


Article

Comparing Commit Messages and Source Code Metrics for the Prediction Refactoring Activities

Priyadarshni Suresh Sagar ¹, Eman Abdulah AlOmar ¹ , Mohamed Wiem Mkaouer ¹, Ali Ouni ²
and Christian D. Newman ^{1,*}

¹ Rochester Institute of Technology, Rochester, New York, NY 14623, USA; ps1862@rit.edu (P.S.S.); eaa6167@rit.edu (E.A.A.); mwmvse@rit.edu (M.W.M.)

² Ecole de Technologie Supérieure, University of Quebec, Quebec City, QC H3C 1K3, Canada; ali.ouni@etsmtl.ca

* Correspondence: cdnvse@rit.edu

Abstract: Understanding how developers refactor their code is critical to support the design improvement process of software. This paper investigates to what extent code metrics are good indicators for predicting refactoring activity in the source code. In order to perform this, we formulated the prediction of refactoring operation types as a multi-class classification problem. Our solution relies on measuring metrics extracted from committed code changes in order to extract the corresponding features (i.e., metric variations) that better represent each class (i.e., refactoring type) in order to automatically predict, for a given commit, the method-level type of refactoring being applied, namely *Move Method*, *Rename Method*, *Extract Method*, *Inline Method*, *Pull-up Method*, and *Push-down Method*. We compared various classifiers, in terms of their prediction performance, using a dataset of 5004 commits and extracted 800 Java projects. Our main findings show that the random forest model trained with code metrics resulted in the best average accuracy of 75%. However, we detected a variation in the results per class, which means that some refactoring types are harder to detect than others.

Keywords: refactoring; software quality; commits; software metrics; software engineering



Citation: Sagar, P.; AlOmar, E.; Mkaouer, MW. Comparing Commit Messages and Source Code Metrics for the Prediction Refactoring Activities. *Algorithms* **2021**, *14*, 289. <https://doi.org/10.3390/a14100289>

Academic Editors: Maurizio Proietti and Frank Werner

Received: 13 July 2021

Accepted: 21 September 2021

Published: 30 September 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Refactoring is the practice of improving software internal design without altering its external behavior. Developers regularly refactor their code by performing various refactoring types, including splitting methods, renaming attributes, moving classes, and merging packages. Recent studies have been focusing on recommending appropriate refactoring types in response to poor code design [1–4] and analyzing how developers refactor code by making mining code changes and commit messages [5–9]. Empirical studies have been focused on mining commit messages to extract developers' intents behind refactoring in terms of optimizing structural metrics (e.g., coupling, complexity, etc.) [10,11] and quality attributes (e.g., reuse, etc.) [12,13]. Commit messages were also used by Rebai et al. [14] to recommend refactoring operations.

To overcome the challenges and limitations of existing studies, we propose a novel approach to predict the type of refactoring through the structural information of the code extracted from the source code metrics (coupling, complexity, etc.). We believe that using code metrics to characterize code is beneficial because code metrics are known to be heavily impacted by refactoring, and this variation in their values can be a learning curve for our model. Our model can learn to detect patterns in metrics values, which can be later combined with textual information in order to support the accurate distinction the refactoring types (move, extract, inline, etc.).

In this paper, we formulate the prediction of refactoring operation types as a multi-class classification problem. Our solution relies on detecting patterns in metric variations to

extract the corresponding features (i.e., keywords and metric values) that better represent each class (i.e., refactoring type) in order to automatically predict, for a given commit, the type of refactoring being applied. In a nutshell, our model takes as input the commit (i.e., code changes) and the metric values associated with the code change in order to predict what type of refactoring was performed by the developer. This model will support developers in accurately choosing which refactoring types to apply when improving the design of their software systems.

To justify the choice of metric information, we challenge the model generated by this combination with state-of-the-art models that use only textual information. Experiments explored in this paper were driven by various research questions, including the following: How accurate is a text-based model in predicting the refactoring type? How accurate is a metric-based model in predicting the refactoring type? Which refactoring classes were most accurately classified by each method? Results show that text-based models produced poor accuracy, whereas supervised machine learning algorithms trained with code metrics as input resulted in the most accurate classifier. Accuracy per class varied for each method and algorithm, and this was expected.

This paper makes the following contributions:

1. We formulate the refactoring type prediction as a multi-class classification problem based on commit mining, and we challenge various models.
2. We evaluate the performance of our prediction model by comparing it against a baseline approach that relies on training with commit messages. To conduct this, we have used a dataset of 5004 commits and extracted 800 Java projects.
3. We also analyze the impact of using textual information vs. structural information in terms of properly identifying refactoring. metric-based modeling, and random forest, more specifically, was found to be the best performing model, with 75% accuracy.
4. We publicly provide our best model and the dataset that served as the *ground-truth*, for replication and extension purposes <https://github.com/smilevo/refactoring-metrics-prediction> (accessed on 20 September 2021).

The remainder of this paper proceeds by discussing the different commit message classification and refactoring prediction techniques implemented by other researchers where a general outline will be provided about research performed in the same field, the approach followed to accomplish the aim of the study, and how implementation has been performed. The last section will mainly focus on results and future work.

2. Related Work

In this section, we summarized the literature related to refactoring documentation and commit classification. A summary of these studies is provided in Table 1.

2.1. What Is Software Refactoring?

Software quality is a multi-faceted feature of any program dependent upon design, complexity, and a myriad of other aspects. It is also an inevitability that, given enough time, changes and additions to the code base will cause its design integrity to deteriorate. Refactoring is the practice of improving the internal software design without changing its behavior. It can be performed by using various types of refactoring operations, including renaming attributes, moving classes, splitting packages, etc. Refactoring code is a necessary step to help reverse the negative effects of continuous development short of starting from scratch every time the design deviates too far from its origins. In this manner, the design of the system can be altered without modifying its behavior or functionality.

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 class;
- *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.

2.2. Refactoring Documentation

Stroggylos and Spinellis [15] searched words stemming from the verb *refactor*, such as *refactoring* or *refactored*, to identify refactoring-related commits. Ratzinger et al. [16,17] also used a similar keyword-based approach to detect refactoring activity between a pair of program versions in order to identify whether a transformation contains refactoring. The authors identified refactorings based on a set of keywords detected in commit messages and focused on the following 13 terms in their search approach: *refactor*, *restruct*, *clean*, *not used*, *unused*, *reformat*, *import*, *remove*, *replace*, *split*, *reorg*, *rename*, and *move*. Later, Murphy-Hill et al. [18] replicated Ratzinger experiment in two open source systems using Ratzinger's 13 keywords. They concluded 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. [19] compared and evaluated three approaches, namely manual analysis, commit message, and dynamic analysis, in order 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. [20] surveyed 328 professional software engineers at Microsoft to investigate when and how they conduct refactoring. They first identified refactoring branches and then asked developers about the keywords that are usually used to mark refactoring events in commit messages. When surveyed, the developers mentioned several keywords to mark refactoring activities. Kim et al. matched the top ten refactoring-related keywords identified from the survey (*refactor*, *clean-up*, *rewrite*, *restructure*, *redesign*, *move*, *extract*, *improve*, *split*, *reorganize*, and *rename*) against the commit messages to identify refactoring commits from version histories. By 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 work [11,21–25] has explored how developers document their refactoring activities in commit messages; this activity is called Self-Admitted Refactoring or Self-Affirmed Refactoring (SAR). In particular, SAR indicates developers' explicit documentation of refactoring operations intentionally introduced during a code change.

