Winter 12-29-2021

On the Documentation of Refactoring Types

Eman Abdullah AlOmar
Rochester Institute of Technology

Jiaqian Liu
SUNY University at Buffalo

Kenneth Addo
University of Maryland, College Park

Mohamed Wiem Mkaouer

Christian D. Newman
Rochester Institute of Technology

See next page for additional authors

Recommended Citation

AlOmar, E.A., Liu, J., Addo, K. et al. On the documentation of refactoring types. Autom Softw Eng 29, 9 (2022). https://doi.org/10.1007/s10515-021-00314-w

Authors
Eman Abdullah AlOmar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian D. Newman, Ali Ouni, and Zhe Yu

Noname manuscript No.
(will be inserted by the editor)

On the Documentation of Refactoring Types

Eman Abdullah AlOmar · Jiaqian Liu · Kenneth Addo · Mohamed Wiem Mkaouer · Christian Newman · Ali Ouni · Zhe Yu

Received: date / Accepted: date

Abstract Commit messages are the atomic level of software documentation. They provide a natural language description of the code change and its purpose. Messages are critical for software maintenance and program comprehension. Unlike documenting feature updates and bug fixes, little is known about how developers document their refactoring activities. Specifically, developers can perform multiple refactoring operations, including moving methods, extracting classes, renaming attributes, for various reasons, such as improving software quality, managing technical debt, and removing defects. Yet, there is no systematic study that analyzes the extent to which the documentation of refactoring accurately describes the refactoring operations performed at the source code level. Therefore, this paper challenges the ability of refactoring documentation, written in commit messages, to adequately predict the refactoring types, performed at the commit level. Our analysis relies on the text mining of commit messages to extract the corresponding features (*i.e.*, keywords) that better represent each class (*i.e.*, refactoring type). The extraction of text patterns, specific to each refactoring type (*e.g.*, rename, extract, move,

Eman Abdullah AlOmar
Rochester Institute of Technology
E-mail: eman.alomar@mail.rit.edu

Jiaqian Liu
University at Buffalo
E-mail: jliu275@buffalo.edu

Kenneth Addo
University of Maryland
E-mail: kaddo1@umbc.edu

Mohamed Wiem Mkaouer · Christian Newman · Zhe Yu
Rochester Institute of Technology
E-mail: {mwmvse,cdnvse,zxyvse}@rit.edu

Ali Ouni
ETS Montreal, University of Quebec
E-mail: ali.ouni@etsmtl.ca

inline, etc.) allows the design of a model that verifies the consistency of these patterns with their corresponding refactoring. Such verification process can be achieved via automatically predicting, for a given commit, the method-level type of refactoring being applied, namely *Extract Method*, *Inline Method*, *Move Method*, *Pull-up Method*, *Push-down Method*, and *Rename Method*. We compared various classifiers, and a baseline keyword-based approach, in terms of their prediction performance, using a dataset of 5,004 commits. Our main findings show that the complexity of refactoring type prediction varies from one type to another. *Rename method* and *Extract method* were found to be the best documented refactoring activities, while *Pull-up Method*, and *Push-down Method* were the hardest to be identified via textual descriptions. Such findings bring the attention of developers to the necessity of paying more attention to the documentation of these types.

Keywords Refactoring · Software Quality · Software Engineering · Machine Learning

1 Introduction

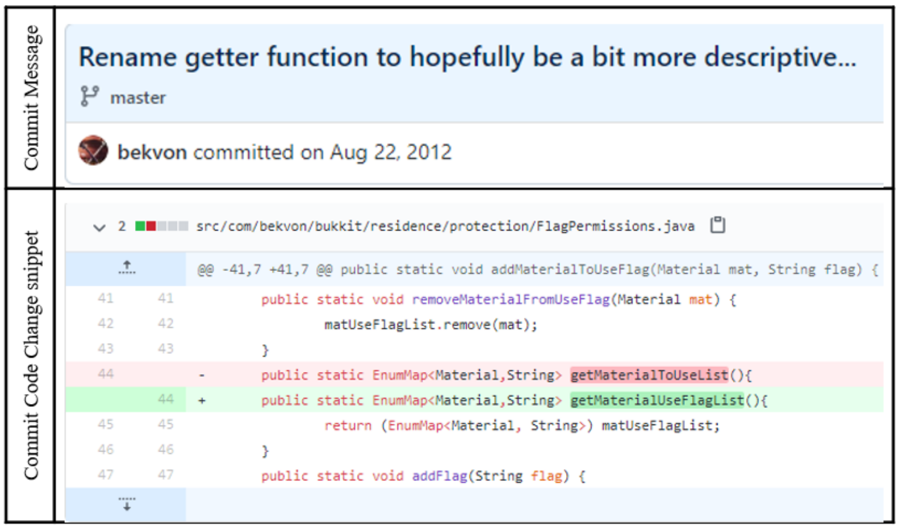
Understanding maintenance activities is critical for practitioners to effectively support the evolution of their projects in terms of enhancing cost-effectiveness, managing technical debt, and better allocation of maintenance related resources. Therefore, a plethora of studies have been performed on 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 activities (Gharbi et al., 2019; Hönel et al., 2020), identification of bug fixes (Zafar et al., 2019), detection of security-relevant changes (Alsolai & Roper, 2020; Sabetta & Bezzi, 2018). Recently, there have been a focus on analyzing commit messages in the context of refactoring.

Refactoring, being the art of improving software internal design without altering its external behavior (AlOmar et al., 2021b), is the *de-facto* way to reduce technical debt (Avgeriou et al., 2016). To help manage this technical debt, a lot of research focus has shifted to analyzing developers' refactoring practices through mining code changes and commit messages (Counsell et al., 2019, 2018; Naiya et al., 2015; Ubayashi et al., 2018; Veerappa & Harrison, 2013). For instance, (AlOmar et al., 2019a) developed a taxonomy of textual patterns, used by developers when documenting their refactoring activities, to understand how developers document these refactoring activities and many empirical studies have focused on mining commit messages to extract the reason behind developers' choice to refactor in terms of optimizing structural metrics, (*e.g.*, coupling, complexity, etc.) (AlOmar et al., 2019b; Pantiuchina et al., 2018), and quality attributes (*e.g.*, readability, etc.) (Fakhoury et al., 2019a). Commit messages were also used by (Rebai et al., 2020) to recommend refactoring operations. While there is a heavy reliance on the valuable information contained in commit messages, little is known about the extent to which such information can properly describe the actual refactoring changes in the source code. Specifically, studies have shown that developers do often misuse refactoring related terminology, in their documentation (Zhang et al., 2018). Because commit message analysis relies on the notion that refactorings are described in such a way that they can be distinguished from one another (*i.e.,* rename is described differently than move method), it is important to know whether this is generally true and in particular *how* refactorings can be distinguished by the way they are described in commit messages.

Recent studies have been heavily investigating how developers document refactoring to gain more insights on how refactoring is being practically applied. They parse commit messages to extract the intent behind the refactoring, then measure the impact of the refactoring on the source code quality, and verify the consistency between what was described in the message with the measurement in the source code. For instance, (Pantiuchina et al., 2018) found a misperception between the state-of-the-art structural metrics, widely used as indicators for refactoring, and what developers actually document as an improvement when they refactor their source code. Similarly, (AlOmar et al., 2019b) have found that not all metrics are equally capturing developers perception of software quality. (Fakhoury et al., 2019a) have found that current readability frameworks are unable to capture what developers intended to be refactorings that improve the source code readability. Such misperception between the theory of detecting refactoring opportunities, through removing code smells and improving structural metrics, and practical intents driving developers to refactor, could explain the shortage of developers adoption of current refactoring tools (Kim et al., 2014; Murphy-Hill et al., 2012). (Arnaoudova et al., 2016) investigated the linguistic antipatterns that are in disjunction with the source code. In another important dimension that can be investigated, is the consistency between the documentation of the refactoring actions, and the refactoring types that were actually performed in the source code. Just like documenting features and bug fixes, recent studies have shown that developers intentionally describe refactoring activities in commit messages, *i.e.,* self-affirm the existence of refactoring activity (AlOmar et al., 2019a; Zhang et al., 2018). Yet, little is known about the extent to which, the description of refactorings, in the commit message, matches the actual refactoring action that was committed.

Therefore, we study the ways in which terminology used to describe refactorings in commit messages to distinguish different refactorings from one another by studying the discriminative power of various machine learning techniques when provided this terminology. As an illustrative example, we refer to the simplified example extracted from the `bekvon/residence` project[1] reported in Figure 1. The commit message states the purpose of refactoring as a rename of getter function for better readability. Based on the developers commit message, can we automatically deduce the existence of a refactoring

[1] https://github.com/bekvon/residence/commit/76c364ea47e5a28b2041a0bb3323cb48bab180c9 (last checked 2020/06/20)



Rename getter function to hopefully be a bit more descriptive...
master
bekvon committed on Aug 22, 2012

```
2 src/com/bekvon/bukkit/residence/protection/FlagPermissions.java
@@ -41,7 +41,7 @@ public static void addMaterialToUseFlag(Material mat, String flag) {
        public static void removeMaterialFromUseFlag(Material mat) {
                matUseFlagList.remove(mat);
        }
-       public static EnumMap<Material,String> getMaterialToUseList(){
+       public static EnumMap<Material,String> getMaterialUseFlagList(){
                return (EnumMap<Material, String>) matUseFlagList;
        }
        public static void addFlag(String flag) {
```

Fig. (1) An example of a refactoring, and its corresponding documentation.

whose type is *Rename Method*. An intuitive solution for this problem is to detect the refactoring type in the source code and string-match it in the commit message to check whether it is mentioned as a form of verification. Such a solution assumes that developers refer to refactorings as they are known in the refactoring catalogue (Fowler et al., 1999; Wake, 2004). Previous studies found that developers misuse refactoring related terms (Soares et al., 2013), which hinders the accuracy of the string matching solution and presents a challenge for any solution that attempts to verify the consistency between refactoring and its corresponding documentation.

