Toward the Automatic Classification of Self-Affirmed Refactoring

Eman Abdullah AlOmar[a,*], Mohamed Wiem Mkaouer[a], Ali Ouni[b]

[a] *Rochester Institute of Technology, Rochester, NY, USA*
[b] *ETS Montreal, University of Quebec, Montreal, QC, Canada*

Abstract

The concept of Self-Affirmed Refactoring (SAR) was introduced to explore how developers document their refactoring activities in commit messages, *i.e.,* developers explicit documentation of refactoring operations intentionally introduced during a code change. In our previous study, we have manually identified refactoring patterns and defined three main common quality improvement categories including internal quality attributes, external quality attributes, and code smells, by only considering refactoring-related commits. However, this approach heavily depends on the manual inspection of commit messages. In this paper, we propose a two-step approach to first identify whether a commit describes developer-related refactoring events, then to classify it according to the refactoring common quality improvement categories. Specifically, we combine the N-Gram TF-IDF feature selection with binary and multiclass classifiers to build a new model to automate the classification of refactorings based on their quality improvement categories. We challenge our model using a total of 2,867 commit messages extracted from well engineered open-source Java projects. Our findings show that (1) our model is able to accurately classify SAR commits, outperforming the pattern-based and random classifier approaches, and allowing the discovery of 40 more relevant SAR patterns, and (2) our model reaches an F-measure of up to 90% even with a relatively small training dataset.

*Corresponding author
Email addresses: eman.alomar@mail.rit.edu (Eman Abdullah AlOmar), mwmvse@rit.edu (Mohamed Wiem Mkaouer), ali.ouni@etsmtl.ca (Ali Ouni)

Keywords: Refactoring, Self-affirmed Refactoring, Commit Classification, Machine Learning

1. Introduction

The role of refactoring has been growing from simply improving the internal structure of the code without altering its external behavior [1] to hold a key driver of the agile methodologies and become one of the main practices to reduce technical debt [2]. According to recent surveys, the research on refactoring has been focused on automating it through recommending candidate code elements to be refactored and which refactoring operations to apply [3, 4, 5, 6]. Yet, more recent studies have shown that fully automated techniques are underused in practice [7]. Indeed, there is a need to minimize the disturbance of the existing design, by performing large refactorings, as developers typically want to recognize and preserve the semantics of their own design, even at the expense of not significantly improving it [7, 8, 9].

Therefore, several studies have taken a developer-centric approach by detecting how developers do refactor their code [10, 11, 12] and how they document their refactoring strategies [13, 14]. The detection of refactoring operations and their documentation allows a better understanding of code evolution, and challenges that trigger refactoring, including the reduction of code proneness to errors, facilitation of API and type migrations, etc. While automating the detection of refactoring operations that are applied in the source code has advanced recently reaching a high accuracy [12], there is a critical need for a deeper analysis of how such refactoring activities are being documented. In this context, recent studies [13, 14] have introduced a taxonomy on how developers actually document their refactoring strategies in commit messages. Such documentation is known as *Self-Admitted* or *Self-Affirmed* refactoring. Documenting refactoring, similarly to any type of code change documentation, is useful to decipher the rationale behind any applied change, and it can help future developers in various engineering tasks, such as program comprehension, design reverse-engineering,

and debugging. However, the detection of such refactoring documentation was hardly manual and limited. There is a need for automating the detection of such documentation activities, with an acceptable level of accuracy. Indeed, the automated detection of refactoring documentation may support various applications and provide actionable insights to software practitioners and researchers, including empirical studies around the developer's perception of refactoring. This can question whether developers do care about structural metrics and code smells when refactoring their code, or if there are other factors that may influence such non-functional changes. Furthermore, our previous study [14] found that there are several intentions behind the application of refactoring, which can be classified as improving internal structural metrics (*e.g.,* cohesion, encapsulation), removing code smells (*e.g.,* God classes, dead code), or optimizing external quality attributes (*e.g.,* testability, readability). Yet, there is no systematic way to classify such refactoring related messages and estimate the distribution of refactoring effort among these categories.

To cope with the above-mentioned limitations, this paper aims to automate the detection and classification of refactoring documentation in commit messages. In particular, our objective is to analyze the feasibility and performance of applying learning techniques to (1) identify and (2) classify refactoring documentation based on commit messages. However, the detection of refactoring documentation is challenging, besides the inherited ambiguity of distinguishing meanings, in any natural language text, a recent study has shown that developers do misuse the term *refactoring* in their documentations [13], which hardens the reliance on that keyword alone. To cope with these challenges, we design our study to harvest a potential taxonomy that can be used to document refactoring activities. Such taxonomy is typically threatened by the potential false-positiveness of the collected samples. Therefore, we develop a baseline of code changes that are known to contain refactoring activities, and we analyze their commit messages, in order to ensure that the collected textual patterns are meant to describe refactoring, and so, to reduce false positives. Our study makes the following contributions:

- We present a two-step approach that firstly distinguishes whether a commit message potentially contains an explicit description of a refactoring effort. Then, secondly classifies it into one of the three common categories identified in previous studies [13, 14]. To the best of our knowledge this is the first attempt to automate the detection and classification of self affirmed refactorings.

- We evaluate the performance of our approach by comparing it against a keyword-based approach that relies on matching messages with known refactoring keywords [7, 13, 15, 16, 17, 18]. Our key findings show that our model not only outperforms the keyword-based approach, but also accurately identifies refactoring related commits with an average accuracy of 98% and F-measure of 98%. Furthermore, we infer which features, *i.e.,* keywords, are relevant for the detection of such refactoring documentation, and we extract them to extend the list of refactoring documentation patterns, identified in previous studies [13, 14].

- We deploy our model as a lightweight web-service that is publicly available for software engineers and practitioners. We also publicly provide our dataset that served us as the *ground-truth*, for replication and extension purposes [19].

This paper is structured as follows. We start by explaining the notion of refactoring documentation (self-affirmed refactoring) and reviewing existing studies that are most related to commit messages classification in Section 3. Next, in Section 4, we detail our two-step classification methodology. More precisely, we elaborate on the data collection and preprocessing, choice of the classification algorithms. Then, we evaluate our approach, in Section 5, and report a comparative study between various classifiers, extracted from previous studies, and we identify most influential features. We report in Section 7 the threats to our work's validity, before concluding and describing our future work in Section 8.

2. Self-Affirmed Refactoring

2.1. Definition

Commit messages are the atomic descriptions of given code change, in natural language. It augments the change with human and machine readable meaning. In this study, we are interested in locating and automatically detecting developer's documentation of refactoring activities in commit messages. *refactoring documentation* is the textual description of what developers considers to be a refactoring performed in their code change. The act of intentionally documenting a refactoring activity is known as *Self-Affirmed Refactoring* (SAR) [14]. SAR is composed of a terminology that was found to be consistently used in refactoring-related commit messages. For example, if we consider the following commit message:

> Refactor createOrUpdate method in MongoChannelStore to extract methods and make code more readable

The developer explicitly mentions the intention of refactoring, using the keyword "*refactor*", along with providing extra information related to the refactoring activity: the developer reports 1) the type of refactoring operation performed, *i.e.*, *extract method*; 2) the code elements involved in the refactoring operation, *i.e.*, *createOrUpdate* and *MongoChannelStore*; and 3) the intent behind the refactoring, *i.e.*, *make the code more readable*. This message is labeled as Self-Affirmed Refactoring (SAR) as it totally or partially documents the refactoring performed in the source code.

The manual inspection of the message's corresponding commit[1], reveals 3 methods extracted from the method *createOrUpdate()* that belongs to the class *MongoChannelStore* along with renaming a parameter to be consistent with the update. So, the documentation has given enough background to explain the rationale behind the refactoring (improving code readability), the operations performed and the code elements involved.

[1] https://github.com/atlasapi/atlas-persistence

Table 1: List of Self-Affirmed Refactoring (SAR) Patterns.

Patterns		
(1) Refactor*	(30) Removed poor coding practice	(59) Change design
(2) Mov*	(31) Improve naming consistency	(60) Modularize the code
(3) Split*	(32) Removing unused classes	(61) Code cosmetics
(4) Fix*	(33) Pull some code up	(62) Moved more code out of
(5) Introduc*	(34) Use better name	(63) Remove dependency
(6) Decompos*	(35) Replace it with	(64) Enhanced code beauty
(7) Reorganiz*	(36) Make maintenance easier	(65) Simplify internal design
(8) Extract*	(37) Code cleanup	(66) Change package structure
(9) Merg*	(38) Minor Simplification	(67) Use a safer method
(10) Renam*	(39) Reorganize project structures	(68) Code improvements
(11) Chang*	(40) Code maintenance for refactoring	(69) Minor enhancement
(12) Restructur*	(41) Remove redundant code	(70) Get rid of unused code
(13) Reformat*	(42) Moved and gave clearer names to	(71) Fixing naming convention
(14) Extend*	(43) Refactor bad designed code	(72) Fix module structure
(15) Remov*	(44) Getting code out of	(73) Code optimization
(16) Replac*	(45) Deleting a lot of old stuff	(74) Fix a design flaw
(17) Rewrit*	(46) Code revision	(75) Nonfunctional code cleanup
(18) Simplif*	(47) Fix technical debt	(76) Improve code quality
(19) Creat*	(48) Fix quality issue	(77) Fix code smell
(20) Improv*	(49) Antipattern bad for performances	(78) Use less code
(21) Add*	(50) Major/Minor structural changes	(79) Avoid future confusion
(22) Modif*	(51) Clean up unnecessary code	(80) More easily extended
(23) Enhanc*	(52) Code reformatting & reordering	(81) Polishing code
(24) Rework*	(53) Nicer code / formatted / structure	(82) Move unused file away
(25) Inlin*	(54) Simplify code redundancies	(83) Many cosmetic changes
(26) Redesign*	(55) Added more checks for quality factors	(84) Inlined unnecessary classes
(27) Cleanup	(56) Naming improvements	(85) Code cleansing
(28) Reduc*	(57) Renamed for consistency	(86) Fix quality flaws
(29) Encapsulat*	(58) Refactoring towards nicer name analysis	(87) Simplify the code

2.2. Categories

In our previous work [14], we manually analyzed commit messages to extract any relevant textual patterns that can be considered as SAR. We provided a set of 87 SAR patterns, identified across 3,795 open source projects. Table 1 demonstrates all of these patterns. Since refactoring research typically focus on the detection of refactoring opportunities in the source code to recommend appropriate operations, we were particularly interested in extracting the intent behind the refactoring, to capture what typically triggers developers to refactor their code. As seen in Table 1, intents can be either 1) *generic*, using high-level keywords, such as *Code cleanup*, *Code revision*, *Code reformatting & reordering* etc.; or 2) *specific*, using keywords that are more in line with the concepts used by tools to recommend refactoring. To further ensure the correctness of our data, we conducted a pilot study with a sample of data to learn, explore, and understand what challenges we faced when classifying commit messages. Based on the pilot study, we define the three SAR categories (*i.e.,* internal, external, and code smell). In particular, developers typically state structural, size, complexity, and Object-Oriented metrics, such as coupling, composition, design size, etc. These metrics are the main drivers for many refactoring techniques [5, 20, 21, 22, 23, 24], and they are known in literature as *internal quality attributes*.

Also, developers do mention the correction and management of bad programming practices, also known in refactoring studies [3, 8, 10, 25, 26, 27] as *code smells*, *anti-patterns*, and *design defects*. Code Smell resolution is the removal of design defects that might violate the fundamentals of software design principles and decrease code quality. Examples of these code smells include duplicate code, dead code, long method, blob class, etc.

Finally, we extracted intents corresponding to what literature considers as *external quality attributes*. External quality attribute is the property or feature that indicates the effectiveness of a system such as understandability and readability. Many refactoring approaches are driven by the optimization of non-functional attributes such as testability, understandability, changeability,

Table 2: Quality Issues (Quality Attribute(s) & Code Smell(s)) Extracted from SAR Commits.

Internal QA	**External QA**	**Code Smell**
Inheritance (31.04%)	Functionality (34.03%)	Duplicate Code (43.52%)
Abstraction (30.63%)	Performance (31.37%)	Dead Code (24.84%)
Complexity (14.30%)	Compatibility (13.61%)	Data Class (22.93%)
Composition (12.53%)	Readability (3.60%)	Long Method (3.82%)
Coupling (3.81%)	Stability (2.64%)	Switch Statement (3.18%)
Encapsulation (3.61%)	Usability (1.60%)	Lazy Class (0.42%)
Design Size (2.11%)	Flexibility (1.58%)	Too Many Parameters (0.42%)
Polymorphism (1.50%)	Extensibility (1.54%)	Primitive Obsession (0.21%)
Cohesion (0.48%)	Efficiency (1.51%)	Feature Envy (0.21%)
	Accuracy (1.05%)	Blob Class (0.21%)
	Accessibility (1.04%)	Blob Operation (0.21%)
	Robustness (0.78%)	
	Testability (0.75%)	
	Correctness (0.65%)	
	Scalability (0.62%)	
	Configurability (0.56%)	
	Simplicity (0.55%)	
	Reusability (0.45%)	
	Reliability (0.43%)	
	Modularity (0.37%)	
	Maintainability (0.26%)	
	Traceability (0.26%)	
	Interoperability (0.24%)	
	Fault-tolerance (0.16%)	
	Repeatability (0.07%)	
	Understandability (0.06%)	
	Effectiveness (0.06%)	
	Productivity (0.06%)	
	Modifiability (0.03%)	
	Reproducibility (0.03%)	
	Adaptability (0.03%)	
	Manageability (0.01%)	

evolvability, and readability [28, 29, 30, 31, 32, 33].

The complete list of identified SAR patterns, per category, is depicted in Table 2. These categories, namely, *internal quality attributes*, *code smells*, and *external quality attributes* represent what existing refactoring techniques are using to identify refactoring opportunities in the source code, in order to recommend *pure* and *root-canal* refactorings, *i.e.,* behavior preserving code changes for the purpose of improving software quality. Figure 1 depicts how our classification clusters the existing refactoring taxonomy reported in the literature [7, 10, 21, 34, 35, 36, 37, 38]. As can be seen, our classification covers the majority of categories.