2.3. Deep Learning

Implementing a deep learning approach for commit message classification resulted in high accuracy. For active learning of classifiers, an unlabeled dataset of commit messages is created, and labeling is performed after performing feature extraction using the Term Frequency Inverse Document. The approach followed the steps such as dataset construction, which includes text preprocessing and a feature extraction step; a multi-label active learning phase during which a classifier model is built and then evaluated and unlabeled instances are queried for labeling by an oracle; and classification of new commit messages. GitCProc [26] is used for data collection from 12 open source projects. Classifiers using active learning are tested by measures such as hamming loss, precision, recall, and F1 score. Active learning multi-label classification technique reduced the efforts needed to assign labels to each instance in a large set of commits. The classifier presented in the study by Gharbi and Sirine et al. [27] can be improved by considering the changes of the nature of the commits using commit time, and their types also automated commit classification written in different languages, i.e., multilingual classification is a gap for betterment. Mining the open source repositories is difficult for the software engineers

because of the error rate in the labeling of commits. Prior to this work, key word-based approaches are used for bug fixing commits classification. The method implemented by Zafar et al. [28] uses the deep learning models, Bidirectional Encoder Representations from Transformers (BERT), which can understand the context of commits and even the semantics for better classification by creating a hand labeled dataset and semantic rules for handling complex bug fix commits, which in turn reduced the error rate of labeling by 10%. Zafar et al. [28] analyzed git commits to check if they are bug fix commits or not; this will help the development team to identify future resources and achieve project goals in time by integrating NLP and BERT for bug fix commit classification. This Implemented approach is based on fine tuning with the deep neural network, which encodes the word relationships from the commits for the bug fix identification task.

2.4. Resampling Technique

Often, commit message datasets are imbalanced by nature, and it is difficult to build a classifier for such a dataset; it might cause undersampling and oversampling. The method proposed in [29] classifies commit messages extracted from GitHub by using the multiple resampling technique for highly imbalanced dataset, resulting in improvements in classification over the other classifiers. Imbalanced datasets often cause problems with the machine learning algorithm. There are three variants of resampling, under sampling, over sampling, and hybrid sampling. The undersampling method balances the class distribution to reduce the skewness of data by removing minority classes, whereas oversampling duplicates the examples from minority classes to minimize skewness, and hybrid sampling uses a combination of undersampling and oversampling. All these methods tend to maintain the goal of statistical resampling by improving the balance between the minority and majority classes. The study performed in [29] first creates the feature matrix, and resampling is performed by using the imbalanced learn sampling method. Here, a 10-fold cross validation is used to ensure consistent results. From the research study of [29], the questions concerning the development process such as “do developers discuss design” is answered.

2.5. DeepLink: Issue-Commit Link Recovery

For the online version of control systems such as GitHub, links are missing between the commits and issues. Issue commit links play an important role in software maintenance as they help understand the logic behind the commit and make the software maintenance easy. Existing systems for issue commit link recovery extracts the features from issue report and commit log but it sometimes results in loss of semantics. Xie and Rui et al. [30] proposed the design of a software that captures the semantics of code and issue-related text. Furthermore, it also calculates the semantics' similarity and code similarity by using support vector machine (SVM) classification. Deeplink followed the process in order to calculate the semantic and code similarity, which includes data construction, generation of code embeddings, similarity calculation, and feature extraction. The result is supported from [30] by the experiment performed on six projects, which answered the research questions relying on the effectiveness of deeplink in order to recover the missing links, effects of code context, and semantics of deeplink providing 90of F1-measure.

2.6. Code Density for Commit Message Classification

The classification of commits support the understanding and quality improvement of the software. The concept introduced by Honel et al. [31] uses code density, i.e., ratio between net and gross size of the code change, where net size is the size of the unique code in the system and gross size includes clones, comments, space lines, etc. Answers for the question are revealed by [31], and the question include the following: What are the statistical properties of commit message dataset? Is there any difference between cross and single project classification; Do classifiers perform better by considering the net size related attributes? Are the size and density related features suitable for commit message

classification? They further developed a git-density tool for analyzing git repositories. This work can be extended by considering the structural and relational properties of commits while reducing the dimensionality of features.

2.7. Boosting Automatic Commit Classification

There are three main categories of maintenance activities: predictive, adaptive, and corrective. Better understanding of these activities will help managers and development team to allocate resources in advance. Previous work performed on commit message classification mainly focused on a single project. The work performed by Levin et al. [32] presented a commit message classifier capable of classifying commits across different projects with high accuracy. Eleven different open source projects were studied, and 11,513 commits were classified with high kappa values and high accuracy. The results from [32] showed that when the analysis is based on word frequency of commits and source code changes, the model boosted the performance. It considered the cross-project classification. The methods are followed by gathering the commits and code changes, sampling to label the commit dataset, developing a predictive model and training on 85% data and testing on 15% of test data from same commit dataset, Levin et al. [32] used naïve Bayes to set the initial baseline on test data. This system of classification motivated us to consider the combinations of maintenance classes such as predictive + corrective. In order to support the validation of labeling mechanisms for commit classification and to generate a training set for future studies in the field of commit message classification work presented by Mauczka, Andreas et al. [33] surveyed source code changes labeled by authors of that code. For this study, seven developers from six projects applied three classification methods to evident the changes made by them with meta information. The automated classification of commits could be possible by mining the repositories from open sources, such as git. Even though precision recall can be used to measure the performance of the classifier, only the authors of commits know the exact intent of the change.

Mockus and Votta [34] designed an automatic classification algorithm to classify maintenance activities based on a textual description of changes. Another automatic classifier is proposed by Hassan [35] to classify commit messages as a bug fix, introduction of a feature, or a general maintenance change. Mauczka et al. [36] developed an Eclipse plug-in named Subcat to classify the change messages into the Swanson original category set (i.e., Corrective, Adaptive, and Perfective [37]), with an additional category, *Blacklist*. Mauczka et al. automatically assessed if a change to the software was due to a bug fix or refactoring based on a set of keywords in the change messages. Hindle et al. [38] performed the manual classification of large commits in order to understand the rationale behind these commits. Later, Hindle et al. [39] proposed an automated technique to classify commits into maintenance categories using seven machine learning techniques. To define their classification schema, they extended the Swanson categorization [37] with two additional changes: Feature Addition and Non-Functional. They observed that no single classifier is the best. Another experiment that classifies history logs was conducted by Hindle et al. [40], in which their classification of commits involves the non-functional requirements (NFRs) a commit addresses. Since the commit may possibly be assigned to multiple NFRs, they used three different learners for this purpose along with using several single-class machine learners. Amor et al. [41] had a similar idea to [39] and extended the Swanson categorization hierarchically. However, they selected one classifier (i.e., naïve Bayes) for their classification of code transactions. Moreover, maintenance requests have been classified by using two different machine learning techniques (i.e., naïve Bayesian and decision tree) in [42]. McMillan et al. [43] explored three popular learners in order to categorize software application for maintenance. Their results show that SVM is the best performing machine learner for categorization over the others.