The goal of our study is to investigate whether different words and phrases found in refactoring commit messages are unique to different types of refactorings (*e.g.*, rename, move, extract, inline, etc.). In pursuit of this goal, we deploy machine learning techniques for the prediction of refactoring operation types based on commit messages. The results of this study can help us determine the types of words and phrases which best discriminate one type of refactoring from another; providing greater insight into the way refactoring is affirmed, which can be used to help automatically document refactorings in a more systematic way. Additionally, this work is critical in supporting refactoring documentation and in reducing the amount of effort needed by developers to appropriately describe what happened during a sequence of changes and help improve comprehension of those changes via commit messages. The work helps us understand how developers discriminate against different refactoring types through human language descriptions. Further, a recent industrial case study at Xerox reveals that developers rarely report specific refactoring operations as part of their documentation when submitting refactoring changes (AlOmar et al., 2021a). With the lack of refactoring documentation guidelines, the reviewers are forced to ask for more details in order to recognize the need for refactoring. The authors designed a procedure for documenting any refactoring review requests, respecting three dimensions that they referred to as the three ***I***s, namely, ***I**ntent*, ***I**nstruction*, and ***I**mpact*. Our study sheds light on the need to improve the quality of documenting refactoring types, which is considered one of the recommended dimensions to include in refactoring documentation.

In this paper, we formulate the prediction of refactoring operation types as a multi-class classification problem. Our solution relies on textual mining of commit messages to extract the corresponding features (*i.e.*, keywords) that better represent each class (*i.e.*, refactoring type) in order to automatically predict, for a given commit, the type of refactoring being applied and documented.

To build our model, we collected a dataset of commits that are known to contain the type of refactorings considered in this study. So, we use Refactoring Miner (Tsantalis et al., 2018) to extract, from different open source projects, commits that are known to contain a refactoring operation. Using Refactoring Miner, we collected a dataset of 5,004 instances, from 800 projects each instance represents a commit message, and a refactoring operation whose type is one of the 6 method-level types considered in this study, namely *Extract Method*, *Inline Method*, *Move Method*, *Pull-up Method*, *Push-down Method*, and *Rename Method*. Then, we use the N-Gram technique (Manning et al., 1999) to identify relevant features, for each of the classes, and which will be used to develop various classifiers, including Random Forest, Logistic Regression, and Gradient Boosted Machine.

Our key findings show that there is no uniform accuracy across all refactoring types, *i.e.,* some refactorings can achieve up to 90% in terms of F-measure, while others achieve 35% at best. This indicates that the documentations of some refactoring types, such as *Rename Method* are likely to follow best documentation practices than others, while some types are harder to distinguish and tend to be more ambiguous , such as *Move Method*, *Pull-up Method*, and *Push-down Method*.

This paper makes the following contributions:

1. We identify the common keywords and phrases developers utilize when describing their refactoring activity in the commit messages. Since there is also a significant amount of ambiguity in the way words are used, our work can reduce this confusion and the keywords that we discuss in this work are a strong starting point for determining what phrases should be used to reduce ambiguity and improve the quality of refactoring documentation. To the best of our knowledge, this is the first work attempted to assess the quality of the documentation of refactoring types using text mining technique.
2. We formulate the refactoring type prediction as a multi-class classification problem based on commit messages mining, and we challenge various models.

3. We evaluate the performance of our prediction model by comparing it against a baseline keyword-based approach that relies on matching messages with known refactoring type (Kim et al., 2014; Ratzinger, 2007; Ratzinger et al., 2008; Soares et al., 2013; Stroggylos & Spinellis, 2007; Zhang et al., 2018).
4. We discuss the inconsistency cases between the documentation of the refactoring actions, and the refactoring types that were actually performed in the source code.
5. We deploy our model as a lightweight web-service that is publicly available for software engineers and practitioners. We publicly provide our best model and the dataset that served as the *ground-truth*, for replication and extension purposes (AlOmar, 2021 (last accessed October 1, 2021)).

The rest of this paper is structured as follows. We review existing studies related to refactoring documentation and commit classification in Section 2. Next, in Section 3, we detail our classification methodology, including the data collection and preprocessing, and choice of the classification algorithms. Then, we evaluate our approach, in Section 4, and report a comparative study between various classifiers, while identifying most influential features. In Section 5, we report the implications of our study, and in Section 6, we discuss the threats to our work's validity. Finally, we conclude and describe our future work in Section 7.

2 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.

2.1 Refactoring 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, 2007) searched words stemming from the verb *"refactor"* such as "refactoring" or "refactored" to identify refactoring-related commits. (Ratzinger, 2007; Ratzinger et al., 2008) 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.*

Table (1) Related Work in Commit Classification Using Machine Learning.

Study	Year	Binary / Multi-class	Category	Machine Learning	Training Size	Result
(Amor et al., 2006)	2006	No/Yes	Swanson's category Administrative	Naive Bayes	400	Accuracy: 70%
(Hindle et al., 2009)	2009	No/Yes	Swanson's category Feature Addition	J48 / Naive Bayes / SMO KStar / IBk / JRip / ZeroR Non-Functional	2000	F-measure: 51% Accuracy: 52%
(Hindle et al., 2011)	2011	No/Yes	Non-Functional	rule / decision trees / vector space SVM / CLR / HOMER / BR	Not mentioned	Receiver Operating Characteristic up to 80%
(Levin & Yehudai, 2017)	2017	No/Yes	Swanson's category	J48 / GBM / RF	1151	Accuracy: 76%
(Hönel et al., 2019)	2019	No/Yes	Swanson's category	LssvmRadical / SVM / GBM xgbTree / LDA / MDA / NN / avNNet C5.0 / RF / Naive Bayes / LogitBoost	1151	Accuracy: up to 89%
(Gharbi et al., 2019)	2019	No/Yes	Swanson's category	DT / kNN / RF / MLP	5000	F-measure: 45.79%
(Krasniqi & Cleland-Huang, 2020)	2020	Yes/Yes	binary: CMR vs non-CMR multi-class: 12 refactoring types	NB / LR / SVM / kNN	1529	F-measure: 84% F-measure: 71%
(AlOmar et al., 2020a)	2020	Yes/Yes	binary: SAR vs non-SAR multi-class: Internal QA / External QA / code smell	RF / LR / GBM / DJ / BPM SVM / LD-SVM / NN / AP	1823 1044	F-measure: 98% F-measure: 93%
(AlOmar et al., 2021c)	2020	No/Yes	multi-class: Internal QA / EXternal QA / code smell Bug Fix / Functional	RF / LR / kNN / DT / SVC Mutlinomial Naive Bayes	1702	F-measure: 87%
(Aniche et al., 2020)	2020	Yes/No	multi-class: 20 refactoring types	LR / NB / SVM / DT / RF / NN	2 million	F-measure: > 90%
(Marmolejos et al., 2021)	2021	Yes/No	SAR vs non-SAR	BPM / AP / LR / GBM / NN	3000	F-measure: 96%
(AlOmar et al., 2021d)	2021	No/Yes	multi-class: Internal QA / External QA / code smell Bug Fix / Functional	RF / LR / kNN / DT / SVC Mutlinomial Naive Bayes	1702	F-measure: 87%

Later, (Murphy-Hill et al., 2012) 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., 2013) compared and evaluated three approaches, namely, manual analysis, commit message (Ratzinger et al.'s approach (Ratzinger, 2007; Ratzinger et al., 2008)), and dynamic analysis (SafeRefactor approach (Soares et al., 2009)) 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., 2014) 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., 2014) matched the top ten refactoring-related keywords identified from the survey 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.

Prior works (AlOmar et al., 2019a; Zhang et al., 2018) have 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., 2019b) 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. Furthermore, (AlOmar et al., 2021d) explored the relationship between developers' experience and refactoring. Their main findings show that refactoring contributors that frequently refactor the code tend to document less than developers that occasionally perform refactoring.

2.2 Commit Classification

(Hindle et al., 2009) 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 (Swanson,

1976) with two additional changes: Feature Addition, and Non-Functional. They observed that no single classifier is the best. (Hindle et al., 2011) conducted another experiment that classifies history logs 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., 2006) had a similar idea to (Hindle et al., 2009) 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 (Mahmoodian et al., 2010). (McMillan et al., 2011) 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 & Yehudai, 2017) 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%). Recently, a replicated study (Hönel et al., 2019) of (Levin & Yehudai, 2017) introduced code density of a commit 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 another study, (Gharbi et al., 2019) proposed a multi-label active learning-based approach to classify commit messages into maintenance categories. Their experimental results showed that the proposed approach achieved an F-measure of 45.79%.

(Krasniqi & Cleland-Huang, 2020) developed a model to first detect refactoring commit messages from non-refactoring commits, and then differentiated between 12 refactoring types. Their findings showed that Naive Bayes and SVM achieved the best performance with an F-measure of 84% and 0.71% for binary and multiclass classification problems, respectively. Another experiment that predicts refactoring was conducted using quality metrics. (Aniche et al., 2020) used a machine learning approach that involves predicting refactoring using code, process, and ownership metrics. The resulting models predict 20 different refactorings at class, method, and variable-levels with an accuracy often higher than 90%. More recently, (AlOmar et al., 2020a) proposed an approach to classify self-affirmed refactoring in commit messages. Their results show that their approach 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, *i.e.,* the keyword-based and the random classifier. In a follow-up work, (AlOmar et al., 2021c) performed a multi-class classification to categorize these commits into three categories, namely, Internal Quality Attribute, External Quality Attribute, and Code Smell Resolution, along with the traditional Bug Fix and Functional categories. This classification challenges the original definition of refactoring, being exclusive to improving software design and fixing code smells. (Marmolejos et al., 2021) proposed a framework to identify refactoring documentation by using different techniques, such as feature hashing and feature selection (Chi-squared and Fisher score), and five machine learning algorithms. As per their results, the combination of Chi-Squared with Bayes point machine and Fisher score with Bayes point machine could be the most efficient when it comes to automatically identifying refactoring documentation, with an F-measure of 96%. We summarize these state-of-the-art studies in Table 1.

Our work is in the intersection of the above-mentioned studies, as we leverage commit classifications techniques to automatically classify refactoring documentation. While prior studies searched for the existence of refactoring documentation, we further challenge it by checking whether the granularity of documentation can reach up to the level of distinguishing the types of executed refactorings. The refactoring types that we want to identify are the following:

– *Extract Method.* creating a new method by extracting a selection of code from inside the body of an existing method.
– *Inline Method.* replacing calls and usages of a method with its body, and potentially removing its declaration.
– *Move Method.* changing the declaration of a method, from one class to another one.
– *Pull-up Method.* moving up a method in the inheritance chain from a child class to a parent class.
– *Push-down Method.* moving down a method in the inheritance chain from a parent class to a child class.
– *Rename Method.* changing the name of a method identifier to a different one.

We chose types that are applied to the same level, *i.e.,* for the sake of consistency. Our approach can also be applied to class-level or package-level refactorings. In the next section, we detail the design of our proposed approach.