Also, it is important to note that existing studies, along with our manual analysis, have pointed out that refactoring can also be interleaved with other development tasks, such as updating functionalities, bug fixes, etc. We do not consider these categories (*e.g.,* Bug Fix, Functional etc.) as part of our classification, since it can be performed using previous studies [39, 40, 41, 42]. More recently, Paixão et al. [43] captured these additional refactoring categories (*i.e.,* Bug Fix and Feature). In the future, we plan to extend our work to capture this taxonomy as well.

It is worth noting that there are many studies analyzing the impact of refactoring on 1) code smells, 2) internal quality attributes, and 3) external quality attributes, but our work focuses on the developer's documentation, and not on the refactoring operations themselves and their impact. Our aim is to classify the intent. For example, when we classify a message stating the removal of duplicate code as a code smell, we are classifying purely the developer's intent of removing duplicate code, so we are not claiming that the performed refactoring operations had an impact on only the removal of the code smell. In fact, these refactoring operations may also have an impact on other internal quality attributes, but such analysis is not what we are trying to achieve in this paper. Simply, we are classifying the developer's intent and not the impact of the refactoring operations. The impact of refactoring operations has been heavily studied in literature and our study complements this effort by exposing what developers do care about when they refactor.

Finally, according to a recently published survey [44], refactoring is typically driven by intents that belong to the categories that we have used in this study.

2.3. Benefits

Commit messages are essential for not only comprehending code changes, but for many other aspects of software development, such as as classification of maintenance effort [39, 41], code change summarization [45], files change-proneness and bug-proneness [46], etc. For instance, recent studies have shown the feasibility of extracting insights of software quality from developers inline

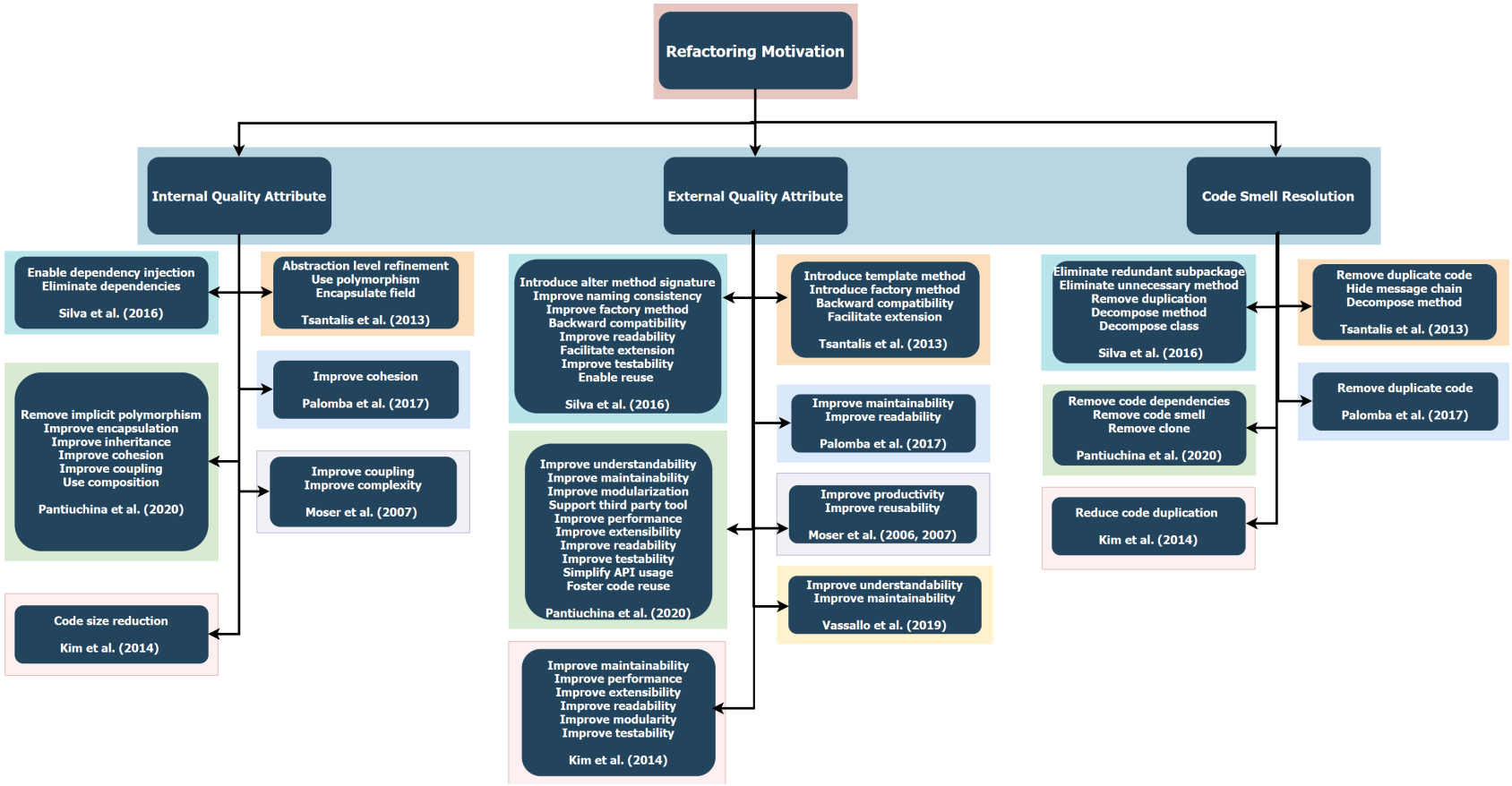

Figure 1: Refactoring Motivation.

documentation. For instance, mining developers comments has unveiled how developers knowingly commit code that is either incomplete, temporary, or faulty. Such phenomenon is known as *Self-Admitted Technical Debt* (SATD) [47]. Similarly, our previous study has introduced *Self-Affirmed Refactoring* (SAR) [14], defined as developers explicit documentation of refactoring operations intentionally introduced as code change. Per analogy to SATD, SAR manifests as a positive phenomenon, known to be one of the primary concepts to manage technical debt [48]. So, it is of particular interest to understand how the developer's intent to refactoring code leads to an adequate corrective action, *i.e.*, SATD resolution, especially that recent studies focus on understanding how SATD is being removed [49, 50, 51, 52].

When it comes to refactoring documentation, revealing the intents that are frequently pushing developers to refactor, is of a major importance for the community, especially that recent surveys have shown that refactoring tools are under-used, and developers are still manually refactoring their code [7, 53]. And so, these patterns can narrow the scope of refactoring towards what developers consider to be relevant, in order to bridge the gap between refactoring tools and their adoption in practice. However, the identification of these SAR patterns, is human-intensive, subjective, and error-prone. Coping with the burden of manual analysis is the main goal of this paper, by initially detecting then classifying these SAR patterns, into the above-mentioned categories. Furthermore, the automated identification of these SAR patterns, is not straightforward, as these keywords, are not necessarily exclusive to refactoring. Even *refactoring*, being the most intuitive keyword used to describe this activity, has been also found to be used out of its context [13].

Refactoring, just like any code change, has to be reviewed, before being merged into the code base. However, little is known about how developers *perceive* and *review* refactoring during the code review process, especially that refactoring, by definition, is not intended to alter to the system's behavior, but to improve its structure, so its review may differ from other code changes. Yet, there is not much research investigating the proper documentation of refactor-

ing, which can facilitate the process of reviewing it. Through the identification of these SAR patterns, many example of documented refactorings can be provided for future investigations and analysis.

3. Related Work

In this section, we report studies related to developer's perception of refactoring and its documentation, along with the current state-of-the-art studies related to commit messages classification.

3.1. Refactoring and its documentation

A number of studies have focused recently on the identification and detection of refactoring activities during the software life-cycle. One of the common approaches to identify refactoring activities is to analyze the commit messages in versioned repositories. Stroggylos & Spinellis [16] searched words stemming from the verb *"refactor"* such as "refactoring" or "refactored" to identify refactoring-related commits. Ratzinger et al. [15, 17] also used a similar keyword-based approach to detect refactoring activity between a pair of program versions to identify whether a transformation contains refactoring. The authors identified refactorings based on a set of keywords detected in commit messages, and focusing on the following 13 terms in their search approach: *refactor, restruct, clean, not used, unused, reformat, import, remove, replace, split, reorg, rename, and move*.

Later, Murphy-Hill et al. [53] replicated Ratzinger's experiment in two open source systems using Ratzinger's 13 keywords. They conclude that commit messages in version histories are unreliable indicators of refactoring activities. This is due to the fact that developers do not consistently document refactoring activities in the commit messages. In another study, Soares et al. [18] compared and evaluated three approaches, namely, manual analysis, commit message (Ratzinger et al.'s approach [15, 17]), and dynamic analysis (SafeRefactor approach [54]) to analyze refactorings in open source repositories, in terms of

behavioral preservation. The authors found, in their experiment, that manual analysis achieves the best results in this comparative study and is considered as the most reliable approach in detecting behavior-preserving transformations.

In another study, Kim et al. [7] surveyed 328 professional software engineers at Microsoft to investigate when and how they do refactoring. They first identified refactoring branches and then asked developers about the keywords that are usually used to mark refactoring events in commit messages. When surveyed, the developers mentioned several keywords to mark refactoring activities. Kim et al. matched the top ten refactoring-related keywords identified from the survey (*refactor, clean-up, rewrite, restructure, redesign, move, extract, improve, split, reorganize, rename*) against the commit messages to identify refactoring commits from version histories. Using this approach, they found 94.29% of commits do not have any of the keywords, and only 5.76% of commits included refactoring-related keywords.

Peruma et al. [55] investigated how method, class, and package identifier names evolve and how this evolution was documented in the commit corresponding messages. By also analyzing their surrounding refactoring operations, the authors observed how names are influenced by their neighboring changes. They extended their work by considering the situation where a rename is applied to an identifier whose data type is changed [56]. Such case is of an interest to the authors as it indicates a potential change in the behavior its associated identifier.

Prior work [57, 13, 14] has explored how developers document their refactoring activities in commit messages; this activity is called Self-Admitted Refactoring or Self-Affirmed Refactoring (SAR). In particular, SAR indicates developers' explicit documentation of refactoring operations intentionally introduced during a code change.

The existence of such patterns unlocks more studies that question the developer's perception of quality attributes (*e.g.,* coupling, complexity), typically used in recommending refactoring. For instance, AlOmar et al. [58] identified which quality models are more in-line with the developer's vision of quality

optimization when they explicitly mention in the commit messages that they refactor to improve these quality attributes. This study shows that, although there is a variety of structural metrics can represent internal quality attributes, not all of them can measure what developers consider to be an improvement in their source code. Based on their empirical investigation, for metrics that are associated with quality attributes, there are different degrees of improvement and degradation of software quality.

3.2. Commit Classification

A wide variety of approaches to categorizing commits have been presented in the literature. The approaches vary between performing manual classification [10, 27, 34, 61, 66, 68], to developing an automatic classifier [59, 62, 65], to using machine learning techniques [40, 42, 60, 63, 64, 69, 70, 71] and developing discriminative topic modeling [67] to classify software changes. We summarize these state-of-the-art approaches in Table 3.

Hattori and Lanza [61] developed a lightweight method to manually classify history logs based on the first keyword retrieved to match four major development and maintenance activities: Forward engineering, Re-engineering, Corrective engineering, and Management activities. Also, Mauczka et al. [66] have addressed the multi-category changes manually using three classification schemes from existing literature. Tsantalis et al. [34] conducted a multidimensional empirical study on refactorings and performed a systematic labeling of the commit messages to better understand the purpose of the applied refactorings. Silva et al. [10] applied a thematic analysis process to reveal the actual motivation behind refactoring instances after collecting all developers' responses. Further, a few studies [27, 68] propose the classification of refactoring instances as root-canal or floss refactoring through the use of manual inspection. Yan et al. [67] used discriminative topic modeling techniques to automatically classifying software changes.

Mockus & Votta [59] designed an automatic classification algorithm to classify maintenance activities based on a textual description of changes. Another

Table 3: Characteristics of Commit Classification Studies.

