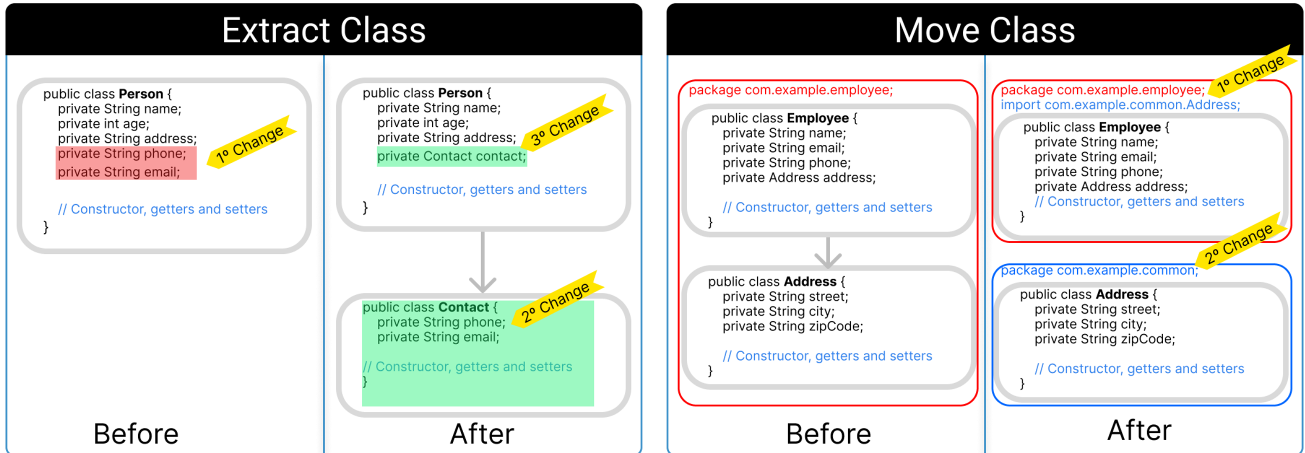

Figure 2. Example of Non-trivial refactoring

the NN can approximate calculations to transform input into outputs (Jin et al., 2000).

- **Random Forest (RF)**: technique capable of creating several independent trees employing many samples of observations and variables, with the main benefit of reduced variance compared to a single tree. It sums the forecasts for each tree to determine an overall forecast for the forest. RF-based algorithms are among the most accurate in many ML problems (Cutler et al., 2012).

2.2.1 *Machine Learning Metrics*

To evaluate ML algorithms' performance, metrics are necessary to measure the quality of a model (Carvalho et al., 2019). The metrics use the values extracted from the confusion matrix: True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN). For this study, we used the following metrics: (i) Accuracy, (ii) Precision, (iii) Recall, and (iv) F1-score (see Table 2). We present a short description of each metric as follows.

Table 2. Metrics for evaluating ML models

ML metrics	Formula
Accuracy	$\frac{TP+TN}{TP+FP+TN+FN}$
Precision	$\frac{TP}{TP+FP}$
Recall	$\frac{TP}{TP+FN}$
F1-score	$2\times\frac{Precision\times recall}{Precision+recall}$
AUC	$\int_0^1 \mathrm{TPR}(\mathrm{FPR})^{-1}\mathrm{dFPR}$

TPR - True Positive Rate
FPR - False Positive Rate

- **Accuracy**: It is the proportion of correctly classified observations among the total observations. The accuracy indicates the overall performance of the model (Davis and Goadrich, 2006).
- **Precision**: It is the proportion of positive observations correctly classified among the predicted positive observations. Accuracy is measured when FP is considered more harmful than FN (Carvalho et al., 2019).
- **Recall**: It is the proportion of positive observations correctly classified among true positive observations. The recall is measured when FNs are considered more harmful than FPs (Carvalho et al., 2019).
- **F1-score**: It is the harmonic mean between the precision and the recall (Chicco and Jurman, 2020).
- **AUC**: Area Under the Curve is a commonly used metric to evaluate the quality of a binary classification model. Represents the area under the ROC (Receiver Operating Characteristic) curve, which is an indicator of the model's ability to distinguish between positive and negative classes (Hanley and McNeil, 1982; Muschelli III, 2020).

2.3 Code Quality Metrics

Software metrics are used to measure and understand software structure. Chidamber and Kemerer (1994) proposed a suite of code metrics for software that adopts the object-oriented paradigm. This suite, one of the precursors, is commonly used in the literature (Padhy et al., 2015; Aggarwal et al., 2006; Malhotra[1] and Chug, 2012).

Our study selected a set of metrics from the CK suite proposed by Chidamber and Kemerer (1994), LOC (Lorenz and Kidd, 1994) and different attributes of the code elements, such as: the number of methods, number of returns, number of variables, etc. The values were extracted using the CK tool (Aniche, 2015) to be used as features to train the prediction models. The set of these metrics and the attributes of the code elements used in this work can be seen in Table 3.

3 Study Settings

This section describes the settings of our study. Section 3.1 introduces the study goal and research questions. Section 3.2 describes each study step and procedure, from data collection to data analysis.

Table 3. Metrics and attributes of code elements used in this work Lorenz and Kidd (1994); Chidamber and Kemerer (1994)

Field	Description
	Metrics
cbo	Coupling between objects. Counts the number of dependencies a class has.
wmc	(Weight Method Class or McCabe's complexity. It counts the number of branch instructions in a class.
noc	Number of Children. It counts the number of immediate subclasses that a particular class has.
rfc	Response for a Class. Counts the number of unique method invocations in a class.
lcom	Lack of Cohesion of Methods a normalized metric that computes the lack of cohesion of class
nosi	Number of static invocations. Counts the number of invocations to static methods
loc	Lines of code. It counts the lines of the count, ignoring empty lines and comments
	Code elements attributes
totalMethodsQty	Counts the number of all methods.
staticMethodsQty	Counts the number of static methods.
publicMethodsQty	Counts the number of public methods.
privateMethodsQty	Counts the number of private methods.
protectedMethodsQty	Counts the number of protected methods.
defaultMethodsQty	Counts the number of default methods.
visibleMethodsQty	Counts the number of visible methods.
abstractMethodsQty	Counts the number of abstract methods.
finalMethodsQty	Counts the number of final methods.
synchronizedMethodsQty	Counts the number of synchronized methods.
totalFieldsQty	Counts the number of all fields
staticFieldsQty	Counts the number of static fields
publicFieldsQty	Counts the number of public fields
privateFieldsQty	Counts the number of private fields
protectedFieldsQty	Counts the number of protected fields
defaultFieldsQty	Counts the number of default fields
finalFieldsQty	Counts the number of final fields
synchronizedFieldsQty	Counts the number of synchronized fields
returnQty	The number of return instructions
loopQty	The number of loops like for, while, do while and enhanced for
comparisonsQty	The number of comparisons == and !=
tryCatchQty	The number of try/catches
parenthesizedExpsQty	The number of expressions inside parenthesis
stringLiteralsQty	The number of string literals
numbersQty	The number of numbers literals int, long, double, float
assignmentsQty	The number of same or different comparisons
mathOperationsQty	The number of math operations (times, divide, remainder, plus, minus, left shit, right shift)
variablesQty	The number of declared variables
maxNestedBlocksQty	The highest number of blocks nested together
anonymousClassesQty	The quantity of anonymous classes
innerClassesQty	The quantity of inner classes
lambdasQty	The quantity of lambda expressions
uniqueWordsQty	The algorithm basically counts the number of words in a class, after removing Java keywords
typeAnonymous	Boolean indicating whether is an anonymous class
typeClass	Boolean indicating whether is a class
typeEnum	Boolean indicating whether is an enum
typeInnerclass	Boolean indicating whether is an inner class
typeInterface	Boolean indicating whether is an interface