3 Study Design

The aim of our work is to reveal the extent to which a clear documentation of refactorings can help in correctly classifying them. The manual search for such correlation between refactoring types and their corresponding proper description can be time-consuming and error prone. We refer to solutions that can properly discriminate, and resolve, textual ambiguity; imitating the human decision making (Murphy, 2012) versus other, simpler techniques such as string-matching (Ratzinger et al., 2005, 2008; Soares et al., 2013; Stroggylos & Spinellis, 2007) which can be used, to some extent, to solve the same problem. We opt for the supervised learning where predictors (*i.e.,* independent variables) are developed to decide about the dependent variable's value, which, in our case, refers to the commit message classification. Thus, our dependent variable is represented by the refactoring types to be predicted. The independent variables will be extracted from the keywords used by developers to describe each type of refactoring in their commit messages. Therefore, we need to first setup a dataset that can characterize each class adequately. Since our aim is to investigate which types of refactoring are more adequately documented than others, we formulate this problem as a multiclass classification problem. Hence, when we build our dataset, we choose commits such that each contains one type of refactoring being performed. Then, we provide, for each class (*i.e.*, refactoring type) a set of commit messages that are meant to document it.

In the following, we elaborate on the technical details of our adopted classification technique, starting from the data collection, through its preparation and finally the models training and validation. The overview of our approach is depicted in Figure 2.

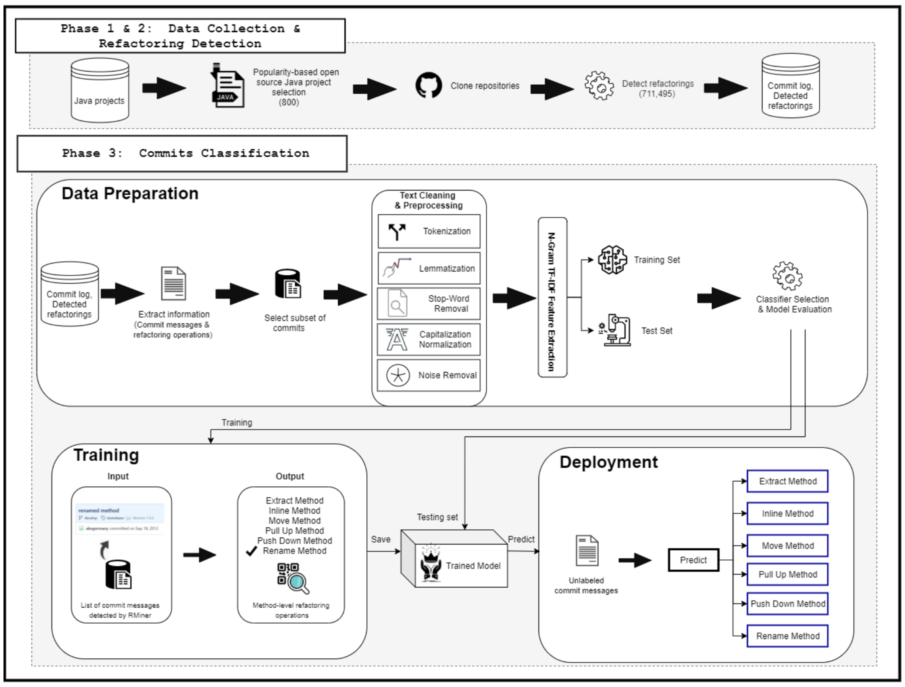

Phase 1 & 2: Data Collection & Refactoring Detection
Java projects
JAVA
Popularity-based open source Java project selection (800)
Clone repositories
Detect refactorings (711,495)
Commit log, Detected refactorings
Phase 3: Commits Classification
Data Preparation
Commit log, Detected refactorings
Extract information (Commit messages & refactoring operations)
Select subset of commits
Text Cleaning & Preprocessing
Tokenization
Lemmatization
Stop-Word Removal
Capitalization Normalization
Noise Removal
N-Gram TF-IDF Feature Extraction
Training Set
Test Set
Classifier Selection & Model Evaluation
Training
Training
Input
renamed method
List of commit messages detected by RMiner
Output
Extract Method
Inline Method
Move Method
Pull Up Method
Push Down Method
Rename Method
Method-level refactoring operations
Save
Testing set
Trained Model
Predict
Deployment
Unlabeled commit messages
Predict
Extract Method
Inline Method
Move Method
Pull Up Method
Push Down Method
Rename Method

Fig. (2) Overall Prediction Framework.

3.1 Overall Classification Framework

In a nutshell, the goal of our work is to automatically identify then classify commit messages containing refactoring documentation. Our approach takes as input, a commit message, and classifies it into one of six common method-level refactoring operations: *Extract Method*, *Inline Method*, *Move Method*, *Pull-up Method*, *Push-down Method*, and *Rename Method*. 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 of real-world documented refactoring operations (*i.e.,* commit messages). In the prediction phase, the built model will be used to predict the type of a given 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. Based on this input, the commit messages are preprocessed, allowing for informative featurization. Thereafter, 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 with 9 commonly used classifiers to evaluate our prediction model, namely, Gradient Boosted Machine (GMB) (Friedman, 2001), Support Vector Machine (SVM) (Wu et al., 2008), Locally Deep SVM (LD-SVM) (Jose et al., 2013), Averaged Perceptron Method (APM) (Collins, 2002), Bayes Point Machine (BPM) (Herbrich et al., 2001), Logistic Regression (LR) (Andrew & Gao, 2007), Random Forest (RF) (Prinzie & Van den Poel, 2008), Decision Jungle (DJ) (Shotton et al., 2013), and Neural Network (NN) (Hansen & Salamon, 1990). We selected these classifiers as they are commonly used in previous commit classification studies as well as several software engineering classification/prediction problems (Amor et al., 2006; Hindle et al., 2011, 2009; Hönel et al., 2019; Levin & Yehudai, 2017, 2019; Mahmoodian et al., 2010), as outlined in Table 1. 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 3.2.4) 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.

3.2 Commit Classification

Our solution design has six main phases: (1) data collection and refactoring detection, (2) data labeling, (3) text cleaning and preprocessing, (4) feature extraction using N-Gram, (5) model training and building, and (6) model evaluation. Since a commit message is written in plain text, we follow the approach provided by (AlOmar et al., 2020a; Kowsari et al., 2019) that discussed a recent trend in text classification techniques and algorithms.

3.2.1 Data Collection & Refactoring Detection

To perform this study, we randomly selected 800 projects, which were curated open-source Java projects hosted on GitHub as described in Table 2. These curated projects were selected from an available dataset by (Munaiah et al., 2017), 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. Additionally, these projects are non-forked (*i.e.,* not cloned from other projects), as forked projects may impact our conclusions by introducing duplicate code and documents. Also, 74.6% of the projects had their most recent commit within the last four years. The 800 selected projects analyzed in this study have a total of 748,001 commits, and a total of 711,495 refactoring operations from 111,884 refactoring commits. Additionally, these projects contain 732 commits and involve 19 developers on average (corresponding to the median of 346.5 commits and 7 developers). An overview of the projects is provided in Table 2.

To extract the entire refactoring history of each project, we use Refactoring Miner because 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% (Silva et al., 2016a; Tsantalis et al., 2018) along with being suitable for our study that requires a high degree of automation in data mining.

Table (2) Projects Overview.

Item	**Count**
Total of projects	800
Total commits	748,001
Refactoring commits	111,884
Refactoring operations	711,495
Considered Projects - Refactored Code Elements	
Code Element	**# of Refactorings**
Method	302,929
Class	228,974
Attribute	80,509
Parameter	42,992
Variable	28,765
Package	2380
Interface	1742

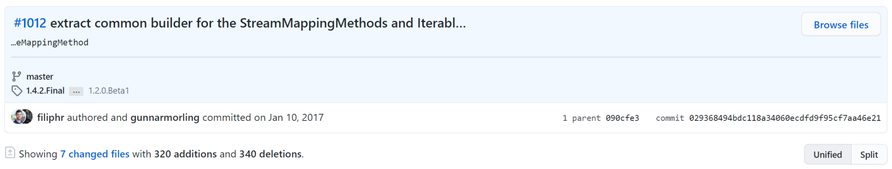
3.2.2 Data Labeling

Our goal is to provide the classifier with sufficient commits that represent the refactoring operations considered 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.,* refactoring types. 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 (Levin & Yehudai, 2017, 2019), we make sure that the classes for multiclass classification problem are equally distributed when preparing the data for the training (*cf.,* Table 3). 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 1). The highest number of commits used in comparable studies was 5,000 commits (Gharbi et al., 2019). Thus, we select a sample of 5,004 commits from 800 projects for each classification model. Below we detail the manual analysis of the data we use for our classification.

To prepare the dataset for the multiclass classification, we first run Refactoring Miner (Tsantalis et al., 2018) on the 800 open-source projects we presented in Table 2, in order to identify all commits containing refactorings. Then, we filter them to only keep commits with at most one refactoring type. Then, we cluster them by the types of refactorings we selected for this study. For each cluster, we start the random sampling of potential commits to include for our training set. For each randomly selected commit, we manually read through its message to verify whether it contains any textual description of the refactoring. Any commit with no such textual description is discarded. In our work, we discard the commits that do not contain any textual description of refactoring to narrow down the commit messages eliminating the ones that are less likely to be classified as one of the refactoring types. It is important to note that we removed these commit messages because (1) these commit messages do not contain enough information and do not describe the code change , and (2) we want to train the classifier on well-documented commit messages, and label commits that contain enough information about refactorings so that we can assess the quality of refactoring documentation. An example of commits that we retain in our dataset is illustrated in Figure 1. An example of commits that we discard documents a pull request, *e.g.,* "*Merge pull request #6 from marcel-blonk/develop make map type handle interfaces correctly*"[2]. Commits whose messages do not contain any kind of refactoring documentation would represent a noise in our dataset. Such commits would have been kept if the problem was formulated to binary identify refactoring documentation, but this is out of the scope of our work. This process resulted in selecting 5,004 stratified samples, divided equally for each stratum.

It is worth noting that upon performing the manual inspection of a subset of commit messages, we noticed that developers mostly document refactoring when they perform one or very few refactoring operations. However, if developers performed multiple refactoring operations, they are unlikely to detail refactoring activity in the commit messages. Figure 3 depicts an example of a commit message in which a developer stated that they performed only Ex-

[2] Commit extracted from sage-bionetworks/schema-to-pojo.



#1012 extract common builder for the StreamMappingMethods and Iterabl…
…eMappingMethod

master
1.4.2.Final … 1.2.0.Beta1

filiphr authored and gunnarmorling committed on Jan 10, 2017

1 parent 090cfe3 commit 029368494bdc118a34060ecdfd9f95cf7aa46e21

Showing 7 changed files with 320 additions and 340 deletions.

