How do Trivial Refactorings Affect Classification Prediction Models?

Darwin Pinheiro
Federal University of Ceará
Quixadá, Brazil
darwinfederal@alu.ufc.br

Carla Bezerra
Federal University of Ceará
Quixadá, Brazil
carlailane@ufc.br

Anderson Uchôa
Federal University of Ceará
Itapajé, Brazil
andersonuchoa@ufc.br

ABSTRACT

Refactoring is defined as a transformation that changes the internal structure of the source code without changing the external behavior. Keeping the external behavior means that after applying the refactoring activity, the software must produce the same output as before the activity. The refactoring activity can bring several benefits, such as: removing code with low structural quality, avoiding or reducing technical debt, improving code maintainability, reuse or readability. In this way, the benefits extend to internal and external quality attributes. The literature on software refactoring suggests carrying out studies that invest in improving automated solutions for detecting and correcting refactoring. Furthermore, few studies investigate the influence that a less complex type of refactoring can have on predicting more complex refactorings. This paper investigates how less complex (trivial) refactorings affect the prediction of more complex (non-trivial) refactorings. To do this, we classify refactorings based on their triviality, extract metrics from the code, contextualize the data and train machine learning algorithms to investigate the effect caused. Our results suggest that: (i) machine learning with tree-based models (Random Forest and Decision Tree) performed very well when trained with code metrics to detect refactorings; (ii) separating trivial from non-trivial refactorings into different classes resulted in a more efficient model, indicative of improving the accuracy of automated solutions based on machine learning; and, (iii) using balancing techniques that increase or decrease samples randomly is not the best strategy to improve datasets composed of code metrics.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Empirical software validation**; • **General and reference → Measurement**.

KEYWORDS

refactoring, machine learning, software quality

ACM Reference Format:
Darwin Pinheiro, Carla Bezerra, and Anderson Uchôa. 2022. How do Trivial Refactorings Affect Classification Prediction Models?. In *16th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS 2022), October 3–4, 2022, Uberlandia, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3559712.3559720

1 INTRODUCTION

During software maintenance, developers can introduce code with low structural quality to be introduced by developers unintentionally or unintentionally [41]. Over time, these low-quality codes end up degrading the code quality, which can lead to crashes in the future [60]. One solution that can solve this problem is to apply transformations to the source code, one type of such transformation is called refactoring [51]. Opdyke [39] introduced the term refactoring, but it only became popular with the book of Martin Fowler [32]. Refactoring is defined as a transformation that changes the internal structure of the source code without changing the external behavior [32]. Keeping the external behavior means that after applying the refactoring activity, the software should produce the same output as it did before the activity.

Researchers have investigated different perspectives for the use of refactoring [9, 20, 33]. Among them: (i) solutions that recommend refactorings for developers [10, 56]; (ii) challenges in applying refactoring [28, 49]; (iii) developers' motivation to refactor the code [43, 51]; and (iv) machine learning-based refactoring detection [6, 8, 38]. Using predictive machine learning (ML) models to assist developers in combating or minimizing design problems is a relatively new area [9]. Some studies use machine learning to detect refactorings through supervised learning [4, 6, 8, 38, 46]. Others investigate refactorings using unsupervised learning [3, 13].

Despite the large number of studies investigating ML as a solution for refactoring activity [3, 6, 8, 13, 38, 44, 46], few works investigate strategies on how to improve the prediction of refactorings by these models. Azeem et al. [9] conducts a literature review and points out that there is room for studies to investigate how ML can be used to detect design problems.

Thus, to fill this gap, this study investigates the influence of less complex refactorings on predicting other refactorings through a supervised learning classification method. For this, we performed the following steps: (i) we selected 884 open-source projects; (ii) classified the refactorings into trivial and non-trivial based on changes in the source code; and (iii) we extracted code metrics from the files involved in operations refactoring to be used as features in supervised learning algorithms to investigate how trivial refactorings can influence the prediction of other refactorings. Our main contributions are:

(1) A study that identifies the effect of less complex refactorings on predicting other refactorings using supervised learning binary classification; and
(2) A better-investigated strategy for improving refactoring detection tools, investigating the combination of features based on the triviality of refactoring operations.

Our main findings show that: (i) separating the less complex from the more complex refactorings into different classes results in a more efficient model, indicative of improving the accuracy of automated solutions based on supervised learning; and, (ii) tree-based ML models such as Random Forest and Decision Tree performed very well when trained with code metrics to detect refactorings.

The rest of this article is organized as follows. Section 2 presents the main concepts used in this article, based on refactoring and machine learning. Section 3 presents a list of related previous studies. In Section 4, we present the research objectives and questions that guided our study. Section 5 describes the phases of the methodology used: selection of repositories, mining of features, selection of contexts, training of models and prediction and analysis of data. In Section 6 we present the results found in this study and Section 7 the discussions of this work. We then discuss threats to the validity of our experiment in Section 8. Finally, Section 9 presents our work's conclusion and main findings.