2.8. Prediction of Refactoring Types

Refactoring is crucial as it impacts the quality of software and developers decide on the refactoring opportunity based on their knowledge and expertise; thus, there is a need for an automated method for predicting the refactoring. Proposed methods by Aniche et al. [44] have shown how different machine learning algorithms can be used to predict refactoring opportunities with a training set of 11,149 real-world projects from the Apache, F-Droid, and GitHub ecosystems and how the random forest classifier provided maximum accuracy out of six algorithms to predict method-level, class-level, and variable-level refactoring after considering the metrics and context of a commit.

Upon a new request to add a feature, developers try to decide on the refactoring in order to improve source code maintainability, comprehensibility, and prepare their systems to adapt to this new requirement. However, this process is difficult and time consuming. A machine learning based approach is a good solution to solve this problem; models trained on history of the previously requested features, applied refactoring, and code pick out information outperformed and provide promising results (83.19% accuracy) with 55 open source Java projects [45]. This study aimed to use code smell information after predicting the need of refactoring. Binary classifiers provide the need of refactoring and are later used to predict the refactoring type based on requested code smell information along with features. The model trained with code smell information resulted in the best accuracy. Table 1 summarizes all the studies relevant to our paper.

Table 1. Summarized literature review.

Study	Methodology	Classification Method	Category	Results
Towards Standardizing and Improving Classification of Bug-Fix Commit [28]	1. Implemented the deep learning model Bidirectional Encoder Representations from Transformers (BERT) which can understand the context of commits.	Deep Learning	1.Maintenance activities 2. Bug fixing	Semantic rules for handling complex bug fix commits reduced the error rate of labeling by 10%.
On the Classification of Software Change Messages using Multi-label Active Learning [27] DeclareRobustCommand	1. Labeled dataset after performing the feature extraction using Term Frequency Inverse Document.	Machine Learning and Systematic Labeling	1. Maintenance activities 2. Corrective Engineering	High accuracy in terms of precision, recall, and hamming loss.
Classifying Commit Messages: A Case Study in Resampling Techniques [29]	1. Applied a variety of resampling methods in different combinations 2. Tested highly imbalanced dataset with classes.	Machine Learning	Re-engineering	10% more accuracy achieved by classifier over the other classifiers.
Boosting Automatic Commit Classification Into Maintenance Activities By Utilizing Source Code Changes [32]	1. When the analysis is based on word frequency of commits and source code changes, model boosted the performance	Automated classifier	Maintenance Activities	Eleven different open source projects were studied, and 1151 commits were classified with high kappa value and high accuracy.

Table 1. Cont.

Study	Methodology	Classification Method	Category	Results
Identifying Unusual Commits on GitHub [46]	1. This model is able of classify large commits. 2. Commits made at the unusual time.	Statistical approaches	1. Forward Engineering	Reduced overwhelming notifications for developers.
Supervised machine learning algorithms to predict software refactoring [44]	1. The proposed methods have shown how different machine learning algorithms can be used to predict refactoring opportunities with a training set of 11,149 real-world projects.	Supervised machine learning approach	1. Maintenance activities	Predicted method-level, class-level and variable-level refactoring after considering metrics and context of commit; the random forest classifier produced maximum accuracy.

3. Research Methodology and Conduction

3.1. Approach Overview

Refactoring is necessary for improving software quality, and developers often perform refactoring to maintain their software systems, add new features, and fix problems with existing ones. While performing these changes, it is quite difficult to identify the correct refactoring. The methods described in this paper can help developers and maintenance teams to decide necessary refactoring for their software. We have implemented multiple machine learning models to predict the correct refactoring type based on the commits of project and code metrics. Our approach will also help the development team to decide if any commit is unusual. Detecting anomalies in commits was necessary since we are in the era of the open source community. The models built for this study are based on two approaches: commit message based and code metrics based. After performing some initial analyses, we found that code metrics based models were producing better accuracy than compared to the commit messages since code metrics are the key factors in deciding cohesion, coupling, and complexity of refactoring class. The following section will discuss the methodology we followed to collect the data, to preprocess it, and the methods used to build various ML models. As depicted in Figure 1, we followed a commit classification approach similar to [23,24,44].

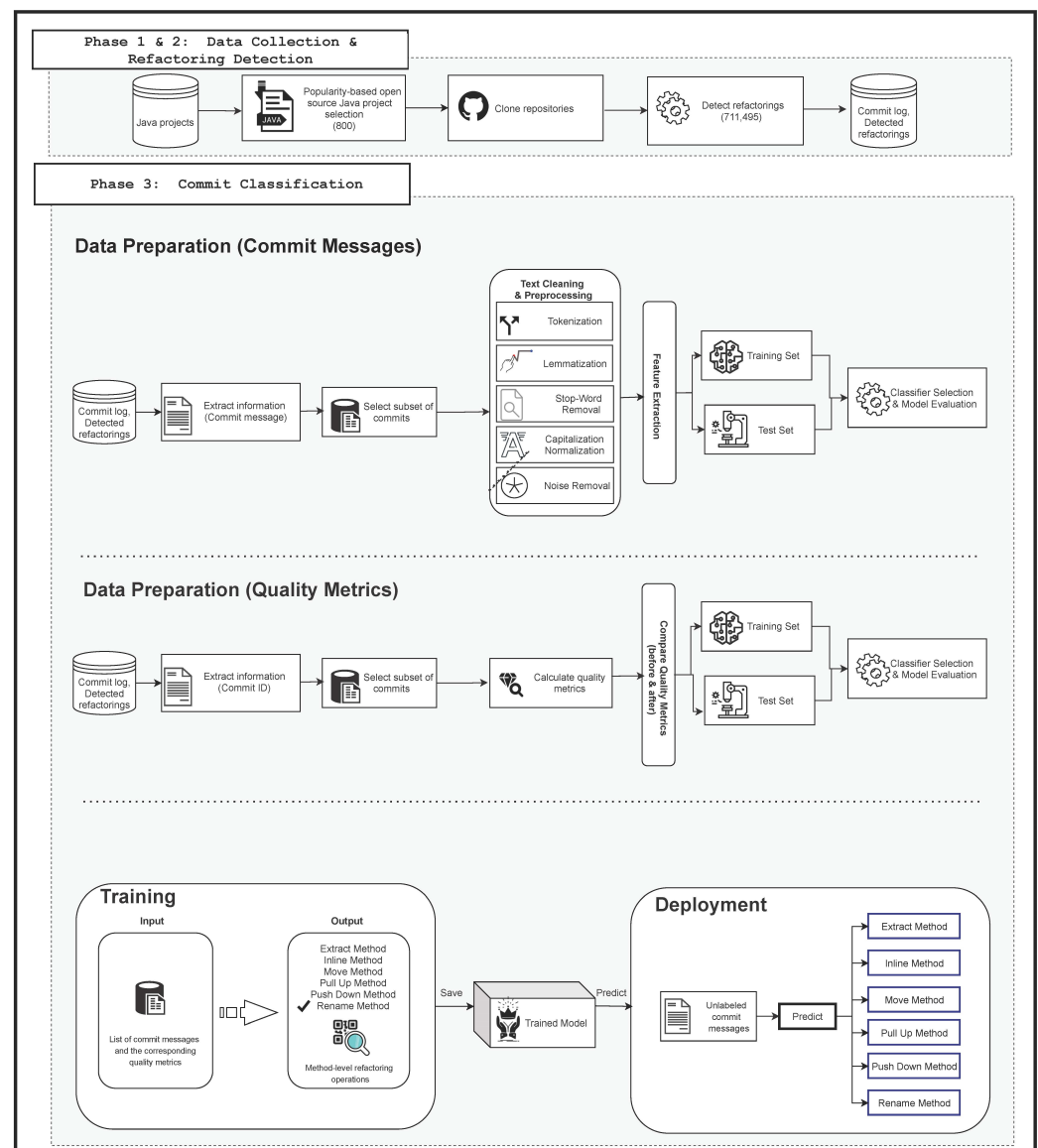


Figure 1. Overall framework.

