On the Employment of Machine Learning for Recommending Refactorings: A Systematic Literature Review

Guisella Angulo Armijo
DC-UFSCar
São Carlos, SP, Brazil
angulogc@gmail.com

Daniel San Martín Santibañez
EIC-UCN
Coquimbo, Chile
daniel.sanmartin@ucn.cl

Rafael Durelli
DCC-UFLA
Lavras, MG, Brazil
rafael.durelli@ufla.br

Valter Vieira de Camargo
DC-UFSCar
São Carlos, SP, Brazil
valtervcamargo@ufscar.br

ABSTRACT

Context and Motivation: Refactoring is a widely recognized technique aimed at enhancing the comprehensibility and maintainability of source code while preserving its external behavior. The widespread adoption of refactorings as a systematic practice is still very dependent on individual expertise and inclination of software engineers. To address this challenge, various approaches have emerged with the objective of automatically suggesting refactorings, thereby alleviating engineers from the manual burden of identifying such opportunities. **Objective:** This study aims to analyze the current landscape of approaches utilizing Machine Learning (ML) for recommending refactorings and discuss their usage. **Method:** A Systematic Literature Review (SLR) was conducted, spanning five scientific databases from 2015 to December 2023. Initially, 177 papers were identified, from which a final set of 27 papers was reached. **Results:** The findings encompass: i) an exploration of the most and least investigated refactorings and ML techniques; ii) an analysis of the datasets used; iii) an examination of the evaluation methodologies employed; and iv) an assessment of recommendation completeness and quality. **Conclusion:** This study has significant potential for further research, as numerous refactorings remain unexplored by existing studies. Furthermore, it highlights that many ML-based approaches fall short in delivering comprehensive recommendations, thus emphasizing the imperative for ongoing investigation and enhancement in this field. All artifacts produced from our research are available on the replication package [1].

KEYWORDS

refactoring recommendation, machine learning

1 INTRODUCTION

Refactoring is the practice of modifying a system's source code to enhance its structure without altering its observable external behavior [19]. The refactoring process encompasses four primary tasks: (i) identifying refactoring opportunities. Example: code smells, architectural smells, software defects and anti-patterns; (ii) identifying one or more refactorings to solve the opportunities; (iii) executing the refactoring(s) by modifying the source code; and iv) ensuring that the external behavior remains unchanged [29, 47].

Identifying refactoring opportunities and deciding which refactoring to apply constitutes the most time-consuming task of the process. Developers must possess the ability to recognize situations where a specific refactoring could be applied and also to choose the adequate refactorings to apply.

In recent years research on machine learning (ML) in the context of refactoring recommendations has grown [14] [8] [31] [15] [54] [11], mostly concentrating on the support for identification of refactoring opportunities. At the same way, many approaches have used ML in the context of smells identification [16] [21] [3] [12]. Although research on smells identification and refactoring recommendations share similarities, they also exhibit significant differences. Firstly, research on smell identification has a narrower focus, concentrating solely on identifying code smells, without considering other potential refactoring opportunities. Secondly, the datasets used for training are also specially tailored just for identifying smells, not being applicable for refactoring recommendations. Thirdly, the main emphasis of smells identification approaches [21] [3] [12] lies in the accuracy of the classifier, usually a smaller attention is given to the refactorings that could be applied to solve the smells.

Although the number of research on recommendation of refactorings based on ML has attacked attention in recent years, there is still a noticeable lack of knowledge on how ML approaches have been adopted for refactoring recommendation and whether there are points of improvement to allow a better recommendation. Our goal is to organize, analyze and explore the current state of the art in this domain. Such a study would enable researchers to gain a clear understanding of this field and identify new research opportunities.

In this paper we present a SLR on how ML has been applied for recommending refactorings [23]. Initially, we extracted a total of 177 papers from the most typical online digital libraries, covering the period from 2015 to December 2023. After applying the exclusion and inclusion criteria and backward and forward snowballing, we got a final set containing 27 papers.

Our SLR aims at providing a comprehensive investigation to elaborate: i) Identification of the most/least studied refactorings taken into account by previous research; ii) the types of ML classifiers exploited by researchers; iii) an analysis and a classification of the datasets used by the approaches; iv) strategies used to evaluate the machine learning models and the identification of the automation level of the approaches; and iv) a discussion on the quality of the recommendations considering their completeness based on W3B (Which, Where, Why and Benefits), a criteria we have proposed.

Section 2 presents the research methodology and the SLR protocol. In Section 3, we present the obtained results. Section 4 discusses the relevant findings. Section 5 addresses the potential threats to validity. Section 6 shows the works related to this work. Finally, in Section 7, we present our conclusions and future works.

2 RESEARCH METHODOLOGY

The focus of this SLR is on studies that have investigated the use of ML for recommending refactorings. Our focus is exclusively on primary studies whose goal is to suggest or recommend refactorings. Studies employing ML solely for aiding in the identification of problematic situations are out of the scope of our investigation. Our inclusion criteria encompass only those studies wherein the ultimate outcome is a recommendation of refactoring. Our SLR followed the guidelines proposed by Kitchenham et al. [22] as shown in Figure 1.

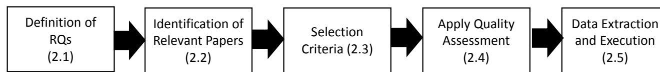

Figure 1: Process defined for SLR

2.1 Research Questions

To guide our investigation, we have formulated three general research questions and five specific ones [23].

RQ1 *Which relationships between Machine Learning and Refactorings Recommendations are explicitly discussed in the literature?*

RQ1.1 *Which refactorings have been investigated?*

RQ1.2 *Which ML algorithms have been used to recommend refactorings?*

RQ1.3 *How datasets and features can be classified?*

RQ2 *Do the way the approaches are evaluated depend on the automation level of them?*

RQ2.1 *How have the approaches been evaluated?*

RQ2.2 *What is the automation level (fully automated, semi-automated, or manual) of the approaches?*

RQ3 *Have the approaches concerned with the quality of the recommendations?*

2.2 Identification of Relevant Papers

Figure 2 shows the base string elaborated around three terms: (*i*) Recommendation; (*ii*) Refactoring, and (*iii*) Machine Learning. This base string was adapted considering alternative spellings and synonyms for each of the 5 digital libraries: Scopus, IEEE Xplore, ACM, Science Direct and Wiley. Therefore, we restricted the search to the period between 2015 and 2023 and, to avoid missing important/relevant articles, we conducted a backward snowball technique using reference lists from the final set of articles.

Figure 2: Search string

```
("Recommendation" OR "Recommend" OR "Recommending" OR
"Identification" OR "Identify" OR "Identifying" OR
"Prediction" OR "Predict" OR "Predicting" OR "Prevision")
AND ( "Refactoring" OR "Refactor") AND ( "Machine Learning"
OR "Supervised Learning" OR "Unsupervised Learning")
```

2.3 Selection Criteria

We have established two inclusion criteria (IC) and six exclusion criteria (EC):

- IC-1: The study elaborates on the use of ML for recommending refactoring.
- IC-2: The research is published in English.
- EC-1: The study does not address Machine Learning;
- EC-2: The study does not address Recommendation;
- EC-3: The study does not address Refactoring;
- EC-4: The study is a secondary study;
- EC-5: The study is not available;
- EC-6: The study is an Abstract, poster, technical report, thesis, book, conference review, or patent.

2.4 Quality assessment

The quality of publications was measured after the final selection process. The following checklist was used to assess the credibility and thoroughness of the selected publications.

(1) Does the paper have a well-defined approach?
(2) Is the refactoring(s) proposed by the approach clearly defined?
(3) Is the machine learner classifier clearly defined?
(4) Does the paper include empirical or theoretical validation?

For each paper, the quality score was calculated by assigning [0, 0.5, 1] to each of the four questions and then adding them up. The artifact result is available on the replication package [1].

2.5 Data Extraction and Execution

The following data we have extracted from the final set of papers:

- Papers metadata: title, authors, publication venue, year, pages, volume, abstract and document type;
- Refactoring researched in the approach;
- Machine Learning techniques employed;
- Dataset characteristics and feature details;
- Evaluation strategy of the approach;
- Automation level of the approach;
- Quality of the Recommendation.