2 BACKGROUND

2.1 Trivial and Non-Trivial Refactorings

Code refactoring is a current practice of software development [28, 37, 52]. Code refactoring was defined by Martin Fowler [32] as a disciplined technique for structuring an existing source code, changing its internal structure without changing the functional behavior of the system. In other words, code refactoring can be understood as applying transformations to the code structures to enhance software maintainability [28, 37]. 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 [52, 58]. 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 [57].

Some studies classify or group refactorings according to their general purpose [22, 48, 54]. Other studies prefer simplifying a binary classification into refactored and unrefactored [21, 38]. AlOmar et al. [6] classifies your refactorings into: internal, external, fix bug, and fix the smell. Other than that, Sellitto et al. [48] groups 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 [32] 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. 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 Superclass operation which aims to separate the responsibilities of a class. The procedure consists of creating a new class and moving responsible attributes and methods. We can perform this operation when: (i) a class does not have a clear responsibility; and (ii) when a subset of attributes and methods appear to form a new set [32]. 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 [32], 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 only changes the name of the refactored class, changing only a single line. Also, it is not possible to split it into other refactoring operations.

In our study, we 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 [27] and more studies of refactorings that prioritize the class level [2]. Additionally, we split these 13 types of transformations into two groups: 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 machine learning (ML) techniques frequently used in literature [5, 6, 8, 38, 44]. 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 [45].
- **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 [12].
- **Naive Bayes (NB)**: a probabilistic classifier based on the application of Bayes' theorem [25]. This technique is highly scalable and completely disregards the correlation between the variables in the training set. It is considered a simple technique to train and fast [47].
- **Neural Network (NN)**: a technique commonly used for deep learning and designed to solve problems that other techniques cannot solve. NNs are made up of many interconnected layers, so when data passes through these layers the NN can approximate calculations to transform input into outputs [24].
- **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

Table 1: Group of Trival and Non-trivial Refactorings used in this study [32]

Group	Refactoring	Problem	Solution
Trivial	Add Class Annotation	When an annotation is needed	Add an annotation
	Add Class Modifier	When it is no longer necessary to use the 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 Class	When a class does nothing or has no responsibilities	Move data from one class to another class. The class you merge is removed and all references are updated
	Move Class 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

algorithms are among the most accurate in many machine learning problems [17].

2.2.1 Machine Learning Metrics. To evaluate ML algorithms' performance, metrics are necessary to measure the quality of a model [14]. 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).

- **Accuracy**: It is the proportion of correctly classified observations among the total observations. The accuracy indicates the overall performance of the model [19].
- **Precision**: Is the proportion of positive observations correctly classified among the predicted positive observations. Accuracy is measured when FP is considered more harmful than FN [14].
- **Recall**: Is the proportion of positive observations correctly classified among true positive observations. Recall is measured when NFs are considered more harmful than FPs [14].
- **F1-score**: It is the harmonic mean between the precision and the recall [15].

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	$2\times\frac{Precision\times recall}{Precision+recall}$

2.3 Code Quality Metrics

Software metrics are used to measure and understand software structure. Chidamber and Kemerer [16] 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 [1, 31, 42].

Our study selected a set of metrics from the CK suite proposed by Chidamber and Kemerer [16], LOC [30] and different attributes of the code elements, such as: number of methods, number of returns, number of variables, etc. The values were extracted using the CK tool [7] 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 RELATED WORK

In the literature, studies are found that point to strategies similar to those used in this work and that investigate, in different contexts, how ML techniques detect refactorings.

Aniche et al. [8] 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 had better performance than the others.

AlOmar et al. [6] 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.

Nyamawe [38] 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. [44] 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, they are the ones that provide greater precision compared to the others used in the study.

Table 3: Metrics and attributes of code elements used in this work [16, 30]

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	

Other studies use ML to adopt unsupervised learning to detect [3, 13] refactorings. Alkhalid et al. [3] uses the Adaptive K-Nearest Neighbor (A-KNN) algorithm to recommend method-level refactoring, providing suggestions to improve cohesion. Bryksin et al. [13] uses various clustering algorithms to try to improve the recommendation of the Move Method refactoring operation. They implemented a plugin for the IntelliJ IDEA integrated development environment.

Reflection about our contribution. The prediction of refactorings using machine learning has been studied extensively from different perspectives. However, it is something new to classify refactorings into trivial and non-trivial based on their level of code change and the effect it has on prediction. For this, we carried out a study analyzing 84,262 refactorings of 884 software projects and introduced the concept of the triviality of refactorings to measure the effect on the prediction of other refactorings.