As an illustrative example, Figure 2 details a commit for which its message states the relocation of the method *classFor(asmType)* to an internal class utility class for the purpose of applying the single responsibility principle and code reusability link to the commit: <https://github.com/modelmapper/modelmapper/commit/6796071fc6ad98150b6faf654c8200164f977aa4> (accessed on 20 September 2021). After running Refactoring Miner, we detected the existence of a *Move method* refactoring from the class *ExplicitMappingVisitor* to the class *Types*. The detected refactoring matches the description of the commit message and provides more insights about the old placement of the method.

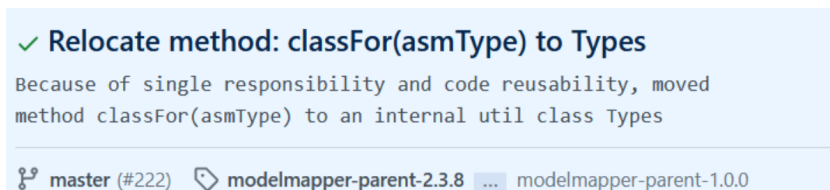


Figure 2. A sample instance of our dataset.

In a nutshell, the goal of our work is to automatically predict refactoring activity from commit messages and code metrics. In the data collection layer, we collected commits for projects from GitHub with web crawling for every project, and we prepared csv files with project commits and code metrics for further machine learning analysis. After this initial collection process, data were preprocessed to remove noise for model building. Extracting features helped us achieve results. Since we were dealing with text data, it was necessary to convert it with useful feature engineering. Preprocessed data with useful features were used for training various supervised learning models. We split our analysis into two parts based on our initial experiments. Only commit messages were not quite robust for predicting the refactoring type; thus, we tried to use code metrics. The following section will briefly describe the procedure used to build models with these three inputs.

As shown in Figure 1, our methodology contained two main phases: data collection phase and commit classification phase. Data collection will detail how we collected the dataset for this study, while the second phase focuses on designing the text-based and metric-based models under test conditions.

3.2. Data Collection

Our first step consists of randomly selecting 800 projects, which were curated open-source Java projects hosted on GitHub. These curated projects were selected from a dataset made available by [47], while verifying that they were Java-based, the only language supported by Refactoring Miner [48]. We cloned the 800 selected projects having a total of 748,001 commits and a total of 711,495 refactoring operations from 111,884 refactoring commits. To extract the entire refactoring history of each project, we used the the Refactoring Miner <https://github.com/tsantalis/RefactoringMiner> (accessed on 20 September 2021) tool introduced by [48], since 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 needed to randomly sample a subset while making sure it equitably represents the featured classes, i.e., refactoring types.

The data collection process has resulted in a dataset with five different refactoring classes, all detected at the method level, namely rename, push down, inline, extract, pull up, and move. The dataset used for this experiment is quite balanced. There are a total of 5004 commits in this dataset (see Table 2).

Table 2. Number of instances per class (Commit Message).

Refactoring Classes	Count
Rename	834
Push down	834
Inline	834
extract	834
Pull up	834
Move	834

3.3. Data Preprocessing

After importing data as panda dataframes, data are checked for duplicate commit IDs and missing fields. To achieve better accuracy, data with duplicate values and missing values should not be considered for further analysis. We also normalized the metric values using standard deviation, randomized the dataset with random sampling, and removed null entries. Since we are dealing with commit messages from VCS, text preprocessing is a crucial step. For commit messages to be classified properly by the classifier, they need to

be preprocessed and cleaned, and converted to a format that an algorithm can process. To extract keywords, we have followed the steps listed below:

—*Tokenization*: For text processing, we used NLTK library from python. The tokenization process breaks a text into words, phrases, symbols, or other meaningful elements called tokens. Here, tokenization is used to split commit text into its constituent set of words.

—*Lemmatization*: The lemmatization process replaces the suffix of a word or removes the suffix of a word to obtain the basic word form. In this case of text processing, lemmatization is used for part of the speech identification and sentence separation and keyphrase extraction. Lemmatization provided the most probable form of a word. Lemmatization considers morphological analysis of words; this was one of the reason of choosing it over stemming, since stemming only works by cutting off the end or the beginning of the word and takes list of common prefixes and suffixes by considering morphological variants. Sometimes this might not provide us with the proper results where sophisticated stemming is required, giving rise to other methodologies such as porter and snowball stemming. This is one of the limitations of the stemming method.

—*Stop Word Removal*: Further text is processed for English stop words removal.

—*Noise Removal*: Since data come from the web, it is mandatory to clean HTML tags from data. The data are checked for special characters, numbers, and punctuation in order to remove any noise.

—*Normalization*: Text is normalized, all converted into lowercase for further processing, and the diversity of capitalization in text is remove.

3.4. Feature Extraction

3.4.1. Text-Based Model

Feature extraction includes extracting keywords from commits; these extracted features are used to build a training dataset. For feature extraction, we have used a word embedding library from Keras, which provides the indexes for each word. Word embedding helps to extract information from the pattern and occurrences of words. It is an advanced method that goes beyond traditional feature extraction methods from NLP to decode the meaning of words, providing more relevant features to our model for training. Word embedding is represented by a single n-dimensional vector where similar words occupy the same vector. To accomplish this, we have used pretrained GloVe word embedding. The GloVe word embedding technique is efficient since the vectors generated by using this technique are small in size, and none of the indexes generated are empty, reducing the curse of dimensionality. On the other hand, other feature extraction techniques such as n-grams, TF-IDF, and bag of words generate very huge feature vectors with sparsity, which causes memory wastage and increases the complexity of algorithm.

Steps followed to convert text into word embedding: We converted the text into vectors by using tokenizer function from Keras, then converted sentences into numeric counterparts and applied padding to the commit messages with shorter length. Once we had the padded number sequence representing our commit message corpus, we then compared it with the pretrained GloVe word embedding and created the embedding matrix that has words from commit and respective values for each GloVe embedding. After these steps, we have word embeddings for all words in our corpus of commit messages.