Total: 45

3.1 Goal and Research Questions

This study aims to investigate the influence of trivial refactorings at the class level in predicting non-trivial refactorings. To this end, we used models based on ML algorithms trained with 45 code metrics. By understanding how trivial refactorings affect the prediction of non-trivial refactorings, we will be able to discover strategies to improve the prediction of refactorings through supervised learning. Furthermore, we investigated whether the trained models are generalized to other contexts. We describe our research questions (RQ_s) as follows.

RQ_1: What is the performance of ML algorithms to predict trivial and non-trivial refactorings? – **RQ_1** aims to investigate the performance of 5 ML algorithms(Random Forest, Decision Tree, Logistic Regression, Naive Bayes and Neural Network) to predict trivial and non-trivial refactorings together. By answering **RQ_1**, we can identify which algorithms produce the best results for our different sets of contexts, considering each context as different set that combines trivial and non-trivial refactorings.

RQ_2: How effective is the inclusion of trivial refactor-

ings to predict non-trivial refactorings? – **RQ**$_2$ aims to compare the performance of trained models to predict non-trivial refactorings by considering different sets combining trivial and non-trivial refactorings. By answering **RQ**$_2$, we can compare and evaluate which combination of trivial and non-trivial refactorings presents better results.

RQ$_3$**: How effective are data balancing techniques in the prediction of trivial and non-trivial refactorings?** – **RQ**$_3$ aims to evaluate the effectiveness of the data balancing technique applied to our different sets of contexts. By answering **RQ**$_3$, we can identify whether there is an imbalance in our data, as well as find the data balancing technique that performs best in our models with our configuration.

RQ$_4$**: Can the best models be carried over to different contexts?** – **RQ**$_4$ aims to understand whether the best models should be trained for a given context and whether it generalizes enough to different contexts. By answering **RQ**$_4$, we can reduce the cost that a new training can bring. In addition, we identified the ability to handle large volumes of data and avoid the cost of identifying complex patterns.

3.2 Study Steps and Procedures

Figure 3 overviews the sequence of five-step that we have followed to answer our RQs: (1) Selection and analysis of open source systems; (2) Detect refactoring opportunities and features mining; (3) Contexts Selection; (4) Training and testing the models; and (5) Evaluation Results. We describe each step as follows.

Step 1: Selection and analysis of open-source software systems. The first step consisted of selecting a set of open-source software systems. For our study, we needed to gather a large number of open-source projects to allow the study replication. For this, we have built three sets of data. We used the dataset used in the last article plus two similar datasets in order to minimize any bias produced by just one dataset. The new ones have the same characteristics and are compatible with the used tools. The first dataset (D1), namely in this study as the base dataset used in the last study (Pinheiro et al., 2022). We selected 884 software projects from a dataset of 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. These projects were chosen because the authors observed evidence of solid software engineering practices, including collaboration, continuous integration, quality, maintainability, sustained evolution, project management, responsibility, and unit testing. All projects were extracted from GitHub by our tool. Table 4 summarizes the data for the selected software systems. The first column contains the name of each ecosystem in the dataset, followed by the number of projects, commits, and refactorings. The replication package of the previous study[1] and extension[2] contains their detailed information.

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.

Table 4. Overview of the selected datasets

Ecosystem	# Projects	# Commits	# Refactorings
D1 (Base)	884	35,838	84,262
D2 (Apache)	207	272,096	1,144,365
D3 (Eclipse)	200	153,610	767,111
Total	**1,291**	**461.544**	**1,995,738**

Activity 1: Code refactorings extraction. We detected refactorings for all selected projects. For this end, we chose RMiner, (version 2.0) as the tool to detect code refactorings due to its high accuracy (Tsantalis et al., 2020). This tool is applied between two versions (commits) and returns the elements that changed from one version to another. It also returns the refactoring type associated with the change. The tool detected a total of 1.995.738 refactoring types used in our study (see Table 4). After the code refactoring extraction, we divided the refactorings into two groups, trivial and non-trivial refactorings, as described in Section 2.1.

Activity 2: Tracking the modified files before and after refactorings. To analyze the prediction of trivial and non-trivial refactorings, we need to track the modified files before and after the refactoring application. Thus, we tracked the version before and after each file undergoing trivial and non-trivial refactoring. To track the modified files, we utilized Pydriller (Spadini et al., 2018) and the Jupyter Notebook (Jupyter, 2022) to process the data. Thus, we tracked the version before and after each file undergoing trivial and non-trivial refactoring. A total of 39,423,447 files involved in refactoring operations were analyzed.

Activity 3: Extracting code metrics for tracked files. In this activity, we extracted the code metrics and some attributes of code elements to be used as features in our study. To this end, we have used the CK tool (Aniche, 2015) to extract each metric and attribute. Additionally, we created a Python script to automatize summarizing the file outputs provided by the CK tool in a single file. For all fields calculated by the tool, after previous data analysis, we decided to use only 45 metrics and attributes as features for our datasets. They can be seen in Table 3.

Step 3: Contexts 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**. The Table 5 presents all the divisions.

The base context (**C0**) is defined by separating features from files that have undergone non-trivial refactoring activity. Class 0 is for features before the activity is executed, while class 1 is for features after the activity is executed.

The next context (**C1**) is defined by file features that

[1]Available at `https://doi.org/10.5281/zenodo.6800385`.
[2]Available at `https://doi.org/10.5281/zenodo.7820168`.