The process for execution the research can be observed in Figure 3. Each stage displays the criteria used and the resulting quantity, culminating in a final set of 27 papers.

2.6 Results

The data analysis was conducted considering *Approaches* and not the papers individually. This was done because some papers belong to the same research group, being just evolution of a same approach. In this case, we grouped them under a unique Approach (Referred as A#) - see the first column of the table. Therefore, although there are 27 papers in the final set, we have 22 approaches. The papers [S04], [S12], and [S19] were grouped as the approach [A04], the papers [S05] [S17] and [S20] were grouped as the approach [A05] and papers [S18] and [S21], were grouped as [A16].

We classified the recommendations into two types: i) unique refactoring recommendation (UniR) and ii) sequence of refactorings recommendations (SeqR). The first type is recommendations that suggest just one refactoring at a time. The second one offers a

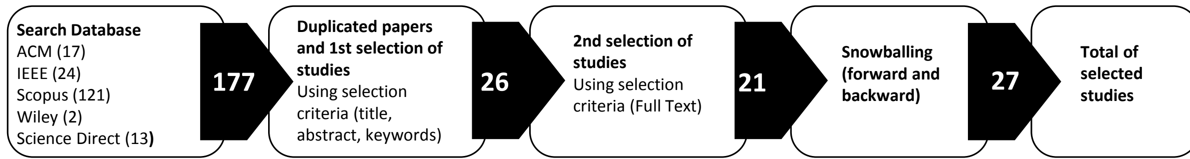

Figure 3: Filtering Process of the Papers

Table 1: The Final Set of Primary Studies

Approach	Paper ID	Paper Title	Authors	Year	Type	Venue
[A01]	[S01]	REMS: Recommending Extract Method Refactoring Opportunities via Multi-view Representation of Code Property Graph	Cui Di et al. [14]	2023	UniR	IEEE ICPC
[A02]	[S02]†	Automatic Refactoring Candidate Identification Leveraging Effective Code Representation	Palit et al. [33]	2023	UniR	ACM
[A03]	[S03]	Just-in-time code duplicates extraction	AlOmar et al. [8]	2023	UniR	Inf Softw Technol
[A04]	[S04]	Mining commit messages to enhance software refactorings recommendation: A machine learning approach	Nyamawe [30]	2022	SeqR	MLWA
	[S12]	Feature requests-based recommendation of software refactorings	Nyamawe et al. [32]	2020	SeqR	Empir. Softw. Eng.
	[S19]	Automated recommendation of software refactorings based on feature requests	Nyamawe et al. [31]	2019	SeqR	RE
[A05]	[S05]	Enabling Decision and Objective Space Exploration for Interactive Multi-Objective Refactoring	Rebai et al. [39]	2022	SeqR	IEEE TSE
	[S17]	Less is more: From multi-objective to mono-objective refactoring via developer's knowledge extraction	Alizadeh et al. [5]	2019	SeqR	SCAM
	[S20]	Reducing Interactive Refactoring Effort via Clustering-Based Multi-objective Search	Alizadeh et al. [6]	2018	SeqR	ASE
[A06]	[S06]	RMove: Recommending Move Method Refactoring Opportunities using Structural and Semantic Representations of Code	Cui Di et al. [15]	2022	UniR	ICSE
[A07]	[S07] †	Class-Level Refactoring Prediction by Ensemble Learning with Various Feature Selection Techniques	Panigrahi et al. [35]	2022	Undef	Applied Sciences
[A08]	[S08]	A Machine Learning Approach to Software Model Refactoring	Brahmaleen et. al. [42]	2022	UniR	IJCA
[A09]	[S09]	A probabilistic-based approach for automatic identification and refactoring of software code smells	Saheb et al. [40]	2022	UniR	Appl. Soft Comput.
[A10]	[S10]	Data-Driven Extract Method Recommendations: A Study at ING	Van der Leij et al. [50]	2021	UniR	ACM
[A11]	[S11]	The effectiveness of supervised machine learning algorithms in predicting software refactoring	Aniche et al. [9]	2020	UniR	IEEE
[A12]	[S13]	Recommendation of Move Method Refactoring Using Path Based Representation of Code	Kurbatova et al. [27]	2020	UniR	ICSEW
[A13]	[S14] †	Application of Naive Bayes classifiers for refactoring Prediction at the method level	Panigrahi et al. [36]	2020	UniR	ICCSEA
[A14]	[S15] †	Harnessing deep learning algorithms to predict software refactoring	Alenezi et al. [4]	2020	Undef	Telkomnika
[A15]	[S16] †	An Automatic Advisor for Refactoring Software Clones Based on Machine Learning	Sheneame [41]	2020	UniR	IEEE Access
[A16]	[S18]	Method Level Refactoring Prediction on Five Open Source Java Projects Using Machine Learning Techniques	Kumar et al. [25]	2019	Undef	ISEC
	[S21]	Application of SMOTE and LSSVM with various kernels for predicting refactoring at method level	Kumar et al. [24]	2018	Undef	ICONIP
[A17]	[S22]	Automatic Clone Recommendation for Refactoring Based on the Present and the Past	Yue et al. [54]	2018	UniR	ICSME
[A18]	[S23] †	Deep Learning Based Feature Envy Detection	Liu et al. [28]	2018	UniR	ASE
[A19]	[S24]	A log-linear probabilistic model for prioritizing extract method refactorings	Xu et al. [52]	2017	UniR	ICCC
[A20]	[S25]	GEMS: An Extract Method Refactoring Recommender	Xu et al. [53]	2017	UniR	ISSRE
[A21]	[S26]	Finding Extract Method Refactoring Opportunities by Analyzing Development History	Imazato et al. [20]	2017	UniR	COMPSAC
[A22]	[S27]	Application of LSSVM and SMOTE on Seven Open Source Projects for Predicting Refactoring at Class Level	Kumar and Sureka [26]	2017	Undef	APSEC

† Papers added in the snowballing stage UniR:Unique refactoring recommendation SeqR: Sequence of refactorings recommendations Undef: Undefined by the authors

sequence of them, for example: *Apply the following refactorings: Extract Class, after Extract Method and after Move Method.*

[A01] presents REMS, which recommends a unique refactoring, the Extract Method, to improve the internal structure of the method and avoid the introduction of code smells. [A02] which is an approach that recommends a unique refactoring, is focused on identifying the refactoring opportunity for applying the Extract Method refactoring. [A03] introduces AntiCopyPaster, a tool that recommends a unique refactoring, the Extract Method, to avoid the introduction of duplicate code into the source code. The approach [A04] can recommend different source-code structures to be refactored following a sequence order. The goal is to prepare the source code for the inclusion of new requirements. [A05] has the capacity to recommend refactorings at attributes, methods, and classes in a sequence order to improve quality attributes QMOOD (reusability, flexibility, understandability, extendibility, and effectiveness). [A06] presents RMove, which recommends a unique refactoring, the Move Method refactoring, to remove the Feature Envy code smell. [A07] identifies the refactoring opportunity, recommending classes for refactoring, promoting cost-effective and consistent software design before writing the source code.

[A08] recommends a unique refactoring, for correcting flaws at UML classes. The goal of the approach is to improve the design quality. [A09] recommends a unique refactoring: removing relations, classes and adding classes. To be applied in class diagram, represented in a probabilistic Bayesian network, for correcting six different code smells at a higher level of granularity. [A10] which is an approach that recommends a unique refactoring to improve the quality of the source code. The approach focuses on the recommendation of the extract Method in the context of ING, a large financial organization. [A11] shows an approach that recommends unique refactoring at class, method, and variable levels. The goal is to investigate the effectiveness of machine learning algorithms in predicting software refactorings. [A12] proposes an approach to recommend a unique refactoring, the Move Method. The approach relies on the path-based representation of code and aims at improving class cohesion. [A13] proposes an approach to recommend a unique refactoring. The approach aims at solving duplicated code and long method code smell for improving the source code quality. [A14] which is an approach that recommends classes to be refactored. This is focused on exploring the effectiveness of deep

learning algorithms in building refactoring prediction models at the class level to improve software quality.