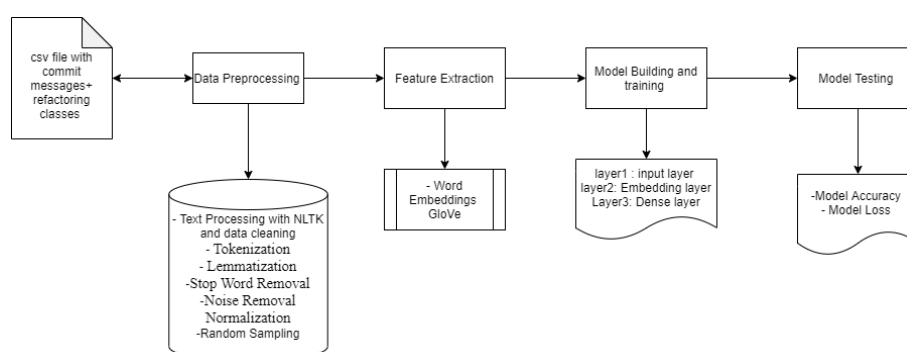
Text-Based Model Building. Model building and training: To build the model with commit messages as input in order to predict the refactoring type (see Figure 3), we used Keras functional API after we obtained the word embedding matrix. We followed the following steps:

- We created a model with an input layer of word embedding matrix, LSTM layer, which provided us with a final dense layer of output.
- For the LSTM layer, we used 128 neurons; for the dense layer, we have five neurons since there are five different refactoring classes.
- We have Softmax as an activation function in the dense layer and categorical_crossentropy as the loss function.

- As shown in Table 3, we also performed parameter hypertuning in order to choose the values of activation function, optimizer, loss function, number of nodes, hidden layers, epoch, number of dense layers, etc. The dataset and source code of this experiments is available on GitHub <https://github.com/smilevo/refactoring-metrics-prediction> (accessed on 20 September 2021).
- We trained this model on 70% of data with 10 epochs.
- After checking training accuracy and validation accuracy, we observed that this model is not overfitting.
- To test the model with only commit messages as input, we used 30% of data, and we used the evaluate function from the Keras API to test the model on test dataset and visualized model accuracy and model loss.

Table 3. Parameter hypertuning for LSTM model.

Parameters Used in LSTM Model	Values
Number of neurons	6
Activation Function	softmax
Loss Function	categorical_crossentropy
Optimizer	adam
Number of dense layers	1
Epoch	5

**Figure 3.** Overview of model with commit messages as input.

3.4.2. Metric-Based Model

We calculated the source code metrics of all code changes containing refactorings. We used "Understand" to extract these measurements <https://www.scitools.com> (accessed on 20 September 2021). These metrics have been previously used to assess the quality of refactoring or to recommend refactorings [3,49–51]. In addition to that, many previous papers have found significant correlation code metrics and refactoring [11,13,52]. Their findings show that metrics can be a strong indicator for refactoring activity, regardless of whether it improves or degrades these metric values. In order to calculate the variation of metrics, for each of the selected commits, we verified the set of Java files impacted by the changes (i.e., only modified files) before and after the changes were implemented by refactoring commits. Then, we considered the difference in values between the commit after and the commit before for each metric.

Metric-Based Model Building. After we split the data as training and test datasets. We built different supervised machine learning models to predict the refactoring class, as depicted in Figure 4. The steps we followed were the following steps:

- We used supervised machine learning models from the sklearn library of python.
- We trained random forest, SVM, and logistic regression classifiers on 70% of data.

- We performed the parameter hypertuning to obtain optimal results. Table 4 shows the selected parameters for each algorithm used in this experiment.
- After checking training accuracy and validation accuracy, we observed this model is not overfitting.
- Built models are tested on 30% of data, and the results were analyzed by varied machine learning measures such as precision, recall, F1- score, accuracy, confusion matrix, etc.

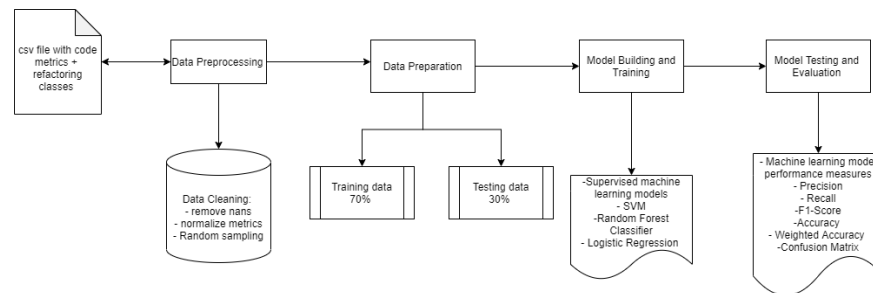


Figure 4. Framework of model with code metrics as input.

Table 4. Parameter hypertuning for Supervised ML Algorithms.

Supervised Learning Models	Parameters	Values
SVM	C	1.0
	Kernel	Linear
	Gamma	auto
	Degree	3
Random Forest	n_estimators	100
	criterion	gini
	min_samples_split	2
Logistic Regression	penalty	12
	dual	False
	tol	1e-4
	C	1.0
	fit_intercept	True
Naive Bayes	solver	lbfgs
	alpha	1.0
	fit_prior	True
	class_prior	None

3.5. Model Evaluation

We computed F-measures for multiclass in terms of precision and recall by using the following formula:

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

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

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

Accuracy is calculated as follows.

$$Accuracy = \frac{Tp + Tn}{Tp + Tn + Fp + Fn}$$

4. Experimental Results and Analysis

The following section will describe the experimental setup and the results obtained, followed by the analysis of research questions. The study performed in this paper can also be extended in the future to identify usual and unusual commits. Building multiple models with combinations of input provided us with better insights of factors impacting refactoring class prediction.

Our experiment is driven by the following research questions:

- **RQ1.** How effective is text-based modeling in predicting the type of refactoring?
- **RQ2.** How effective is metric-based modeling in predicting the type of refactoring?

4.1. RQ1. How Effective Is Text-Based Modeling in Predicting the Type of Refactoring?

Tables 5 and 6 show that the model produced a total of 54% accuracy on 30% of test data. With the "evaluate" function from keras, we were able to evaluate this model. The overall accuracy and model loss show that only commit messages are not very robust inputs for predicting the refactoring class; there are a number of reasons why the commit messages are unable to build robust predictive models. Often, the task of dealing with text to build a classification model is challenging, and feature extraction helped us to achieve this accuracy. Most of the time, the use of limited vocabulary by developers makes commits unclear and difficult to follow for fellow developers.

Table 5. Results of LSTM model with commit messages as input.

Model Accuracy	54.3%
Model Loss	1.401
F1-score	0.21035261452198029
Precision	1.0
Recall	0.1176215410232544

Table 6. Metrics per class.

	Precision	Recall	F1-Score	Support
Extract	0.56	0.66	0.61	92
Inline	0.47	0.43	0.45	84
Rename	0.56	0.68	0.62	76
Push down	0.37	0.39	0.38	87
Pull up	0.41	0.27	0.32	89
Move	0.97	0.95	0.96	73
Accuracy			0.55	501
Macro avg	0.56	0.56	0.56	501
Weighted avg	0.54	0.55	0.54	501

RQ1. Conclusion. One of the very first experiments performed provided us with the answer to this question, where we used only commit messages to train the LSTM model to predict the refactoring class. The accuracy of this model was 54%, and it was not up to expectations. Thus, we concluded that only commit messages are not very effective in

predicting refactoring classes; we also noticed that the developers' ability to use minimal vocabulary while writing code and committing changes on version control systems could be one of the reasons for inhibited prediction.

4.2. RQ2. How Effective Is Metric-Based Modeling in Predicting the Type of Refactoring?

To perform this experiment, we considered source code metrics as input for our classification model. There are 64 code metrics in our dataset; source code metrics helps to understand cohesion, coupling, and results in more accurate prediction of refactoring class since these metrics are closely related to each other. Metrics also help to identify which refactoring class is suitable for changes made in the source code and leverages traceability. We only dealt with numbers while performing this experiment. We trained different supervised machine learning models to predict refactoring class based on code metrics. We initially analyzed this dataset and carried out exploratory data analysis.