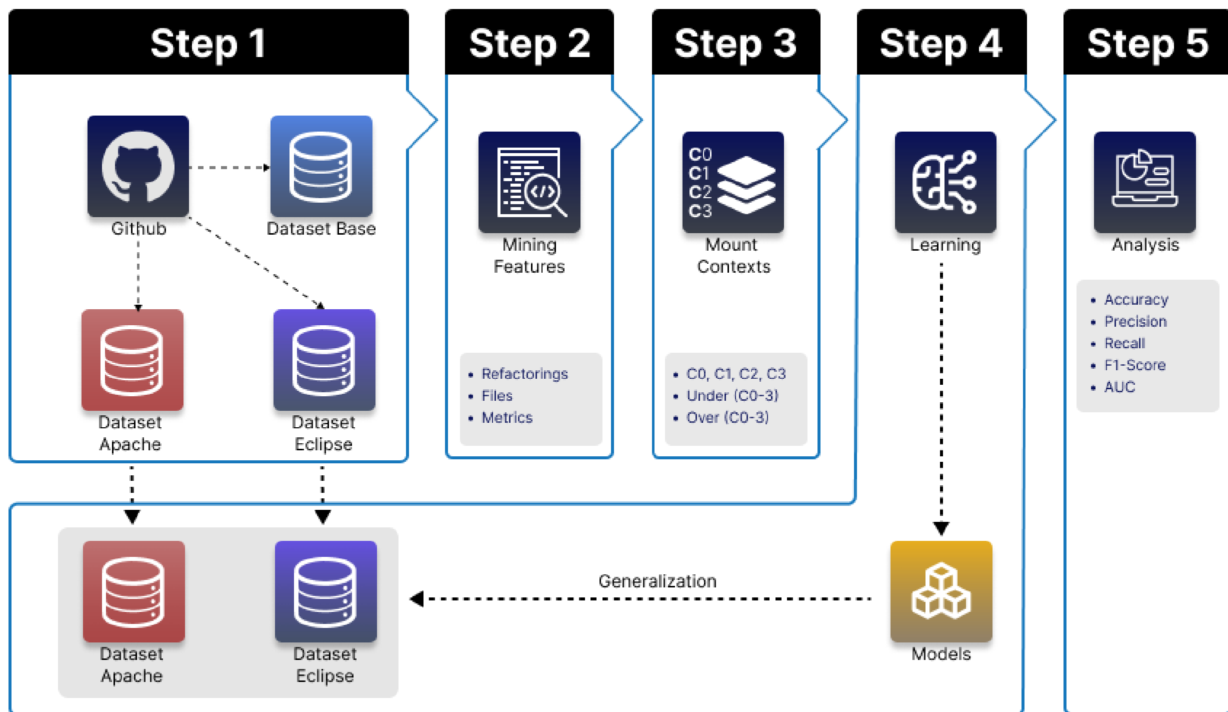

Figure 3. Overview of the research methodology

have undergone the trivial and non-trivial refactoring activity. Class 0 is for features after the trivial refactoring activity is performed, while class 1 is for features after the non-trivial refactoring activity is performed.

Context two (**C2**) is defined by file features that have undergone trivial and non-trivial refactoring activity. Class 0 is for features before the trivial and non-trivial refactoring activity is performed, while class 1 is for features after the trivial and non-trivial refactoring activity is performed.

Context three (**C3**) is defined by file features that have undergone trivial and non-trivial refactoring activity. Class 0 is for features before the trivial refactoring activity is performed, while class 1 is for features after the non-trivial refactoring activity is performed.

It is important to highlight that most contexts included refactorings since we sought to investigate how they can affect the prediction of non-trivial refactorings. The number of instances of each context can be seen in Table 5.

Table 5. Instance numbers of contexts

Context	D1		D2		D3	
class	0	1	0	1	0	1
C0	251,416	258,010	1,004,983	992,991	625,274	649,768
C1	232,468	258,010	1,265,956	992,991	446,880	649,768
C2	364,015	377,879	1,633,881	1,630,119	845,395	876,527
C3	112,599	258,010	628,828	992,991	220,121	649,768

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. The first four algorithms were used through the Scikit-learn library[3], while the Neural Network was used through TensorflowKeras[4]. After training, each generated model was validated by predicting the refactorings of the features in the test set.

Furthermore, we consider that the prediction of ML algorithms can be negatively affected by an unbalanced dataset (number of different samples between classes). Therefore, we applied two balancing techniques for each combination: Random Under Sampler and SMOTE (Chawla et al., 2002). These balancing techniques were chosen because they are commonly used in recent studies (Moreo et al., 2016; Hasanin and Khoshgoftaar, 2018; Tabassum et al., 2023) and because they facilitate the comparison of efficiency when used together (Mohammed et al., 2020). In addition to unbalanced contexts, the balancing techniques were applied individually by context **C0**, **C1**, **C2** and **C3**, creating eight more combinations: **C0 Under**, **C1 Under**, **C2 Under**, **C3 Under**, **C1 over**, **C1 Over**, **C2 Over** and **C3 Over**. After training, we tested the models to predict refactoring activity on the datasets. Then, we tested the generated models from the base dataset of the context that obtained better results in the other datasets.

Step 5: Evaluation Results. Finally, we calculated accu-

[3]`https://scikit-learn.org/stable/`
[4]`https://www.tensorflow.org/api_docs/python/tf/keras`

racy, precision, recall, F1-score, and Area Under the Curve metrics to evaluate the trained models and compared the results by context. We decided to use the mean as we needed a value to represent the data. Next, we observed: (i) whether the presence of trivial refactorings affects the prediction of other refactorings; (ii) which algorithm obtained better results; (iii) whether data balancing techniques had any effect; and, (iv) whether the models were able to generalize to other contexts. With this, it was possible to answer our research questions. We present the results in the next section.

4 Results and Discussions

In this section, we describe our results. We present an overview of calculating metrics for the contexts mentioned in Table 6 and Table 8. 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. AUC provides a robust measure of the model's discriminative ability in binary scenarios, while the F1-score balances precision and recall, proving particularly useful in cases of class imbalance. When used together, these metrics offer a more comprehensive analysis, enhancing the reliability and validity of the presented results.

We also present the generalization of the models. In the following subsections, we answer each of the RQs.

4.1 Performance of ML algorithms to predict trivial and non-trivial refactorings ($\mathbf{RQ_1}$)

To answer $\mathbf{RQ_1}$, we combined several datasets and evaluated the performance of the ML algorithm in predicting trivial refactorings (present in all contexts). Table 6 presents the performance of each ML algorithm by ML metric and context specified in this study.