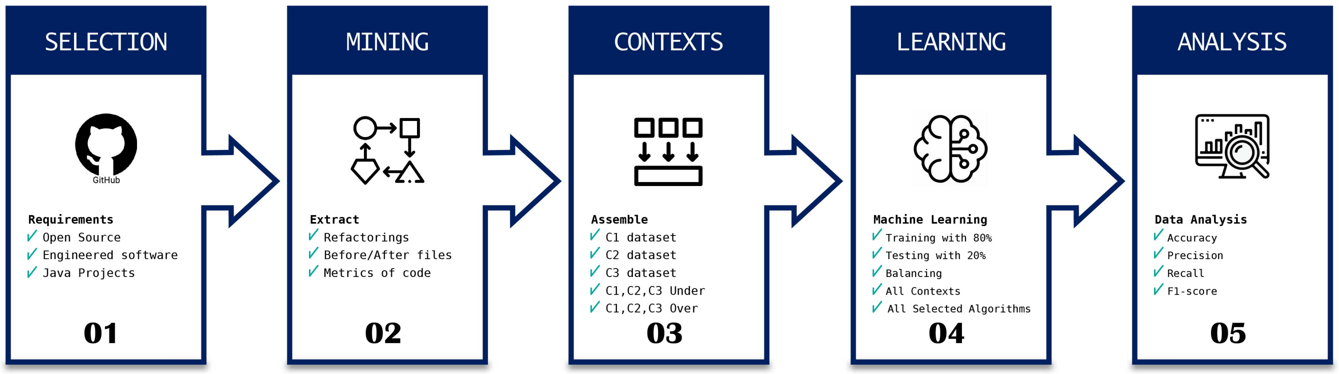

Figure 1: Overview of the research methodology

4 STUDY SETTINGS

4.1 Goal and Research Questions

This study investigates the influence of trivial class-level refactorings on the prediction of non-trivial refactorings. Our choice was made due to the need for studies that prioritize this level [2]. Models based on ML algorithms trained with code metrics were used to understand this influence. Thus, knowing how trivial refactorings affect the prediction, it is possible to discover strategies that improve the prediction of refactorings through supervised learning.

Kaur and Singh [26] and Singh and Kaur [53] point out that studies are needed that promote automated solutions involving refactoring in their results. One strategy currently used to detect refactorings is ML. However, there is still a gap in studies that explore this strategy to solve design problems [9]. In addition, it is also pointed out as necessary studies that show the accuracy of the refactoring detection tools [29].

We divide the goals as follows: (i) Build a dataset based on the code metrics of the selected projects to serve as a basis for training ML models; (ii) Configure ML models to obtain the best performance; and (iii) Analyze the results to identify how trivial refactorings can affect non-trivial ones. To this end, we detail our three research questions (RQ$_s$) as follows.

RQ$_1$: *What is the performance of ML algorithms to predict non-trivial refactorings?* – **RQ$_1$** aims at investigating the use of ML algorithms to predict non-trivial refactorings. Thus, we compare the performance of five ML algorithms: Decision Tree, Logistic Regression, Naive Bayes, Neural Network, and Random Forest. We chose these algorithms, since they provide an intuitive and easy to explain model [11, 40]. For this purpose, we train each algorithm using source code metrics extracted from classes that have undergone non-trivial refactorings as features.

RQ$_2$: *How effective is the inclusion of trivial refactorings to predict non-trivial refactorings?* – **RQ$_2$** aims at evaluating and comparing the performance of ML algorithm to predict non-trivial refactorings by considering different sets combining trivial and non-trivial refactorings. These combinations are further explained in Section 5.

RQ$_3$: *How effective are data balancing techniques in the prediction of trivial and non-trivial refactorings?* – Complementary to the previous research question the **RQ$_3$** aims at evaluating the effectiveness in the prediction of trivial and non-trivial refactorings of two data balancing techniques: Random Over Sampler and Under Sampler for imbalanced data. We evaluate each technique by considering the different sets mentioned in RQ$_2$.

5 DATA COLLECTION AND ANALYSIS

Figure 1 overviews the sequence of five phases followed to answer our RQs: (1) Selection of software projects; (2) Refactoring and feature mining; (3) Selection of contexts; (4) Training and testing ML-based models; and (5) Evaluation of results. We describe each step as follows. We used a dataset with the refactorings and software metrics collected for the 884 software projects [18]. An overview of the sample projects used in our study with commits and refactorings can be seen in Table 4.

Table 4: Overview of the sample projects

Dataset	Total
Number of Projects	884
Number of Commits	35,838
Number of Refactorings	84,262

5.1 Phase 1: Selection of software projects

The first phase consisted of choosing open-source projects. For our study, we needed to gather a large number of open-source projects to allow the study replication. For this purpose, we selected 884 software projects from a dataset composed of *engineered* software projects extracted from GitHub. This dataset was proposed by Munaiah et al. [36]. Such projects were cataloged by the authors based on evidence of the use of solid software engineering practices, such as: *collaboration*; *continuous integration*; *quality*; *maintainability*; *sustained evolution*; *project management*; *responsibility* and *unit testing* [36]. The evidence mentioned earlier in the repositories brings greater applicability to our research as refactoring is a widely used practice in software engineering.