4.2.1. Classification Using Random Forest

We used the inbuilt functions from scikit-learn library of python to build a random forest model and to test it. We followed the below approach to set up this experiment:

- Imported pandas, numpy, and sklearn library from python;
- Imported dataset as data frame;
- Removed null entries, randomized the dataset, and normalized it with statistical modeling;
- Trained the random forest model and tested it.

This model produced overall 75% accuracy, and this was the best accuracy we have achieved so far. We used precision, recall, F1-score and support as measuring units to evaluate the model. Precision, recall, and F1-score for this model are shown in Table 7. Precision provides us with the percentage of positive predictions made correctly, whereas recall depicts the percentage of positive instances predicted correctly by the classifier over all positive instances in the dataset. The F-1 score is nothing but a harmonic mean of precision and recall.

Table 7. Results of random forest algorithm.

	Precision	Recall	F1-Score	Support
Extract	0.91	0.47	0.62	395
Inline	0.99	1.00	1.00	422
Move	0.94	0.69	0.80	429
Pull up	0.57	0.65	0.61	413
Push down	0.90	0.69	0.78	373
Rename	0.56	0.98	0.71	449
Accuracy			0.75	2481
Macro avg	0.81	0.75	0.75	2481
Weighted avg	0.81	0.75	0.75	2481

4.2.2. Classification Using Logistic Regression

Logistic regression is a predictive analysis algorithm based on the concept of probability, and it uses the sigmoid function to predict the classes. We have used the in-built functions from the sklearn library of Python to achieve the results. As it can be observed from Table 8, this model achieved only 47% accuracy, with best class accuracy for the inline class.

Table 8. Results of Logistic Regression model.

	Precision	Recall	F1-Score	Support
Extract	0.56	0.29	0.38	380
Inline	0.79	0.72	0.76	417
Move	0.94	0.69	0.80	429
Pull up	0.41	0.13	0.20	418
Push down	0.62	0.40	0.48	400
Rename	0.32	0.90	0.47	443
Accuracy			0.47	2481
Macro avg	0.53	0.46	0.45	2481
Weighted avg	0.53	0.47	0.45	2481

4.2.3. Classification Using Support-Vector Machines

Support-vector machines (SVMs) are one of the robust algorithms for multiclass classification due to its ability to not overfit models with multiple classes. SVM predicts the classes based on the separation distance in the hyperplane of data points, called support vectors. For this experiment, we have used linear kernel function. Table 9 shows the overall accuracy and class accuracies. This model achieved 44% of accuracy, with the best class accuracy for inline class of 79%.

Table 9. Results of SVM model.

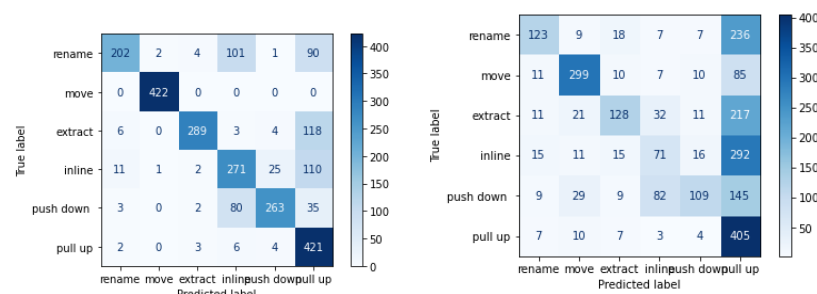
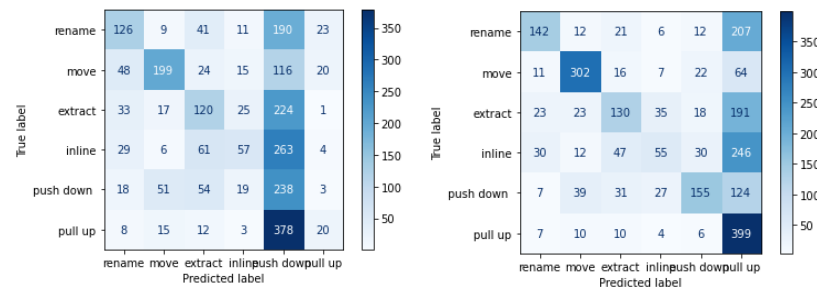
	Precision	Recall	F1-Score	Support
Extract	0.64	0.24	0.35	380
Inline	0.79	0.73	0.76	417
Move	0.69	0.30	0.42	423
Pull up	0.29	0.11	0.16	418
Push down	0.61	0.29	0.40	400
Rename	0.29	0.92	0.44	443
Accuracy			0.44	2481
Macro avg	0.55	0.43	0.42	2481
Weighted avg	0.55	0.44	0.42	2481

4.2.4. Classification Using Naive Bayes

Naive Bayes is one of the intuitive classification algorithms. However, it only works best with categorical data. It takes the freight class occurrence with correlation with other classes into consideration. From Table 10, it is clear that naive Bayes is not very robust for code metrics, and this model was only 35% accurate in predicting refactoring class. Figures 5 and 6 demonstrate the confusion matrix for each of the classifier.

Table 10. Results of Naive bayes model.

	Precision	Recall	F1-Score	Support
Extract	0.20	0.73	0.31	380
Inline	0.65	0.60	0.62	417
Move	0.39	0.25	0.31	423
Pull up	0.47	0.13	0.20	418
Push down	0.50	0.34	0.41	400
Rename	0.72	0.08	0.14	443
Accuracy			0.35	2481
Macro avg	0.49	0.35	0.33	24811
Weighted avg	0.49	0.35	0.33	2481

**Figure 5.** Confusion Matrix for random forest classifier and SVM (Metrics only).**Figure 6.** Confusion Matrix for naive Bayes classifier and logistic regression classifier (Metrics only).

RQ2. Conclusion. Random forest multiclass classification model built with source code metrics resulted in good accuracy, whereas the other three supervised learning models resulted in poor accuracy. In this case, our experiments show in Table 11 that random forest classification model is significantly robust in predicting refactoring classes when built on code metrics input.

Table 11. Results of LSTM model with commit messages and code metrics as input.

Model Accuracy	40.67%
Model Loss	1.310
F1-score	0.014
Precision	0.666
Recall	0.0071

For the study where we considered commit messages as input for the LSTM model, the "move" class was the most accurately predicted. After training supervised learning algorithms with code metrics as input, the random forest algorithm predicted inline class with 99% accuracy, whereas logistic regression predicted "move" with 94% of accuracy. The overall accuracy of the logistic regression classifier was noticeably poorer. SVM and naive Bayes predicted inline and rename refactoring classes more accurately.

4.3. Limitations

We have only considered six method-level refactoring classes: inline, extract, pull up, push down, move, and rename. In the future, we can extend the scope of our study by considering more classes. Code metrics are the key factors in deciding cohesion, coupling, and complexity of refactoring class; we have considered the dataset with 64 code metrics. We could also extend our research by taking more metrics into account. This will extend the scope of our study. The limited set of code metrics and refactoring classes was one of the limitations of this study. The methods used to deal with text data and numeric data were different and using a deep learning method with code metrics dataset was another option to consider. Another limitation of this study is focusing on only LSTM. In the future, we will also test different deep learning algorithms.