In summary, our results indicate that the Random Forest algorithm achieved the best performance indices, considering the general average in all contexts, with an average of 0.71, 0.72, 0.74, 0.73 and 0.70 for the accuracy metrics, precision, recall, F1-score, and AUC, respectively. This Random Forest algorithm stood out in the first context, with a remarkable balance (see Table 6). For the context **C1**, the performance was 0.84, 0.86, 0.84, 0.85, and 0.84 for the metrics of accuracy, precision, recall, F1-score, and AUC, respectively. For **C1** with undersampling balancing, the performance was 0.84, 0.86, 0.82, 0.84, and 0.84 for the metrics of accuracy, precision, recall, F1-score, and AUC, respectively.

While for **C1** with oversampling balancing, 0.84, 0.86, 0.83, 0.84, and 0.84 were obtained for the metrics of accuracy, precision, recall, F1-score, and AUC, respectively. Furthermore, in the **C3** context, the models showed even better results with 0.88, 0.89, 0.94, 0.91, and 0.84 for the metrics of accuracy, precision, recall, F1-score, and AUC without balancing the data, respectively. With undersampling balancing, the performance was 0.85, 0.86, 0.84, 0.85, and 0.85. With the data oversampling, the values reached 0.88, 0.89, 0.86, 0.87, and 0.88, respectively.

> **Finding 1:** Models based on the Random Forest algorithm were the best in general contexts, with highlighting for the contexts **C1** and **C3**.

In Table 6, we also observed that the Decision tree in the **C3**, without balancing data, achieved equivalent results of the Random Forest. The indices were (accuracy, precision, recall, F1-score, and AUC) 0.88, 0.90, 0.93, 0.91, and 0.85, against 0.88, 0.89, 0.94, 0.91, and 0.84 of the Random Forest. With both balancing techniques, the results remained equivalent.

> **Finding 2:** Decision tree and Random Forest were the algorithms that achieved better results in the **C3** context, using or not the balancing technique.

As shown in Table 6, the model built by a Neural Network using a balancing technique with data oversampling showed the best results, considering the area under the ROC curve (**AUC**). This model achieved a significant AUC index of 0.91, followed by Random Forest and Decision Tree.

> **Finding 3:** The Neural Network was the algorithm that created the best classification model, considering the model's ability to distinguish between classes, regardless of the chosen cutoff point.

Table 8 presents the values of the metrics obtained from the models trained with data from datasets D2 and D3 in the **C3** context, as well as from the **base dataset** models applied to datasets D2 and D3 in the same context, represented in the Table 8 by D1_D2 and D1_D3.

The models based on **Random Forest**, **Neural Network**, and **Decision Tree** showed great results both in dataset D2 and D3. In D2 the **Decision Tree** model obtained a F1-score and AUC of 95% and 94%, respectively. In the case of **Neural Network**, a F1-score and AUC of 86% and 90% were obtained, respectively. For the **Random Forest** model was obtained a F1-score and AUC of 95% and 93%, respectively.

Similarly, in D3, the performance was also optimistic, in which the **Decision Tree** model obtained a F1-score and AUC of 95% and 90%, respectively. The **Neural Network** model obtained a F1-score and AUC of 89% and 87%, respectively. Finally, the **Random Forest** model obtained a F1-score and AUC of 95% and 89%, respectively. On the other hand, the models based on **Logistic Regression** and **Naive Bayes** algorithms presented inferior results in both datasets. In D2, the **Logistic Regression** model obtained a F1-score and AUC of 75% and 52%, respectively, while with **Naive Bayes** resulted in a F1-score and AUC of 74% and 51%, respectively.

> **Finding 4:** Tree-based and neural network models tend to be more efficient regardless of the dataset.

Implications of RQ_1. 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.

In the context of our research, it is crucial to acknowledge that each metric addresses specific facets of the model's performance, and the declaration of an algorithm as the best according to a particular metric does not necessarily imply overall superiority.

For instance, when considering the performance of algorithms such as Neural Networks, which are identified as the best based on the AUC metric, it suggests an exceptional capability for classification in terms of discriminating between classes, as measured by the ROC curve. However, alternative metrics like the F1 score prioritize different aspects, such as precision and recall, potentially yielding disparate conclusions regarding the overall model performance. This discrepancy underscores the significance of selecting evaluation metrics aligned with the specific objectives of the given task.

Furthermore, using a balancing technique, either through undersampling or oversampling of the data, did not significantly improve the results. Therefore, the results obtained in RQ_1 can help data scientists and developers of automated refactoring tools to make more informed decisions about which algorithms to use when investigating refactorings based on metrics and code attributes.

4.2 The effectiveness of including trivial refactorings to predict new refactorings (RQ_2)

To answer RQ_2, 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 6, 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. Thus, the **Decision Tree** model obtained an increase of 39% and 33% for F1-Score and AUC, respectively, when including trivial refactorings in the configuration of **C3**. In the same line, the **Logistic Regression** model obtained an increase of 17% (F1-Score) and 4% (AUC) in the configuration of **C3**. The model based on **Naive Bayes** showed a significant increase in the F1-score of 68% and 2% in AUC. **Neural Network** model obtained an increase of 21% and 36% for F1-score and AUC, respectively. Finally, models based on **Random Forest** achieved similar scores to those based on **Decision Tree**, with increases of 37% (F1-Score) and 31% (F1-Score).

Finding 5: Adding trivial refactorings to different classes along with non-trivial refactorings resulted in a more effective model. This suggests that including trivial refactorings is important for improving the prediction of new refactorings.

For **C2**, For C2, we added file features before passing through a trivial or non-trivial refactoring in one class of the machine learning algorithm, while in the other class, we kept the features of the corresponding files with the refactoring already performed. The dataset has grown by 45%, adding 232,468 rows.

Considering the F1-score and AUC, we obtained similar results. The combination **C2** showed an increase of only 6% for both metrics, using the **Decision Tree** model. We also can observe that no increase in the metrics was observed for the model based on **Logistic Regression**. The same happened with the model based on **Naive Bayes**, but with a loss in the F1-score of 1%. The Neural Network model lost 6% of F1-score and gained 1% of AUC. Finally, Random Forest had a slight increase of 6% in both F1-score and 5% in AUC.

Finding 6: Combining trivial and non-trivial refactorings in the same class does not change the results significantly. This indicates that the presence of trivial refactorings to be positive for refactoring prediction will depend on how they are combined in the dataset.