Fig. (3) An example of multiple refactorings, and its corresponding documentation.

tract refactoring operations. Yet, when running the Refactoring Miner tool, it shows that there are 36 refactoring operations performed in this commit message, namely, *Extract Method*, *Extract Superclass*, *Pull up Attribute*, *Pull up Method*, and *Rename Method*.

Table (3) Number of Refactoring Instances per Class.

Dataset	Extract	Inline	Move	Pull Up	Push Down	Rename
5,004 instances	834	834	834	834	834	834

3.2.3 Text Cleaning & Preprocessing

After the data preparation phase, we applied a similar methodology explained in (Kochhar et al., 2014; Kowsari et al., 2019) 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 (Kowsari et al., 2019). 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). We opted to use lemmatization over stemming, as the lemma of a word is a valid English word (Lane et al., 2019). 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 (Saif et al., 2014).
– **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.

3.2.4 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 (Kowsari et al., 2019). In general, N-Gram term has more semantic than an isolated word. Some of the keywords (*e.g.,* "*extract*") do not provide much information when used on its own. However, when collecting N-Gram from commit message (*e.g., Refactor createOrUpdate method in MongoChannelStore to extract methods and make code more readable*), the keyword "extract" clearly indicates that this refactoring commit belongs to *Extract Method* refactoring. In our classification, we use bigrams since it is very common to enhance the performance of text classification (Tan et al., 2002), and we select Fisher Score filter-based feature selection (Duda et al., 2012; Gu et al., 2012) to *featurize* text and manage the size of the text feature vector like (Kochhar et al., 2014). As for the weighting function, we used the standard Term Frequency-Inverse Document Frequency (TF-IDF) (Manning et al., 2008) as it is commonly used in the literature (Gharbi et al., 2019; Le et al., 2015; Lin et al., 2013; Ouni et al., 2016). The value for each N-Gram is proportional to 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.

3.2.5 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.

3.2.6 Classifier Selection and Model Evaluation

Selecting the proper classifier for optimal classification of the commits is a rather challenging task (Fernández-Delgado et al., 2014). Best practices suggest that developers properly document their commits by providing a commit message along with every commit they make to the repository. These commit messages are typically written using natural language, and generally convey some descriptive information about the commit changes they represent. In this study, we are dealing with multiclass classification problem since the commit messages are categorized into six different types. Since we have a predefined set of categories (*i.e.,* refactoring types), 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 various 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 (Kochhar et al., 2014; Levin & Yehudai, 2017, 2019). These studies found that Random Forest (RF) often achieves high performance. We investigated each classifier in our study using common statistical measures (*precision, recall, and F-measure*) of classification performance to compare each of them based on Azure Machine Learning (Azure ML) (Mund, 2015). It is important to note that the calculation of F-measure for multiclass classification is not supported by Azure ML. Thus, we compute F-measure using the following formula:

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

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

$$P = \frac{tp}{tp + fp}, \qquad R = \frac{tp}{tp + fn} \tag{2}$$

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 (Lorena et al., 2009). Thus, to ensure fairness, we use One-vs-All strategy for multiclass classification when using the following five classifiers: Gradient Boosted Machine (GMB), Support Vector Machine (SVM), Locally Deep SVM (LD-SVM), Averaged Perceptron Method (APM), and Bayes Point Machine (BPM). The remaining classifiers, consider in this study, are: Logistic Regression (LR), Random Forest (RF), Decision Jungle (DJ), and Neural Network

(NN). Our experiment is conducted using Microsoft Azure Machine Learning platform (Azure ML) (Mund, 2015), as it provides a built-in web-service once the classification models are deployed. We provide, in Table 4, the default parameter values of the classification algorithms in our study replicability purposes.

Table (4) Default Parameter Values for the Classification Algorithms.

Algorithm	Parameter	Description	Default Value
Random Forest	n_estimators	Number of decision trees	8
	max_depth	Maximum depth of the decision trees	32
	n_samples_leaf	Number of random splits per node	128
	min_samples_split	Minimum number of samples per leaf node	1
Logistic Regression	optimiz_tol	Optimization tolerance	1E-07
	1_weight	L1 regularization weight	1
	L2_weight	L2 regularization weight	1
	memory_L_BFGS	Memory size for L-BFGS	20
Gradient Boosted Machine	max_n_leaf	Maximum number of leaves per tree	20
	min_samples_leaf	Minimum number of samples per leaf node	10
	learning_rate	Learning rate	0.2
	n_tree	Number of trees constructed	100
Decision Jungle	n_estimators	Number of decision directed acyclic graphs	8
	max_depth	Maximum depth of the decision directed acyclic graphs	32
	max_width	Maximum of the decision directed acyclic graphs	128
	n_optimiz	Number of optimization steps per decision directed acyclic graphs layer	2048
Support Vector Classification	n_iter	Number of iterations	1
	Lambda	Lambda	0.001
Locally Deep SVM	max_depth	Depth of the tree	3
	lam_weight	Lambda weight	0.1
	n_theta	Lambda Theta	0.01
	n_theta_Prime	Lambda Theta Prime	0.01
	n_sigmoid	Sigmoid sharpness	1
	n_iter	Number of iterations	15000
Neural Network	n_nodes	Number of hidden nodes	100
	learning_rate	The learning rate	0.1
	n_learning_rate	Number of learning iterations	100
	learning_rate_weights	Initial learning weights diameter	0.1
	momentum	Momentum	0
Average Perceptron Method	learning_rate	Learning rate	1
	m_iter	Maximum number of iterations	10
Bayes Point Machine	n_training_iter	Number of training iterations	30

Table (5) Performance of Each Model, in Terms of Precision (P), Recall (R), and F-measure (F1), per Refactoring Type (a set of 5,004 commits).

Random Forest				***Logistic Regression***				***One-vs-All Gradient Boosted Machine***			
Refactoring type	**P**	**R**	**F1**	**Refactoring type**	**P**	**R**	**F1**	**Refactoring type**	**P**	**R**	**F1**
Extract Method	0.58	0.65	0.62	Extract Method	0.63	0.64	0.63	Extract Method	0.71	0.68	0.69
Inline Method	0.41	0.46	0.44	Inline Method	0.43	0.48	0.45	Inline Method	0.45	0.44	0.45
Move Method	0.57	0.67	0.61	Move Method	0.57	0.61	0.59	Move Method	0.61	0.66	0.63
Pull Up Method	0.41	0.31	0.35	Pull Up Method	0.41	0.38	0.40	Pull Up Method	0.42	0.41	0.42
Push Down Method	0.42	0.32	0.36	Push Down Method	0.40	0.36	0.38	Push Down Method	0.44	0.41	0.42
Rename Method	0.89	0.92	0.91	Rename Method	0.93	0.87	0.90	Rename Method	0.91	0.94	0.93

Decision Jungle				***One-vs-All Support Vector Machine***				***One-vs-All Locally Deep SVM***			
Refactoring type	**P**	**R**	**F1**	**Refactoring type**	**P**	**R**	**F1**	**Refactoring type**	**P**	**R**	**F1**
Extract Method	0.54	0.66	0.59	Extract Method	0.55	0.56	0.55	Extract Method	0.54	0.54	0.54
Inline Method	0.40	0.43	0.42	Inline Method	0.38	0.39	0.39	Inline Method	0.35	0.35	0.35
Move Method	0.58	0.73	0.65	Move Method	0.50	0.51	0.50	Move Method	0.47	0.46	0.47
Pull Up Method	0.39	0.21	0.27	Pull Up Method	0.37	0.36	0.36	Pull Up Method	0.34	0.38	0.36
Push Down Method	0.38	0.27	0.31	Push Down Method	0.37	0.38	0.37	Push Down Method	0.41	0.39	0.40
Rename Method	0.90	0.96	0.93	Rename Method	0.86	0.81	0.84	Rename Method	0.85	0.78	0.81

Neural Network				***One-vs-All Averaged Perceptron Method***				***One-vs-All Bayes Point Machine***			
Refactoring type	**P**	**R**	**F1**	**Refactoring type**	**P**	**R**	**F1**	**Refactoring type**	**P**	**R**	**F1**
Extract Method	0.58	0.50	0.54	Extract Method	0.54	0.53	0.53	Extract Method	0.49	0.46	0.48
Inline Method	0.37	0.37	0.37	Inline Method	0.36	0.38	0.37	Inline Method	0.33	0.35	0.34
Move Method	0.50	0.44	0.47	Move Method	0.45	0.48	0.46	Move Method	0.40	0.49	0.44
Pull Up Method	0.36	0.35	0.35	Pull Up Method	0.36	0.37	0.36	Pull Up Method	0.36	0.35	0.36
Push Down Method	0.37	0.46	0.41	Push Down Method	0.39	0.38	0.39	Push Down Method	0.38	0.36	0.37
Rename Method	0.82	0.86	0.84	Rename Method	0.85	0.81	0.83	Rename Method	0.70	0.61	0.65

4 Results & Discussions

In this section, we assess the performance of our approach, and aim at answering the following research questions:

- **RQ1. (effectiveness)** How effective is our supervised learning in predicting the type of refactoring?
- **RQ2. (baseline comparison)** How do our model compare with keyword-based classification?
- **RQ3. (terminology)** What are the frequent terms utilized by developers when documenting refactoring types?
- **RQ4. (inconsistency)** How useful is our approach in analyzing the inconsistency types between source code and documentation?

Replication package. We provide our comprehensive experiments package available in (AlOmar, 2021 (last accessed October 1, 2021) to further replicate and extend our study.

4.1 RQ1. How effective is our supervised learning in predicting the type of refactoring?

Table 5 reports the performance results of each classifier, in terms of precision, recall and F-measure, broken down per class, *i.e.,* refactoring type.

According to Table 5, Random Forest (RF), Gradient Boosting Machine (GBM), and Logistic Regression (LR) are performing relatively higher than their competitor classifiers, in terms of F-measure, across the majority classes. We also observe that the GBM was able to achieve the highest average F-measure of 0.59, in comparison with RF and LR, whose F-measure is respectively 0.54 and 0.55. 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.