Our procedure was to take Java projects classified as *engineered.* We focused on Java projects because Java is a very popular programming language, and it was also targeted by related studies [5, 6, 8, 38]. Furthermore, we also selected projects in Java due to the availability of tools to identify refactorings [50, 56, 59] and collect software metrics [7]. Thus, we selected Java projects with diverse structures, sizes and popularity. The replication package [18] contains their detailed information.

5.2 Phase 2: Refactoring and feature mining

This phase consisted of extracting data of refactorings and code metrics (used as features) for all selected projects. This phase is composed of three steps: (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 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 [59]. This tool is applied between two versions (commits) and returns the elements that changed from one version to the other. It also returns the refactoring type associated with the change. The tool detected a total of 84,262 refactoring types used in our study (Section 2.1). After the code refactoring extraction, we divided the refactorings into two groups as described in (Section 2.1).

Step 2: Tracking the modified files before and after refactorings. In order 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 [55], and a python script to summarize the 767,698 files involved in refactorings.

Step 3: Extracting code metrics for tracked files. In this step, 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 [7] 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. After the data collection, we split the generated dataset into two parts. For all fields calculated by the tool, after previous data analysis, we decided to use only 45 metrics and attributes as features for our dataset. They can be seen in Table 3.

Table 5: Trivial and Non-Trivial Refactorings

Type	Name	Qty
Trivial	Add Class Annotation	926
Trivial	Add Class Modifier	346
Trivial	Change Class Access Modifier	418
Trivial	Modify Class Annotation	407
Trivial	Remove Class Annotation	578
Trivial	Remove Class Modifier	196
Trivial	Rename Class	1209
	Total Trivial	**4,080**
Non-Trivial	Extract Class	357
Non-Trivial	Extract Subclass	28
Non-Trivial	Extract Superclass	218
Non-Trivial	Merge Class	27
Non-Trivial	Move Class	5045
Non-Trivial	Move and Rename Class	430
	Total Non-Trivial	**6,105**
	Total	**10,185**

5.3 Phase 3: Selection of contexts

In this step, we create several datasets with different combinations of refactoring types. We called context each combination in this work. The first context (**C1**) is defined by combining features from files that have undergone trivial and non-trivial refactoring operations. The second context (**C2**) is the combination of the features of the files before being refactored (trivial and non-trivial) with the features of the refactored files (trivial and non-trivial). Finally, the third context (**C3**) is the combination of file features before and after undergoing trivial refactorings.

We emphasize that every context had the presence of trivial refactorings, as we sought to investigate how they can affect the prediction of non-trivial refactorings.

5.4 Phase 4: Training and testing the models

In this phase, we used the datasets constructed by the combinations **C1**, **C2** and **C3** created in the previous phase to predict refactorings. All contexts have been tested, with some changes to the processing pipeline. In this phase, 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 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[1], while the Neural Network was used through TensorflowKeras[2]. After training, each generated model was validated by predicting refactorings of the features in the test set.

Furthermore, we consider it a problem where the prediction of ML algorithms can be negatively affected by an unbalanced dataset (number of different samples between classes). So for each combination, we applied two balancing techniques: Random Under Sampler and Random Over Sampler. These balancing techniques were chosen because they are commonly used in recent studies [23, 35] and because they facilitate the comparison of efficiency when used together [34]. The balancing techniques were applied individually by context **C1**, **C2** and **C3**, creating six more combinations: **C1 Under**, **C2 Under**, **C3 Under**, **C1 Over**, **C2 Over** and **C3 Over**.

5.5 Phase 5: Evaluation

Finally, we calculated accuracy, precision, recall, and F1-score metrics to evaluate the trained models and compared the results by context. Next, we observed: (i) whether trivial refactorings affect the prediction of other refactorings; (ii) which algorithm obtained better results; and (iii) whether data balancing techniques had any effect. With this, it was possible to answer our research questions. We present the results in the next section.

6 RESULTS

In this section, we report our results. We present an overview of metrics calculation for the contexts mentioned in Table 6. In the following subsections, we answer each of the RQs.

[1]https://scikit-learn.org/stable/
[2]https://www.tensorflow.org/api_docs/python/tf/keras