Implications of RQ_2. 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.

4.3 Effectiveness of data balancing techniques in predicting trivial and non-trivial refactorings (RQ_3)

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, we observed a significant increase only in contexts that received trivial refactorings. The outliers are presented in Table 7.

By applying the Undersampling and Oversampling balancing techniques in the **C0** context, which does not have trivial refactorings, we observed that the results obtained were little significant or negative in all algorithms. In Table 7, we highlight that the technique of Undersampling had a significant negative impact on the F1-score of the **Logistic Regression** model, with a worsening of 16%. In the other models, the variation of worsening was from 0% to 4% in the F1-score. In the same way with the Oversampling technique, we obtain the same negative value for models based on **Logistic**

Regression in F1-score, with a negative variation between 1% and 5%.

In the other contexts - **C1**, **C2**, and **C3** - we have observed a worsening in almost all algorithms. The context **C2** stood out negatively, using the **Neural Network** algorithm, with a loss of 24% in the F1-score using Undersampling and 49% in the Oversampling of the data.

On the other hand, the model based on the **Decision Tree** algorithm stood out positively in the **C1** context, with F1-score increasing by 33% in both Undersampling and Oversampling. Similarly, AUC (Area Under the Curve) also increased by 32% with Undersampling and 33% with Oversampling. Furthermore, Table 7 presents the model based on the **Naive Bayes** algorithm increased its F1-score by 46% with the applied techniques.

> **Finding 7:** For our problem, balancing the dataset up or down usually keeps the same result or makes the model worse.

The algorithm **Navie Bayes** in the **C1** context obtained a lower result, with a recall of 12% and an F1-score of 20%. However, when applying the Undersampling and Oversampling balancing techniques in this context, the model obtained a significant improvement of 81% and 46% in recall and F1-score, respectively.

Furthermore, the models based on the **Naive Bayes** algorithm in the **C3** context showed good recall and F1-score indices, with 95% and 80%. respectively. However, we observed a worsening with the use of balancing techniques, in which the use of Undersampling and Oversampling resulted in a worsening of 15% and 14% in the F1-score, respectively.

> **Finding 8:** The model with the worst results in the **C1** context obtained the best use of the balancing techniques.

We also observed that the model based on **Logistic Regression** obtained the worst results when applying data balancing techniques. In the **C0** context, using the undersampling technique resulted in a 44% and 16% reduction in recall and F1-Score, respectively. Similarly, using the oversampling technique resulted in a reduction of 45% and 16% in recall and F1-Score, respectively.

In the **C1** context, the reduction in recall and F1-Score were 11% and 4% with undersampling and 15% and 6% with Oversampling. In the case of the **C2** context, the reduction was even more significant, with a worsening of 56% and 23% in recall and F1-Score with the use of Undersampling and 48% (recall) and 18% (F1-Score) of worsening in the use of Oversampling. Finally, in the **C3** context, the reduction was 26% and 18% with undersampling and 23% and 17% with oversampling in the values of recall and F1-Score, respectively.

> **Finding 9:** The Logistic Regression algorithm was the one that deteriorated the most with the use of balancing techniques.

Implications of RQ$_3$. 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.

4.4 Generalization of the best model in other data context domain (RQ$_4$)

To answer RQ$_4$, 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**.

Table 9, shows the values of the differences of the metrics obtained from the model trained with data from the base dataset applied in dataset D2 and D3 with the values obtained from the models trained in the same data domain of D2 and D3. All models showed low or no variation, by presenting values between 0% and 5%, except for the **Neural Network** model which showed significant variation. Additionally, the values obtained from the variation of the **Neural Network** models trained in the base dataset and applied in D2 were 17% in the F1-score and 41% in the AUC metric.

Similarly, a variation was observed in D3, in which was obtained at 8% in F1-score and 37% in AUC for less. The values of the AUC metrics were the ones that most distanced themselves from the values obtained by the models when trained in the domain itself.

> **Finding 10:** Most of the models trained by the base dataset obtained satisfactory results when generalized to other domain contexts.

Table 8 presents the values obtained from the application of the data balancing techniques in D2 and D3. We can observe that in D2, the models that underwent the Undersampling technique obtained the worst results. Models based on **Decision Tree** had a negative decrease of -1% in the F1-score metric and -1% in the AUC.

The **Logistic Regression** based models also had a negative decrease of -17% in the F1-score metric, but a 6% increase in AUC. Additionally, those based on **Naive Bayes** also had a negative decrease of -9% in the F1-score metric and maintained the same value in the AUC metric. Furthermore, the models based on **Neural Network** obtained a negative decrease of -5% in the F1-score metric and -1% in the AUC.

However, only the **Random Forest** based models showed even a small improvement with the balancing technique, with an increase in the AUC metric by +1%. Furthermore, still in D2, but with the Oversampling technique, the results were

very similar, with a slight loss of the F1-score metric of -4% in the models based on **Neural Network**.

In D3, with the same balancing technique, the models obtained slightly different results when compared to D2. The **Decision Tree** based models obtained a negative decrease of -4% in the F1-score metric and an increase of +1% in the AUC. Similarly, the **Logistic Regression** based models also obtained a negative decrease of -25% in the F1-score metric and an increase of +8% in the AUC. In the case of **Naive Bayes** based models, we also observed a negative decrease of -19% in the F1-score metric and an increase of +1% in the AUC metric. Furthermore, the **Neural Network** based models obtained a negative decrease of -11% in the F1-score metric and -2% in the AUC. Finally, those **Random Forest** based models obtained a -5% decrease in the F1-score metric and +1% increase in AUC.

We also observed that in the same dataset (D3), the Oversampling technique obtained similar results, with emphasis on the **Random forest** based models, which presented an increase in the AUC by +3% and a decrease of the F1-score to -3%. Additionally, in the case of **Neural Network** based models, we observed an increase in AUC of +3% and a decrease of F1-score to -7%.

Similarly, the balancing technique applied to the models trained with the base dataset and applied to the D2 and D3 datasets obtained little variation, between -6% and 7%. Except in the case of **Neural Network** based models. In D2, the generalized models based on the Neural Network algorithm obtained a difference between the model of the domain itself of -17% for the F1-score (Table 9). This difference increased to -27% with the use of the Undersampling technique and to -33% with the Oversampling technique. Similarly, on D3, the values obtained for models based on Neural Network were -8% in F1-score. In the case of the Undersampling technique, was obtained -23% (F1-score), and Oversampling with -31% (F1-score).

Finding 11: The balancing techniques applied to models from other domains generated negative or little positive results.

Implications of RQ$_4$. 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.