Study	Year	Manual/Automatic	Classification Method	Category	Machine Learning	Training Size	Result
Mockus & Votta [59]	2000	No/Yes	Automated Classifier	Maintenance Activities	N/A	40 maintenance requests (8 participants)	Accuracy: ∼ 61%
Amor et al. [60]	2006	No/Yes	Machine Learning	Swanson's category Administrative	NaiveBayes	400 commits (1 participant)	Accuracy: 70%
Hattori & Lanza [61]	2008	No/Yes	Keywords-based Search	Forward Engineering Reengineering Corrective Engineering Management	N/A	1088 commits	F-measure: 76%
Hassan [62]	2008	No/Yes	Automated Classifier	Bug Fixing General Maintenance Feature Introduction	N/A	18 commits (6 participants)	Agreement: 70%
Hindle et al. [39]	2008	Yes/No	Systematic Labeling	Swanson's category Feature Addition Non-Functional	N/A	2000 commits	Not mentioned
Hindle et al. [63]	2009	No/Yes	Machine Learning	Swanson's category Feature Addition Non-Functional	J48 / NaiveBayes / SMO KStar / IBk / JRip / ZeroR	2000 commits	F-measure: 51% Accuracy: 52%
Mahmoodian et al. [64]	2010	No/Yes	Machine Learning	Corrective & Adaptive	NaiveBayes / ADtree	1700 requests	Accuracy: 78%
Hindle et al. [40]	2011	No/Yes	Machine Learning	Non-Functional	rule / decision trees / vector space SVM / CLR / HOMER / BR	Not Mentioned	Receiver Operating Characteristic up to 80%
Mauczka et al. [65]	2012	No/Yes	Automated Classifier (Subcat tool)	Swanson's category Blacklist	N/A	21 commits (5 participants)	Precision: 92% Recall: 85%
Tsantalis et al. [34]	2013	Yes/No	Systematic Labeling	Code Smell Resolution Extension Backward Compatability Abstraction Level Refinement	N/A	Not Mentioned	Manual
Mauczka et al. [66]	2015	Yes/No	Systematic Labeling	Swanson's category Hattori & Lanza category [61] Non-Functional	N/A	967 commits	Manual
Yan et al. [67]	2016	No/Yes	Topic Modeling	Swanson's category	N/A	80 commits (5 participants)	F-measure: 76%
Silva et al. [10]	2016	Yes/No	Systematic Labeling	Refactoring's Motivation	N/A	N/A	Manual
Chavez et al. [68]	2017	Yes/No	Systematic Labeling	Floss Refactoring Root-canal Refactoring	N/A	sample of 2119	Manual
Cedrim et al. [27]	2017	Yes/No	Systematic Labeling	Floss Refactoring Root-canal Refactoring	N/A	part of sample of 2584	Manual
Levin & Yehudai [42]	2017	No/Yes	Machine Learning	Swanson's category	J48 / GBM / RF	1151 commits	Accuracy: 76%
Levin & Yehudai [69]	2019	No/Yes	Machine Learning	Swanson's category	J48 / GBM / RF	1151 commits	Accuracy: 76%
Honel et al. [70]	2019	No/Yes	Macine Learning	Swanson's category	LssvmRadical / SVM / GBM xgbTree / LDA / MDA / NN / avNNet C5.0 / RF / Naive Bayes / LogitBoost	1151 commits	Accuracy: up to 89%
This work		**No/Yes**	**Machine Learning**	**SAR & non-SAR** **Internal QA** **External QA** **Code Smell**	**LR / RF / GBM / DJ / SVM** **LD-SVM / NN / APM / BPM**	**1823 commits** (two-class) **1044 commits** (multiclass)	**Accuracy: 98%** **F-mesaure: 98%** **Accuracy: 93%** **F-mesaure: 93%**

automatic classifier is proposed by Hassan [62] to classify commit messages as a bug fix, introduction of a feature, or a general maintenance change. Mauczka et al. [65] developed an Eclipse plug-in named Subcat to classify the change messages into Swanson's original category set (*i.e.,* Corrective, Adaptive and Perfective [72]), with an additional category "Blacklist". He automatically assessed if a change to the software was due to a bug fix or refactoring based on a set of keywords in the change messages. Hindle et al. [39] performed a manual classification of large commits to understand the rationale behind these commits. Later, Hindle et al. [63] proposed an automated technique to classify commits into maintenance categories using seven machine learning techniques. To define their classification schema, they extended Swanson's categorization [72] with two additional changes: Feature Addition, and Non-Functional. They observed that no single classifier is the best. Another experiment that classifies history logs was conducted by Hindle et al. [40], in which their classification of commits involves the non-functional requirements (NFRs) a commit addresses. Since the commit may possibly be assigned to multiple NFRs, they used three different learners for this purpose along with using several single-class machine learners. Amor et al. [60] had a similar idea to [63] and extended the Swanson categorization hierarchically. They, however, selected one classifier (*i.e.,* Naive Bayes) for their classification of code transactions. Moreover, maintenance requests have been classified using two different machine learning techniques (*i.e.,* Naive Bayesian and Decision Tree) in [64]. McMillan et al. [71] explored three popular learners to categorize software application for maintenance. Their results show that SVM is the best performing machine learner for categorization over the others.

Levin and Yehudai [42] automatically classified commits into three main maintenance activities using three classification models namely, J48, Gradient Boosting Machine (GBM), and Random Forest (RF). They found that the RF model outperforms the two other models (accuracy: 76% versus 70% and 72%). In their extended work [69], the RF model showed a promising accuracy of 76%. More recently, a replicated study [70] of [42] introduced code density of a com-

mit to study the purpose of a change. Using code-density based classification, they achieved up to 89% accuracy for cross project commit classification using LogitBoost classifier.

In this paper, we build on top of these techniques to leverage an automated identification and classification of SARs. Although the manual summarization of SAR is useful, it is considered as a time-consuming task because of the required manual effort to derive the list of patterns. Although much work has been done on automatically classifying commits in general, there is no currently automatic way to identify SAR patterns specifically. Several studies [39, 42, 60, 61, 63, 65, 66, 67, 69, 70] have discussed how to automatically classify change messages into Swanson's general maintenance categories (*i.e.,* Corrective, Adaptive, Perfective). Refactoring, in general, has been classified as a sub-type of "Perfective" in this maintenance category. Currently, there is no study that reports specific subcategories of refactoring extracted from real-world scenarios of commit messages and performs an automated classification of SAR commits. Therefore, in this paper, we push the Self-Affirmed Refactoring research a step forward by introducing an automatic classification approach to (1) determine whether a commit contains SAR or not (*cf.*, Table 1), and (2) classify SAR into its three categories (see Table 2). Compared with the pattern-based approach, our automated approach can identify more SAR patterns that can complement and extend the list of patterns identified in [14].

Further, in this work, we are detecting the indicators of refactoring to understand how developers document refactoring. We are not labelling refactoring operations themselves; we are instead labelling the commit messages that are found to contain refactoring operations. The existence of refactoring operations, in the studied commits, can be verified by running state-of-the-art tools, such as Refactoring Miner [12] and RefDiff [11] tools. Both of these studies indicated that their tool achieves high accuracy (precision of 98% and 100%, and recall of 87% and 88%, respectively), which gives us confidence to use one of these tools as a form of validation that the commits contain refactoring.

As can be seen in table 3, commit messages are extensively used in existing

literature to classify several maintenance-related tasks. Studies that focus on classifying bug and feature requests have used commit messages as a primary source of information to generate high accuracy and applicable results. However, our approach is not restricted to a specific source of textual information. Future work could replicate our approach with other types of metadata, *e.g.,* issue descriptions.

4. Approach

In this section, we first provide an overview of our approach. Then, we elaborate on the technical details of the adopted classification technique, in the following subsection. The overview of our approach is depicted in Figure 2, and a sample of commit messages is demonstrated in Figure 3.

4.1. Data Collection

To collect the necessary commits, we refer to an existing large dataset of links to GitHub repositories [73]. We perform an initial filtering, using Reaper [74], to only navigate through well-engineered projects while verifying that they were Java-based; the only language supported by Refactoring Miner. The authors of this dataset classified "well-engineered software projects" based on the projects' use of software engineering practices such as documentation, testing, and project management. So, we ended up reducing the number of selected projects from 57,447 to 3,795.

Using "well-engineered" and "well-documented" kind of interchangeably - although we acknowledge the potential value of having a more diverse set of projects, and our findings may not extend to projects that are not as well-documented, because our primary research methods rely on documentation, we chose to focus on projects that would be likely to have high-quality documentation (*i.e.,* commit messages) consistently available.

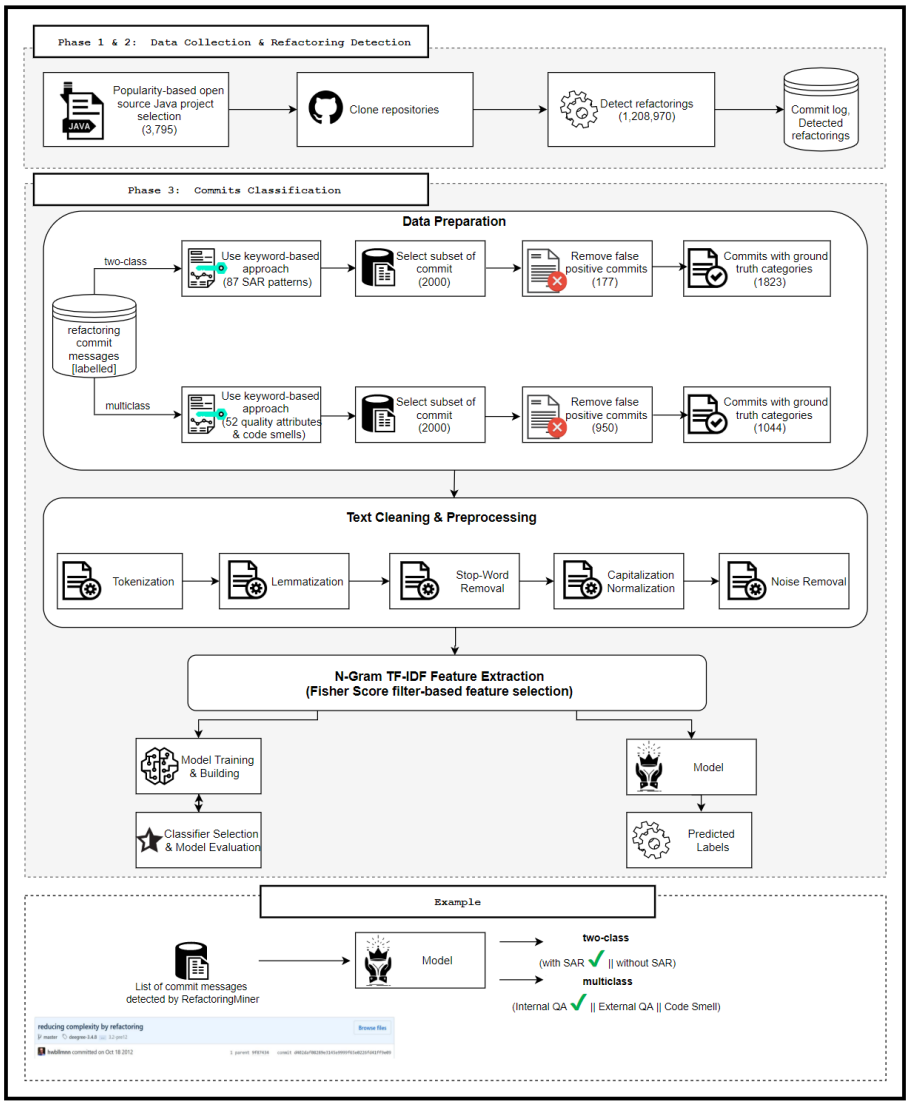

Figure 2: Overall Classification Framework.

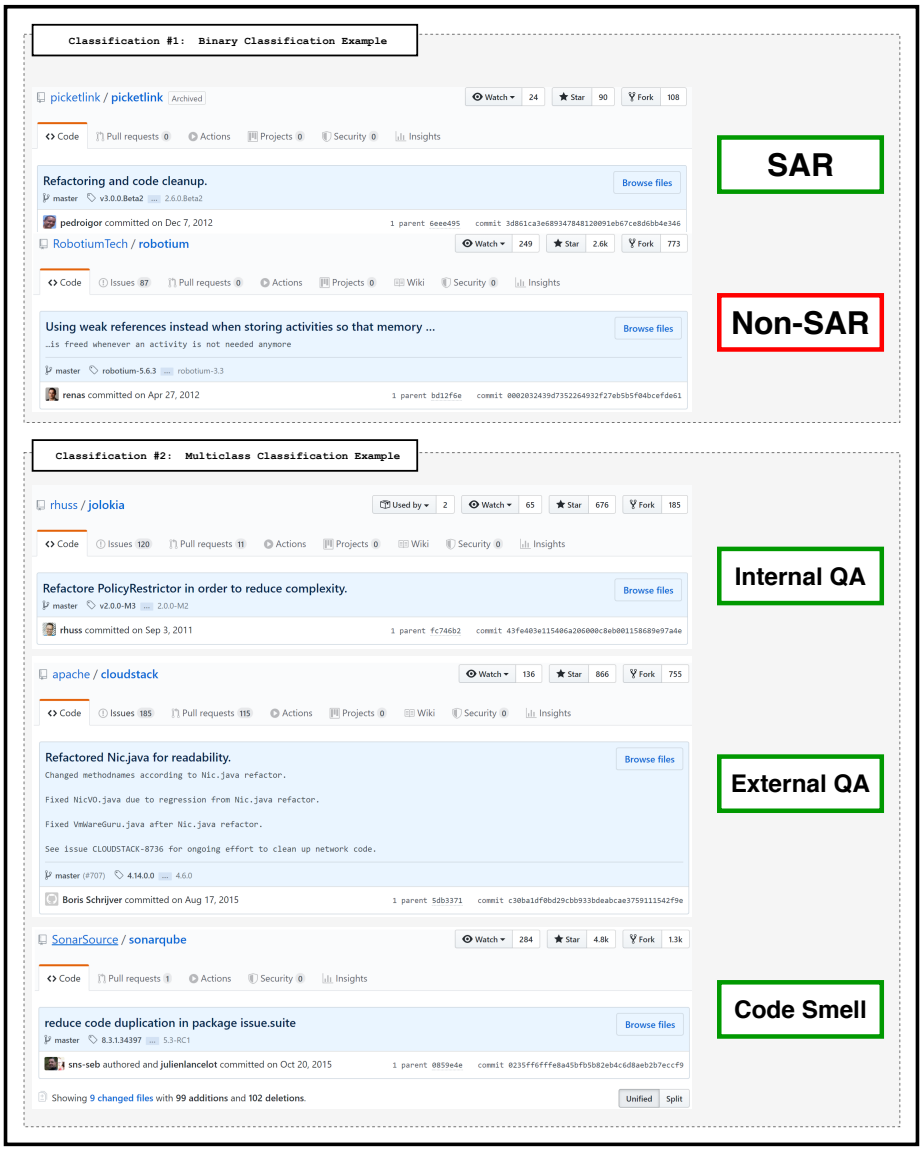

Figure 3: Commit Message Examples for Binary and Multiclass Classification.

4.2. Refactoring Detection

To extract the entire refactoring history in each project, we use the popular refactoring mining tool, *i.e.,* Refactoring Miner [10]. Our choice to use Refactoring Miner is justified by the fact that it achieved the highest accuracy in detecting refactorings compared to the state-of-the-art available tools, with a precision of 98% and recall of 87% [10, 12] along with being suitable for our study that requires a high degree of automation in data mining. In this phase, We collect a total of 1,208,970 refactoring operations from 322,479 commits, applied during a period of 23 years (1997-2019).