Overall, there is an interesting pattern that we can observe across all classifiers: there is an agreement between all models that the *Rename Method* refactoring is the easiest to classify, with an F-measure starting from 0.65 (Bayes Point Machine) and reaching up to 0.93 (GBM). The *Extract Method* refactoring classification was the second highest for all classifiers except Decision Jungle. Its F-measure varies from 0.48 (Bayes Point Machine) to 0.69 (GBM). Furthermore, we observe that for the *Move Method* refactoring, the classifiers' performance varies between 0.46 (Averaged Perceptron Method) and 0.63 (GBM). As for the remaining classes, the performance of classifiers was similar and relatively low, when compared with the previous classes. For instance, the classifiers' performance, for the *Inline Method* refactoring varies between 0.34 (Bayes Point Machine) and 0.45 (GBM). For the *Pull-up Method* and the *Push-down Method* refactorings, the highest F-measure scored across all classifiers was 0.42. To gain a better understanding on why there exists such differences in the prediction between the refactoring types, we further analyzed the confusion matrix of the GBM classifier. During our qualitative analysis, we made the following observations:

Table (6) Examples of Wrongly Predicted Commit Messages, by the Gradient Boosting Machine (GBM).

Observation	**Ref. Operation**	**Commit Message Example**
Similar Expression	Extract Method	"*fcrepo-1029:* ***move*** *purge code* ***to*** *separate method*"
	Inline Method	"*ISQReader:* ***move*** *the dialog code* ***into*** *run() and tidy up*"
	Move Method	"***Move*** *send/receive code* ***from*** *SMTPSession* ***to*** *TextProtocolTester [...]*"
	Pull Up Method	"*HV-1239* ***Moving*** *shared code* ***up*** *to CascadableConstraintMapping[...]*"
	Push Down Method	"***Move*** *group communication* ***down*** *to jvstm-ispn only [...]*"
Inadequate Expression	Extract Method	"***Merged*** *updateTopic and updateTopicInline.*"
	Inline Method	"***Extracting*** *transactions from HadoopArchiveFileSystem. [...]*"
	Move Method	"*Improve code structure.* ***Added*** *tests.*"
	Pull Up Method	"***split*** *out into ERXAjaxContext so you can [...]*"
	Push Down Method	"***removed*** *deprecated method getConfigServer()*"
	Rename Method	"***Added*** *extended names for mixins.*"

Observation # 1. Similar Expressions.

Our first observation relates to the terminology and keywords developers use to describe each refactoring type. We notice that *Rename Method* has the highest accuracy across all classifiers because developers typically use the keyword *rename* to describe renaming methods. However, for the other types, developers do not stick to how these types are named in the refactoring catalog, and use various terminologies, to describe them. We enumerate, in Table 6, examples from messages belonging to *Extract/Inline/Pull-up/Push-down Method* classes, and which were wrongly predicted as *Move Method*. For instance, the process of extracting a method was described in one of the commits as "*moving* purge code to a *separate method*". While we can induce the extraction of the method, it was mislabeled by GBM classifier.

Observation # 2. Inadequate Expressions.

Occasionally, some messages contain keywords that are counter-intuitive to our model, resulting in a misclassification. Table 6 contains samples of misclassified commits, we report the correct label, while keywords that induced the wrong prediction are in bold. Let us take the following message: "*Merged updateTopic and updateTopicInline*", which documents inlining two methods, namely `updateTopic()` and `updateTopicInline()`, however, Refactoring Miner has detected an extraction of the method. To further understand this, we conducted a manual analysis of random samples. Our verification indicates that the keywords used by our model are not necessarily meant to document the underlying refactoring, as developers may document other changes performed in the commit.

It is worth noting that a recent study has reported that developers do misuse refactoring-related terms in their documentations (Zhang et al., 2018). Such cases will also hinder the accuracy of our prediction.

Summary. The accuracy of refactoring prediction is not uniform across all types. Some types are easier to predict than others. The prediction results for *Rename Method*, *Extract Method*, and *Move Method* were ranging from 63% to 93% in terms of F-measure. However, our model was not able to accurately distinguish between *Inline Method*, *Pull-up Method*, and *Push-down Method*, as its F-measure was between 42% and 45%.

4.2 RQ2. How do our model compare with keyword-based classification?

We opt to test the keyword-based approach because it was used to identify refactoring commits in previous studies (Kim et al., 2014; Mauczka et al., 2012; Murphy-Hill et al., 2012; Ratzinger et al., 2005, 2008; Stroggylos & Spinellis, 2007; Zhang et al., 2018). The keyword-based approach also measures the extent to which developers explicitly mention their refactoring operations in their commit messages.

The keyword-based approach simply uses the following keywords, namely "*extract*", "*inlin*", "*mov*", "*pull*", "*push*", and "*renam*", to perform the prediction. Note that we manually check the results to remove any false matching, *e.g.,* for the keyword *mov*, we filtered matchings like *movie* and *movement.*

Figures 4, 5, and 6 present the experimental results of our approach compared with the keyword-based prediction. Our approach provides an F-measure improvement across all refactoring types. One case in which the keyword-based approach could not detect the type of refactoring but the ML-based approach detects correctly is best illustrated in the following commit message: "*Change name of 'Decorator' to 'Events'*". The keyword-based approach does not capture this message as it does not contain the keyword "*renam*". This is intuitive since the model has identified a set of keywords that were also used to indicate a given refactoring type. For example, if we refer to Table 7, the *Inline Method* refactoring was found to be documented using various keywords such as *combine*, *gather*, and *merge.* Similarly with the *Extract Method* refactoring, whose documentation contained *add*, *create*, *split*, and *separate.*

It is worth noting that the highest performance of the keyword-based approach was achieved when predicting the *Move Method* refactoring, being able to capture the vast majority of commits containing this type (true positives), along with many other commits containing mainly the *Pull-up Method*, and *Push-down Method* refactortings, because developers typically document them using the "*move*" keyword, as we illustrated in Table 6.

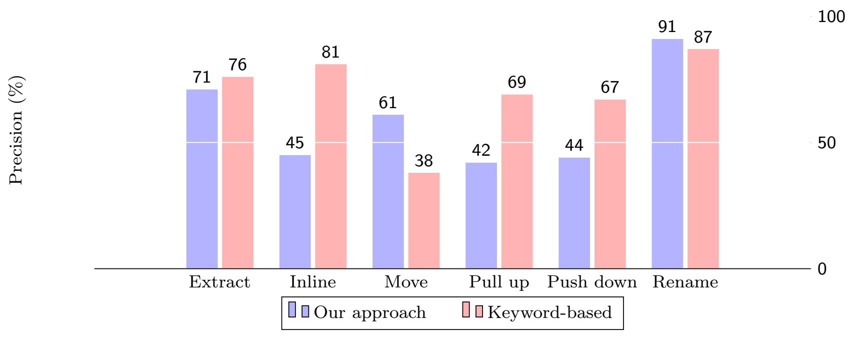


Fig. (4) Visualization of the Precision for Different Approaches.

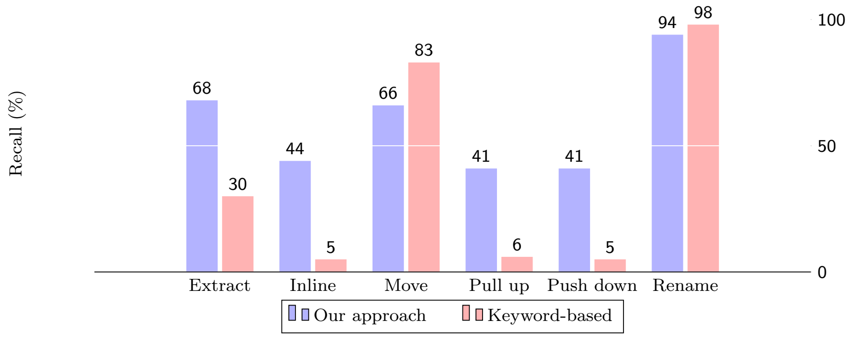


Fig. (5) Visualization of the Recall for Different Approaches.

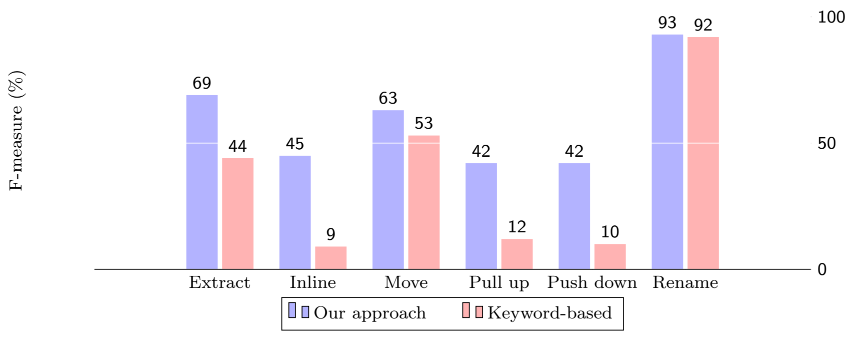


Fig. (6) Visualization of the F-measure for Different Approaches.

Summary. The keyword-based approach performs significantly lower than ML models. It assumes that developers are familiar with the catalog of refactorings, or refactoring types being offered in the IDEs. Our findings show that developers tend to document refactoring using the same set of patterns. The keyword-based approach scored relatively better performance for the *Rename Method* type because its keyword (*i.e.,* rename) is intuitive, in contrast with other types, such as *Inline Method* and *Push-down Method*.

Table (7) Relevant Features per Class.

Extract	**Inline**	**Move**	**Pull Up**	**Push Down**	**Rename**
Add	Combine	Move	Move	Move	Change
Create	Gather	Add	Pull	Push	Fix
Extract	Inline		Shift	Reduce	Improve
Move	Merge			Remove	Rename
Separate	Move				Update
Split					
Break up					

4.3 RQ3. What are the frequent terms utilized by developers when documenting refactoring types?

This research question examines the textual content of the commit messages to determine the frequent refactoring types-related terminology developers utilize when documenting their refactoring activity. In this RQ, we utilize natural language processing techniques, more specifically bigram analysis, to extract the frequent bigrams developers utilize in describing their refactoring activity for each refactoring type considered in our study. Bigrams are a sequence of two adjacent words in a sentence; in this instance, the commit messages. We also look at trigrams to locate sets of common terms. Unlike unigrams, bigrams and trigrams provide a certain level of context for terms, which helps our analysis by reducing the chance of making false presumptions. Before our extraction, we first run Refactoring Miner in order to identify commits containing refactorings from each type of refactoring operations considered in this study as discussed in Section 3.

Upon a closer inspection of the refactoring patterns in Tables 8, 9, and 10, we have made several observations: (1) the keywords and phrases used in renaming refactorings are the most discriminative, indicating that these terms are strongly associated with the action of renaming, (2) the patterns used for extract refactorings are associated with the motivation behind refactoring, *e.g.,* remove duplication, improve clarity, and improve reusability, (3) for move, pull up, and push down, developers used the term “move” interchangeably as the main action of these refactoring operations involve moving the code elements, and (4) the terms used in inlining refactorings are limited as developers mainly used specific keywords to demonstrate the action.