5 Related Work

Similar strategies to those used in this work have been investigated in the literature, exploring how machine-learning techniques detect refactoring opportunities in various contexts.

Aniche et al. (2020) conducted a large-scale empirical study on 11,149 projects to verify the effectiveness of ML algorithms in predicting refactoring recommendations. The authors identified that supervised algorithms are effective in predicting refactorings. The results indicate that the resulting models achieved accuracy greater than 90%, and models based on Random Forest performed better than the others.

AlOmar et al. (2021) conducted a study using 800 projects to understand what motivates developers to apply refactorings. As a basis for the study, the comments on the commits made by developers were used. For this, the authors used supervised ML with multi-class models defining categories for types of refactorings. The authors identified that the developers' motivation to refactor is not only to correct code smell, it is also motivated by error correction, changes in requirements, optimization of the design structure and improvement in quality attributes. In addition, the authors identified the most commonly used textual patterns in refactorings.

AlOmar et al. (2022) conducted a study to analyze the relationship between documentation and refactoring operations, comparing two distinct approaches. They performed text mining on commit messages, extracting keywords that best represent the type of refactoring. Then, they trained multiclass classification models to predict the types of refactoring. The results indicate that each type of refactoring operation presents a different complexity for prediction.

Nyamawe (2022) used ML with commit history to predict refactorings. The author implements a binary classifier to predict the need for refactorings and a multi-label classifier to recommend refactoring. The author's results suggest that leveraging confirmation messages significantly improved the accuracy of recommending refactorings.

Panigrahi et al. (2020) conducted a study in which they proposed models based on Naive Bayes classifiers (Gaussian, Multinomial and Bernoulli) to predict method-level software refactorings. In addition, the authors used techniques such as *SMOTE*, *UPSAMPLE* and *RUSBOOTS* for data balancing. The results indicate that among the classifiers, Naives Bayes and Bernaulli are the ones that provide greater precision compared to the others used in the study.

Sheneamer (2020) proposes a study that presents a learning method that automatically extracts features from detected code clones and uses classification models, such as Random Forest, ForestPA, Bagging, and K-nearest neighbors, to advise developers on the need and type of refactoring that a clone requires.

Peruma et al. (2020) conducted a study on 800 Java projects, with the aim of identifying the influence of changes in data types on the structure and meaning of a renaming refactoring. The authors did not group the refactorings and found that some developers with little experience tend to perform only renaming refactorings instead of other types of refactorings. Based on these results, the authors seek to offer improvements in support for renaming recommendations.

Kumar et al. (2019b) conducted a study aimed at developing classifier models capable of predicting the need for method-level refactoring. To achieve this, five systems were used to build the dataset, and 25 code metrics extracted from SourceMeter software at the method level were used as features for supervised learning. Additionally, balancing techniques such as SMOTE, UPSAMPLE, and RUSBoost were employed.

Other studies use ML to adopt unsupervised learning to detect (Alkhalid et al., 2010; Bryksin et al., 2018) refactorings. Alkhalid et al. (2010) uses the Adaptive K-Nearest Neighbor (A-KNN) algorithm to recommend method-level refactoring, providing suggestions to improve

Table 6. 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

Table 7. Performance of algorithms in contexts

	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

cohesion. Bryksin et al. (2018) uses various clustering algorithms to try to improve the Move Method refactoring operation recommendation. They implemented a plugin for the IntelliJ IDEA integrated development environment.

Reflection about our contribution. Machine learning-based prediction of refactorings has been extensively studied from various perspectives. However, classifying refactorings into trivial and non-trivial categories based on the level of code change and their effect on prediction is a novel approach. To investigate this, we conducted a study analyzing over 1 million refactorings across 1,291 software projects and developed the concept of trivial refactoring to quantify their impact on predicting other refactorings.

6 Threats to Validity

This section discusses threats to the validity of the study according to the classification of Wohlin et al. (2012).

Internal validity. In our study, we used RefactoringMiner (Tsantalis et al., 2020), a high-precision tool to detect refactoring opportunities in commits, Pydriller (Spadini et al., 2018) to extract source code from files and CK-Tool (Aniche, 2015) to obtain code metrics of files involved in refactorings. Despite the high accuracy, these tools may still fail during the process of mining. To mitigate this problem, we have repeated some steps of the process when necessary. Additionally, to find out the impact that trivial refactorings have on the prediction of other refactorings, we grouped the features of a set of refactoring operations in the same class and this can cause a drop in the performance of the models.

External validity. Despite a large number of projects

Table 8. Result of generalization in the best context

	None					Under					Over				
Alg	M	D2	D1_D2	D3	D1_D3	M	D2	D1_D2	D3	D1_D3	M	D2	D1_D2	D3	D1_D3
Decision	**acc**	0.94	0.91	0.92	0.89	**acc**	0.94	0.90	0.91	0.85	**acc**	0.94	0.92	0.92	0.89
	pre	0.95	0.92	0.94	0.92	**pre**	0.94	0.90	0.91	0.85	**pre**	0.94	0.92	0.93	0.90
	rec	0.95	0.94	0.95	0.94	**rec**	0.94	0.91	0.90	0.84	**rec**	0.94	0.92	0.92	0.89
	f1	0.95	0.93	0.95	0.93	**f1**	0.94	0.91	0.91	0.85	**f1**	0.94	0.92	0.92	0.89
	auc	0.94	0.91	0.90	0.85	**auc**	0.94	0.90	0.91	0.85	**auc**	0.94	0.92	0.92	0.89
Logistic	**acc**	0.62	0.62	0.75	0.75	**acc**	0.58	0.58	0.59	0.58	**acc**	0.58	0.57	0.59	0.59
	pre	0.62	0.62	0.75	0.75	**pre**	0.58	0.58	0.58	0.59	**pre**	0.58	0.56	0.58	0.57
	rec	0.96	0.97	0.99	0.99	**rec**	0.58	0.57	0.62	0.57	**rec**	0.58	0.58	0.66	0.68
	f1	0.75	0.76	0.85	0.85	**f1**	0.58	0.57	0.60	0.58	**f1**	0.58	0.61	0.62	0.62
	auc	0.52	0.52	0.51	0.51	**auc**	0.58	0.58	0.59	0.58	**auc**	0.58	0.57	0.59	0.59
Navie	**acc**	0.60	0.61	0.74	0.74	**acc**	0.51	0.51	0.51	0.51	**acc**	0.51	0.51	0.51	0.51
	pre	0.61	0.61	0.75	0.74	**pre**	0.50	0.50	0.50	0.50	**pre**	0.50	0.50	0.50	0.50
	rec	0.94	0.94	0.98	0.98	**rec**	0.94	0.94	0.96	0.96	**rec**	0.94	0.93	0.97	0.96
	f1	0.74	0.74	0.85	0.85	**f1**	0.65	0.65	0.66	0.66	**f1**	0.65	0.65	0.66	0.66
	auc	0.51	0.51	0.50	0.50	**auc**	0.51	0.51	0.51	0.51	**auc**	0.51	0.51	0.51	0.51
Neural	**acc**	0.82	0.56	0.84	0.69	**acc**	0.91	0.49	0.77	0.49	**acc**	0.82	0.49	0.82	0.51
	pre	0.83	0.61	0.85	0.75	**pre**	0.91	0.49	0.77	0.49	**pre**	0.80	0.49	0.82	0.51
	rec	0.89	0.81	0.94	0.88	**rec**	0.80	0.60	0.79	0.61	**rec**	0.84	0.48	0.82	0.51
	f1	0.86	0.69	0.89	0.81	**f1**	0.81	0.54	0.78	0.55	**f1**	0.82	0.49	0.82	0.51
	auc	0.90	0.49	0.87	0.50	**auc**	0.89	0.49	0.85	0.49	**auc**	0.90	0.49	0.90	0.50
Random	**acc**	0.94	0.91	0.92	0.89	**acc**	0.84	0.91	0.90	0.85	**acc**	0.94	0.92	0.92	0.90
	pre	0.95	0.92	0.94	0.91	**pre**	0.94	0.90	0.91	0.85	**pre**	0.94	0.92	0.93	0.90
	rec	0.95	0.94	0.95	0.95	**rec**	0.93	0.91	0.90	0.86	**rec**	0.94	0.92	0.91	0.89
	f1	0.95	0.93	0.95	0.93	**f1**	0.94	0.91	0.90	0.85	**f1**	0.94	0.92	0.92	0.89
	auc	0.93	0.91	0.89	0.84	**auc**	0.94	0.91	0.90	0.85	**auc**	0.94	0.92	0.92	0.90