4.3. Overall Framework

In a nutshell, the goal of our work is to automatically identify then classify commit messages containing refactoring documentation, *i.e.*, Self-Affirmed Refactorings, (for the sake of simplicity, we refer to them as SAR). Our approach takes as input, a commit message, and makes a binary decision on whether it contains SAR or not. If a SAR is detected, it classifies it into one of of three common categories: (*i*) internal quality attribute (*ii*) external quality attribute, and (*iii*) code smell [14]. The overall framework of our approach is depicted in Figure 2. We formulate a two-phased approach that consists of a model building phase and a prediction phase. In the model building phase, our goal is to build a model from a corpus real world documented refactoring operations (*i.e.,* commit messages). In the prediction phase, the model created in the previous phase will be used to predict categories of new refactoring-related commit messages.

Our framework takes commit messages along with their ground truth categories obtained by manual inspection as input for the training procedure extracted from different projects, provided by a previous study [14]. Based on this input, the commit messages are preprocessed, allowing for informative featurization. Next, for each commit message, we extract features (*i.e.,* words) to create a structured feature space. Then, we use the extracted features to build the training set. In total, we experimented 9 commonly used classifiers to evaluate our model for prediction. We selected these classifiers as they are typically used

in previous commit classification studies as well as several software engineering classification/prediction problems [40, 42, 60, 63, 64, 69, 70], as outlined in Table 3. After training all models, we use a testing set to challenge the performance. Since the model has already learned the vocabulary of N-Gram (discussed in Section 4.4.3) and their weights from the training dataset, we extract features from the test data based on that vocabulary and weights, and input them to the model. Finally, the classifier will output the predicted label for each tested commit message.

4.4. Commit Classification

Our classification process has five main phases: (1) data preparation, (2) text cleaning and preprocessing, (3) feature extraction using N-Gram, (4) model training and building, and (5) classifier selection and model evaluation. Since a commit message is written in plain text, we follow the approach provided by Kowsari et al. [75] that discussed a recent trend in text classification techniques and algorithms.

4.4.1. Data Preparation

Our goal is to provide the classifier with sufficient commits that represent the categories analyzed in this study. Since the number of candidate commits to classify is large, we cannot manually process them all, and so we need to randomly sample a subset while making sure it equitably represents the featured classes, *i.e.,* categories. Since an imbalanced training dataset or class starvation (*i.e.,* not having adequate instances of a certain class) could worsen the performance of the model [42, 69], we make sure that the classes for two-class (*i.e.,* with or without SAR) and multiclass (*i.e.,* Internal QA, External QA, and Code Smell) classification problems are equally distributed when preparing the data for the training (cf. Table 4). The classification process has been performed by the authors of the paper. To approximate the needed number of commits to add, we reviewed the thresholds used in the studies related to commit classification (see Table 3). The highest number of commits used in comparable studies was

around 2,000 commits [39, 63, 68]. Thus, we select a sample of 2,000 commits from 3,795 projects for each classification model. Below we detail the manual analysis of the data we use for our classification.

Table 4: Number of Instances per Class.

Dataset	**with SAR**	**without SAR**	
1,823 instances	912	911	
Dataset	**Internal QA**	**External QA**	**Code Smell**
1,044 instances	348	348	348

For data preparation, building the ground truth is challenging since we are looking for a particular set of commits. To prune the search space, we started with using an existing dataset of commits [14], manually inspected and validated for containing refactoring operations and an associated description at the commit message. We intend to build our own dataset by choosing a subset of this dataset, in a way to serve the purpose of the binary and multiclass classification.

To prepare the dataset of the binary classification, we need to create two groups of commits, *i.e.,* commits with or without SAR. The first group (with SAR) is created by randomly sampling commits, previously known to contain SAR patterns listed in Table 1. We further perform another round of individual verification by the authors before adding them to the group. Commits for which there was no full agreement by the authors were excluded from our dataset. The second group (without SAR) can be easily created by randomly choosing commit messages that simply do not contain these SAR patterns, but since we do want to strengthen our decision boundary, we intend to choose commits that are closest to neighboring regions between the two classes. To do so, for each commit from the first group (with SAR), we locate the set of its contiguous commits (committed either before or after), and performed by the same committer, then we randomly sample one of them to be added to the second group (without SAR), after manually verifying that it does not contain any description of a refactoring activity.

For the multiclass classification, we build it by making sure the chosen commits belong to one of the three categories listed in Table 2. To avoid involving our interpretation, it is important to note that the description of the categories listed in Table 2 needs to be explicit in the commit messages. We used stratified sampling to select 2,000 commit messages for manual classification, divided equally for each stratum. To ensure that these commits reported developers' intention to perform refactoring, and to improve quality attributes or fix code smells, we inspected these commits to remove false positives.

To avoid having false positive commits, we applied the filtering to narrow down the commit messages eliminating the ones that are less likely to be classified as self-affirmed refactoring. We designed the filtering to help ensure that we only trained the algorithm on higher-quality commit messages [76].

We followed the process from existing papers in filtering commit messages [65, 77, 78]. For example, Fu et al. [77] filtered out short commit messages. Mauczka et al. [65] used the "Blacklist" category to filter all commits, which underlying modifications were not carried out by humans or which do not actually include any source code modifications. In our work, we apply five filtering heuristics to narrow down the commit messages eliminating the ones that are less likely to be classified as SAR. It is important to note that we removed short commit messages from the training, but not from the testing set because (1) short commit messages do not contain enough information and do not clearly describe the purpose of code change , and (2) we want to train the classifier on well-documented commit messages, and label commits that contain enough information about refactorings. Prior study has pruned short commit messages since these will be noise for the classifiers, and they did not record the cause of the changes [77]. Some criteria we used for filtering were as follows:

- If a commit contains an alternative form of the word "refactor" such as "re-factor*", the commit was classified as SAR commit.
- If a commit message contains a pattern that is in a slightly different form of one of the patterns, such as "simplify the code" and "simplify code",

the commit was classified as SAR commit.

- Commits that were either too short or ambiguous were discarded. Some examples of hard-to-classify commit messages are: "*Solr Indexer ready*"[2], "*allow multiple collections*"[3], and "*Auto configuration of AgiScripts*"[4].

- If one commit could belong to more than one class, it was excluded.

- If the quality attribute is a part of the identifier name, the commits were excluded, *e.g.,* "*SONARJS-541 Precise issue location for ExpressionComplexity (S1067)*". We discarded this commit because "complexity" is referring to a part of a class name and not a quality attribute.

The above-mentioned examples of ambiguous commit messages prevent us from being confident, and hence, for each discarded commit message, we randomly sampled another replacement. We repeated this process until we found the commit message that we were able to confidently classify. Because of the random nature of the process, some classes were saturated faster than others, so we kept increasing the number of instances only for the underrepresented classes, until we find the right balance between all classes. The criteria listed above reduced the number of commits and helped us focus on the most insightful commit messages. For the binary classification, 177 commits were removed because of them either being short or ambiguous. Also, in our case, any message with less than 7 characters was too short for us to decide. The evaluation resulted in keeping 1,823 commits and 1,044 commits, respectively for two-class and multiclass classification problems. To mitigate the risk of having a biased dataset and to inspect the level of agreement of the manual classification, we extract stratified sample of our dataset that are classified by the first author, and have these sample commits independently classified again by the second author. Particularly, similar to [69], in order to inspect manual classification agreement,

[2]https://github.com/01org/graphbuilder
[3]https://github.com/0install/java-model
[4]https://github.com/1and1/attach-qar-maven-plugin

we randomly classified a 10% sample of commits, *i.e.,* 186 and 105 commits out of the 1823 and 1044 for two-class and multiclass classification problems, respectively. This quantity roughly equates to a sample size with a confidence level of 95% and a confidence interval of 8. We used Cohen's Kappa coefficient [79] to evaluate the inter-rater agreement level for the categorical classes. We achieved an agreement level of 0.96 for the two-class classification, and 0.87 for multiclass classification. According to Fleiss et al. [80], these agreement values are considered to have an almost *perfect agreement* (*i.e.*, 0.81–1.00).

The result of this classification is available in the reproduceability package of this work, thus, it can be reused and extended [19].

4.4.2. Text Cleaning & Preprocessing

After the data preparation phase, we applied a similar methodology explained in [75, 81] for text pre-processing. In order for the commit messages to be classified into correct categories, they need to be preprocessed and cleaned; put into a format that the classification algorithms will process. This way, the noise will be removed, allowing for informative featurization. To extract features (*i.e.,* words), we preprocess the text as follows:

- **Tokenization:** The goal of tokenization is to investigate the words in a sentence. The tokenization process breaks a stream of text into words, phrases, symbols, or other meaningful elements called tokens [75]. In our work, we tokenize each commit by splitting the text into its constituent set of words. We also split tokens on special characters (*e.g.,* the string "package-level" would be separated into two tokens, "package" and "level").

- **Lemmatization:** The lemmatization process either replaces the suffix of a word with a different one or removes the suffix of a word to get the basic word form (lemma). In our work, the lemmatization process involves sentence separation, part-of-speech identification, and generating dictionary form. We split the commit messages into sentences, since input text

could constitute a long chunk of text. The part-of-speech identification helps in filtering words used as features that aid in key-phrase extraction. Lastly, since the word could have multiple dictionary forms, only the most probable form is generated.

- **Stop-Word Removal:** Stop words (*i.e.,* words and common English words such as "is", "are", "if", etc) are removed since they do not play any role as features for the classifier [82].

- **Capitalization Normalization:** Since text could have a diversity of capitalization to form a sentence and this could be problematic when classifying large commits, all the words in the commit messages are converted to lower case and all verb contractions are expanded.

- **Noise Removal:** Special characters and numbers are removed since they can deteriorate the classification. More specifically, we remove all numeric characters, unique and duplicate special characters, email addresses and URLs.

Table 5: Performance of Different Classifiers (Binary Classification).

Classifier	**Precision**	**Recall**	**Accuracy**	**F-measure**
Logistic Regression	0.98	0.93	0.96	0.95
Random Forest	0.98	0.98	0.98	0.98
Gradient Boosted Machine	0.98	0.98	0.98	0.98
Decision Jungle	0.97	0.94	0.95	0.95
Support Vector Machine	0.96	0.94	0.95	0.95
Locally Deep SVM	0.97	0.93	0.95	0.95
Neural Network	0.98	0.92	0.95	0.95
Averaged Perceptron Method	0.97	0.93	0.95	0.95
Bayes Point Machine	0.83	0.85	0.84	0.84

Table 6: Performance of Different Classifiers (Multiclass Classification).

Classifier	**Precision**	**Recall**	**Accuracy**	**F-measure**
Logistic Regression	0.93	0.93	0.93	0.93
Random Forest	0.93	0.93	0.93	0.93
One-vs-All Gradient Boosted Machine	0.93	0.93	0.93	0.93
Decision Jungle	0.89	0.88	0.88	0.88
One-vs-All Support Vector Machine	0.91	0.91	0.91	0.91
One-vs-All Locally Deep SVM	0.90	0.90	0.90	0.90
Neural Network	0.91	0.91	0.91	0.91
One-vs-All Averaged Perceptron Method	0.91	0.90	0.90	0.91
One-vs-All Bayes Point Machine	0.83	0.83	0.83	0.83

4.4.3. Feature Extraction Using N-Gram

After cleaning and preprocessing the text, we apply feature extraction to extract only the most useful information from text strings to differentiate classes in both classification problems. In particular, we selected the N-Gram technique for feature extraction. The N-Gram technique is a set of *n-word* that occurs in a text set and could be used as a feature to represent that text [75]. In general, N-Gram term has more semantic than an isolated word. Some of the keywords (*e.g.,* "improve") do not provide much information when used on its own. However, when collecting N-Gram from commit message (*e.g., Refactor:Remove redundant method names, extract method, improve usability*), the keyword "improve" clearly indicates that this is a SAR commit. In our classification, we use bigrams since it is very common to enhance the performance of text classification [83], and we select Fisher Score filter-based feature selection [84, 85] to *featurize* text and manage the size of the text feature vector, similar to [81]. As for the weighting function, we used the standard Term Frequency-Inverse Document Frequency (TF-IDF) [86] due to its popularity in the research community (the value for each N-Gram is its TF score multiplied by its IDF score). Thus, each preprocessed word in the commit message is assigned a value which is the weight of the word computed using this weighting scheme. TF-IDF gives greater weight (*e.g.,* value) to words which occur frequently in fewer

documents rather than words which occur frequently in many documents.

4.4.4. Model Training and Building

In this phase, we performed the 10-fold cross-validation technique to assess the variability and reliability of the classifier. Specifically, for each of the classification methods, we combined the commit messages into a single large dataset. Then, we split the dataset into ten folds, where each fold contained an equal proportion of commit messages. Thereafter, we performed ten evaluation rounds with different testing dataset in which nine folds were used as training dataset and the remaining one of the ten folds is used as the testing dataset. We aggregated the results of the ten evaluation rounds and reported the average performance for each classifier.

4.4.5. Classifier Selection and Model Evaluation

Selecting the proper classifier for optimal classification of the commits is a rather challenging task [87]. Best practices suggest that developers document their commits by providing a commit message along with every commit they make to the repository. These commit messages are usually written using natural language, and generally convey some information about the commit they represent. In this study, we are dealing with two-class and multiclass classification problems since the commit messages are categorized into two and three different types as explained in Table 1 and 2, respectively. Because we have a predefined set of categories, our approach relies on supervised machine learning algorithms to assign each commit message to one category. Since it is very important to come up with an optimal classifier that can provide satisfactory results, several studies have compared several classifiers such as K-Nearest Neighbor (KNN), Naive Bayes Multinomial, Gradient Boosting Machine (GBM), and Random Forest (RF) in the context of commit classification into similar categories [42, 69, 81]. These studies found that Random Forest (RF) achieves high performance. We investigated each classifier ourselves using common statistical measures (*precision, recall, accuracy, and F-measure*) of classification perform-

ance to compare each. It is important to note that the calculation of F-measure for multiclass classification is not supported by Azure Machine Learning (Azure ML). Thus, to facilitate comparison and to have all statistical measures that are consistent with two-class classification, we compute F-measure for multiclass in terms of precision and recall using the following formula:

$$F = 2 * \left(\frac{Precision * Recall}{Precision + Recall} \right) \tag{1}$$

where Precision (P) and Recall (R) are calculated as follows:

$$P = \frac{tp}{tp + fp}, R = \frac{tp}{tp + fn}$$

It is worth noting that a few models that we consider are inherently binary classifiers. In order to adjust for multiclass classification, each classifier applies the One-vs-All strategy for issues that require multiple output classes [88]. Thus, to ensure fairness, we use One-vs-All strategy for multiclass classification when using the following five classifiers: Gradient Boosted Machine (GMB) [89], Support Vector Machine (SVM) [90], Locally Deep SVM (LD-SVM) [91], Averaged Perceptron Method (APM) [92], and Bayes Point Machine (BPM) [93]. The remaining classifiers, considered in this study, are: Logistic Regression (LR) [94], Random Forest (RF) [95], Decision Jungle (DJ) [96], and Neural Network (NN) [97]. Our experiment is conducted using Microsoft Azure Machine Learning (Azure ML [98]), as it provides a built-in web-service once the classification models are deployed.