Table 6: Results of the different ML models after trained and tested

	Decision Tree				Logistic Regression				Navies Bayes				Neural Network				Random Forest			
Context	**Acc**	**Pr**	**Re**	**F1**	**Acc**	**Pr**	**Re**	**F1**	**Acc**	**Pr**	**Re**	**F1**	**Acc**	**Pr**	**Re**	**F1**	**Acc**	**Pr**	**Re**	**F1**
C1	0.85	0.78	0.73	0.75	0.70	0.68	0.10	0.17	0.67	0.44	0.07	0.12	0.82	0.78	0.60	0.68	0.85	0.78	0.74	0.76
C2	0.58	0.59	0.60	0.59	0.51	0.51	0.93	0.66	0.49	0.51	0.04	0.07	0.54	0.52	0.95	0.67	0.58	0.58	0.62	0.60
C3	0.52	0.53	0.51	0.52	0.5	0.5	0.96	0.66	0.49	0.50	0.12	0.19	0.50	0.50	0.65	0.57	0.53	0.54	0.57	0.55
C1 Under	0.84	0.82	0.87	0.84	0.60	0.62	0.58	0.60	0.52	0.63	0.11	0.19	0.80	0.80	0.81	0.80	0.83	0.82	0.86	0.84
C1 Over	0.85	0.83	0.89	0.86	0.61	0.63	0.55	0.59	0.52	0.63	0.11	0.19	0.81	0.80	0.83	0.81	0.85	0.83	0.88	0.85
C2 Under	0.85	0.60	0.50	0.55	0.50	0.50	0.56	0.53	0.50	0.51	0.07	0.12	0.52	0.86	0.06	0.11	0.58	0.60	0.50	0.55
C2 Over	0.58	0.60	0.50	0.55	0.50	0.50	0.53	0.51	0.50	0.50	0.75	0.60	0.51	0.89	0.02	0.05	0.58	0.59	0.53	0.56
C3 Under	0.52	0.53	0.50	0.51	0.50	0.50	0.95	0.66	0.49	0.50	0.12	0.19	0.51	0.53	0.27	0.36	0.53	0.54	0.57	0.55
C3 Over	0.52	0.53	0.50	0.51	0.50	0.50	0.93	0.65	0.49	0.50	0.12	0.19	0.51	0.51	0.90	0.65	0.53	0.54	0.57	0.55

6.1 RQ$_1$: What is the performance of ML algorithms to predict non-trivial refactorings?

To answer **RQ$_1$** we combined several datasets and verified the algorithm's performance for predicting trivial refactorings (present in all contexts). In Table 6, we can observe the performance of each ML algorithm by ML metric and the context specified in this study.

In summary, Random Forest achieved the best indexes considering the general average of the contexts, with an average of 0.64, 0.63, 0.64 and 0.63 for the metrics accuracy, precision, recall and f1-score, respectively. The algorithm stood out mainly in the first context and its balance (see Table 6). In **C1** we got 0.85, 0.78, 0.74 and 0.76. In **C1** with Random Undersampling Balancing we got 0.83, 0.82, 0.86 and 0.84. Finally, in **C1** with Random Oversampling Balancing we obtained 0.85, 0.83, 0.88 and 0.85, respectively.

Finding 1: Models based on the Random Forest algorithm were the best in general contexts.

In Table 6, we also observed that the Decision tree in the **C1** with the balancing technique Random Oversampling is equivalent to or better than Random Forest. The best indices were (accuracy, precision, recall, f1-score) 0.85, 0.83, 0.89 and 0.86, against 0.85, 0.83, 0.88 and 0.85 for Random Forest (the best of this work overall).

Finding 2: Decision tree and Random Forest were the algorithms that achieved better results in the **C1** context, using the Random Oversampling balancing technique.

6.2 RQ$_2$: How effective is the inclusion of trivial refactorings to predict non-trivial refactorings?

To answer **RQ$_2$**, we performed some combinations of trivial refactorings with non-trivial refactorings, in which each combination corresponds to a context in our study. In summary, three different contexts were created and two balancing techniques were applied to each of them. The first context (**C1**) is divided into two classes, one is equivalent to trivial refactorings and the other to non-trivial ones. To determine the effectiveness of including trivial refactorings, we compared the result obtained in this context with the result obtained only using non-trivial refactorings.

Looking at Table 6, we can observe that the indexes involving non-trivial refactorings were smaller than the indexes involving **C1**. We used accuracy and F1-Score as key metrics to compare models. Thus, we observed that the Decision Tree model had an increase of 33% and 23%, respectively. Following the same logic, the Logistic Regression model had an increase of 19% in accuracy and a decrease of 38% in the F1-score. The Navies Bayes model showed an increase in accuracy of 18% and a loss of 7% in the F1 score. The Neural Network model recorded an increase of 31% and 18%, respectively. Models based on Random Forest achieved similar marks to those based on Decision Tree with increases of 33% and 22%. Overall, we concluded that we effectively increased all model accuracy.

Finding 3: Separating trivial from non-trivial refactorings into different classes results in a more efficient model. This indicates that the presence of trivial refactorings is important to improve the prediction of new refactorings.

For **C2**, we combined the trivial and non-trivial refactorings in a single class, while the other class has the corresponding unrefactored instances. The dataset has increased by 45%, by adding 232,468 rows. Considering accuracy and F1-score, we obtained similar results. The combination **C2** showed an increase in accuracy of only 6% and an F1-score of 7% for the Decision tree-based model. No accuracy increase and 11% F1-score for the Logistic Regression-based model. Similar was the case with the model based on Navies Bayes, which had no increase in accuracy. However, in the F1-score, there was a loss of 12%. Neural Network increased accuracy by 3% and F1-score by 17%. Finally, Random Forest had a slight 6% increase in both accuracy and F1-score. Therefore, we can conclude that using this context brings little significant results with very similar results.