[A15] presents an approach to recommend unique refactoring for solving code clone smells. The approach focuses on three types of refactorings: Move method, Pull up Method, and Extract Method. [A16], which is an approach that recommends Methods to be refactored. This is focused on exploring the effectiveness of ten supervised machine learning models in building refactoring prediction at the method-level to improve software quality. [A17] introduces a ML-based approach called CREC, that recommends a unique refactoring for solving code clones. The approach is focused on the recommendation of the Extract Method and it points out the clones that need to be refactored, improving the software maintenance. [A18] shows a deep learning-based approach that recommends unique refactoring. This approach focuses on recommending the Move Method to identify and remove feature envy code smell. [A19] is an approach that recommends a unique refactoring, the Extract Method. This approach is based on probabilistic techniques for improving the software maintainability. [A20] introduces a tool called GEMS that recommends a unique refactoring. This approach provides a ranking of refactoring opportunities to apply the Extract Method aiming at improving quality attributes of source code. [A21] is an approach that recommends a unique refactoring and has as goal the recommendation of the Extract Method for improving the software maintainability. [A22] presents an approach that recommends a unique refactoring. This approach based on the analysis of 102 metrics aims at improving the structure of source code.

Table 2: Refactorings addressed by the selected papers

Fowler Refactorings		
Refactorings	**#N**	**Approaches (A#)**
Extract Function/ Extract Method	13	[A01],[A02] [A03], [A04], [A05], [A10], [A11], [A13], [A15], [A19], [A17], [A20], [A21]
Move Function/ Move Method	8	[A04],[A05], [A06], [A08],[A11], [A12], [A15], [A18]
Pull Up Method	6	[A04], [A05], [A08],[A11], [A13], [A15]
Extract Superclass	4	[A04], [A05], [A08], [A11]
Push Down Method	4	[A04], [A05], [A08], [A11]
Rename Function /Rename Method	4	[A04], [A05], [A08], [A11]
Move Field	3	[A04], [A05], [A08]
Pull Up Field	3	[A04], [A05], [A08]
Push Down Field	3	[A04], [A05], [A08]
Extract Class	3	[A05], [A08], [A11]
Replace Type Code with Subclasses / Extract Subclass	3	[A05], [A08], [A11]
Encapsulate Variable /Encapsulate Field / Self-Encapsulate Field	2	[A04], [A05]
Inline Function	2	[A04], [A11]
Rename Field	1	[A08]
Introduce Parameter Object	1	[A13]
Preserve Whole Object	1	[A13]
Replace Function with Command / Replace Method with Method Object	1	[A13]
Replace Temp with Query	1	[A13]
Substitute Algorithm	1	[A13]
Extract Variable	1	[A11]
Inline Variable	1	[A11]
Rename Variable	1	[A11]
Refactorings proposed by other authors (non-Fowler Refactorings)		
Refactorings	**#N**	**Approaches (A#)**
Rename Class	2	[A04], [A11]
Extract Interface	2	[A04], [A11]
Move Class	2	[A08], [A11]
Increase-Decrease Field Security	1	[A05]
Increase-Decrease Method Security	1	[A05]
Extract Associated Class	1	[A08]
Decomposition objects	1	[A07]
Removing Relations	1	[A09]
Removing Classes	1	[A09]
Adding Classes	1	[A09]

3 ANSWERS FOR RESEARCH QUESTIONS

RQ1 - *Which relationships between Machine Learning and Refactorings Recommendations are explicitly discussed in the literature?*
We divided RQ1 in three sub-questions RQ1.1, RQ1.2 and RQ1.3.
RQ1.1 - *Which refactorings have been investigated?*

Table 2 shows the 32 refactorings addressed in the final set of papers; 22 were proposed by Fowler and 10 by other authors. Some refactorings are known by different names, so they are grouped and separated by a bar. As can be seen, *Extract Function* (aka *Extract Method*) and the *Move Function* (aka *Move Method*) are the two most investigated refactorings, researched by 13 and 8 approaches respectively. According to the authors, Extract Method is widely used in literature to solve different issues like reducing code duplication, removing the Long Method and Feature Envy code smell. In this regart, Silva et al. [44] stated that Extract Method is the most versatile refactoring that serves for 11 different purposes.

Besides the most investigated refactorings, it is also important to be aware of the least investigated ones. Out of Fowler's catalog of 67 refactorings, 45 did not appear in our final set, which accounts for 67% of the catalog. If we expand this analysis by counting the refactorings that do not appear plus the ones that appear just once (nine ones), this percentage grows to 81%, corresponding to 54 refactorings from 67. Therefore, a lot of effort is concentrated on just two or three refactorings, and around 54 are left out.

It is difficult to precise about why some refactorings are little researched. Some reasons can be the unpopularity of such refactoring or the lack of metrics/tools able to detect them. Regardless of the reason, it is clear that some refactorings are more challenging to recommend than others. For example, the refactorings Rename Class and Introduce Parameter Object are very little investigated. In the case of Introduce Parameter Object, the reason for being little research may be the lack of low-level metrics (parameter level) able to detect/characterize the problem. The analysis should reveal that a set of method parameters should be encapsulated in a new class. This requires a semantic analysis to discover which are the parameters and whether they make sense or not by putting them as fields of a new class; a challenging task.

The approaches [A07], [A14], [A16], and [A22] have been omitted from Table 2 because they do not provide specific Fowler/ non-Fowler refactorings. These approaches identified components (Classes or Methods) and recommended them to be refactored.

RQ1.2- *Which ML algorithms have been used to recommend refactorings?*

Table 3 presents the ML models and algorithms that appear in the final set of papers. Remarkably, the most employed learning process is Supervised Learning involving 28 algorithms. Answering the RQ1.2, the most used algorithms are: Random Forest (RF) researched by 11 approaches; Logistic Regression (LR) researched by 9 approaches; and Support Vector Machine (SVM) researched by 8 approaches. Note that some approaches have employed more

than one algorithm. For example, the approach [A01] makes use of 9 different Supervised learning algorithms.

The motivation behind the election of some algorithms over others is not clear. Some authors explain their choices, for [A01], the authors selected the 9 algorithms because they considered that these models are lightweight and efficient, which can further be integrated into IDE for interactive refactoring by users.

In the case of [A04], the authors select the algorithms (SVM, RF, LR, and DT) because of the classification problem addressed and the effectiveness of the algorithms. For [A06], the authors employed the SVM because of its effectiveness in code smell detection problems.

In the same way, the authors of [A13] use only Naive Bayes because it over performance other supervised ML algorithms with less training data. Finally, in the case of [A03], the authors claim that the Convolutional Neural Network (CNN) proved to be overcome at selecting useful features and building complex mapping from input to output automatically.

Table 3: Algorithms used by the selected papers

L.P.	Algorithm	#A	Approach
Supervised	Random Forest (RF)	11	[A01], [A02], [A04], [A06], [A10], [A11], [A15], [A16], [A17], [A19], [A22]
	Logistic Regression (LR)	9	[A01], [A04], [A16], [A06], [A07],[A10], [A11], [A20], [A22]
	Support Vector Machine (SVM)	8	[A01], [A04], [A06], [A10], [A11], [A12], [A20], [A22]
	Naïve Bayes (NB)	7	[A01], [A16], [A06], [A10], [A11], [A13], [A17]
	Decision Tree (DT)	6	[A01], [A04], [A06], [A07], [A10], [A11],
	Least-squares support-vector machine (LSSVM)	4	[A16], [A07], [A22], [A22]
	K-nearest neighbors (KNN)	4	[A01], [A07], [A15], [A20]
	Bayesian Network (BN)	4	[A09], [A16], [A22], [A22]
	Gradient Descent (GD)	3	[A16], [A06], [A08]
	Gradient boosting classifier	2	[A19], [A20]
	AdaBoost	2	[A16], [A17]
	Extreme Gradient Boosting	2	[A01], [A06]
	Multinomial naive bayes (MNB)	1	[A04]
	Bagging	1	[A15]
	LogitBoost	1	[A16]
	J48	1	[A22]
	C4.5	1	[A17]
	Sequential minimal optimization	1	[A17]
	ForestPA	1	[A15]
	Convolutional Neural Network [1]	6	[A01], [A03], [A04], [A06], [A11], [A18]
	Artificial Neural Network (ANN)[1]	2	[A16], [A07]
	Levenberg Marquardt Algorithm[1]	2	[A16], [A06]
	Radial Basis Function Network [1]	2	[A16], [A06]
	Gated Recurrent Units Recurrent Neural Network (GRU) [2]	2	[A06], [A14]
	Long Short Term Memory Recurrent Neural Network (LSTM) [2]	2	[A01], [A06]
	Deep neural network model [2]	1	[A08]
	Extreme Learning Machine [2]	1	[A07]
	Multilayer Perceptron [1]	1	[A16]
Un-supervised	Clustering Ensembles with Pareto-front	1	[A05]
	Density-based algorithm	1	[A15]
	Clustering + Algorithm NSGA-II	1	[A05]
	Clustering + Genetic Algorithm	1	[A05]

[1] Artificial Neural Network [2] Deep Learning Models

Regarding the most researched algorithms. There are many reasons why Random Forest (RF) is one of the most-used algorithms mainly due to its simplicity and diversity. In the context of refactoring recommendations, where large volumes of information, characteristics, and diversity of domains are necessary, RF has demonstrated its robustness to deal with high dimensional data and correlation between features, besides presenting further strong advantages by its ability to deal with outliers.