5. Results & Discussion

In this section, we conduct an empirical study to assess the performance of our approach. To evaluate different commit classification models, we used standard statistical measures to measure the performance of the classification (*Precision, Recall, Accuracy and F-measure*). In the following, we report the results of our research questions.

Replication package. We provide our comprehensive experiments package available in [19] to further replicate and extend our study.

5.1. *RQ1: Is it possible to accurately perform two-class and multi-class SAR classification using our machine learning technique?*

As shown in previous work [14], SAR can be extracted from commit messages. However, there is a lack of automatic techniques to classify them. In this work, we performed an automated approach to classify SAR to determine if the classification using machine learning techniques can result in high accuracy. A comparison between classification algorithms is reported in Table 5 and 6. The best performing model was used to classify the test dataset. Based on our findings, the F-measure of Random Forest (RF) and Gradient Boosting Machine (GBM) are respectively 98% and 98% which are clearly higher than their competitors for the two-class classification. For the multiclass, in addition to RF and GBM, Logistic Regression (LR) outperforms the other models with F-measure of 93%. Figures 4, 5, and 6 show the detailed performance for the best multiclass classifiers.

Random Forest and boosting learning machines belong to the family of ensemble learning machines, and have typically yielded superior predictive performance mainly due to the fact that they both aggregate several learnings. As for Logistic Regression, the fact that Logistic Regression achieves comparable performance as Random Forest and Boosting can be explained by the fact that the underlying true model for the text data has an inherent structure that matches the logistic regression assumption.

Another observation with regard to the classifiers accuracy is that few of the classifiers we considered in our study (GBM, SVM, LD-SVM, APM, and BPM) are inherently binary classifiers, and we used the One-vs-All strategy to adapt them for multiclass. Hence, these classifiers give us higher accuracy when performing binary classification compared to multiclass classification (98% vs 93%, respectively).Another reason for getting a different accuracy value when identifying multiclass labels vs two-class is that some commit messages could potentially belong to multiple categories. Hence, the machine learning classifiers, considered in this study, got confused when classifying such commit messages. Figures 7 and 8 show two cases of commit messages that confused the classifiers

when performing two-class and multiclass classifications, respectively. The first commit message (Figure 7) contains a pattern (*i.e.,* changing package name) that is a synonym of the patterns "renam*" or "use better name". The second commit message (Figure 8) contains more code element-related keywords such as "method" or "class" tend to be classified as "Code Smell" since code smell-related commits usually contain more description of the code elements that need to be optimized. This commit example targets to improve the flexibility at the design phase, which should be classified as "External QA".

Moreover, it is important to note that the classes used in this study categorize mainly the refactoring documentation and do not reflect the overall activities of the commits. Hence, these commit messages do not strictly contain refactoring code changes, especially that we noticed that refactoring tends to be interleaved with other software engineering tasks, such as fixing bugs, migrating type changes etc. Therefore, it is important to consider such *context* to better understand the intention behind the application of refactoring, and this will be our main future research direction.

Summary. We find that our approach is accurately identifying the SAR patterns and the three common quality improvements with an F-measure of 98% and 93% for the two-class and multiclass classification problems, respectively.

5.2. ***RQ2: How effective is our machine learning approach in classifying SAR?***

The main goal of this research is to propose an automatic approach to classify SAR commits that can effectively outperform the classification over the current state-of-the-art baselines, *i.e.,* Pattern-based [14] and Random classifier [78]. The selection of the two baseline approaches to compare against our approach was similar to Da Silva et al. [78]. We opt to choose a pattern-based approach because the methods used so far to identify refactoring commits [7, 13, 15, 16, 18, 30, 53, 65] and analyze refactoring activity [18] heavily

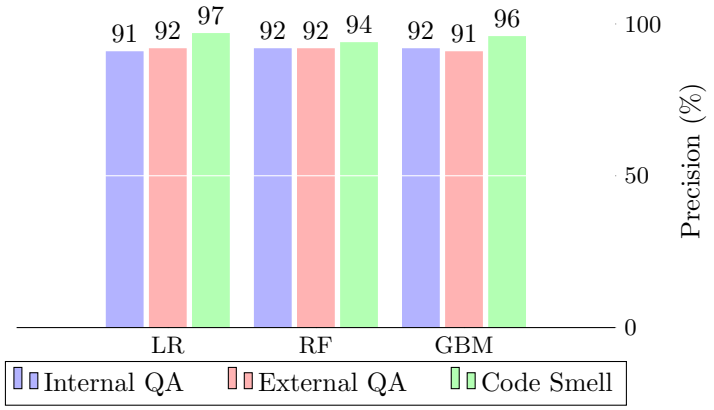

Figure 4: Visualization of the Precision for Different Classifiers (Multiclass)

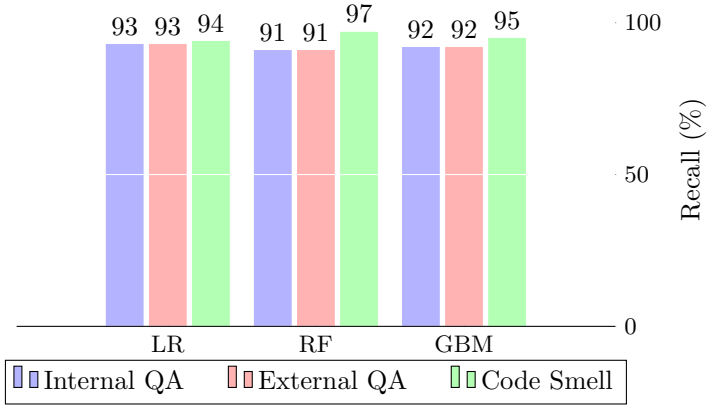

Figure 5: Visualization of the Recall for Different Classifiers (Multiclass).

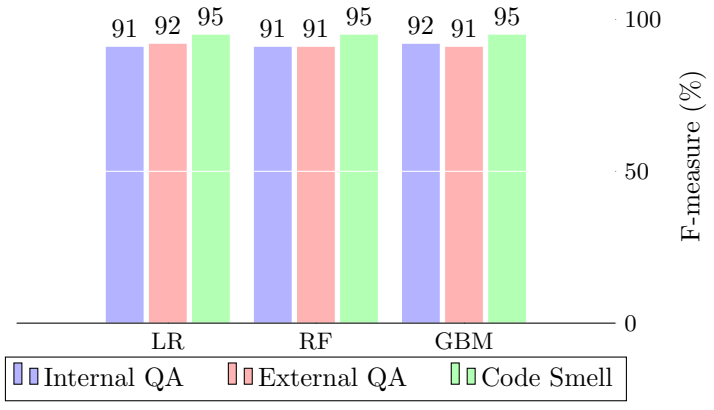

Figure 6: Visualization of the F-measure for Different Classifiers (Multiclass).

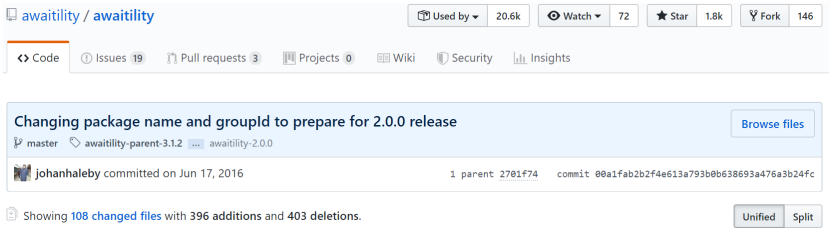

Figure 7: Example of Refactoring Commit Message that Confused the Classifiers (Two-class).

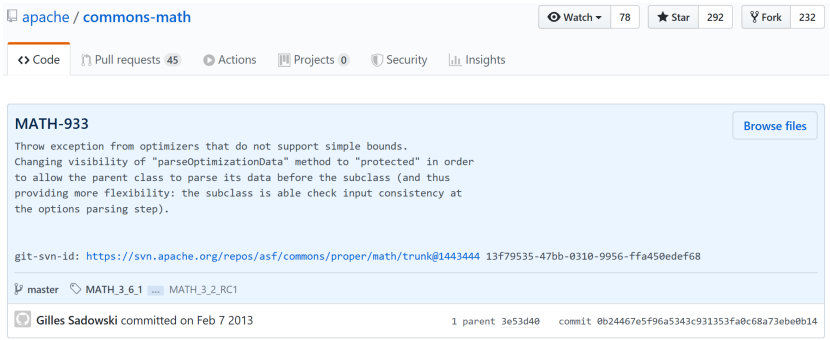

Figure 8: Example of Refactoring Commit Message that Confused the Classifiers (Multiclass).

rely on string matching. Other studies (*e.g.*, [65]) that focused on classifying commit messages on Swanson's categories (Corrective, Adaptive, Perfective) also used keyword-based approach. Currently, there is no evidence on how well pattern-based approaches perform. The choice of random classifier was similar to [78] that assumes that the detection of self-affirmed refactoring is random. Existing studies (*cf.*, Table 3) that have applied machine learning techniques in similar contexts (i.e., text classification) usually evaluate their approach using different classifiers. To compare their approach against others, they consider the keyword-based approach. To our knowledge, the only study that considers additional approach (*i.e.*, random classifier) is the study by Da Silva et al. [78]. Thus, we consider keyword-based and random classifiers to compare against our approach.

Answering this research question would shed light on whether the classification of SAR is a learning problem or not. We hypothesize that if learning

algorithms cannot outperform a String matching algorithm, then there is no need for proposing such a framework. The two chosen baselines, for this investigation, are listed below:

- **Baseline 1 (Pattern-based technique):** The pattern-based approach in identifying SAR is proposed by AlOmar et al. [14]. In their work, they identified 87 recurring patterns in SAR commit messages. We use these patterns as indicators of refactoring activities, *i.e.,* if a pattern exists in a commit, it is then classified as a SAR.

  In order to calculate the standard statistical metrics for this baseline, we use a set of 1,823 and 1,044 commit messages (*cf.,* Table 4) from the list of SAR and non-SAR commits and from each class of the multiclass classification respectively. We use them to perform a manual inspection to identify true positives (*tp*), true negatives (*tn*), false positives (*fp*), and false negatives (*fn*). True positives are cases when the pattern-based approach correctly identified SAR commits, and true negatives are commits correctly classified as without SAR. Similarly, false positives are commits classified as being SAR when they are not and finally false negatives are commits classified as without SAR when they are really SAR commits. Thus, using the *tp*, *tn*, *fp*, and *fn* values, we compute the precision, recall, and F-measure.

- **Baseline 2 (Random classifier):** Similar to Da Silva et al. [78], we consider Random classifier as one of the baselines to compare against our approach. The rationale behind using this random classifier to hold our approach accountable for providing significantly better results in comparison with a random classification. The precision of this approach is calculated by taking the total number of SAR over the total number of commit messages for all projects. As for the recall, there is a 50% chance that commit messages will be classified as SAR. The calculation of F-measure is explained previously in Section 4.

Table 7 and Figure 9 present the experimental results of our approach com-

pared with baseline 1 (Pattern-based), and baseline 2 (Random classifier). For our approach, we consider the highest F-measure score to compare against the other two baselines. Our approach provides an improvement over the comment patterns, outperforming it by 1.53 times and 1.45 times for two-class and multiclass respectively. We can see from Table 7 that our approach outperforms the simple Random baseline by 1.84 times and by 22.14 times respectively for two-class and multiclass classifications.

To better analyze our findings, after deploying our models as a web-service, we validate the two-class and multiclass models by randomly selecting 500 and 363 new commit messages, respectively. These new commit messages contain all types of commits (*e.g.,* short commit messages, commits with more than two classes, and commits with quality attributes as part of the identifier names). We manually read through commit messages that were classified as SAR commits in the prediction phase, and were classified as non SAR according to the pattern-based approach. Intuitively, such results induce the existence of features that represent the refactoring activity, and they are not captured by the previous study. Indeed, we found a set of featured keywords that do indicate refactoring activities (*e.g.,* "Tidy code", "repackage", and "fix bad merge and coding style issues"), and were not reported by any of the previous studies related to refactoring documentation. Such featured patterns could complement the list of manually identified 87 SAR patterns. Figure 10 reveals examples of these new patterns.

Table 7: Comparision of Statistical Measures between our Approach, Pattern-based and the Random Classifier.