Finding 4: 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.

In **C3** it was put in a class only trivial refactorings instances while in the other instances your non-refactored correspondents' instances. It was identified as a result of not so important and unnecessary efforts. To all of the used algorithms, accuracy had a variation between 1%. In models based on Decision Tree and Navies Bayes, the F1-score does not change. Models built under Random Forest thinking in other contexts were better. There was only a discreet variation on the F1-score of 1%. The biggest variation was in the models based on Logistic Regression and Neural Network with an improvement between 11% and 7%, respectively.

Finding 5: The use of trivial refactorings by itself to identify non-trivial refactorings does not improve the results. It is implicated that using only trivial refactorings to identify non-trivial is unnecessary.

6.3 RQ$_3$: How effective are data balancing techniques in the prediction of trivial and non-trivial refactorings?

Trying to increase the performance between the models by decreasing the discrepancy among classes, two balancing techniques were adopted: Random Over Sampler and Random Under Sampler. In conclusion, there was a higher increase in F1-Score in several contexts, but the general performance from models measured by accuracy did not change greatly.

When applying the balancing technique Random Over Sampler in the **C1**, **C2** and **C3** contexts it was not possible to get better improvements in accuracy to the group of data used in this study. The way on the other side, the accuracy decreased in other contexts.

Finding 6: To balance the group of data increasing randomly could get results equivalent or negative in the model performance.

For the **C1**, **C2** and **C3** contexts the models based in Decision Tree reached accuracy of 85%, 58% and 52%, respectively; the models Random Forests reached 85%, 58% and 53%, respectively. It was applied the balancing technique Random Over Sampler to increase the samples of these models was not identified great changes. In the models based on Logistic Regression, Navies Bayes and Neural Network the accuracy fell between 9%, 15% and 1%, respectively, to **C1**. Beyond that, there was a fall until 3% in **C2** to the three models above. In **C3** does not had changes.

Finding 7: The context with the bigger number of samples was the only one that greatly improved accuracy with the balancing on the model based on Decision Tree.

C2 combines in a unique class every trivial and non-trivial refactorings. It was the only one that greatly improved with the Random Under Sampler technique applied to a model based on Decision Tree. The model without balancing got an accuracy of 58%, after applying the technique the model obtained 85%, an improvement of 27%.

Finding 8: The balancing technique which increases the number of samples had a great emphasis in models with low F1-Score.

The model based on Logistic Regression in **C1** got 17% of F1-Score. After applying the Random Over Sampler technique the index grows up to 59%, a great improvement of 42%. Another model with a similar result was based on Navies Bayes applied to **C2** got a low F1-Score of 7% after applying the same technique reached 60% on the same metric, a great improvement of 53%. After all, these answers indicate that trivial refactorings predict other refactorings positively when configured in several ways.

7 DISCUSSIONS

This section discusses our findings in future directions for research involving software refactoring.

Improved predictive models. One of the main findings shows that trivial refactorings can increase the efficiency in predicting non-trivial refactorings. This finding points us to a way to improve the efficiency of models that can predict more complex refactorings. In some cases, developers lose confidence when using automated solutions for refactoring because they encounter false positives during the process, thus considering it an impeding and demonizing obstacle for the refactoring activity [21, 49]. Additionally, although some studies provide preditors with high accuracy [8], they still do not guarantee 100% accuracy. It is necessary to investigate ways to increasingly improve the efficiency of these models to motivate their use by developers.

Lesson 1: Failing to improve the efficiency of predictive models can lead to developers losing confidence in using automated solutions involving refactoring activity by developers.

Features for algorithms. In our findings, some ML algorithms performed better than others. This is common, however in our analysis, we noticed that the models with the best performance were Random Forest and Decision Tree. These models work by creating trees with independent feature nodes. It is believed that a better selection and even a weighting of the metrics and attributes of the code elements can increase ML algorithms' efficiency.

Lesson 2: Investigating which features to use can be a way to improve the performance of ML algorithms.

8 THREATS TO VALIDITY

We discuss threats to the study validity as follows.

Internal Validity. In our study, we run tools to detect refactorings in commits, extract source code from commits and extract source code metrics from files involved in refactoring. They have very high accuracy rates and are commonly used in empirical studies but are prone to failure. Therefore, during the phase of extraction and detection of refactorings, some failures suppressed some of the data needed to train the algorithms. However, it was necessary to perform some tasks repeatedly. In addition, only class-level refactorings were chosen for our study, and there may be different results with the inclusion of other refactoring levels. However, our choice was based on the need for further studies involving more of the chosen level. Another issue to consider was establishing a few criteria for choosing the balancing techniques to randomly increase or decrease the samples of the unbalanced data sets. However, it was only considered that any commonly used balancing technique would increase the efficiency of the models.