5. Threats to Validity

External Validity: This refers to the generalizability of the treatment outcomes. 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.

Construct Validity: This refers to the extent to which a test measures what it claims to be measuring. 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 impacts on our findings.

Internal Validity: This refers to the extent to which a piece of evidence supports the claim. 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 [48,53] 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.

6. Conclusions and Future Work

In this paper, we implemented different supervised machine learning models and LSTM models in order to predict the refactoring class for any project. To begin with, we implemented a model with only commit messages as input, but this approach led us to more research with other inputs. Combining commit messages with code metrics was our second experiment, and the model built with LSTM produced 54.3% of accuracy. Sixty-four different code metrics dealing with cohesion and coupling characteristics of the code are among one of the best performing models, producing 75% accuracy when tested with 30% of data. Our study significantly proved that code metrics are effective in predicting the refactoring class since the commit messages with little vocabulary are not sufficient for training ML models.

In the future, we would like to extend the scope of our study and build various models in order to properly combine both textual information with metrics information to benefit from both sources. Ensemble learning and deep learning models will be compared with respect to the combination of data sources.

Author Contributions: Data curation, E.A.A.; Investigation, P.S.S.; Methodology, P.S.S. and C.D.N.; Software, E.A.A.; Supervision, M.W.M.; Validation, E.A.A.; Writing—original draft, P.S.S. and A.O. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Tsantalis, N.; Chaikalis, T.; Chatzigeorgiou, A. JDeodorant: Identification and removal of type-checking bad smells. In Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering, Athens, Greece, 1–4 April 2008; pp. 329–331.
2. Zhang, M.; Baddoo, N.; Wernick, P.; Hall, T. Prioritising refactoring using code bad smells. In Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, 21–25 March 2011; pp. 458–464.
3. Mkaouer, W.; Kessentini, M.; Shaout, A.; Koligheue, P.; Bechikh, S.; Deb, K.; Ouni, A. Many-objective software remodularization using NSGA-III. *ACM Trans. Softw. Eng. Methodol.* **2015**, *24*, 1–45.
4. Ouni, A.; Kessentini, M.; Sahraoui, H.; Inoue, K.; Deb, K. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Trans. Softw. Eng. Methodol.* **2016**, *25*, 23.
5. Veerappa, V.; Harrison, R. An empirical validation of coupling metrics using automated refactoring. In Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, 10–11 October 2013; pp. 271–274.
6. Naiya, N.; Counsell, S.; Hall, T. The Relationship between Depth of Inheritance and Refactoring: An Empirical Study of Eclipse Releases. In Proceedings of the 2015 41st Euromicro Conference on Software Engineering and Advanced Applications, Madeira, Portugal, 26–28 August 2015; pp. 88–91.
7. Ubayashi, N.; Kamei, Y.; Sato, R. Can Abstraction Be Taught? Refactoring-based Abstraction Learning. In Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, Madeira, Portugal, 22–24 January 2018; pp. 429–437.
8. Counsell, S.; Swift, S.; Arzoky, M.; Destefanis, G. Do developers really worry about refactoring re-test? An empirical study of open-source systems. In Proceedings of the International Conference on Product-Focused Software Process Improvement, Wolfsburg, Germany, 28–30 November 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 159–166.
9. Counsell, S.; Arzoky, M.; Destefanis, G.; Taibi, D. On the Relationship Between Coupling and Refactoring: An Empirical Viewpoint. In Proceedings of the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Recife, Brazil, 19–20 September 2019; pp. 1–6.
10. Pantiuchina, J.; Lanza, M.; Bavota, G. Improving Code: The (Mis) perception of Quality Metrics. In Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 23–29 September 2018; pp. 80–91.
11. AlOmar, E.A.; Mkaouer, M.W.; Ouni, A.; Kessentini, M. On the impact of refactoring on the relationship between quality attributes and design metrics. In Proceedings of the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Recife, Brazil, 19–20 September 2019; pp. 1–11.
12. AlOmar, E.A.; Rodriguez, P.T.; Bowman, J.; Wang, T.; Adepoju, B.; Lopez, K.; Newman, C.; Ouni, A.; Mkaouer, M.W. How do developers refactor code to improve code reusability? In Proceedings of the International Conference on Software and Software Reuse, Hammamet, Tunisia, 9–11 November 2020; Springer: Berlin/Heidelberg, Germany, 2020; pp. 261–276.
13. Alrubaye, H.; Alshoaibi, D.; Alomar, E.; Mkaouer, M.W.; Ouni, A. How does library migration impact software quality and comprehension? an empirical study. In Proceedings of the International Conference on Software and Software Reuse, Hammamet, Tunisia, 9–11 November 2020; Springer: Berlin/Heidelberg, Germany, 2020; pp. 245–260.
14. Rebai, S.; Kessentini, M.; Alizadeh, V.; Sghaier, O.B.; Kazman, R. Recommending Refactorings via Commit Message Analysis. *Inf. Softw. Technol.* **2020**, *126*, 106332.
15. Stroggylos, K.; Spinellis, D. Refactoring—Does It Improve Software Quality? In Proceedings of the Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007), Minneapolis, MN, USA, 20–26 May 2007; pp. 10–10.
16. Ratzinger, J.; Sigmund, T.; Gall, H.C. On the Relation of Refactorings and Software Defect Prediction. In Proceedings of the 2008 International Working Conference on Mining Software Repositories, Leipzig, Germany, 10–11 May 2008; ACM: New York, NY, USA, 2008; MSR '08; pp. 35–38. doi:10.1145/1370750.1370759.
17. Ratzinger, J. sPACE: Software Project Assessment in the Course of Evolution. Ph.D. Thesis, Vienna University of Technology, Vienna, Austria. 2007.
18. Murphy-Hill, E.; Parnin, C.; Black, A.P. How we refactor, and how we know it. *IEEE Trans. Softw. Eng.* **2012**, *38*, 5–18.
19. Soares, G.; Gheyri, R.; Murphy-Hill, E.; Johnson, B. Comparing approaches to analyze refactoring activity on software repositories. *J. Syst. Softw.* **2013**, *86*, 1006–1022.
20. Kim, M.; Zimmermann, T.; Nagappan, N. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Trans. Softw. Eng.* **2014**, *40*, 633–649.