Classification	**Our approach**			**Pattern-based**			**Random Classifier**		
	Precision	**Recall**	**F-measure**	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F-measure**
Two-class	0.98	0.98	0.98	1.00	0.47	0.64	0.61	0.5	0.53
Multiclass	0.93	0.93	0.93	0.97	0.48	0.64	0.02	0.5	0.042
Two-class Improve.	–	–	–	0.98 x	2.08 x	1.53 x	1.60 x	1.96 x	1.84 x
Multiclass Improve.	–	–	–	0.95 x	1.93 x	1.45 x	46.5 x	1.86 x	22.14 x

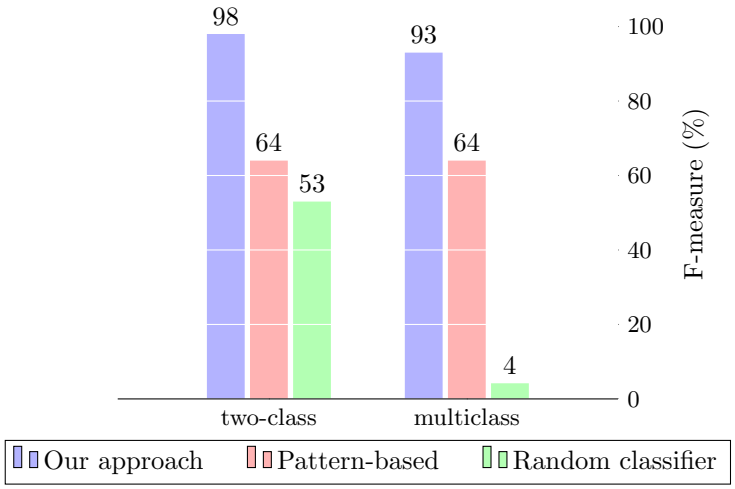

Figure 9: Visualization of the F-measure for Different Approaches.

Summary. We find that our approach can effectively outperform the classification over the current state-of-the-art baselines. We achieved an F-measure of 98% when identifying SAR commits (an average improvement of 1.53 x and 1.84 x over the state of the art approaches), and an F-measure of 93% when identifying the common quality improvement categories (an average improvement of 1.45 x and 22.14 x over the state of the art approaches). Additionally, our approach identifies more patterns that complement the list of manually identified 87 SAR patterns.

5.3. ***RQ3: How much training dataset is needed to effectively classify self-affirmed refactoring?***

After assessing the accuracy of our approach in classifying SAR commits, we want to investigate the amount of training data that is needed to effectively classify SAR. Our approach will be easily extended if a small dataset can be used for SAR identification. On the other hand, if a large number of commits are required, then our approach requires considerable time and effort.

To answer this research question, we incrementally add training data and

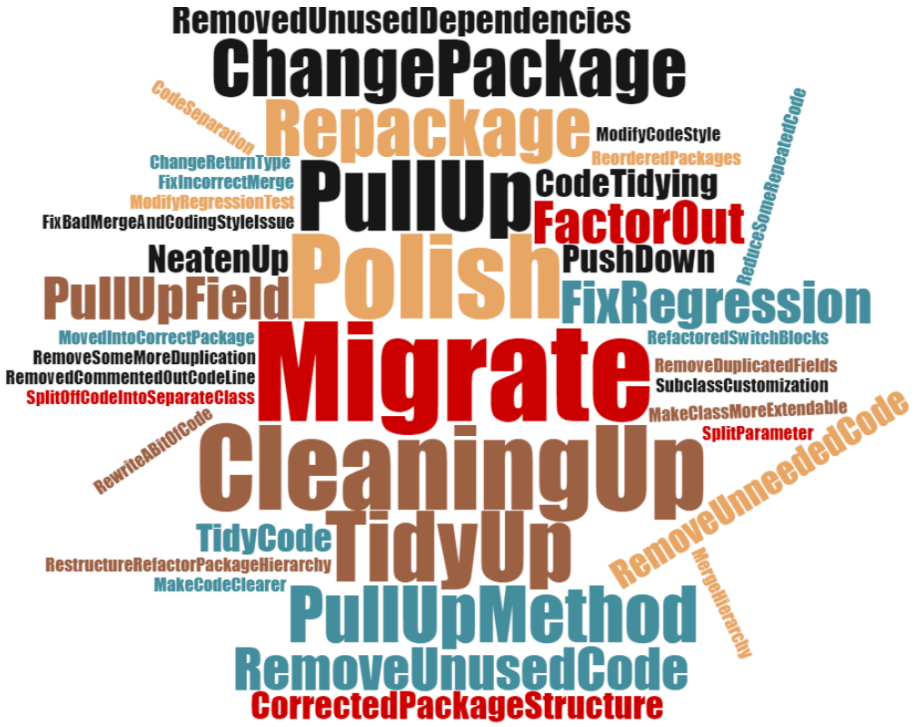

Figure 10: Sample of Patterns Identified by Automatic Classification.

assess the classifier's performance. We start by randomly selecting a stratified sample of 11,000 commits for each stratum (*i.e.,* SAR and non-SAR commits) provided by the authors of [14], and combining these commits into a single large dataset. Then, we follow the classification process discussed in Section 4, which results in 5,000 equally divided for each class. We then split the dataset into 10 folds with equal size, ensuring that each partition has the same ratio of SAR and non-SAR commits. For the multiclass classification problem, however, we use only a stratified sample of 1,044 commits discussed in Section 4. The reason for only considering these commits is that we are restricted by the minimum number of commits belonging to the code smell category. Thus, to avoid having an imbalanced training dataset, we keep the training size as it was originally set up. We discard the 44 commits since this number is less than the selected batch training size.

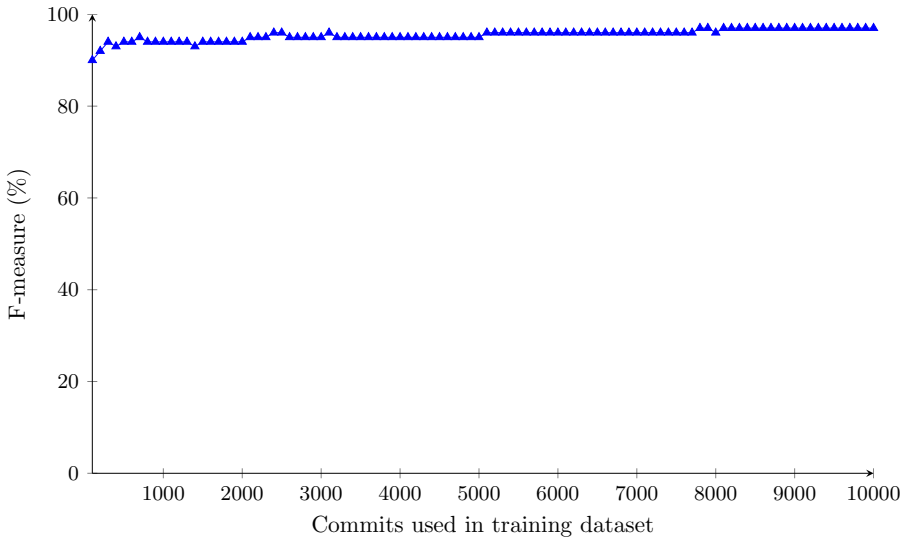

Figure 11: F-measure Achieved by Incrementally Adding Training Data Size for Two-class Classification.

For both classification methods, we run our approach using the 10-fold cross-validation technique, using nine folds as training data and the remaining one for testing. Because our target is to examine the impact of the quantity of training data on the performance of the classification, we train the classifier adding batches of 100 commits at a time similar to [78], and evaluate their performance on the testing dataset. For each batch of commits, we maintain the same ratio of SAR and non-SAR commits. The process ends when all of the training dataset is used. After each iteration, we report the average performance for all of the folds.

Figure 11 reveals the F-measure scores when identifying SAR and non-SAR commits. Overall, we find that the F-measure maintains almost the same level with no significant improvement, in terms of accuracy, as the dataset size increases. As can be seen, we obtain a high F-measure value starting with less than 1000 commits. We conclude that only one fold of the training dataset is sufficient to identify SAR commits with F-measure of 90%. To achieve F-measure higher than 90%, at least one fold of 1000 is needed. Figure 12 shows the

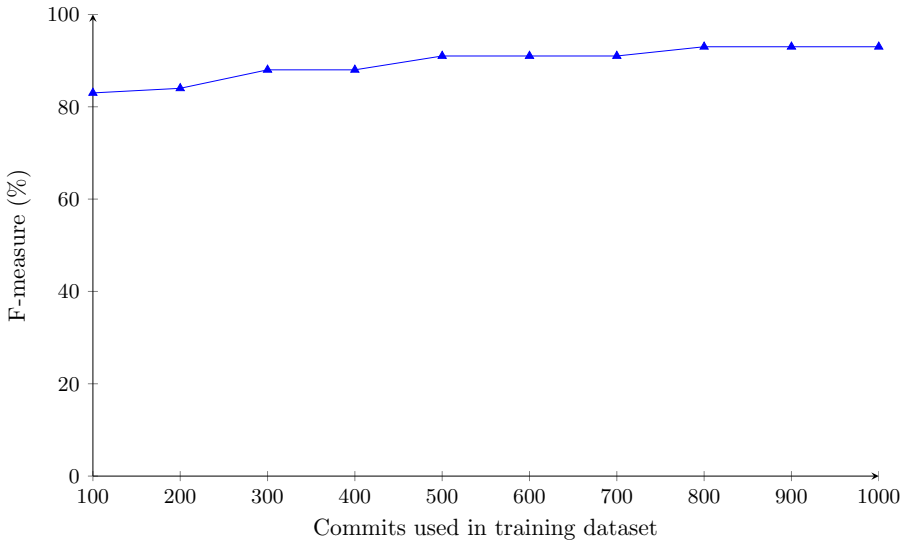

Figure 12: F-measure Achieved by Incrementally Adding Training Data Size for Multiclass Classification.

F-measure values when classifying Internal QA, External QA, and Code Smell commits (multiclass). In general, we notice that the F-measure value slightly increases as we increase the number of commits in the training dataset. To get at least 90% F-measure, more than 400 commits are needed. We conclude that to achieve a performance equivalent to 80% and 90% of the high F-measure score, only 10% and 40% of the commit messages are required respectively. To test the significance of the difference in F-measure values, we applied the Mann-Whitney U Test and found that the differences are not statistically significant.

Summary. We find that to achieve a performance equivalent to 90% of the high F-measure score, only one fold of commit messages is required for the two-class and multiclass classification problems, respectively.

6. Research Implications

This section further discusses positions our work in the spectrum of existing studies and how it implicates current research and practice.

6.1. Implications for practitioners

From a practitioner's point of view, giving enough background related to the performed refactorings is important to facilitate the code review process. Since there is no consensus on how to formally document refactoring activities, our model can provide various examples of how refactoring activity has been documented. Such information can be valuable to provide examples either to learn from or criticize. Also, since documenting code changes is enforced practice for some companies, then our tool can be used, in synchrony with other refactoring miners to detect when a refactoring, in the source code level, has no "expected" documentation in the commit message level. Such a quick sanity check can remind developers of adding any missing information. Furthermore, the review process heavily relies on understanding the context of the preformed refactoring, and since refactoring impact cannot be narrowed into one category, authors have to clearly state their intention in order for the reviewers to properly assess it.

Further, understanding maintenance activities is critical for practitioners to effectively direct the evolution of their projects in terms of enhancing cost-effectiveness, managing technical debt, and better planification of maintenance related resources. Therefore, a plethora of studies have been performed on automatic classification of repository artifacts (*e.g.,* bug reports, issues, code changes) in general, and commit messages in particular for several purposes, including the approximation of maintenance activity [41, 42, 70], security-relevant changes [99, 100], bug proneness [100, 101], bug fixes [102, 103]. Our work extends this existing effort by adding another dimension of the localization of refactoring effort. The end goal of estimating maintenance activities is to support managers and developers in better evaluating the quality of their projects, and so being more sensitive to anomalies that may arise, and the way to cope with them.

These three categories provide software practitioners with a catalog of common refactoring documentation patterns which represent concrete examples of common ways to document refactoring activities in commit messages. Having

these higher-level categories helps developers find the specific refactoring patterns they are looking for faster. Generally, in industry, there is no guideline on how to structure commit messages. This catalog of SAR patterns can encourage developers to follow best documentation patterns and also to further extend these patterns to improve refactoring documentation in particular and code changes in general. This work will also help developers to improve the quality of the refactoring documentation and trigger the need to explore the motivation behind refactoring. Further, these categories tell the opinion of developers, so it is important for managers to learn developers' opinions and feelings especially for distributed software development practices. If developers did not document, managers will not know their intention. Since software engineering is a human-centric process, it is important for managers to understand the intention of people working on the team. In this work, we (1) learn about how people self-report their types of work to evaluate progress with respect to goals for improving code quality, and (2) examine changes over time in how developers report their own activity in order to gain insight into patterns/find areas for improvement.

Moreover, for refactoring recommendations, if we know the intention of developers (*e.g.,* fix code smell), we can recommend refactoring based on the intention. From refactoring commit messages, we learn from these commit message examples and know what code elements they change, we then can optimize our refactoring recommendation to just work on code elements they are changing. This work will help refactoring recommending systems by narrowing their scope (*e.g.,* working on code fragments that developers are interested in). Current recommender system did not look at the intention, they excluded completely the intention of developers. Thus, these recommender systems are underused because they did not consider this important aspect.

6.2. Implications for researchers

From a research perspective, recent studies have been focusing on automatically identifying any execution of a refactoring operation in the source code

[11, 12, 104]. The main purpose of the automatic detection of refactoring is to better understand how developers cope with their software decay by extracting any refactoring strategies that can be associated with removing code smells [3, 4], or improving the design structural measurements [5, 105]. However, these techniques only analyze the changes at the source code level, and provide the operations performed, without associating it with any textual description, which may infer the rationale behind the refactoring application. Our proposed model intends to bridge this gap by complementing the existing effort in accurately detecting refactorings, by augmenting with any description that was intended to describe the refactoring activity. As previously shown in Tables 1 and 2, developers tend to add a high-level description of their refactoring activity, and occasionally mention their intention behind refactoring (remove duplicate code, improve readability), along with mentioning the refactoring operations they apply (type migration, inline methods, etc.). Our model, combined with the detection of refactoring operations, serves as a solid background for various empirical investigations. For instance, previous studies have analyzed the impact of refactoring operations on structural metrics [35, 106, 107]. One of the main limitations of these studies is the absence of any context related to the application of refactorings, *i.e.,* it is not clear whether developers did apply these refactoring with the intention of improving design metrics. Therefore, the use of our model will allow the consideration of commits whose commit messages specifically express the refactoring for the purpose of optimizing structural metrics, such as coupling, and complexity, and so, many empirical studies can be revisited with a more adequate dataset.