External Validity. We used 884 open-source projects with a history of 35,838 commits and 84,262 refactorings. Despite a large number of projects and code refactorings analyzed, different results may be obtained if the domain of the systems is different in: (i) programming language; (ii) maintainability; (iii) paradigm used; or (iv) software quality.

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. However, we do not know if it is the right size to find the best solution to our problem. Finding the solution with different dataset sizes may yield more efficient results. Another important threat refers to the metrics used to build the dataset. Some important metrics for evaluating the models were not used, such as: AUC and ROC. 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 time of prediction. This may affect our conclusion.

9 CONCLUSION

In this study, we investigate how trivial class-level refactorings affect the prediction of non-trivial refactorings using ML techniques. To do this we select 884 open-source projects and extract the type of refactoring from classes involved in some operation and code metrics. We grouped the refactorings into trivial and non-trivial ones based on their level of change. Furthermore, we proposed contexts based on combinations of the refactoring types that made it possible to increase the accuracy of supervised learning models.

Our main findings were: (i) ML with tree-based models, such as Random Forest and Decision Tree, performed very well when trained with code metrics to detect refactorings; (ii) separating trivial and non-trivial refactorings into different classes resulted in a more efficient model, indicative to improve the accuracy of ML-based automated solutions; and, (iii) using balancing techniques that increase or decrease samples randomly is not the best strategy to improve datasets composed of code metrics.

In future work we envision: (i) create a triviality index that better defines the type of the refactoring operation and calculates the amount of change of a refactoring operation; (ii) identify which balancing techniques produce more efficient results with the use of code metrics for the prediction of refactorings; (iii) conduct an in-depth investigation of other algorithms and metrics that may perform better in predicting trivial refactorings; and, (iv) investigate how models that predict trivial refactorings impact the detection of refactorings performed by automated solutions.

REFERENCES

[1] KK Aggarwal, Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. 2006. Empirical Study of Object-Oriented Metrics. *J. Object Technol.* 5, 8 (2006), 149–173.
[2] Mansi Agnihotri and Anuradha Chug. 2020. A systematic literature survey of software metrics, code smells and refactoring techniques. *Journal of Information Processing Systems* 16, 4 (2020), 915–934.
[3] Abdulaziz Alkhalid, Mohammad Alshayeb, and Sabri Mahmoud. 2010. Software refactoring at the function level using new Adaptive K-Nearest Neighbor algorithm. *Advances in Engineering Software* 41, 10-11 (2010), 1160–1178.
[4] Eman Alomar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. 2022. On the documentation of refactoring types. *Automated Software Engineering* 29 (05 2022). https://doi.org/10.1007/s10515-021-00314-w
[5] Eman Abdullah AlOmar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. 2022. On the documentation of refactoring types. *Automated Software Engineering* 29, 1 (2022), 1–40.
[6] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane Kessentini. 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 (2021), 114176.
[7] Maurício Aniche. 2015. *Java code metrics calculator (CK)*. Available in https://github.com/mauricioaniche/ck/.
[8] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius Durelli. 2020. The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.3021736
[9] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* 108 (2019), 115–138.
[10] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107 (2015), 1–14.
[11] Massimo Bertolini, Davide Mezzogori, Mattia Neroni, and Francesco Zammori. 2021. Machine Learning for industrial applications: A comprehensive literature review. *Expert Systems with Applications* 175 (2021), 114820.
[12] Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern recognition and machine learning*. Vol. 4. Springer.
[13] Timofey Bryksin, Evgenii Novozhilov, and Aleksei Shpilman. 2018. Automatic recommendation of move method refactorings using clustering ensembles. In *Proceedings of the 2nd International Workshop on Refactoring*. 42–45.
[14] Diogo V. Carvalho, Eduardo M. Pereira, and Jaime S. Cardoso. 2019. Machine Learning Interpretability: A Survey on Methods and Metrics. *Electronics* 8, 8 (2019). https://doi.org/10.3390/electronics8080832
[15] Davide Chicco and Giuseppe Jurman. 2020. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC genomics* 21, 1 (2020), 1–13.
[16] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.
[17] Adele Cutler, D Richard Cutler, and John R Stevens. 2012. Random forests. In *Ensemble machine learning*. Springer, 157–175.
[18] Dataset. 2022. Dataset of code metrics. https://doi.org/10.5281/zenodo.6800385
[19] Jesse Davis and Mark Goadrich. 2006. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd international conference on Machine learning*. 233–240.
[20] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. 2018. A systematic literature review on bad smells–5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering* 47, 1 (2018), 17–66.
[21] Andre Eposhi, Willian Oizumi, Alessandro Garcia, Leonardo Sousa, Roberto Oliveira, and Anderson Oliveira. 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)*. IEEE, 148–153.
[22] Eduardo Fernandes, Alexander Chávez, Alessandro Garcia, Isabella Ferreira, Diego Cedrim, Leonardo Sousa, and Willian Oizumi. 2020. Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology* 126 (2020), 106347. https://doi.org/10.1016/j.infsof.2020.106347
[23] Tawfiq Hasanin and Taghi Khoshgoftaar. 2018. The effects of random undersampling with simulated class imbalance for big data. In *2018 IEEE international conference on information reuse and integration (IRI)*. IEEE, 70–79.
[24] Wen Jin, Zhao Jia Li, Luo Si Wei, and Han Zhen. 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, Vol. 3. IEEE, 1647–1649.