RQ1.3 -*How datasets and features can be classified?*

To answer this RQ we performed two analyses. The first provided a classification for the datasets and the second provided a classification for the features of the datasets. In the first we classified the datasets in two types:

1) Code Smell-based (CS): the instances/samples of the dataset are methods/classes that have a code smell. These instances are extracted from the current version of the project, so previous project commits are not considered. This type makes the ML model able to identify just code smell issues.

2) Refactoring-based (RB): The model trained with methods/classes sample that underwent refactorings in the past commits. Thus, the trained instances remain in the project history and are extracted using mining tools. In this case, it is expected that the trained model can identify a broader spectrum of refactoring opportunities; code smells, and any other situation that a refactoring can be applied to.

Table 4 shows in the last column the Dataset type (DT). It is noticeable that 15 approaches (representing more than 68%) use RB-datasets. An interesting point is the pattern between the 15 approaches. Half of them explore the history of the projects for Extract Method refactorings [A01], [A02], [A03], [A10], [A11], [A17], [A19], [A20] and [A21]. On the other hand, the approaches [A04], [A07], [A13], [A14], [A16], and [A22] choose as samples the Methods/Classes that underwent any type of refactoring. These approaches aim at building generic refactoring recommendations.

Table 4 also shows the column called "extraction tool" and "Label" to shed light on the tools used and the additional efforts invested in building the dataset. For the value "NA" (no apply), it means that the authors did not utilize tools/ techniques for extracting the values for the features, i.e, they have used a ready and already populated dataset available in the literature. In the case of [A01], the authors use the Silva dataset [44] and Xu dataset [53]. For [A09], the authors use the Fontana dataset [10] and the Palomba dataset [34]. In the case of [A10], the authors use part of the Aniche et al. dataset [A11]. The approaches [A07], [A13], [A14], [A16] and [A22] use the Dataset Tera-promise, which is a well-known repository in literature composed of source code metrics and refactorings extracted from two successive releases of 7 open source Java projects.

The second analysis was focused on classifying the features of the datasets. Thus, we have identified 5 type of features: **C1 – Model Metrics**. It encompasses metrics extracted from UML diagrams and graph models like Probabilistic graphical model (PGM). For example, for UML metric: number of generalizations, number of associations, and number of classes; for PGM metric: Depth of inheritance tree and number of relations between classes; **C2 – Source Code Metrics**. It is about the well-known metrics of complexity, coupling and cohesion. For example: Lines of code (LOC), response for class (RFC), and coupling between objects (CBO); **C3 – Tool-based information**. It encompasses information extracted from the issue tracker like Jira framework as summary, description and status; and Github information as quantity of commits, commit

Table 4: Dataset features by the selected papers

App.	C1	C2	C3	C4	C5	Metric Extraction Tool	Lbl.	D.T.
[A01]	✓					NA	NA	RB
[A02]				✓		Autoencoder technique	A	RB
[A03]		✓				Non-specified	A	RB
[A04]			✓			Python Natural Language Processing Toolkit/Codacy tool	A	RB
[A06]		✓		✓		SRCML, ASTMiner and DEPENDS tools	A	CS
[A07]		✓				NA	NA	RB
[A08]	✓					SDMetrics tool	M	CS
[A09]	✓					NA	NA	CS
[A10]		✓				NA	NA	RB
[A11]		✓	✓			SourceMeter	A	RB
[A12]		✓		✓		code2vec technique	A	CS
[A13]		✓				NA	NA	RB
[A14]		✓				NA	NA	RB
[A15]		✓				Java Development Tool (JDT)	NS	CS
[A16]		✓				NA	NA	RB
[A17]		✓			✓	MCIDiff	M	RB
[A18]		✓		✓		Distance Metric and Word2vector Technique	A	CS
[A19]		✓				Non-specified	M	RB
[A20]		✓				Proprietary Algorithm for Extracting code features	A	RB
[A21]		✓				H. Murakami technique	A	RB
[A22]		✓				NA	NA	RB

Lbl → A: Automated M: Manual NS: Non-Specified NA: Non-Applied
D.T. → CS: Code Smell - based RB: Refactoring - based

message, and date; **C4 – Semantic information Metrics**. It comprises metrics related to the extraction of relationships between the source code. For example: the relation between the method names and class names; and finally **C5 – Project History Metrics**. Metrics that encompass project versions. The intuition is that a project's evolution history may imply its future evolution. For example: "a percentage of change commits among all commits".

Table 4 shows the type of features in the dataset (C1 to C5). It is remarkable that C2 (Source Code Metrics) emerges as the most prevalent category with 16 out of 22 approaches, representing more than 72%. We claim this category is the most researched because these metrics allow developers to identify code smells and to improve the quality attributes, topics that have been widely investigated in literature. Regarding the C5 group, only 1 approach use the information about the history of the project. Thus, [A18] incorporated this type of metrics because the authors claim that a project evolution history may imply its future evolution. The intuition is that if a component suffers with refactorings repetitively, it may imply this component is a strong candidate to be refactored again.

RQ2-*Does the way the approaches are evaluated depend on the automation level of them?*

This question is broken down in two sub-questions RQ2.1 and RQ2.2 aiming at describing the two main concepts involved: the evaluation method and the automation level of the approaches. So, Table 5 was elaborated to understand how the approaches have performed the evaluation. In the first column there is the Approach ID; from the second to the fifth there are the evaluation types (I, II, III or IV); in the sixth one the abstraction level of the evaluation; the seventh one shows the result of the evaluation for the type of evaluation II and III; the eighth one shows the limitation of the evaluation and the last one shows the Automation level.

RQ2.1-*How have the approaches been evaluated?*

We identified 4 evaluation methods in the final set of papers:
(I) Comparison with others state-of-the-art approaches/tools. The evaluation consists in comparing the performance of the approach with other similar approaches and/or tools;
(II) Evaluation of the classifiers. It is a kind of internal evaluation where the goal is to identify the best classifier among the classifiers used in the approach;
(III) Evaluation of one Classifier. Another kind of internal validation. In this case, the authors evaluated the performance of the only one classifier used in the approach;
(IV) Controlled Experiment. This evaluation involves a group of participants whose goal is to evaluate the usefulness of the approach/tool when compared with other approach/tools.

In Table 5, we can realize that 14 approaches, representing 73% of the total, applied the evaluation type II and III. As was mentioned, this type of evaluation attempts to identify the best classifier. In this regard, the column "Evaluation Result" shows that among all the evaluations the models with better performance are SVM, NB and AdaBoost, despite presenting different evaluation conditions.

We believe that SVM outperforms other classifiers due to the following reasons: i) handling high dimensional, SVM employs overfitting protection and has the ability to learn which can be independent from the dimensionality of the feature space and ii) handling problems with sparse instances and dense concepts.

For type IV, controlled experiments, only five approaches [A01], [A03], [A05], [A06] and [A10] conducted this evaluation method. These experiments focused on evaluating the usability and effectiveness of the proposed tool/approach. In the case of [A01], the authors conducted a study with 10 experienced participants to evaluate the usefulness of the tool called REMS. Thus, the refactoring recommendations generated by different tools are shown for the participants and they must evaluate the recommendations and complete a questionnaire. In the case of [A03], the authors make available the plugin of AntiCopyPaster tool and a video demonstrating how to use it. Then, they ask about the usefulness, usability, and functionality of the proposed tool through a 21-question survey.