Summary. Developers discriminate against different refactoring types through human language descriptions. The terminology used in rename refactorings are the most discriminative, indicating that these terms are strongly associated with the action of renaming.

Table (8) Relevant Terms per Refactoring Types.

Rename	**Extract**
alter* method name for more consistency	add* a new method
better method name	add* method
chang* method name	add* new [] function
chang* method name for clarity	add* new method
chang* method name for consistency	add* several methods
chang* some method name	add* some convenience functions
chang* test method name	add* the [] method
chang* the method name	add* the method []
chang* the name	break* up the jumbo methods
clarif* method name	brok* up long methods into a bunch of smaller methods
clean* up method name	brok* up the [] method into a separate []
correct* a method name	creat* a higher level [] method
correct* method name	creat* a new method
fix* a typo in a method name	creat* method
fix* confusing method name	creat* separate method
fix* inconsistent method name	extract* common code from
fix* incorrect method name	extract* a few methods out
fix* method name	extract* a method
fix* method name conflict	extract* abstract method
fix* method name typo	extract* common code
fix* misspelled method name	extract* common method
fix* several method names	extract* method
fix* spelling for method name	extract* out a method
fix* typo in method name	extract* out function
improv* method name	extract* out the method
improv* the name	extract* some methods
made the method name a bit more explicit	extract* some methods for code clarity sake
method name chang*	extract* the [] method from []
method name fix*	extract* some stuff to a method
method name improv*	fix* for method code size
method name refactor*	mov* [] into separate methods
method names in tests changed	refactor* duplicate code into separate method
minor change to method name	refactor* some methods
minor refactorings to method name	refactor*: Introduc* a method
modif* test method name	separat* [] from []
more meaningful method name	separat* a method
normaliz* getter method name	split* [] into separate methods
polish test method name	split* into separate functions
refactor* method name	split* into smaller pieces first
refactor* some method names	split* into some smaller assert to reuse
renam* factory methods	split* the [] into component parts for clarity
renam* for clarification	split* the [] method in several sub-methods
renam* for clarity	split* the [] method into a []
renam* for consistency	split* the code into [] and []
renam* method	split* the HUGE generate method into different methods
renam* method name	split* up
renam* misleading method name	split* up [] a bit more neatly
renam* of code	split* up a complex method
renam* of component	split* up the [] method
renam* of function name	split* up the [] method into some methods
renam* some internal variables and methods	
renam* some methods	
renam* the method	
shorten* method name	
simplif* user method name	
solv* typo in method name	
standardization of method name	
tid* up method naming	
tid* up test method name	
unif* execution method name	
uniformiz* method name	
updat* method name	
updat* the test name	
using more correct method name	

Table (9) Relevant Terms per Refactoring Types (cont.).

Move	**Inline**
mov* [] to []	add* methods for merge operation
mov* [] to new method	combin* method
mov* all code into the only implementing class	consolidat* methods
mov* all utility methods into the same class	consolidat* some code
mov* around some methods	delet* unused method
mov* code around	inlin* helper methods
mov* formerly static methods to new	inlin* method
mov* from [] to []	inlin* method only called once
mov* into	inlin* private method
mov* method	inlin* some methods
mov* out of	inlin* some trivial method
mov* some	inlin* the simplest method
mov* some code into a static utility method	merg* [] and [] into 1 method
mov* some methods and/or classes around	merg* [] and [] methods
mov* some methods to	merg* code into static method
mov* some of it's responsibilities out to other classes	merg* refactoring
mov* some of the methods into a class	merg* some code simplification
mov* some static methods to Utils	more cleanup and merge resolution
mov* some stuff	refactor* [] into []
mov* static methods to a util class	refactor*: remov* some unused methods
mov* stuff out of the	remov* unused methods
mov* the []	simplif* things my inlining both the method and the argument
mov* the implementation of the methods to	some consolidation of methods
mov* the method tests in their own class	useless method inlined
mov* the methods	
mov* the notion of [] from [] to []	
mov* to []	
mov* util methods	
refactor* : mov* code	
refactor* out the methods into separate class	
refactor* some methods	
refactor* some methods to external helper class	
refactor* the code to move the [] to the []	
refactor* to move the [] to []	
refactor*: Move helper method to helper class	
refactor*: move to a helper method	
some static methods were moved from [] to []	

4.4 RQ4. How useful is our approach in analyzing the inconsistency types between source code and documentation?

Although our approach attempted to thoroughly predict method-level refactoring types, several inconsistency types between source code and documentation might occur. Several studies (Arnaoudova et al., 2016; Fakhoury et al., 2019b; Kim & Kim, 2016) have identified and detected recurring poor practices related to inconsistencies among the documentation and implementation of the code elements. Because such inconsistencies can affect software comprehensibility and maintainability, this research question aims at exploring the frequency of different inconsistency types that might help in reporting any early inconsistency between refactoring types detected by refactoring detector tools and their documentation. Specifically, we are studying the following inconsistency types:

Case # 1. Refactoring of type A is detected based on the source code but the description does not correspond to any refactoring.

To obtain the data for this type of inconsistency, we need to add a set of commits in which the documentation does not correspond to any type of refactorings considered in this study. We started by randomly selecting 834

Table (10) Relevant Terms per Refactoring Types (cont.).

Pull Up	**Push Down**
bunch of methods pulled up	chang* to shift functions
mov* common code in	minimal code duplication
mov* common code into	mov* common parts of
mov* common code to	mov* references to [] and [] into subclass
mov* more methods to	mov* test sections out of
mov* the common unit test setup to a base class	mov* [] from superclass
mov* the implementation to the superclass	mov* [] implementations into subclasses
mov* to	mov* some methods off [] onto a [] subclass
pull* to class level	push [] into []
pull* common	push to method level
pull* from	push* down
pull* from a specified	push* down to
pull* out	push* entities around
pull* out some common functionality	push* the [] code down into the
pull* out test methods into common area	push* to
pull* reusable	push* some stuff down from
pull* reusable code out of	reduc* the amount of implementation-specific code
pull* to	remov* dependency on
pull* up	remov* duplicate
pull* up common methods	remov* redundant
pull* up more properties to the base type	remov* redundant functions
pull* up some functionality from	stuff moved to separate
pull* up some methods	
pull* up to	
pull* out common code	
refactor* to "pull up"	
shift* further method to parent	

refactoring commits detected by Refactoring Miner while making sure no specific documentation about refactoring is reported. For example, we excluded the terms "*extract*", "*inlin*", and "*mov*" since these terms correspond to the method-level refactoring operations. The 834 commits equated to the number of commits per refactoring type, as shown in Table 3. We then had to manually examine the list of commits to determine their appropriateness for this analysis. Next, we built a new model by considering adding this set to the training data with a "None" label. Since RQ1 shows that the GBM was able to achieve the highest average F-measure of 0.59, we used the GBM for our model, and we achieved the average F-measure of 0.58. Using a confidence level of 99% and an interval of 5%, we constructed a sample size of 588 commits for the manual analysis. The majority of these commits (85.03 %) indicated there is a consistency between the refactoring detector and the model prediction, whereas a minority of these commits (14.96%) shows inconsistent results.

The main challenge that we observed across various commits, is the tendency of developers to provide a high-level description of their refactoring, through the use of general expressions and patterns, such as *refactor*, *restructure*, and *code clean up*, etc. Such patterns cannot be framed into one single type, *i.e.,* they can be used to describe all refactoring types. The following example demonstrates such a case:

"Just cleaned up the code a bit."

Quote (1) Inconsistency type (Case # 1)

This phenomenon of using high level description to document low-level changes is also observed frequently in bug fix commit messages, where text messages would just contain the popular pattern of "*fix bug X*". However, this is less problematic in the context of bugs because developers can still use the bug number (X) to locate the corresponding bug report, and so access the bug's proper documentation in the bug report. Whereas, for refactoring documentation, this is a persistent problem since without providing the rationale and the appropriate explanation of the change, there is no way to trace back such information anywhere in the project.

In practice, developers perform refactorings as singular transformations and in conjunction with other refactorings (*i.e.,* batch or composite refactorings). Previous studies (*e.g.,* (Bibiano et al., 2020)) explored how single or composite refactorings contribute to the code smell removal or internal quality attribute improvement. Since developers perform these kinds of refactorings at the source code level, we expect that developers apply such practice of single or multiple transformation types at the documentation level on real development practices. Our previous studies on refactoring documentation showed that developers self-affirmed the action of refactoring in both open source (*e.g.,* (AlOmar et al., 2019a, 2020a, 2021c)) and industry (AlOmar et al., 2021a) at different levels of granularity including the high-level and fine-grained descriptions. A previous study (Yamashita et al., 2020) on tailoring untangled changes pointed out that developers often mix changes in different intentional tasks in one comment. The authors proposed an approach that regards a sequence of fine-grained changes that are about to be committed as a single commit by developers to merge and split change clusters to support the manual tailoring of untangling changes.

From a practical point of view, researchers and practitioners can benefit from the proposed model to detect inconsistency types between refactoring detectors at the source code and documentation level, and to accelerate code review process since recent studies expressed the need to improve the quality of documentation for refactoring and non-refactoring changes (AlOmar et al., 2021a; Ebert et al., 2021).

Case # 2. Refactoring of type A is detected based on the description but the source code change does not correspond to any refactoring.

To perform our analysis, we need to include a set of commits that do not correspond to any refactoring operations and then feed this set into the training data with a "None" label. Thus, after running Refactoring Miner on a set of commits, we randomly selected 834 non-refactoring commits as indicated by Refactoring Miner. The selection of 834 commits was due to the count of refactoring types (see Table 3). We then built a model considering adding the set of non-refactoring commits in the training data. Similarly to Case #1, we consider using the GBM for the newly created model, and we achieve the average F-measure of 0.56. To better understand the nature of this type of inconsistency, we performed a manual validation of 588 commits from the test data, this sample corresponds to a confidence level of 99% and a confidence interval of 5%. The majority of the commits (436 instances or 74.14%) shows an agreement between the results obtained from the tool and our model, and (152 instances or 25.85%) illustrates the disagreement case.

(Soares et al., 2020) reported that such type of inconsistency might indicate that developers apply refactorings that are different from refactorings defined by (Fowler et al., 1999). Moreover, we observe in our study that developers are documenting what they consider to be refactoring in non source code files. These files include configuration files, maven file, or database are not associated with refactoring operations detected by the tool even though the description contains refactoring operation-related keywords. The following example demonstrates such a case:

"Renamed table. The same table name was used in another test, which made this test fail when running all tests."