[25] Michael I Jordan and Tom M Mitchell. 2015. Machine learning: Trends, perspectives, and prospects. *Science* 349, 6245 (2015), 255–260.

[26] Satnam Kaur and Paramvir Singh. 2019. How does object-oriented code refactoring influence software quality? Research landscape and challenges. *Journal of Systems and Software* 157 (2019), 110394.

[27] Zeba Khanam. 2018. Analyzing Refactoring Trends and Practices in the Software Industry. *International Journal of Advanced Research in Computer Science* 10, 5 (2018).

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

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

[30] Mark Lorenz and Jeff Kidd. 1994. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc.

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

[32] Kent Beck Martin Fowler. 2000. *Refactoring: Improving the Existing Code Design* (1st ed.). Bookman Co., Inc.

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

[34] Roweida Mohammed, Jumanah Rawashdeh, and Malak Abdullah. 2020. Machine learning with oversampling and undersampling techniques: overview study and experimental results. In *2020 11th international conference on information and communication systems (ICICS)*. IEEE, 243–248.

[35] Alejandro Moreo, Andrea Esuli, and Fabrizio Sebastiani. 2016. Distributional random oversampling for imbalanced text classification. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. 805–808.

[36] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating github for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.

[37] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2011. How we refactor, and how we know it. *IEEE Transactions on Software Engineering* 38, 1 (2011), 5–18.

[38] Ally S. Nyamawe. 2022. Mining commit messages to enhance software refactorings recommendation: A machine learning approach. *Machine Learning with Applications* 9 (2022), 100316. https://doi.org/10.1016/j.mlwa.2022.100316

[39] William F Opdyke. 1992. Refactoring object-oriented frameworks. (1992).

[40] Abdelfettah Ouadah, Leila Zemmouchi-Ghomari, and Nedjma Salhi. 2022. Selecting an appropriate supervised machine learning algorithm for predictive maintenance. *The International Journal of Advanced Manufacturing Technology* (2022), 1–25.

[41] Ali Ouni, Marouane Kessentini, Slim Bechikh, and Houari Sahraoui. 2015. Prioritizing code-smells correction tasks using chemical reaction optimization. *Software Quality Journal* 23, 2 (2015), 323–361.

[42] Neelamadhab Padhy, Rasmita Panigrahi, and Sarada Baboo. 2015. A Systematic Literature Review of an Object Oriented Metric: Reusability. In *2015 International Conference on Computational Intelligence and Networks*. 190–191. https://doi.org/10.1109/CINE.2015.44

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

[44] Rasmita Panigrahi, Sanjay Kumar kuanar, and Lov Kumar. 2020. Application of Naïve Bayes classifiers for refactoring Prediction at the method level. In *2020 International Conference on Computer Science, Engineering and Applications (ICCSEA)*. 1–6. https://doi.org/10.1109/ICCSEA49143.2020.9132849

[45] J Ross Quinlan. 2014. *C4. 5: programs for machine learning*. Elsevier.

[46] Irina Rish et al. 2001. An empirical study of the naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, Vol. 3. 41–46.

[47] Irina Rish et al. 2001. An empirical study of the naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, Vol. 3. 41–46.

[48] Giulia Sellitto, Emanuele Iannone, Zadia Codabux, Valentina Lenarduzzi, Andrea Lucia, Fabio Palomba, and Filomena Ferrucci. 2021. Toward Understanding the Impact of Refactoring on Program Comprehension.

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

[50] Danilo Silva, Joao Silva, Gustavo Jansen De Souza Santos, Ricardo Terra, and Marco Tulio O Valente. 2020. RefDiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering* (2020).

[51] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*. 858–870.

[52] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*. 858–870.

[53] Satwinder Singh and Sharanpreet Kaur. 2018. A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal* 9, 4 (2018), 2129–2151.

[54] Paraskevi Smiari, Stamatia Bibi, Apostolos Ampatzoglou, and Elvira-Maria Arvanitou. 2022. Refactoring embedded software: A study in healthcare domain. *Information and Software Technology* 143 (2022), 106760.

[55] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 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*. ACM Press, New York, New York, USA, 908–911. https://doi.org/10.1145/3236024.3264598

[56] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 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)*. IEEE, 4–14.

[57] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 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)*. IEEE, 4–14.

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

[59] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* (2020), 21 pages. https://doi.org/10.1109/TSE.2020.3007722

[60] Aiko Yamashita and Leon Moonen. 2012. Do code smells reflect important maintainability aspects?. In *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE, 306–315.