In the case of [A05], the evaluation measures the user's perception of how meaningful the recommended refactorings are compared to the other tools. Besides, it measures the time (T) that developers spent to identify the best refactoring strategies and the number of interactions. Finally, in [A06], the participants were provided with the source code of the *FreeMind* project and a list of refactoring solutions generated by four different tools. Then, the authors asked the participants to complete a questionnaire about the refactoring tools and their solutions. For the case of [A10], the authors elaborated a 30-question survey and ask 5 engineers seniors to decide whether or not s/he would refactor that method.

Table 5: Evaluation and automation of approach in the selected papers

ID	I	II	III	IV	Level	Evaluation Result (II and III)	Limitation	Automation
[A01]	✓	✓		✓	Method	The KNN outperformed the other 8 classifiers. The combination with CodeBERT embedding technique shows a better performance	The small size of the dataset can compromise the result of the trained model.	Semi-Aut
[A02]	✓				Method	The comparison was against the state-of-the-art approach from Aniche et al. [11], taking as baseline the random forest model.	Other existing approaches in the literature are not used for comparison.	Semi-Aut
[A03]		✓		✓	Method	Convolutional Neural Network, Random Forest and Support Vector Machine are the models with better performance	The authors only use Convolutional Neural Network (CNN) for their propose approach.	Fully-Aut
[A04]	✓	✓			Class/Method/ Attribute	For binary Classifier MNB outperformed the rest of the classifiers. For Multilabel Classifiers, SVM had the best performing classifier.	The proposed approach is compared with its previous version	Semi-Aut
[A05]	✓			✓	Class/Method/ Attribute	The authors conduct a experiment where developers manually evaluate solutions to estimate the relevance of refactorings.	No measures or internal quality indicators for estimating the relevance of refactorings were used.	Fully-Aut
[A06]	✓	✓		✓	Method	The authors train classifiers with various embedding techniques. The Code2Vec+SDNE (CV+SN) embedding technique with NB (Naive Bayes) was the most effective combination.	Deep learning classifiers do not perform as well as the authors expected in recommending move method refactorings. The authors believe this is cause by type of the data.	Semi-Aut
[A07]		✓			Class	The MVE (maximum voting ensemble) with upsampling shows a better performance when compared with the other classifiers in the proposed approach.	PROMISE is a well-known and widely used dataset in this context. A comparative evaluation with other approaches/tools could be conducted to demonstrate the performance.	Semi-Aut
[A08]			✓		Class (Model)	The authors show that with the use of deep neural network it is possible to detect models as flawed by functional decomposition (FD) with a precision of 0.87.	The evaluation focused on the identification of the FD and does not make it clear how to apply the recommended solution, what is the order of application and where it should be applied.	Semi-Aut
[A09]	✓				Class	The approach was compared with the work of Di Nucci et al.	The authors only use Bayesian networks model.	Semi-Aut
[A10]				✓	Method	The survey consists of 30 questions. Additionally, the authors performed a comparison between Datasets.	The paper is a case study within a single organization, ING, a large financial organization.	Semi-Aut
[A11]		✓			class, method, and variable-levels	Random Forest has the highest overall accuracy among all the 6 models	The approach is only intended to compare the models, and not to solve any problem or provide any recommendations.	Semi-Aut
[A12]	✓				Method	Datasets: JMove's dataset and MoveMethodDataset	The authors only use SVM classifier.	Semi-Aut
[A13]			✓		Method	Three Naïve Bayes classifiers were used (GNB, MNB, BNB) and the BNB presented the best performance in terms of AUC and Accuracy.	In this study, the author only use Naive Bayes and do not experiment with other classifiers.	Semi-Aut
[A14]			✓		Class	Compares the performance of GRU classifiers using balanced and imbalanced datasets. The balanced one had the best performance.	In this study, the author only use GRU and do not experiment with other classifiers.	Semi-Aut
[A15]	✓	✓			Method	ForestPA and RF achieved the best results among all the classifiers.	The Dataset used for evaluation are small.	Semi-Aut
[A16]		✓			Method	AdaBoost and ANN+GD classifiers outperformed the other classifiers. In addition, the authors show that using balance techniques can produce statistically significant differences in performances.	PROMISE is a well-known and widely used dataset in this context. A comparative evaluation with other approaches/tools could be conducted.	Semi-Aut
[A17]	✓	✓			Method	AdaBoost suggests clones for refactoring with high accuracy.	Comparison with an only one approach developed in 2014.	Semi-Aut
[A18]	✓				Method	The destination part of the recommendation was evaluated, i.e., the correct identification of the target class.	The Dataset was generated artificially, i.e., all the Feature Envy smell were made moving the methods manually.	Semi-Aut
[A19]	✓				Method	Dataset: 5 open source software projects	Dataset small, composed by only 267 Extract Method instances.	Semi-Aut
[A20]	✓	✓			Method	GB classifier presents better performance of the others.	The Dataset used for evaluation was small - only 267 instances.	Semi-Aut
[A21]		✓			Method	SVM has high Precision, but lower Recall. On the other hand, the algorithms based on decision tree (J48 and RandomForest) record over 89% for both of Precision and Recall.	There are other approaches/tools in the literature that recommend the "Extract Method" and it would have been interesting to observe the performance of the proposal compared to them.	Semi-Aut
[A22]			✓		Class	The authors demonstrated that LS-LSM RBF kernel variant outperforms linear and polynomial kernel.	In this study, the author only one classifier Least Squares Support Vector Machines (LSSVM).	Semi-Aut

Table 6: Tools used in evaluation type I

		#P	Approach
Tool Name	JDeodorant	7	[A01], [A05], [A06], [A12], [A19], [A18], [A20]
	JMove	3	[A06], [A12], [A18]
	JExtract	3	[A01], [A19], [A20]
	SEMI	2	[A01], [A20]
	PathMove	1	[A06]
	GEMS	1	[A01]
	Segmentation	1	[A01]
Unknown tool	Wang and Godfrey	2	[A15][A17]
	Charalampidou et al.	1	[A04]
	Ouni et al.	1	[A05]
	Mkaouer et al.	1	[A05]
	Alizadeh et al.	1	[A05]
	FR-Refactor (Nyamawe et al.)	1	[A04]
	CREC (Yue et al.)	1	[A15]
	Di Nucci et al.	1	[A09]
	Aniche et al.	1	[A02]

Regarding Type I, comparison with others tools, the goal is identifying the tools used by the approaches. We elaborated the Table 6 that shows the distribution of some well-known tools and other unknown tools (we have included the authors' names for identification). It is noteworthy that *JDeodorant* [18] is the most used tool. It is able to identify five kinds of bad smells and resolve them, recommending and applying refactorings. Other popular tools are *JMove* [48] and *Jextract* [43] used by 3 approaches. Regarding unkwnon tools, the approach of Wang and Godfrey [51], which focuses on recommending code clones, serves as a baseline for comparison by [A16] and [A18].

RQ2.2-*What is the automation level of the approaches?*

To classify our final set we took into consideration the works of [7] and [46], but we have extended the classification given by them. Therefore, our classification consider the following ones:

- **Fully automated**. The approach involves a tool which is able to identify the refactoring opportunity, provides the recommendation to the developer and also allows the developer to apply the refactoring.

- **Partial/semi automated**. The approach does not have a tool integrated in an IDE. The steps are usually performed in steps/phases using trained classifiers.

Table 5 shows the automation level of the approaches, 20 of the 22 papers in the final set do not provide any support to guide the developer in the refactoring process, which means than 90% of the approaches are Semi-Automated. The only two approaches that provide fully automated support are [A03] and [A05], providing the support for identifying the refactoring opportunities. [A03]

presents an AntiCopyPaster plugin that monitors the introduction of potential duplicate code and recommends its refactoring using the IDE's Extract Method feature. [A05] presents DOIMR tool that allows the interaction with developers. The tool shows a list of refactoring recommendations solution and the developer can evaluate them based on their preferences.