Quote (2) Inconsistency type (Case # 2)

Such changes would not be detected by Refactoring Miner or any other detection tool because these tools are conceived to operate on only source files. Interestingly, our model results show that developers would also perform what they call refactoring on other files. If we refer to the original definition of refactoring, these changes may not be necessarily considered as refactorings, but with the rise of continuous integration, and infrastructure as service, many non-source files are now evolving as part of the project's ecosystem. These files undergo maintenance and evolution as well (updating dependencies, changing configurations, etc.). Therefore, there is a need for the refactoring community to properly taxonomize changes to these files, and evolve its toolset to detect them as well. Existing studies on configuration files have focused on the interactions between Java and XML configuration files (Chen & Johnson, 2008), the identification and detection of CI configuration bad practices that violate the best practices in CI configuration files (*e.g.,* redirecting scripts into interpreters, bypassing security checks, and using commands in an incorrect phase) and the prevalence of these anti-patterns in CI specifications (Gallaba & McIntosh, 2018; Zampetti et al., 2020). Since refactoring on other files is under research, future CI research and tooling needs to focus on the development of automated CI anti-pattern detectors and refactoring recommenders, and avoid the consequences of misusing CI features.

Case # 3. Refactoring of type A is detected based on the source code, refactoring of type B is detected based on the description and A is different from B.

Previous studies investigated the case when there is a disagreement between source code and its documentation in the context of programming misconception (Swidan et al., 2018), linguistic anti-patterns (Arnaoudova et al., 2016), bug localization (Fakhoury et al., 2019b), and code review (Ebert et al., 2021). In their study on misconceptions in programming education for school students, (Swidan et al., 2018) observed that younger learners hold common programming misconceptions that cause them to make errors. The authors recommended developing intervention methods to catch those misconceptions as early as possible. Further, (Arnaoudova et al., 2016) investigated developers' perception of linguistic anti-patterns and developed a catalog of 17 types of linguistic anti-patterns related to inconsistencies, findings that the majority of the participants perceive linguistic anti-patterns as poor practices and must be avoided. (Fakhoury et al., 2019b) showed that inconsistencies in the source code have a significant effect on cognitive load, success, and time spent on program comprehension. More recently, (Ebert et al., 2021) discussed how developers deal with confusion in code reviews caused by unclear commit messages and lack of documentation. According to their survey with developers, one of the most frequent reasons for confusion is lack of documentation and missing code change rationale.

Since the presence of inconsistencies can mislead developers, we aim to investigate this phenomenon. For this type of inconsistency, we randomly selected 588 refactoring commits to check the percentage of the agreement and the mismatch between refactoring types detected by the Refactoring Miner and our model. This quantity roughly equates to a sample size with a confidence level of 99% and a confidence interval of 5%. We then run our deployed model on these commits in order to compare our results with that obtained by the Refactoring Miner. The result shows that the inconsistency case represents 60.20% of the commits whereas only 39.79% of the commits are consistent.

Concerning our manual analysis, we observe that developers provide inadequate description of the code changes. The following example demonstrates such a case in which the tool detected composite refactoring operations *i.e., Extract*, *Rename*, and *Move* whereas our model predicted the commit as *Extract* based on the description:

"Extract BindingHelper for re-use in wizards."

Quote (3) Inconsistency type (Case # 3)

Our analysis for the three types of inconsistency shows that there is a need to improve the quality of refactoring documentation, and encourage the invention of the refactoring documentation generator. This offers a valuable opportunity to improve and standardize the format of the documentation. We believe that by combining the documentation with the state-of-the-art refactoring detectors, we can better understand the applied refactoring. For future work, we plan to perform an in depth study and extensive manual low-level source code inspection to better understand the phenomenon (*i.e.,* inconsistency cases).

Summary. Our model can work in conjunction with refactoring detectors (Silva & Valente, 2017; Tsantalis et al., 2018) in order to report any early inconsistency between refactoring types and their documentation.

5 Research Implications

The main implications of this study are as follows:

1. While existing studies, in classifying code changes using their commit messages (Gharbi et al., 2019; Levin & Yehudai, 2017, 2019), have been achieving relatively higher accuracies in comparison with our model, this reveals a lack of refactoring documentation *culture*, unlike documenting other code changes such as, API migration, bug fixes, and feature updates. However, the end goal our model is not to detect refactorings, but to work in conjunction with refactoring detectors (Silva et al., 2016b; Tsantalis et al., 2018) in order to report any early inconsistency between refactoring types and their documentation. This is useful not only to improve the quality of documentation, which has been found to be lacking when it comes to describing code changes (Treude et al., 2020), but also to improve the understandability of code changes for code review and evolution purposes. For instance, a recent study has found that revealing more details about refactoring, such as types and intents, helps in facilitating its acceptance in code reviews (Bibiano et al., 2020).
2. The words and phrases used in rename refactorings are the most discriminative, indicating that these terms are strongly associated with the action of renaming. Future work to help document rename refactorings, which are shown to be under-documented at between 1 and 6% of the time (Arnaoudova et al., 2014; Peruma et al., 2020), can use our approach to determine what keywords they should use, or recommend to developers, when generating commit messages.
3. Refactorings are generally associated with a specific set of keywords and phrases found in commit messages. However, there is also a significant amount of ambiguity in the way words are used; particularly for pull-up and push-down refactorings. A system which recommends how to document refactorings can reduce this confusion and the keywords that we discuss in this work are a strong starting point for determining what phrases should be used to reduce ambiguity.
4. Our approach can be used to study the discriminative terms found in commit messages and can be used to detect the common words and phrases which describe different types of refactorings. In this study, we used this approach on a large number of systems but it could also be used on singular systems to detect project-specific ways of describing refactorings; further bolstering any future recommendation system's ability to tailor recommended commit messages/keywords to a specific project.
5. Our study helps us understand refactoring documentation practices that trigger the need to explore the motivation behind refactoring. The study helps future developers to follow best documentation practices and improve the quality of the refactoring documentation. Further, the refactoring motivations 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 do not document, managers will not

know their intention. Since software engineering is a human-centric process, it is important for managers to understand the people's intention to work on the team through their documentation.

6 Threats to Validity

In this section, we describe potential threats to validity of our research method, and the actions we took to mitigate them.

Internal Validity. Our analysis is mainly threatened by the accuracy of the Refactoring Miner tool because the tool may miss the detection of some refactorings. However, previous studies (Silva et al., 2016b; Tsantalis et al., 2018) report that Refactoring Miner has high precision and recall scores (*i.e.*, a precision of 98% and a recall of 87%) compared to other state-of-the-art refactoring detection tools and is frequently utilized in refactoring studies (*e.g.,*(AlOmar et al., 2019a, 2020a,b,c, 2021e; Aniche et al., 2020; Chávez et al., 2017; Peruma et al., 2020)). A recent survey (Tan & Bockisch, 2019) compares several refactoring detection tools and shows that Refactoring Miner is currently the most accurate refactoring detection tool, which gives us confidence in using the tool.

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. 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. Another threat to validity can be related to the list of keywords that we used to identify set of commits for keyword-based approach as developers might use other keywords when documenting refactoring. However, the impact of this threat was limited to the refactoring operation-related keywords detected by Refactoring Miner.

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. Further, since a commit message could potentially belong to multiple refactoring types, our model does not consider such cases. However, exploring how to automatically classify commits into this kind of hybrid categories is an interesting direction for future work.

7 Conclusion

In this paper, we formulated the prediction of refactorings as a multiclass classification problem, *i.e.,* classifying refactoring commits into six method-level refactoring operations, applying nine supervised machine learning algorithms. We compared the performance of our approach to the keyword-based baseline and our results show that our approach outperforms the keyword-based approach. Specifically, our main findings show that (1) the prediction results for *Rename Method*, *Extract Method*, and *Move Method* were ranging from 63% to 93% in terms of F-measure. However, our model was not able to accurately distinguish between *Inline Method*, *Pull-up Method*, and *Push-down Method*, as its F-measure was between 42% and 45%, (2) the keyword-based approach performs significantly lower than ML models, (3) developers discriminate against different refactoring operations through human language descriptions, and (4) there is a need to improve the quality of refactoring documentation and encourage the invention of the refactoring documentation generator.

In the future, we plan to study the applicability of our approach to other projects developed in different programming languages, and to other domains, *i.e.,* consider using commit messages written in different programming languages to predict refactoring and compare findings. We also plan to use the extension of Refactoring Miner (Tsantalis et al., 2020) that supports low-level refactorings. Another interesting research direction is to investigate if our approach can be applied to statement-level refactoring (*e.g., Extract Variable*). Additionally, since a commit message could potentially belong to multiple categories (*e.g.*, *Extract Method* and *Move Method*), future research could usefully apply multi-label classification to automatically classify commits into this kind of hybrid categories. Further, 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 test our approach using other types of information, including issue descriptions.

8 Acknowledgments

This material is based on work supported by the National Science Foundation under Grant No. 1757680.

References

AlOmar, E., Mkaouer, M. W., & Ouni, A. (2019a). Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)* (pp. 51–58). IEEE.

AlOmar, E. A. (2021 (last accessed October 1, 2021)). *self-affirmed-refactoring repository*. URL: https://smilevo.github.io/self-affirmed-refactoring/.

AlOmar, E. A., AlRubaye, H., Mkaouer, M. W., Ouni, A., & Kessentini, M. (2021a). Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (pp. 348–357). IEEE.

AlOmar, E. A., Mkaouer, M. W., Newman, C., & Ouni, A. (2021b). On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology*, (p. 106675).

AlOmar, E. A., Mkaouer, M. W., & Ouni, A. (2020a). Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, (p. 110821).

AlOmar, E. A., Mkaouer, M. W., Ouni, A., & Kessentini, M. (2019b). 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)* (pp. 1–11). IEEE.

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

AlOmar, E. A., Peruma, A., Mkaouer, M. W., Newman, C. D., & Ouni, A. (2021d). Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process*, (p. e2395).

AlOmar, E. A., Peruma, A., Newman, C. D., Mkaouer, M. W., & Ouni, A. (2020b). On the relationship between developer experience and refactoring: An exploratory study and preliminary results. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (pp. 342–349).

AlOmar, E. A., Rodriguez, P. T., Bowman, J., Wang, T., Adepoju, B., Lopez, K., Newman, C., Ouni, A., & Mkaouer, M. W. (2020c). How do developers refactor code to improve code reusability? In *International Conference on Software and Software Reuse* (pp. 261–276). Springer.