Table 9. 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

(1,291) and code refactorings analyzed (1,995,738), different results can be obtained depending on the domain of systems in terms of: (i) programming language, (ii) maintainability, (iii) used programming paradigm, or (iv) software quality. Regarding data balancing, we emphasize that the unbalance of instances and refactorings accounted in each dataset can negatively influence the result during the process of predicting refactorings. To mitigate this threat, we applied balancing techniques in different contexts: Oversampling strategy with SMOTE and Random Undersampling. However, there was an unbalance in the generalization, as the models trained on the base dataset had the lowest proportion and were compared to the models trained on the datasets with the highest proportions.

Construct validity. One threat to validity may be the size of the dataset chosen for the study. This size was chosen based on previous studies on refactorings (Aniche et al., 2020; AlOmar et al., 2021; Peruma et al., 2020). However, we do not know if it is the right size to find the best solution to our problem. This, finding the solution with different dataset sizes may yield more efficient results. Another important threat refers to the metrics used to build the dataset. However, we have used well-known metrics in the literature: accuracy, precision, recall, F1-score, and Area Under the Curve metrics. In addition, it is necessary to investigate and systematize the choice of metrics based on the object-oriented paradigm.

Conclusion validity. We investigated the effect of trivial refactorings on the prediction of non-trivial ones. To identify how the former affects the latter, data from files involved in both types of refactorings are used and tested on the same and different classes in the prediction models. This relationship may cause some bias in the results at the prediction time. This may affect our conclusion.

7 Concluding Remarks

Our study investigated how trivial (class-level) refactorings can affect the prediction of non-trivial refactorings across attributes and code metrics. Our experiment was carried out on 1,291 open-source projects and used the following algorithms as a supervised learning technique to create classifiers: Decision Tree, Logistic Regression, Navies Bayes, Random Forest and Neural Network. Our study also used two data balancing techniques: Random Oversampling and SMOTE Oversampling. We grouped refactorings according to their triviality and proposed contexts based on combinations of refactoring types. In addition, we separated the datasets to identify possible generalizations of the models.

Our main findings: (i) 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. However, only the first two are able to reach a good generalization in other data domain contexts of refactoring; (ii) separating trivial and non-trivial refactorings into different classes still results in a more efficient model, even on different datasets; and (iii) 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. (iv) We understand that a possible explanation for the performance improvements when ”trivial refactorings” are included is that the machine learning models have increased knowledge of what is not non-trivial refactoring, thus improving their prediction.

In future work, we intend to: (i) Create a triviality index that best defines a trivial refactoring operation and quantifies that triviality; (ii) identify other attributes and metrics that can produce more efficient results for predicting refactorings; (iii) perform an in-depth investigation of other algorithms that may perform better in predicting refactorings; and, (iv) investigate how models that predict trivial refactorings impact the detection of refactorings performed by automated solutions.

Acknowledgements

This work is partially supported by the Cearense Foundation of Scientific and Technological Support (FUNCAP) grant BP5-00197-00042.01.00/22, and by the National Council for Scientific and Technological Development (CNPq) grant 404406/2023-8.

References

Aggarwal, K., Singh, Y., Kaur, A., and Malhotra, R. (2006). Empirical study of object-oriented metrics. *J. Object Technol.*, 5(8):149–173.

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.

Al Dallal, J. (2012). Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *Information and Software Technology*, 54(10):1125–1141.

Alkhalid, A., Alshayeb, M., and Mahmoud, S. (2010). Software refactoring at the function level using new adaptive k-nearest neighbor algorithm. *Advances in Engineering Software*, 41(10-11):1160–1178.

Alkhalid, A., Alshayeb, M., and Mahmoud, S. A. (2011). Software refactoring at the package level using clustering techniques. *IET software*, 5(3):274–286.

AlOmar, E. A., Liu, J., Addo, K., Mkaouer, M. W., Newman, C., Ouni, A., and Yu, Z. (2022). On the documentation of refactoring types. *Automated Software Engineering*, 29(1):1–40.

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. (2015). *Java code metrics calculator (CK)*. Available in https://github.com/mauricioaniche/ck/.

Aniche, M., Maziero, E., Durelli, R., and Durelli, V. (2020). The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, pages 1–1.

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.

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.

Bavota, G., Oliveto, R., De Lucia, A., Antoniol, G., and Guéhéneuc, Y.-G. (2010). Playing with refactoring: Identifying extract class opportunities through game theory. In *2010 IEEE International Conference on Software Maintenance*, pages 1–5. 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.