Furthermore, our study provides software practitioners with a catalog of common refactoring documentation patterns (cf. Tables 1 and 2) which would represent concrete examples of common ways to document refactoring activities in commit messages. This catalog of SAR patterns can encourage developers follow best documentation patterns and also to further extend these patterns to improve refactoring changes documentation in particular and code changes in general. Indeed, reliable and accurate documentation is always of crucial

importance in any software project. The presence of documentation for low level changes such as refactoring operations and commit changes helps to keep track of all aspects of software development and it improves on the quality of the end product. Its main focuses are learning and knowledge transfer to other developers.

Another important research direction that requires further attention concerns the documentation of refactoring. It has been known that there is a general shortage of refactoring documentation, as developers typically focus on describing their functional updates and bug patches. Also, there is no consensus about how refactoring should be documented, which makes it subjective and developer specific. Moreover, the fine-grained description of refactoring can be time consuming, as typical description should contain indication about the operations performed, refactored code elements, and a hint about the intention behind the refactoring. In addition, the developer specification can be ambiguous as it reflects the developer's understanding of what has been improved in the source code, which can be different in reality, as the developer may not necessarily adequately estimate the refactoring impact on the quality improvement. Therefore, our model can help to build a corpus of refactoring descriptions, and so many studies can better analyze the typical syntax used by developers in order to develop better natural language models to improve it, and potentially automate it, just like existing studies related to other types of code changes [108, 109, 110].

This work can help researchers to investigate the consistency between code changes and the actual intention and explore whether there is an overlap or not.

6.3. Implications for educators

From an educator point of view, this study helps to teach the new generation of developers or engineers the best practice to document their refactoring activity.

7. Threats to Validity

In this section, we identify potential threats to the validity of our approach and our experiments.

Construct Validity: Since our approach heavily depends on commit messages, we used well-commented Java projects when performing our study. Thus, the quality and the quantity of commit messages might have an impact on our findings. Additionally, a well-commented project might not contain SAR as developers might not document refactoring activities in the commit messages. We mitigate this risk by choosing projects that are appropriate for our analysis. Another potential threat relates to manual classification. Since the manual classification of training commit messages is a human intensive task and it is subject to personal bias, we mitigate manual classification related errors by discarding short and ambiguous commits from our dataset and replacing them with other commits. Another important limitation concerns the size of the dataset used for training and evaluation. The size of the used dataset was determined similarly to previous commit classification studies, but we are not certain that this number is optimal for our problem. It is better to use a systematic technique for choosing the size of the evaluation set. Concerning the relationship between refactoring and quality issues, we designed our study with the goal of classifying refactoring documentation. We have not explored if the refactoring operations detected by the Refactoring Miner tool are related to the corresponding quality issues documented by developers in the commit messages. Further, recent studies [67, 111, 112] indicate that commit messages could capture more than one type of classification (*i.e.,* mixed maintenance activity). Figure 13 shows a commit message could belong to internal quality attribute (since it discusses code complexity reduction), external quality attribute (since it points out scalability improvement), and code smell (since it explains duplicate code removal). In this work, we have not yet investigated whether a significant number of commits can belong to more than once class, and if so, we plan on exploring a multi-label classification in our future work.

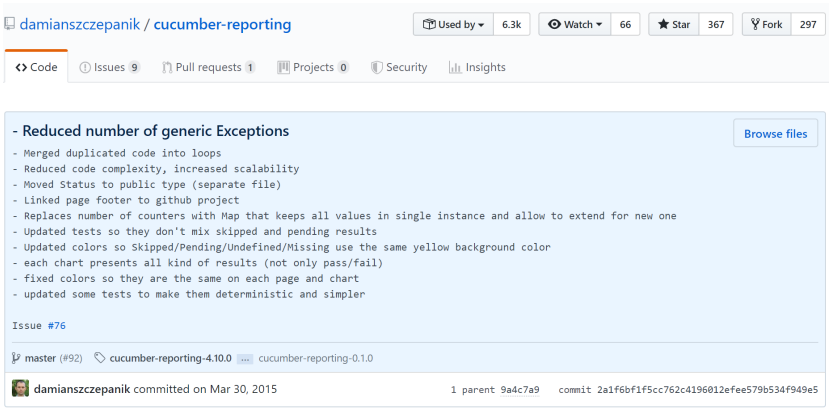

Figure 13: Example of Multi-label Refactoring Commit Message that Would Confuse the Classifiers.

External Validity: The first threat relates to the commits that are extracted only from open source Java projects. Our results may not generalize to commercially developed projects, or to other projects using different programming languages. Another threat concerns the generalization of SAR patterns in the commit messages. Since a commit is considered SAR commits if it only contains any of SAR patterns, this may not generalize to other projects (*e.g.,* outside the Java developers community) as it may have additional expression that could belong to SAR category.

Although we used commit messages as our primary source of text, our approach is not restricted to a specific source of textual information. In our future work, we can replicate our approach with other types of metadata, including issue descriptions. For this study, we chose to focus on commit messages rather than issue descriptions, since issue descriptions can be very high level, may not go into code change details, may not always be available, and may refer to multiple changes in the code that span or mix different purposes (*e.g.,* bug fix and feature request). Besides, not all projects are using issue tracker. If the issue tracker is guaranteed to be available, it could be used as an additional source of information.

The use of well engineered projects is a double-edged sword, while it guarantees an easier labeling process, and providing less noisy data for the approach, it

hinders its generalizability since these projects represent only a subset of all projects. So, our model may not achieve similar (high) performance across many projects. We tried to mitigate this concern by considering different types of projects, belonging to different domains. We shuffled commit messages during the training and testing to avoid any biases.

8. Conclusion

In this paper, we proposed an approach to identify and classify self-affirmed refactoring in commit messages. We compared the performance of our approach to pattern-based and simple random baselines. Our results show that our approach (1) is able to accurately classify SAR commits with accuracy of 98% and 93% for two-class and multiclass classification methods, respectively, outperforming the two state-of-the-art approaches considered in this study, and (2) can achieve F-measure of 90% using only 1% and 40% of the commits when performing two-class and multiclass SAR classifications respectively. This indicates a relatively small training dataset is sufficient to classify SAR commits.

In the future, we plan to study the applicability of our approach to other projects developed in different programming languages, and to other domains. Another potential research direction is to use the current findings to build a tool that supports the identification and detection of self-affirmed refactoring commits. We also plan to conduct different user studies with our industrial partner to predict the refactoring intention of the developers and further assess whether it aligns with what happened to his source code after applying refactoring.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, d. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
URL http://dl.acm.org/citation.cfm?id=311424

[2] M. Fowler, J. Highsmith, et al., The agile manifesto, Software Development 9 (8) (2001) 28–35.

[3] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, Jdeodorant: Identification and removal of type-checking bad smells, in: 2008 12th European Conference on Software Maintenance and Reengineering, IEEE, 2008, pp. 329–331.

[4] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia, An empirical study on the developers' perception of software coupling, in: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013, pp. 692–701.

[5] G. Bavota, S. Panichella, N. Tsantalis, M. Di Penta, R. Oliveto, G. Canfora, Recommending refactorings based on team co-maintenance patterns, in: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, ACM, 2014, pp. 337–342.

[6] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, P. Avgeriou, Identifying extract method refactoring opportunities based on functional relevance, IEEE Transactions on Software Engineering 43 (10) (2016) 954–974.

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

[8] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, K. Deb, Multi-criteria code refactoring using search-based software engineering: An industrial case study, ACM Transactions on Software Engineering and Methodology (TOSEM) 25 (3) (2016) 23.

[9] A. Ouni, M. Kessentini, H. Sahraoui, M. S. Hamdi, Search-based refactoring: Towards semantics preservation, in: 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 347–356.

[10] D. Silva, N. Tsantalis, M. T. Valente, Why we refactor? confessions of github contributors, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, ACM, New York, NY, USA, 2016, pp. 858–870. doi:10.1145/2950290.2950305.
URL http://doi.acm.org/10.1145/2950290.2950305

[11] D. Silva, M. T. Valente, Refdiff: detecting refactorings in version histories, in: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, 2017, pp. 269–279.

[12] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, D. Dig, Accurate and efficient refactoring detection in commit history.

[13] D. Zhang, L. Bing, L. Zengyang, P. Liang, A preliminary investigation of self-admitted refactorings in open source software (S), in: The 30th International Conference on Software Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, July 165-168, 2018. [13], pp. 165–168. doi:10.18293/SEKE2018-081.
URL https://doi.org/10.18293/SEKE2018-081

[14] E. A. AlOmar, M. W. Mkaouer, A. Ouni, Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages, in: Proceedings of the 3nd International Workshop on Refactoring-accepted. IEEE, 2019.

[15] J. Ratzinger, T. Sigmund, H. C. Gall, On the relation of refactorings and software defect prediction, in: Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR '08, ACM, New York, NY, USA, 2008, pp. 35–38. doi:10.1145/1370750.1370759.
URL http://doi.acm.org/10.1145/1370750.1370759

[16] K. Stroggylos, D. Spinellis, Refactoring–does it improve software quality?, in: Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007), IEEE, 2007, pp. 10–10.

[17] J. Ratzinger, sPACE: Software Project Assessment in the Course of Evolution, Ph.D. thesis (2007).
URL http://www.infosys.tuwien.ac.at/Staff/ratzinger/publications/ratzinger_phd-thesis_space.pdf

[18] G. Soares, R. Gheyi, E. Murphy-Hill, B. Johnson, Comparing approaches to analyze refactoring activity on software repositories, Journal of Systems and Software 86 (4) (2013) 1006–1022.

[19] E. A. AlOmar, self-affirmed-refactoring repository (2020 (last accessed September 13, 2020)).
URL https://smilevo.github.io/self-affirmed-refactoring/

[20] B. Du Bois, S. Demeyer, J. Verelst, Refactoring-improving coupling and cohesion of existing code, in: 11th working conference on reverse engineering, IEEE, 2004, pp. 144–151.

[21] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, G. Succi, A case study on the impact of refactoring on quality and productivity in an agile team, in: IFIP Central and East European Conference on Software Engineering Techniques, Springer, 2007, pp. 252–266.

[22] V. Singh, V. Bhattacherjee, Evaluation and application of package level metrics in assessing software quality, International Journal of Computer Applications 58 (21).

[23] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, A. Ouni, Many-objective software remodularization using nsga-iii, ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (3) (2015) 17.

[24] D. Silva, R. Terra, M. T. Valente, Recommending automated extract method refactorings, in: Proceedings of the 22nd International Conference on Program Comprehension, ACM, 2014, pp. 146–156.

[25] N. Moha, Y.-G. Gueheneuc, L. Duchien, A.-F. Le Meur, Decor: A method for the specification and detection of code and design smells, IEEE Transactions on Software Engineering 36 (1) (2009) 20–36.

[26] G. Szóke, G. Antal, C. Nagy, R. Ferenc, T. Gyimóthy, Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?, in: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, IEEE, 2014, pp. 95–104.

[27] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, A. Chávez, Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 465–475.

[28] B. Du Bois, S. Demeyer, J. Verelst, Does the" refactor to understand" reverse engineering pattern improve program comprehension?, in: Ninth European Conference on Software Maintenance and Reengineering, IEEE, 2005, pp. 334–343.

[29] B. Geppert, A. Mockus, F. Robler, Refactoring for changeability: A way to go?, in: 11th IEEE International Software Metrics Symposium (METRICS'05), IEEE, 2005, pp. 10–pp.

[30] J. Ratzinger, M. Fischer, H. Gall, Improving evolvability through refactoring, Vol. 30, ACM, 2005.

[31] C. D. Newman, M. W. Mkaouer, M. L. Collard, J. I. Maletic, A study on developer perception of transformation languages for refactoring, in: Proceedings of the 2nd International Workshop on Refactoring, 2018, pp. 34–41.

[32] C. D. Newman, M. J. Decker, R. S. AlSuhaibani, A. Peruma, D. Kaushik, E. Hill, An empirical study of abbreviations and expansions in software

artifacts, in: Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2019.

[33] C. D. Newman, R. S. AlSuhaibani, M. J. Decker, A. Peruma, D. Kaushik, M. W. Mkaouer, E. Hill, On the generation, structure, and semantics of grammar patterns in source code identifiers, Journal of Systems and Software 170 (2020) 110740.

[34] N. Tsantalis, V. Guana, E. Stroulia, A. Hindle, A multidimensional empirical study on refactoring activity, in: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '13, IBM Corp., Riverton, NJ, USA, 2013, pp. 132–146.
URL http://dl.acm.org/citation.cfm?id=2555523.2555539

[35] F. Palomba, A. Zaidman, R. Oliveto, A. De Lucia, An exploratory study on the relationship between changes and refactoring, in: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), IEEE, 2017, pp. 176–185.

[36] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, A. Bacchelli, A large-scale empirical exploration on refactoring activities in open source software projects, Science of Computer Programming 180 (2019) 1–15.

[37] J. Pantiuchina, F. Zampetti, S. Scalabrino, V. Piantadosi, R. Oliveto, G. Bavota, M. Di Penta, Why developers refactor source code: A mining-based study.

[38] R. Moser, A. Sillitti, P. Abrahamsson, G. Succi, Does refactoring improve reusability?, in: International Conference on Software Reuse, Springer, 2006, pp. 287–297.

[39] A. Hindle, D. M. German, R. Holt, What do large commits tell us?: A taxonomical study of large commits, in: Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR '08, ACM, New York, NY, USA, 2008, pp. 99–108. doi:10.1145/1370750.

1370773.
URL http://doi.acm.org/10.1145/1370750.1370773

[40] A. Hindle, N. A. Ernst, M. W. Godfrey, J. Mylopoulos, Automated topic naming to support cross-project analysis of software maintenance activities, in: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, ACM, New York, NY, USA, 2011, pp. 163–172. doi:10.1145/1985441.1985466.
URL http://doi.acm.org/10.1145/1985441.1985466

[41] S. Gharbi, M. W. Mkaouer, I. Jenhani, M. B. Messaoud, On the classification of software change messages using multi-label active learning, in: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, ACM, 2019, pp. 1760–1767.