AlOmar, E. A., Wang, T., Vaibhavi, R., Mkaouer, M. W., Newman, C., & Ouni, A. (2021e). Refactoring for reuse: An empirical study. *Innovations in Systems and Software Engineering*, (pp. 1–31).

Alsolai, H., & Roper, M. (2020). A systematic literature review of machine learning techniques for software maintainability prediction. *Information and Software Technology*, *119*, 106214.

Amor, J., Robles, G., Gonzalez-Barahona, J., Navarro Gsyc, A., Carlos, J., & Madrid, S. (2006). Discriminating development activities in versioning systems: A case study, .

Andrew, G., & Gao, J. (2007). Scalable training of l1-regularized log-linear models. In *International Conference on Machine Learning*.

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

Arnaoudova, V., Di Penta, M., & Antoniol, G. (2016). Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, *21*, 104–158.

Arnaoudova, V., Eshkevari, L. M., Penta, M. D., Oliveto, R., Antoniol, G., & Guéhéneuc, Y. (2014). Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, *40*, 502–532.

Avgeriou, P., Kruchten, P., Ozkaya, I., & Seaman, C. (2016). Managing technical debt in software engineering (dagstuhl seminar 16162). In *Dagstuhl Reports*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik volume 6.

Bibiano, A. C., Soares, V., Coutinho, D., Fernandes, E., Correia, J., Santos, K., Oliveira, A., Garcia, A., Gheyi, R., Fonseca, B. et al. (2020). How does incomplete composite refactoring affect internal quality attributes. In *28th IEEE/ACM International Conference on Program Comprehension (ICPC)*.

Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., & Garcia, A. (2017). How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering* (pp. 74–83). ACM.

Chen, N., & Johnson, R. (2008). Toward refactoring in a polyglot world: extending automated refactoring support across java and xml. In *Proceedings of the 2nd Workshop on Refactoring Tools* (pp. 1–4).

Collins, M. (2002). 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* (pp. 1–8). Association for Computational Linguistics.

Counsell, S., Arzoky, M., Destefanis, G., & Taibi, D. (2019). On the relationship between coupling and refactoring: An empirical viewpoint. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1–6). IEEE.

Counsell, S., Swift, S., Arzoky, M., & Destefanis, G. (2018). Do developers really worry about refactoring re-test? an empirical study of open-source systems. In *International Conference on Product-Focused Software Process Improvement* (pp. 159–166). Springer.

Duda, R. O., Hart, P. E., & Stork, D. G. (2012). *Pattern classification*. John Wiley & Sons.

Ebert, F., Castor, F., Novielli, N., & Serebrenik, A. (2021). An exploratory study on confusion in code reviews. *Empirical Software Engineering*, *26*, 1–48.

Fakhoury, S., Roy, D., Hassan, S. A., & Arnaoudova, V. (2019a). Improving source code readability: theory and practice. In *Proceedings of the 27th International Conference on Program Comprehension* (pp. 2–12). IEEE Press.

Fakhoury, S., Roy, D., Ma, Y., Arnaoudova, V., & Adesope, O. (2019b). Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empirical Software Engineering*, (pp. 1–39).

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

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

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

Gallaba, K., & McIntosh, S. (2018). Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci. *IEEE Transactions on Software Engineering*, *46*, 33–50.

Gharbi, S., Mkaouer, M. W., Jenhani, I., & Messaoud, M. B. (2019). On the classification of software change messages using multi-label active learning. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (pp. 1760–1767).

Gu, Q., Li, Z., & Han, J. (2012). Generalized fisher score for feature selection. *arXiv preprint arXiv:1202.3725*, .

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

Herbrich, R., Graepel, T., & Campbell, C. (2001). Bayes point machines. *Journal of Machine Learning Research*, *1*, 245–279.

Hindle, A., Ernst, N. A., Godfrey, M. W., & Mylopoulos, J. (2011). 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 (pp. 163–172). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/1985441.1985466. doi:10.1145/1985441.1985466.

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

Hönel, S., Ericsson, M., Löwe, W., & Wingkvist, A. (2019). 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*.

Hönel, S., Ericsson, M., Löwe, W., & Wingkvist, A. (2020). Using source code density to improve the accuracy of automatic commit classification into maintenance activities. *Journal of Systems and Software*, (p. 110673).

Jose, C., Goyal, P., Aggrwal, P., & Varma, M. (2013). Local deep kernel learning for efficient non-linear svm prediction. In *International conference on machine learning* (pp. 486–494).

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

Kim, S., & Kim, D. (2016). Automatic identifier inconsistency detection using code dictionary. *Empirical Software Engineering*, *21*, 565–604.

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

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

Krasniqi, R., & Cleland-Huang, J. (2020). Enhancing source code refactoring detection with explanations from commit messages. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 512–516). IEEE.

Lane, H., Hapke, H., & Howard, C. (2019). *Natural Language Processing in Action: Understanding, Analyzing, and Generating Text with Python*. Manning Publications Company.

Le, T.-D. B., Linares-Vásquez, M., Lo, D., & Poshyvanyk, D. (2015). Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In *2015 IEEE 23rd International Conference on Program Comprehension* (pp. 36–47). IEEE.

Levin, S., & Yehudai, A. (2017). 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 (pp. 97–106). New York, NY, USA: ACM. URL: http://doi.acm.org/10.1145/3127005.3127016. doi:10.1145/3127005.3127016.

Levin, S., & Yehudai, A. (2019). Towards software analytics: Modeling maintenance activities. *arXiv preprint arXiv:1903.04909*, .

Lin, S., Ma, Y., & Chen, J. (2013). Empirical evidence on developer's commit activity for open-source software projects. In *SEKE* (pp. 455–460). volume 13.

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

Mahmoodian, N., Abdullah, R., & Murad, M. A. A. (2010). Text-based classification incoming maintenance requests to maintenance type. In *2010 International Symposium on Information Technology* (pp. 693–697). volume 2. doi:10.1109/ITSIM.2010.5561540.

Manning, C. D., Manning, C. D., & Schütze, H. (1999). *Foundations of statistical natural language processing*. MIT press.

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

Marmolejos, L., AlOmar, E. A., Mkaouer, M. W., Newman, C., & Ouni, A. (2021). On the use of textual feature extraction techniques to support the automated detection of refactoring documentation. *Innovations in Systems and Software Engineering*, (pp. 1–17).

Mauczka, A., Huber, M., Schanes, C., Schramm, W., Bernhart, M., & Grechenig, T. (2012). Tracing your maintenance work – a cross-project validation of an automated classification dictionary for commit messages. In J. de Lara, & A. Zisman (Eds.), *Fundamental Approaches to Software Engineering: 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings* (pp. 301–315). Berlin, Heidelberg: Springer Berlin Heidelberg. URL: https://doi.org/10.1007/978-3-642-28872-2_21. doi:10.1007/978-3-642-28872-2_21.

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

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

Mund, S. (2015). *Microsoft azure machine learning*. Packt Publishing Ltd.

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

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

Naiya, N., Counsell, S., & Hall, T. (2015). The relationship between depth of inheritance and refactoring: An empirical study of eclipse releases. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications* (pp. 88–91). IEEE.

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

Pantiuchina, J., Lanza, M., & Bavota, G. (2018). Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 80–91). IEEE.

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

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

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

Ratzinger, J., Fischer, M., & Gall, H. (2005). *Improving evolvability through refactoring* volume 30. ACM.

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

Rebai, S., Kessentini, M., Alizadeh, V., Sghaier, O. B., & Kazman, R. (2020). Recommending refactorings via commit message analysis. *Information and Software Technology*, (p. 106332).

Sabetta, A., & Bezzi, M. (2018). A practical approach to the automatic classification of security-relevant commits. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 579–582). IEEE.

Saif, H., Fernández, M., He, Y., & Alani, H. (2014). On stopwords, filtering and data sparsity for sentiment analysis of twitter, .

Shotton, J., Sharp, T., Kohli, P., Nowozin, S., Winn, J., & Criminisi, A. (2013). Decision jungles: Compact and rich models for classification. In *Proc. NIPS*. URL: https://www.microsoft.com/en-us/research/publication/decision-jungles-compact-and-rich-models-for-classification/.

Silva, D., Tsantalis, N., & Valente, M. T. (2016a). Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 858–870). ACM.

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

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

Soares, G., Cavalcanti, D., Gheyi, R., Massoni, T., Serey, D., & Cornélio, M. (2009). Saferefactor-tool for checking refactoring safety, .

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

Soares, V., Oliveira, A., Pereira, J. A., Bibano, A. C., Garcia, A., Farah, P. R., Vergilio, S. R., Schots, M., Silva, C., Coutinho, D. et al. (2020). On the relation between complexity, explicitness, effectiveness of refactorings and non-functional concerns. In *Proceedings of the 34th Brazilian Symposium on Software Engineering* (pp. 788–797).

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

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

Swidan, A., Hermans, F., & Smit, M. (2018). Programming misconceptions for school students. In *Proceedings of the 2018 ACM Conference on International Computing Ed-*

ucation Research (pp. 151–159).

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

Tan, L., & Bockisch, C. (2019). A survey of refactoring detection tools. In *Software Engineering (Workshops)* (pp. 100–105).

Treude, C., Middleton, J., & Atapattu, T. (2020). Beyond accuracy: Assessing software documentation quality. *arXiv preprint arXiv:2007.10744*, .

Tsantalis, N., Ketkar, A., & Dig, D. (2020). Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, .

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., & Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering* (pp. 483–494). ACM.

Ubayashi, N., Kamei, Y., & Sato, R. (2018). Can abstraction be taught? refactoring-based abstraction learning. In *MODELSWARD* (pp. 429–437).

Veerappa, V., & Harrison, R. (2013). An empirical validation of coupling metrics using automated refactoring. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (pp. 271–274). IEEE.

Wake, W. C. (2004). *Refactoring workbook*. Addison-Wesley Professional.

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

Yamashita, S., Hayashi, S., & Saeki, M. (2020). Changebeadsthreader: An interactive environment for tailoring automatically untangled changes. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 657–661). IEEE.

Zafar, S., Malik, M. Z., & Walia, G. S. (2019). Towards standardizing and improving classification of bug-fix commits. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1–6). IEEE.

Zampetti, F., Vassallo, C., Panichella, S., Canfora, G., Gall, H., & Di Penta, M. (2020). An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*, *25*, 1095–1135.

Zhang, D., Li, B., Li, Z., & Liang, P. (2018). A preliminary investigation of self-admitted refactorings in open source software. doi:10.18293/SEKE2018-081.