Bishop, C. M. and Nasrabadi, N. M. (2006). *Pattern recognition and machine learning*, volume 4. Springer.

Bryksin, T., Novozhilov, E., and Shpilman, A. (2018). Automatic recommendation of move method refactorings using clustering ensembles. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 42–45.

Carvalho, D. V., Pereira, E. M., and Cardoso, J. S. (2019). Machine learning interpretability: A survey on methods and metrics. *Electronics*, 8(8).

Cassell, K., Andreae, P., and Groves, L. (2011). A dual clustering approach to the extract class refactoring. In *SEKE*, pages 77–82.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357.

Chicco, D. and Jurman, G. (2020). The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC ge-*

nomics, 21(1):1–13.

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

Cutler, A., Cutler, D. R., and Stevens, J. R. (2012). Random forests. In *Ensemble machine learning*, pages 157–175. Springer.

Davis, J. and Goadrich, M. (2006). The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240.

de Mello, R., Oliveira, R., Uchôa, A., Oizumi, W., Garcia, A., Fonseca, B., and de Mello, F. (2022). Recommendations for developers identifying code smells. *IEEE Software*, 40(2):90–98.

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.

Du Bois, B., Demeyer, S., and Verelst, J. (2004). Refactoring-improving coupling and cohesion of existing code. In *11th working conference on reverse engineering*, pages 144–151. IEEE.

Eposhi, A., Oizumi, W., Garcia, A., Sousa, L., Oliveira, R., and Oliveira, A. (2019). Removal of design problems through refactorings: are we looking at the right symptoms? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 148–153. IEEE.

Fernandes, E., Chávez, A., Garcia, A., Ferreira, I., Cedrim, D., Sousa, L., and Oizumi, W. (2020). Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology*, 126:106347.

Hanley, J. A. and McNeil, B. J. (1982). The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143(1):29–36.

Hasanin, T. and Khoshgoftaar, T. (2018). The effects of random undersampling with simulated class imbalance for big data. In *2018 IEEE international conference on information reuse and integration (IRI)*, pages 70–79. IEEE.

Jin, W., Li, Z. J., Wei, L. S., and Zhen, H. (2000). The improvements of bp neural network learning algorithm. In *WCC 2000-ICSP 2000. 2000 5th international conference on signal processing proceedings. 16th world computer congress 2000*, volume 3, pages 1647–1649. IEEE.

Jordan, M. I. and Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260.

Jupyter, P. (2022). Notebook jupyter. `https://jupyter.org/`.

Khanam, Z. (2018). Analyzing refactoring trends and practices in the software industry. *International Journal of Advanced Research in Computer Science*, 10(5).

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

Kumar, L., Lal, S., Goyal, A., and Murthy, N. B. (2019a). Change-proneness of object-oriented software using combination of feature selection techniques and ensemble learning techniques. In *Proceedings of the 12th Innovations on Software Engineering Conference (formerly known as India Software Engineering Conference)*, pages 1–11.

Kumar, L., Satapathy, S. M., and Murthy, L. B. (2019b). Method level refactoring prediction on five open source java projects using machine learning techniques. In *Proceedings of the 12th Innovations on Software Engineering Conference (Formerly Known as India Software Engineering Conference)*, ISEC'19, New York, NY, USA. Association for Computing Machinery.

Lorenz, M. and Kidd, J. (1994). *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc.

Malhotra[1], R. and Chug, A. (2012). Software maintainability prediction using machine learning algorithms. *Software engineering: an international Journal (SeiJ)*, 2(2).

Martin Fowler, K. B. (2000). *Refactoring: Improving the Existing Code Design*. Bookman Co., Inc., 1st edition.

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

Mohammed, R., Rawashdeh, J., and Abdullah, M. (2020). Machine learning with oversampling and undersampling techniques: overview study and experimental results. In *2020 11th international conference on information and communication systems (ICICS)*, pages 243–248. IEEE.

Moreo, A., Esuli, A., and Sebastiani, F. (2016). Distributional random oversampling for imbalanced text classification. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 805–808.

Murphy-Hill, E., Parnin, C., and Black, A. P. (2011). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18.

Muschelli III, J. (2020). Roc and auc with a binary predictor: a potentially misleading metric. *Journal of classification*, 37(3):696–708.

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*. University of Illinois at Urbana-Champaign.

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.

Padhy, N., Panigrahi, R., and Baboo, S. (2015). A systematic literature review of an object oriented metric: Reusability. In *2015 International Conference on Computational Intelligence and Networks*, pages 190–191.

Paixão, M., Uchôa, A., Bibiano, A. C., Oliveira, D., Garcia, A., Krinke, J., and Arvonio, E. (2020). Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 125–136.

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. IEEE.

Panigrahi, R., kuanar, S. K., and Kumar, L. (2020). Application of naïve bayes classifiers for refactoring prediction at the method level. In *2020 International Conference on Computer Science, Engineering and Applications (ICCSEA)*, pages 1–6.

Peruma, A., Mkaouer, M. W., Decker, M. J., and Newman, C. D. (2020). Contextualizing rename decisions using refactorings, commit messages, and data types. *Journal of Systems and Software*, 169:110704.

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*, SBCARS '22, page 81–90, New York, NY, USA. Association for Computing Machinery.

Quinlan, J. R. (2014). *C4. 5: programs for machine learning*. Elsevier.

Rish, I. et al. (2001). An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46.

Sellitto, G., Iannone, E., Codabux, Z., Lenarduzzi, V., Lucia, A., Palomba, F., and Ferrucci, F. (2021). Toward understanding the impact of refactoring on program comprehension.

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

Sheneamer, A. M. (2020). An automatic advisor for refactoring software clones based on machine learning. *IEEE Access*, 8:124978–124988.

Silva, D., Tsantalis, N., and Valente, M. T. (2016a). 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.

Silva, D., Tsantalis, N., and Valente, M. T. (2016b). 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.

Smiari, P., Bibi, S., Ampatzoglou, A., and Arvanitou, E.-M. (2022). Refactoring embedded software: A study in healthcare domain. *Information and Software Technology*, 143:106760.

Spadini, D., Aniche, M., and Bacchelli, A. (2018). PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA. ACM Press.

Tabassum, N., Namoun, A., Alyas, T., Tufail, A., Taqi, M., and Kim, K.-H. (2023). Classification of bugs in cloud computing applications using machine learning techniques. *Applied Sciences*, 13(5).

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. IEEE.

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

Tsantalis, N., Ketkar, A., and Dig, D. (2020). Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Yamashita, A. and Moonen, L. (2012). Do code smells reflect important maintainability aspects? In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 306–315. IEEE.