RQ3-*Have the approaches concerned with the quality of the recommendations?*

This RQ aimed to analyze to what extent the approaches have concerned with the quality of the recommendation they provide from the developer point of view. We consider a high-quality recommendation one that gives the developers all the information they need to decide whether to accept the recommendation or not. In this paper we prefer to use the term "complete" for classifying the recommendations [38]. We have devised our own definition of "complete" to enable a meaningful comparison of the approaches. This was necessary because the approaches differ substantially in terms of the recommendation quality. While some recommendations are quite complete, providing software engineers with detailed information, others lack in details, suggesting just the name of a refactoring or the place in the source code where the refactoring must be applied.

From our perspective, if a refactoring recommendation satisfies our criterion, we consider it as *complete*, meaning that the recommendation includes all necessary information to support the user. Therefore, to facilitate the analysis of the completeness of recommendations, we introduce a classification criteria named W3B (**W**hich, **W**here, **W**hy and **B**enefits), explained below:

WHICH: This involves a precise identification of **which** refactoring(s) should be applied, specifying the name of the refactoring, such as Extract Method or Inline Function. Some approaches recommend a single refactoring at a time, while others suggest an ordered sequence of refactorings.

WHERE: This entails a clear identification of **where** the refactoring must be applied, indicating the specific part(s) of the source code (e.g., method, field, class, parameters) requiring modifications. The *where* element is complex, involving sub parts dependent on the recommended refactoring.

WHY: This involves a clear explanation of *why* a particular refactoring was recommended. The easiest way of thinking about this is asking "WHY the refactoring R was recommended?". The answer must elucidate the features that contribute the most to the model decision. From a developer's point of view, it is crucial to clarify why such a refactoring was recommended. Consequently, a recommendation should convey this information explicitly, such as: *"This refactoring is being recommended because this piece of code (method/class) has a cohesion around X and a coupling with n other classes"*. The provision of a transparent explanation is crucial for users to comprehend and trust ML recommendations, and this objective can be achieved through the application of Explainable Artificial Intelligence (XAI) techniques [17].

BENEFITS: This involves a clear identification of the benefits resulting from applying the refactoring. This element must anticipate to the developer the advantages/benefits of the new state of the source code, i.e., after the application of the refactoring.

We have used the criterion **W3B** for comparing the approaches of our final set. Table 7 shows in the first column the ID of the approach, in the second one the ID of the paper and from the third column to the sixth the W3B elements - which, where, why and benefits. The last column denotes whether the approach covers all criteria points, marked accordingly. It is evident that approaches [A01], [A03], [A05], [A06], [A09], [A12], [A17], [A18], [A19], and [A20] have successfully identified **Which** refactorings should be recommended and provided comprehensive insights into the precise **Where** these refactorings should be applied. This means that 45% of the approaches minimally identified these two elements of the criterion, with the majority of them offering recommendations that involve only a single refactoring.

Regarding the **Where** element, the term "Complete" or "Incomplete" depends on the type of refactoring recommended. In the case of the approaches [A04] and [A08], they do not point out the source code component where the refactoring must be applied, leaving this tedious task to the developers. It is interesting to mention that in the approach [A04] (papers [S04], [S12] and [S19]), the authors recognize this weakness and suggest using another tool to complete this part of the recommendation.

In the case of approaches [A07], [A14], [A16], and [A22] the approaches do not identify the **Which** element. So, the approaches focus on the identification and recommendation of the component (method or class) that must be refactored, providing a more general and wide recommendation. On the other hand, the lack of identification of the **Which** element affects directly the completeness of the **Where** part because as we said before, the sub elements inside the where depends on the type of refactoring.

Regarding the **Why**, only approaches [A03] and [A05] show the reasons behind the recommendation. The [A03] shows a pop-up notification at the bottom of the screen, alerting the developer of the opportunity to apply "Extract Method", explaining the reason why this refactoring must be conducted. On the other hand, the [A05] uses interactive tables and charts alongside extensive analysis to explain the recommendation. As can be observed, almost 90% of the papers do not explain the rationale behind a recommendation.

Finally, regarding the **Benefits** for applying the refactoring recommended, it is observable that 11% approaches [A01], [A03], [A06], [A08], [A09], [A12], [A13], [A15], [A17], [A19] and [A18] aim at solving a code smell. Refactoring and smells have been well researched by the software-engineering research community these past decades. Several secondary studies have been published about the relationship between the code smell and the refactoring that can resolve it. So it is expected to observe this relationship in our final set of articles. Another benefit is "improve software maintainability" researched by 6 approaches [A07], [A14], [A16], [A20], [A21], [A22]. This goal is a generic and broad one because the authors do not make it clear specifically which characteristics related to maintenance they want to improve.

Importantly, based on the reviewed approaches, only the approaches [A03] and [A05] achieve the criterion **W3B**, which means that more than 90% of the approaches do not have the elements to be considered complete. Our results provide evidence that there is a lack in the refactoring recommendation area and at the same time it opens up new opportunities for future works.

Table 7: W3B Criterion - Rules for *Complete* Recommendations

ID	Paper	**Which** refactoring is recommended	Where	Why	Benefits	W3B
[A01]	[S01]	Extract Method	Complete	Undefined	Solving Duplicated Code, Feature Envy and Long Method	✗
[A02]	[S02]	Extract Method	Incomplete	Undefined	Undefined	✗
[A03]	[S03]	Extract Method	Complete	Pop-up notification	Solving Code clone smell	✓
[A04]	[S04] [S12] [S19]	Extract Interface, Extract Method, Extract Superclass, Inline Method, Move And Rename Class, Move Attribute, Move Class, Move Method, Pull Up Attribute, Pull Up Method, Push Down Attribute, Push Down Method, Rename Class, Rename Method,	Undefined	Undefined	Adapting the system for new requirements; improve code cohesion and keep the conformity to OOP principles	✗
[A05]	[S05] [S17] [S20]	Extract Class, Extract SubClass, Extract SuperClass, Extract Method, Move Method/Field, PullUp Field, PullUp Method, PushDown Field/Method, Encapsulate Field, Increase Field Security, Decrease Field Security, Increase Method Security, Decrease Method Security	Complete	Graphical charts and tables	Improving quality attributes QMOOD, in terms of Reusability, Flexibility, Understandability, Functionality, Extendibility, Effectiveness.	✓
[A06]	[S04]	Move Method	Complete	Undefined	Solving Feature Envy	✗
[A07]	[S07]	Undefined by the authors	Incomplete	Undefined	Improve the software maintainability.	✗
[A08]	[S08]	Move Operation, Move Attribute, Extract Class, Extract Associated Class, Extract Subclass, Extract Superclass, Pull Up Operation, Pull Up Attribute, Push Down Operation, Push Down Attribute, Rename Class, Rename Operation, Rename Attribute.	Undefined	Undefined	Identifying Functional decomposition in UML diagrams.	✗
[A09]	[S09]	Remove relations, Remove classes and Add classes	Complete	Undefined	Solving God Class, Data Class, Feature Envy, Complex Class, Spaghetti Class, and Speculative Generality.	✗
[A10]	[S10]	Extract Method	Incomplete	Undefined	Undefined	✗
[A11]	[S11]	Extract Class, Extract Subclass, Extract Super-class, Extract Interface, Move Class, Rename Class, Move and Rename Class, Extract Method, Inline Method, Move Method, Pull Up, Push Down Method, Rename Method, Extract And Move Method, Extract Variable, Inline Variable, Rename Variable.	Incomplete	Undefined	Undefined	✗
[A12]	[S13]	Move Method	Complete	Undefined	Solving Feature Envy smell; reducing the coupling between classes	✗
[A13]	[S14]	Extract Method, Pullup Method, substitution Algorithm, Replace Temp with Query, Introduce parameter Object, Preserve the whole object, Replace method with method object, decomposition objects	Incomplete	Undefined	Duplicate code and Long Method code smells	✗
[A14]	[S15]	Undefined by the authors	Incomplete	Undefined	Improve the software maintainability;	✗
[A15]	[S16]	Move Method, Pull up Method, Extract Method	Incomplete	Undefined	Solving code clone type I, II and III	✗
[A16]	[S18] [S21]	Undefined by the authors	Incomplete	Undefined	Improve the software maintainability	✗
[A17]	[S22]	Extract Method	Complete	Undefined	Solving Code clone smell	✗
[A18]	[S23]	Move Method	Complete	Undefined	Solving Feature Envy code smell	✗
[A19]	[S24]	Extract Method	Complete	Undefined	Long Method smell	✗
[A20]	[S25]	Extract Method	Complete	Undefined	Improving the software maintainability and source code readability	✗
[A21]	[S26]	Extract Method	Incomplete	Undefined	Improving the software maintainability	✗
[A22]	[S27]	Undefined by the authors	Incomplete	Undefined	Improve the software maintainability	✗

[1] Undefined = The authors do not explain this element for their recommendations.