[42] S. Levin, A. Yehudai, Boosting automatic commit classification into maintenance activities by utilizing source code changes, in: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE, ACM, New York, NY, USA, 2017, pp. 97–106. doi:10.1145/3127005.3127016.
URL http://doi.acm.org/10.1145/3127005.3127016

[43] M. Paixão, A. Uchôa, A. C. Bibiano, D. Oliveira, A. Garcia, J. Krinke, E. Arvonio, Behind the intents: An in-depth empirical study on software refactoring in modern code review, 17th MSR.

[44] G. Lacerda, F. Petrillo, M. Pimenta, Y. G. Guéhéneuc, Code smells and refactoring: A tertiary systematic review of challenges and observations, Journal of Systems and Software (2020) 110610.

[45] P. W. McBurney, S. Jiang, M. Kessentini, N. A. Kraft, A. Armaly, M. W. Mkaouer, C. McMillan, Towards prioritizing documentation effort, IEEE Transactions on Software Engineering 44 (9) (2017) 897–913.

[46] X. Xia, D. Lo, X. Wang, X. Yang, Collective personalized change classification with multiobjective search, IEEE Transactions on Reliability 65 (4) (2016) 1810–1829.

[47] A. Potdar, E. Shihab, An exploratory study on self-admitted technical debt, in: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, IEEE, 2014, pp. 91–100.

[48] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al., Managing technical debt in software-reliant systems, in: Proceedings of the FSE/SDP workshop on Future of software engineering research, 2010, pp. 47–52.

[49] F. Palomba, A. Zaidman, R. Oliveto, A. D. Lucia, An exploratory study on the relationship between changes and refactoring, in: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), 2017, pp. 176–185. doi:10.1109/ICPC.2017.38.

[50] G. Bavota, B. Russo, A large-scale empirical study on self-admitted technical debt, in: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), 2016, pp. 315–326. doi:10.1109/MSR.2016.040.

[51] E. d. S. Maldonado, R. Abdalkareem, E. Shihab, A. Serebrenik, An empirical study on the removal of self-admitted technical debt, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2017, pp. 238–248.

[52] F. Zampetti, A. Serebrenik, M. Di Penta, Automatically learning patterns for self-admitted technical debt removal, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2020, pp. 355–366.

[53] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering 38 (1) (2012) 5–18.

[54] G. Soares, D. Cavalcanti, R. Gheyi, T. Massoni, D. Serey, M. Cornélio, Saferefactor-tool for checking refactoring safety.

[55] A. Peruma, M. W. Mkaouer, M. J. Decker, C. D. Newman, Contextualizing rename decisions using refactorings and commit messages, in: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2019, pp. 74–85.

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

[57] A. Peruma, M. W. Mkaouer, M. J. Decker, C. D. Newman, An empirical investigation of how and why developers rename identifiers, in: International Workshop on Refactoring 2018, 2018. doi:10.1145/3242163.3242169.
URL http://doi.acm.org/10.1145/3242163.3242169

[58] E. A. AlOmar, M. W. Mkaouer, A. Ouni, M. Kessentini, On the impact of refactoring on the relationship between quality attributes and design metrics, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2019.

[59] A. Mockus, L. G. Votta, Identifying reasons for software changes using historic databases., in: icsm, 2000, pp. 120–130.

[60] J. Amor, G. Robles, J. Gonzalez-Barahona, A. Navarro Gsyc, J. Carlos, S. Madrid, Discriminating development activities in versioning systems: A case study.

[61] L. P. Hattori, M. Lanza, On the nature of commits, in: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops, 2008, pp. 63–71. doi:10.1109/ASEW.2008.4686322.

[62] A. E. Hassan, Automated classification of change messages in open source projects, in: Proceedings of the 2008 ACM Symposium on Applied

Computing, SAC '08, ACM, New York, NY, USA, 2008, pp. 837–841. doi:10.1145/1363686.1363876.
URL http://doi.acm.org/10.1145/1363686.1363876

[63] A. Hindle, D. M. German, M. W. Godfrey, R. C. Holt, Automatic classication of large changes into maintenance categories, in: 2009 IEEE 17th International Conference on Program Comprehension, 2009, pp. 30–39. doi:10.1109/ICPC.2009.5090025.

[64] N. Mahmoodian, R. Abdullah, M. A. A. Murad, Text-based classification incoming maintenance requests to maintenance type, in: 2010 International Symposium on Information Technology, Vol. 2, 2010, pp. 693–697. doi:10.1109/ITSIM.2010.5561540.

[65] A. Mauczka, M. Huber, C. Schanes, W. Schramm, M. Bernhart, T. Grechenig, Tracing Your Maintenance Work – A Cross-Project Validation of an Automated Classification Dictionary for Commit Messages, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 301–315. doi:10.1007/978-3-642-28872-2_21.
URL https://doi.org/10.1007/978-3-642-28872-2_21

[66] A. Mauczka, F. Brosch, C. Schanes, T. Grechenig, Dataset of developer-labeled commit messages, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015, pp. 490–493. doi:10.1109/MSR.2015.71.

[67] M. Yan, Y. Fu, X. Zhang, D. Yang, L. Xu, J. D. Kymer, Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project, Journal of Systems and Software 113 (Supplement C) (2016) 296 – 308. doi:https://doi.org/10.1016/j.jss.2015.12.019.
URL http://www.sciencedirect.com/science/article/pii/S016412121500285X

[68] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, A. Garcia, How does refactoring affect internal quality attributes?: A multi-project study, in: Proceedings of the 31st Brazilian Symposium on Software Engineering, SBES'17, ACM, New York, NY, USA, 2017, pp. 74–83. doi:10.1145/3131151.3131171.
URL http://doi.acm.org/10.1145/3131151.3131171

[69] S. Levin, A. Yehudai, Towards software analytics: Modeling maintenance activities, arXiv preprint arXiv:1903.04909.

[70] S. Hönel, M. Ericsson, W. Löwe, A. Wingkvist, Importance and aptitude of source code density for commit classification into maintenance activities, in: The 19th IEEE International Conference on Software Quality, Reliability, and Security, 2019.

[71] C. McMillan, M. Linares-Vasquez, D. Poshyvanyk, M. Grechanik, Categorizing software applications for maintenance, in: Proceedings of the 2011 27th IEEE International Conference on Software Maintenance, ICSM '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 343–352. doi:10.1109/ICSM.2011.6080801.
URL http://dx.doi.org/10.1109/ICSM.2011.6080801

[72] E. B. Swanson, The dimensions of maintenance, in: Proceedings of the 2Nd International Conference on Software Engineering, ICSE '76, IEEE Computer Society Press, Los Alamitos, CA, USA, 1976, pp. 492–497.
URL http://dl.acm.org/citation.cfm?id=800253.807723

[73] M. Allamanis, C. Sutton, Mining source code repositories at massive scale using language modeling, in: Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, 2013, pp. 207–216.

[74] N. Munaiah, S. Kroh, C. Cabrey, M. Nagappan, Curating github for engineered software projects, Empirical Software Engineering 22 (6) (2017) 3219–3253.

[75] K. Kowsari, K. Jafari Meimandi, M. Heidarysafa, S. Mendu, L. Barnes, D. Brown, Text classification algorithms: A survey, Information 10 (4) (2019) 150.

[76] S. Jiang, A. Armaly, C. McMillan, Automatically generating commit messages from diffs using neural machine translation, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2017, pp. 135–146.

[77] Y. Fu, M. Yan, X. Zhang, L. Xu, D. Yang, J. D. Kymer, Automated classification of software change messages by semi-supervised latent dirichlet allocation, Information and Software Technology 57 (2015) 369–377.

[78] E. da Silva Maldonado, E. Shihab, N. Tsantalis, Using natural language processing to automatically detect self-admitted technical debt, IEEE Transactions on Software Engineering 43 (11) (2017) 1044–1062.

[79] J. Cohen, A coefficient of agreement for nominal scales, Educational and psychological measurement 20 (1) (1960) 37–46.

[80] J. L. Fleiss, B. Levin, M. C. Paik, et al., The measurement of interrater agreement, Statistical methods for rates and proportions 2 (212-236) (1981) 22–23.

[81] P. S. Kochhar, F. Thung, D. Lo, Automatic fine-grained issue report reclassification, in: Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on, IEEE, 2014, pp. 126–135.

[82] H. Saif, M. Fernández, Y. He, H. Alani, On stopwords, filtering and data sparsity for sentiment analysis of twitter.

[83] C.-M. Tan, Y.-F. Wang, C.-D. Lee, The use of bigrams to enhance text categorization, Information processing & management 38 (4) (2002) 529–546.

[84] R. O. Duda, P. E. Hart, D. G. Stork, Pattern classification, John Wiley & Sons, 2012.

[85] Q. Gu, Z. Li, J. Han, Generalized fisher score for feature selection, arXiv preprint arXiv:1202.3725.

[86] C. D. Manning, P. Raghavan, et al., Schü tze h. introduction to information retrieval (2008).

[87] M. Fernández-Delgado, E. Cernadas, S. Barro, D. Amorim, Do we need hundreds of classifiers to solve real world classification problems, J. Mach. Learn. Res 15 (1) (2014) 3133–3181.

[88] A. C. Lorena, A. C. P. L. F. de Carvalho, J. M. P. Gama, A review on the combination of binary classifiers in multiclass problems, Artificial Intelligence Review 30 (1) (2009) 19. doi:10.1007/s10462-009-9114-9. URL https://doi.org/10.1007/s10462-009-9114-9

[89] J. H. Friedman, Greedy function approximation: a gradient boosting machine, Annals of statistics (2001) 1189–1232.

[90] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip, et al., Top 10 algorithms in data mining, Knowledge and information systems 14 (1) (2008) 1–37.

[91] C. Jose, P. Goyal, P. Aggrwal, M. Varma, Local deep kernel learning for efficient non-linear svm prediction, in: International conference on machine learning, 2013, pp. 486–494.

[92] M. Collins, Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms, in: Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10, Association for Computational Linguistics, 2002, pp. 1–8.

[93] R. Herbrich, T. Graepel, C. Campbell, Bayes point machines, Journal of Machine Learning Research 1 (Aug) (2001) 245–279.

[94] G. Andrew, J. Gao, Scalable training of l1-regularized log-linear models, in: International Conference on Machine Learning, international conference on machine learning Edition, 2007.

[95] A. Prinzie, D. Van den Poel, Random forests for multiclass classification: Random multinomial logit, Expert systems with Applications 34 (3) (2008) 1721–1732.

[96] J. Shotton, T. Sharp, P. Kohli, S. Nowozin, J. Winn, A. Criminisi, Decision jungles: Compact and rich models for classification, in: Proc. NIPS, proc. nips Edition, 2013.
URL https://www.microsoft.com/en-us/research/publication/decision-jungles-compact-and-rich-models-for-classification/

[97] L. K. Hansen, P. Salamon, Neural network ensembles, IEEE Transactions on Pattern Analysis & Machine Intelligence (10) (1990) 993–1001.

[98] S. Mund, Microsoft azure machine learning, Packt Publishing Ltd, 2015.

[99] C. Rosen, B. Grawi, E. Shihab, Commit guru: analytics and risk prediction of software commits, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 966–969.

[100] J. Eyolfson, L. Tan, P. Lam, Correlations between bugginess and time-based commit characteristics, Empirical Software Engineering 19 (4) (2014) 1009–1039.

[101] J. Eyolfson, L. Tan, P. Lam, Do time of day and developer experience affect commit bugginess?, in: Proceedings of the 8th Working Conference on Mining Software Repositories, ACM, 2011, pp. 153–162.

[102] S. Zafar, M. Z. Malik, G. S. Walia, Towards standardizing and improving classification of bug-fix commits, in: 2019 ACM/IEEE International Sym-

posium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2019, pp. 1–6.

[103] A. Sadiq, M. Mostafa, K. Sakib, On the evolutionary relationship between change coupling and fix-inducing changes, in: Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2019, SCITEPRESS - Science and Technology Publications, Lda, Portugal, 2019, pp. 494–501. doi:10.5220/0007758804940501.
URL https://doi.org/10.5220/0007758804940501

[104] M. Kim, M. Gee, A. Loh, N. Rachatasumrit, Ref-finder: a refactoring reconstruction tool based on logic query templates, in: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, ACM, 2010, pp. 371–372.

[105] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, M. Ó Cinnéide, Recommendation system for software refactoring using innovization and interactive dynamic optimization, in: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, ACM, 2014, pp. 331–336.

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

[107] D. Cedrim, L. Sousa, A. Garcia, R. Gheyi, Does refactoring improve software structural quality? a longitudinal study of 25 projects, in: Proceedings of the 30th Brazilian Symposium on Software Engineering, ACM, 2016, pp. 73–82.

[108] R. P. Buse, W. Weimer, Automatically documenting program changes., in: ASE, Vol. 10, 2010, pp. 33–42.

[109] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, D. Poshyvanyk, Changescribe: A tool for automatically generating commit messages, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2, IEEE, 2015, pp. 709–712.

[110] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, X. Wang, Neural-machine-translation-based commit message generation: how far are we?, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 373–384.

[111] H. Kirinuki, Y. Higo, K. Hotta, S. Kusumoto, Hey! are you committing tangled changes?, in: Proceedings of the 22Nd International Conference on Program Comprehension, ICPC 2014, ACM, New York, NY, USA, 2014, pp. 262–265. doi:10.1145/2597008.2597798.
URL http://doi.acm.org/10.1145/2597008.2597798

[112] P. J. Guinan, J. G. Cooprider, S. Faraj, Enabling software development team performance during requirements definition: A behavioral versus technical approach, Information Systems Research 9 (2) (1998) 101–125. arXiv:https://doi.org/10.1287/isre.9.2.101, doi:10.1287/isre.9.2.101.
URL https://doi.org/10.1287/isre.9.2.101