4 DISCUSSION

Our results let emerge some topics worth further discussion:
Refactoring researched - Extract Method - Table 2 shows that the three most researched refactorings are: Extract Method, Move Method, and Pull-up Method. The high interest in these refactorings may indicate their importance in the industrial sector and suggest that these activities are more frequently applied in practice than other activities. In addition, we realized that most of the refactoring proposed by Fowler (45 out of 67 refactoring) was not considered in any of the papers in our final set. This result shows a gap between the refactoring practice and the research in the area of identifying refactoring opportunities.
Supervised learning techniques are favored over unsupervised learning techniques - This numerical superiority is due to the nature of recommendations. To recommend refactorings it is necessary to learn how to identify *the refactoring opportunity*. In literature, extensive research has been conducted on this topic exploring different indicators such as code smell and software qualities. Thus, there is a vast amount of information that can be used to set up a data repository. Another important element is the apogee in mining tools such as Rminer[49] and Rdiff [45] that allow the extraction of past refactorings, helping to build a reliable and solid Dataset.
The nature of the datasets impact in the refactoring opportunities - Regarding the type of Datasets (Refactoring-based (RB) and Code smell-based (CS)), we believe an important difference between the samples/instances of an RB over CS is the potential valuable information that can be explored. Classifiers trained with refactoring-based datasets have the advantage of identifying a wider range of refactoring opportunities. However as the instances, and all feature values for them, must reflect a snapshot before the application of a refactoring, it is harder to do the data collection and processing before populating the dataset. *Features explorer in Datasets* - The results reported in Table 4 show that the most frequent group of features used to comfort the Dataset are the source code metrics. Clearly, the features used are concentrated on static analysis. This observation indicates that there is a gap in the use of other kinds of metrics. Researchers are encouraged to explore different sources of features and to compare if the trained model with other non-classic features improves their prediction.
Lack of mature tools - Over 80% of the reviewed articles indicate manual evaluations as the prevailing method. Authors typically employ a single dataset to train multiple ML, comparing their accuracy. The lack of consensus among authors on the best algorithm is attributed to factors like problem specifics, data characteristics, size, and additional techniques employed, such as data balancing and embedding techniques. Conversely, just two papers propose an automated approach with tool support, but the adoption of these tools in the software engineering industry is hindered by the lack of mature supporting tools, as shown in Table 5.

Low alignment with the W3B criterion - Our analysis using the W3B criterion showed that formulating a complete recommendation is a complex task. So, 90% of the reviewed approaches do not provide a complete recommendation. The quality of a recommendation has a total impact on whether it will be accepted or not. Studies [37] [5] indicate that users do not accept refactoring recommendations because they do not explain why they should be applied or what the benefits they will bring to the system. It may cause the rejection of the recommendation and the loss of all the benefits it brings.

An interesting challenge is regarding the **Benefits** element. Although all approaches have a broad goal defined, each recommendation should provide a clear identification of the benefit of applying that refactoring. That is, the broad benefit must be broken down into specific benefits. For example, a recommender system's overarching objective might be to enhance method cohesion. However, the same system could suggest the Extract Method refactoring to enhance the testability of that method or recommend other improvements. This poses a challenge, particularly in supervised learning, where the training dataset should encompass features that differ from conventional ones.

5 THREATS TO VALIDITY

Internal validity: All relevant papers were retrieved using the search strings across major databases and through snowballing techniques from reference lists. We covered seven key publication venues, including general databases like Scopus and specific ones like IEEE.

External validity: The collected papers are constrained primarily to academic works, limiting generalizability to industry contexts despite providing a solid foundation for academic insights.

6 RELATED WORKS

In this section, we categorize the related works into two groups. The first group encompasses a wide spectrum of secondary studies related to refactorings [13] [2]. For example, Abid et al. [2] provided an overview of the last 30 years of research on software refactorings. They classified the approaches based on different criteria such as: refactoring life-cycle (included recommendation and prediction); refactoring objectives (included to improve internal or external software quality); refactoring techniques (included ML algorithms); and evaluation strategy. This study differs from ours since it is broader and the focus is on the classification of various refactoring techniques using a taxonomy proposed by the authors.

In the second group we consider studies that deal with the use of ML for smells identification. These works stand out as the most extensively researched in the literature [21] [3] [12]. Ahmed et al. [3] performed a SLR on the use of ML to detect code smells from different perspectives: i) the smells employed in the experiments; ii) the types of ML techniques; iii) a comparison between these models in terms of prediction accuracy and performance; iv) the datasets and features used and v) the tool utilized to implement the ML models. At the same way, Azeema et al. [12] also conducted a SLR focused on: i) the code smells and the ML technique employed; ii) the evaluation strategy applied; iii) the features used in dataset and the iv) performance meta-analysis. Another interesting work was done by Manpreet et al. [21]. They dedicated their SLR in the use of ML in research related to code clones code smell. The studies of the second group differ from ours as they are focused on code smells or code anomalies (anti-patterns, bugs, etc), i.e., they are concentrated on the problem space. Besides, the general focus of these works are on the ML pipeline and on the accuracy analysis of ML models. Our work is devoted to analyze the consequences of using ML on the refactoring/solution space, including the quality characteristics of the recommendations.

7 CONCLUSION AND FUTURE DIRECTIONS

This paper reports a SLR on the use of ML in the context of refactoring recommendations. A total of 177 potential articles were identified in five scientific digital libraries in the period from 2015 and 2023. After screening the articles, our final set resulted in 27 papers, of which we grouped them in 22 approaches.

The results of the SLR reveal a significant interest in providing complete refactoring recommendations for users, despite it being a difficult task due to its subjective nature. Based on the 27 selected papers, we identified a criterion called W3B focused on four key aspects of Refactoring recommendation *Which*, *Where*, *Why* and *Benefits*. Using this criterion, we analyzed the final set of approaches. As a result, the approaches [A03] and [A05] are the only ones that provide a complete recommendation, achieving four elements of criterion. Thus, 40% of the approaches (9 approaches) achieves only 3 element and the remaining approaches, more than 60%, complete less than 3 elements of the criterion. This result is important as it opens several possibilities for future work.

A critical area needing more attention is the data used to construct datasets, categorized into refactoring-based and code smell-based types. Emphasizing learning from past refactoring instances is more insightful but often requires external mining tools and is time-consuming. Future work should compare these dataset types and identify optimal configurations for refactoring recommendations, including historical information volume, dataset size, feature selection, balancing techniques, and data standardization methods.

This SLR also reveals an opportunity to explore the inclusion of user feedback in the formulation of the recommendations. So, the feedback is an important aspect to understand the preference of the user and to improve the ML model, allowing better recommendations over time. In the work of [37], they indicate that including developer feedback in the automated generation of refactoring solutions can help in reaching solutions that are (i) well-suited for the specific developer, and (ii) further refined when needed.

8 ACKNOWLEDGEMENTS

We would like to thank the financial support provided by FAPESP, SP, Brazil, process number (2024/13184-7) and by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

REFERENCES

[1] 2024. *On the Employment of Machine Learning for Recommending Refactorings: A Systematic Literature Review*. Zenodo.

[2] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira, and Danny Dig. 2020. 30 years of software refactoring research: a systematic literature review. *arXiv preprint arXiv:2007.02194* (2020).

- [3] Ahmed Al-Shaaby, Hamoud Aljamaan, and Mohammad Alshayeb. 2020. Bad smell detection using machine learning techniques: a systematic literature review. *Arabian Journal for Science and Engineering* 45, 4 (2020), 2341–2369.
- [4] Mamdouh Alenezi, Mohammed Akour, and Osama Al Qasem. 2020. Harnessing deep learning algorithms to predict software refactoring. *Telkomnika* 18, 6 (2020), 2977–2982.
- [5] Vahid Alizadeh, Houcem Fehri, and Marouane Kessentini. 2019. Less is More: From Multi-objective to Mono-objective Refactoring via Developer's Knowledge Extraction. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 181–192.
- [6] Vahid Alizadeh and Marouane Kessentini. 2018. Reducing interactive refactoring effort via clustering-based multi-objective search. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 464–474.
- [7] Vahid Alizadeh, Mohamed Amine Ouali, Marouane Kessentini, and Meriem Chater. 2019. RefBot: Intelligent software refactoring bot. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 823–834.
- [8] Eman Abdullah AlOmar, Anton Ivanov, Zarina Kurbatova, Yaroslav Golubev, Mohamed Wiem Mkaouer, Ali Ouni, Timofey Bryksin, Le Nguyen, Amit Kini, and Aditya Thakur. 2023. Just-in-time code duplicates extraction. *Information and Software Technology* 158 (2023), 107169.
- [9] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius HS Durelli. 2020. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1432–1450.
- [10] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21 (2016), 1143–1191.
- [11] Guisella A Armijo and Valter V de Camargo. 2022. Refactoring Recommendations with Machine Learning. In *Anais Estendidos do XXI Simpósio Brasileiro de Qualidade de Software*. SBC, 15–22.
- [12] 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.
- [13] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. 2020. Automatic software refactoring: a systematic literature review. *Software Quality Journal* 28, 2 (2020), 459–502.
- [14] Di Cui, Qiangqiang Wang, Siqi Wang, Jianlei Chi, Jianan Li, Lu Wang, and Qingshan Li. 2023. REMS: Recommending Extract Method Refactoring Opportunities via Multi-view Representation of Code Property Graph. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 191–202.
- [15] Di Cui, Siqi Wang, Yong Luo, Xingyu Li, Jie Dai, Lu Wang, and Qingshan Li. 2022. RMove: Recommending Move Method Refactoring Opportunities using Structural and Semantic Representations of Code. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 281–292.
- [16] Warteruzannan Soyer Cunha, Guisella Angulo Armijo, and Valter Vieira de Camargo. 2020. Investigating Non-Usually Employed Features in the Identification of Architectural Smells: A Machine Learning-Based Approach *(SBCARS '20)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/3425269.3425281
- [17] Arun Das and Paul Rad. 2020. Opportunities and challenges in explainable artificial intelligence (xai): A survey. *arXiv preprint arXiv:2006.11371* (2020).
- [18] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. JDeodorant: identification and application of extract class refactorings. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE.
- [19] M. Fowler and K. Beck. 2019. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
- [20] Ayaka Imazato, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2017. Finding extract method refactoring opportunities by analyzing development history. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 190–195.
- [21] Manpreet Kaur and Dhavleesh Rattan. 2023. A systematic literature review on the use of machine learning in code clone research. *Computer Science Review* 47 (2023), 100528.
- [22] Barbara Kitchenham, O Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. 2009. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology* 51, 1 (2009), 7–15.
- [23] Barbara Ann Kitchenham and Stuart Charters. 2007. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE 2007-001.
- [24] Lov Kumar, Shashank Mouli Satapathy, and Aneesh Krishna. 2018. Application of smote and lssvm with various kernels for predicting refactoring at method level. In *International Conference on Neural Information Processing*. Springer, 150–161.
- [25] Lov Kumar, Shashank Mouli Satapathy, and Ashish Sureka. 2015. Method Level Refactoring Prediction on Five Open Source Java Projects using Machine Learning Techniques.
- [26] Lov Kumar and Ashish Sureka. 2017. Application of LSSVM and SMOTE on seven open source projects for predicting refactoring at class level. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 90–99.
- [27] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. 2020. Recommendation of Move Method Refactoring Using Path-Based Representation of Code. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 315–322.
- [28] Hui Liu, Zhifeng Xu, and Yanzhen Zou. 2018. Deep learning based feature envy detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 385–396.
- [29] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139.
- [30] Ally S Nyamawe. 2022. Mining commit messages to enhance software refactorings recommendation: A machine learning approach. *Machine Learning with Applications* (2022), 100316.
- [31] Ally S Nyamawe, Hui Liu, Nan Niu, Qasim Umer, and Zhendong Niu. 2019. Automated recommendation of software refactorings based on feature requests. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*. IEEE, 187–198.
- [32] Ally S Nyamawe, Hui Liu, Nan Niu, Qasim Umer, and Zhendong Niu. 2020. Feature requests-based recommendation of software refactorings. *Empirical Software Engineering* 25, 5 (2020), 4315–4347.
- [33] Indranil Palit, Gautam Shetty, Hera Arif, and Tushar Sharma. 2023. Automatic refactoring candidate identification leveraging effective code representation. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 369–374.
- [34] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In *Proceedings of the 40th International Conference on Software Engineering*. 482–482.
- [35] Rasmita Panigrahi, Sanjay Kumar Kuanar, Sanjay Misra, and Lov Kumar. 2022. Class-Level Refactoring Prediction by Ensemble Learning with Various Feature Selection Techniques. *Applied Sciences* 12, 23 (2022), 12217.
- [36] Rasmita Panigrahi, Lov Kumar, et al. 2020. Application of Naïve Bayes classifiers for refactoring Prediction at the method level. In *2020 International Conference on Computer Science, Engineering and Applications (ICCSEA)*. IEEE, 1–6.
- [37] Jevgenija Pantiuchina, Bin Lin, Fiorella Zampetti, Massimiliano Di Penta, Michele Lanza, and Gabriele Bavota. 2021. Why Do Developers Reject Refactorings in Open-Source Projects? *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2021), 1–23.
- [38] Ivens Portugal, Paulo Alencar, and Donald Cowan. 2018. The use of machine learning algorithms in recommender systems: A systematic review. *Expert Systems with Applications* 97 (2018), 205–227.
- [39] Soumaya Rebai, Vahid Alizadeh, Marouane Kessentini, Houcem Fehri, and Rick Kazman. 2020. Enabling decision and objective space exploration for interactive multi-objective refactoring. *IEEE Transactions on Software Engineering* (2020).
- [40] Raana Saheb-Nassagh, Mehrdad Ashtiani, and Behrouz Minaei-Bidgoli. 2022. A probabilistic-based approach for automatic identification and refactoring of software code smells. *Applied Soft Computing* 130 (2022), 109658.
- [41] Abdullah M Sheneamer. 2020. An automatic advisor for refactoring software clones based on machine learning. *IEEE Access* 8 (2020), 124978–124988.
- [42] Brahmaleen Kaur Sidhu, Kawaljeet Singh, and Neeraj Sharma. 2022. A machine learning approach to software model refactoring. *International Journal of Computers and Applications* 44, 2 (2022), 166–177.
- [43] Danilo Silva, Ricardo Terra, and Marco Túlio Valente. 2015. Jextract: An eclipse plug-in for recommending automated extract method refactorings. *arXiv preprint arXiv:1506.06086* (2015).
- [44] 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.
- [45] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 269–279.
- [46] Jocelyn Simmonds and Tom Mens. 2002. A comparison of software refactoring tools. *Programming Technology Lab* (2002).
- [47] Cleiton Silva Tavares, Amanda Santana, Eduardo Figueiredo, and Mariza Bigonha. 2020. Revisiting the Bad Smell and Refactoring Relationship: A Systematic Literature Review. In *Conferencia Iberoamericana de Software Engineering*.
- [48] Ricardo Terra, Marco Tulio Valente, Sergio Miranda, and Vitor Sales. 2018. JMove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software* 138 (2018), 19–36.
- [49] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 483–494.
- [50] David van der Leij, Jasper Binda, Robbert van Dalen, Pieter Vallen, Yaping Luo, and Maurício Aniche. 2021. Data-driven extract method recommendations: a study at ING. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1337–1347.

[51] Wei Wang and Michael W Godfrey. 2014. Recommending clones for refactoring using design, context, and history. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 331–340.

[52] Sihan Xu, Chenkai Guo, Lei Liu, and Jing Xu. 2017. A log-linear probabilistic model for prioritizing extract method refactorings. In *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. IEEE, 2503–2507.

[53] Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. 2017. Gems: An extract method refactoring recommender. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 24–34.

[54] Ruru Yue, Zhe Gao, Na Meng, Yingfei Xiong, Xiaoyin Wang, and J David Morgenthaler. 2018. Automatic clone recommendation for refactoring based on the present and the past. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 115–126.