21. Zhang, D.; Bing, L.; Zengyang, L.; Liang, P. A Preliminary Investigation of Self-Admitted Refactorings in Open Source Software (S). In Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, 1–3 July 2018; [21] pp. 165–168. doi:10.18293/SEKE2018-081.
22. AlOmar, E.A.; Mkaouer, M.W.; Ouni, A. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In Proceedings of the 3rd International Workshop on Refactoring-Accepted, Montreal, QC, USA, 28 May 2019.
23. AlOmar, E.A.; Peruma, A.; Mkaouer, M.W.; Newman, C.; Ouni, A.; Kessentini, M. How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Syst. Appl.* **2020**, *167*, 114176.
24. AlOmar, E.A.; Mkaouer, M.W.; Ouni, A. Toward the automatic classification of self-affirmed refactoring. *J. Syst. Softw.* **2020**, *171*, 110821.
25. AlOmar, E.A.; Mkaouer, M.W.; Ouni, A. Mining and Managing Big Data Refactoring for Design Improvement: Are We There Yet? In *Knowledge Management in the Development of Data-Intensive Systems*; Taylor & Francis: Abingdon, UK, 2020; pp. 127–140.
26. Casalnuovo, C.; Suchak, Y.; Ray, B.; Rubio-González, C. Gitcproc: A tool for processing and classifying github commits. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, 10–14 July 2017; pp. 396–399.
27. Gharbi, S.; Mkaouer, M.W.; Jenhani, I.; Messaoud, M.B. On the classification of software change messages using multi-label active learning. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, Limassol, Cyprus, 8–12 April 2019; pp. 1760–1767.
28. Zafar, S.; Malik, M.Z.; Walia, G.S. Towards standardizing and improving classification of bug-fix commits. In Proceedings of the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Recife, Brazil, 19–20 September 2019; pp. 1–6.
29. Shekarforoush, S.; Green, R.; Dyer, R. Classifying commit messages: A case study in resampling techniques. In Proceedings of the 2017 International Joint Conference on Neural Networks (IJCNN), Anchorage, AK, USA, 14–19 May 2017; pp. 1273–1280.
30. Xie, R.; Chen, L.; Ye, W.; Li, Z.; Hu, T.; Du, D.; Zhang, S. DeepLink: A code knowledge graph based deep learning approach for issue-commit link recovery. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; pp. 434–444.
31. Hönel, S.; Ericsson, M.; Löwe, W.; Wingkvist, A. Importance and aptitude of source code density for commit classification into maintenance activities. In Proceedings of the 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), Sofia, Bulgaria, 22–26 July 2019; pp. 109–120.
32. Levin, S.; Yehudai, A. 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, Toronto, Canada, 8 November 2017; pp. 97–106.
33. Mauczka, A.; Brosch, F.; Schanes, C.; Grechenig, T. Dataset of developer-labeled commit messages. In Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, Italy, 16–17 May 2015; pp. 490–493.
34. Mockus, A.; Votta, L.G. Identifying Reasons for Software Changes using Historic Databases. In Proceedings of the 2000 International Conference on Software Maintenance, ICSM, San Jose, CA, USA, 11–14 October 2000; pp. 120–130.
35. Hassan, A.E. Automated Classification of Change Messages in Open Source Projects. In Proceedings of the 2008 ACM Symposium on Applied Computing, Ceara, Brazil, 16–20 March 2008; ACM: New York, NY, USA, 2008; SAC '08; pp. 837–841. doi:10.1145/1363686.1363876.
36. Mauczka, A.; Huber, M.; Schanes, C.; Schramm, W.; Bernhart, M.; Grechenig, T. Tracing Your Maintenance Work—A Cross-Project Validation of an Automated Classification Dictionary for Commit Messages. In *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, 24 March–1 April 2012. Proceedings*; de Lara, J.; Zisman, A., Eds.; Springer: Berlin, Heidelberg, 2012; pp. 301–315. doi:10.1007/978-3-642-28872-2_21.
37. Swanson, E.B. The Dimensions of Maintenance. In Proceedings of the 2Nd International Conference on Software Engineering, San Francisco, CA, USA, 13–15 October 1976; IEEE Computer Society Press: Los Alamitos, CA, USA, 1976; pp. 492–497.
38. Hindle, A.; German, D.M.; Holt, R. What Do Large Commits Tell Us?: A Taxonomical Study of Large Commits. In Proceedings of the 2008 International Working Conference on Mining Software Repositories, Leipzig, Germany, 10–11 May 2008; ACM: New York, NY, USA, 2008; pp. 99–108. doi:10.1145/1370750.1370773.
39. Hindle, A.; German, D.M.; Godfrey, M.W.; Holt, R.C. Automatic classification of large changes into maintenance categories. In Proceedings of the 2009 IEEE 17th International Conference on Program Comprehension, Vancouver, BC, Canada, 17–19 May 2009; pp. 30–39. doi:10.1109/ICPC.2009.5090025.
40. Hindle, A.; Ernst, N.A.; Godfrey, M.W.; Mylopoulos, J. Automated Topic Naming to Support Cross-project Analysis of Software Maintenance Activities. In Proceedings of the 8th Working Conference on Mining Software Repositories, Leipzig, Germany, 10–11 May 2008; ACM: New York, NY, USA, 2011; pp. 163–172. doi:10.1145/1985441.1985466.
41. Amor, J.; Robles, G.; Gonzalez-Barahona, J.; Navarro Gsyc, A.; Carlos, J.; Madrid, S. Discriminating development activities in versioning systems: A case study. 2006. Available online: https://www.researchgate.net/profile/Jesus-Gonzalez-Barahona/publication/228968358_Discriminating_development_activities_in_versioning_systems_A_case_study/links/0c9605200b2fd8eed900000/Discriminating-development-activities-in-versioning-systems-A-case-study.pdf (accessed on 30 September 2021).

42. Mahmoodian, N.; Abdullah, R.; Murad, M.A.A. Text-based classification incoming maintenance requests to maintenance type. In Proceedings of the 2010 International Symposium on Information Technology, Kuala Lumpur, Malaysia, 15–17 June 2010; Volume 2; pp. 693–697. doi:10.1109/ITSIM.2010.5561540.
43. McMillan, C.; Linares-Vasquez, M.; Poshyvanyk, D.; Grechanik, M. Categorizing Software Applications for Maintenance. In Proceedings of the 2011 27th IEEE International Conference on Software Maintenance, Williamsburg, VA, USA, 25–30 September 2011; IEEE Computer Society: Washington, DC, USA, 2011; pp. 343–352. doi:10.1109/ICSM.2011.6080801.
44. Aniche, M.; Maziero, E.; Durelli, R.; Durelli, V. The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring. *IEEE Trans. Softw. Eng.* 2020. doi:10.1109/TSE.2020.3021736.
45. Nyamawe, A.S.; Liu, H.; Niu, N.; Umer, Q.; Niu, Z. Feature requests-based recommendation of software refactorings. *Empir. Softw. Eng.* **2020**, *25*, 4315–4347.
46. Goyal, R.; Ferreira, G.; Kästner, C.; Herbsleb, J. Identifying unusual commits on GitHub. *J. Softw. Evol. Process.* **2018**, *30*, e1893.
47. Munaiah, N.; Kroh, S.; Cabrey, C.; Nagappan, M. Curating GitHub for engineered software projects. *Empir. Softw. Eng.* **2017**, *22*, 3219–3253.
48. Tsantalis, N.; Mansouri, M.; Eshkevari, L.M.; Mazinanian, D.; Dig, D. Accurate and efficient refactoring detection in commit history. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg Sweden, 27 May–3 June 2018; pp. 483–494.
49. Mkaouer, M.W.; Kessentini, M.; Bechikh, S.; Deb, K.; Ó Cinnéide, M. High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III. In Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, Vancouver, Canada, 12–16 July 2014; pp. 1263–1270.
50. Mkaouer, M.W.; Kessentini, M.; Bechikh, S.; Cinnéide, M.Ó.; Deb, K. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empir. Softw. Eng.* **2016**, *21*, 2503–2545.
51. Mkaouer, M.W.; Kessentini, M.; Cinnéide, M.Ó.; Hayashi, S.; Deb, K. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empir. Softw. Eng.* **2017**, *22*, 894–927.
52. Hamdi, O.; Ouni, A.; AlOmar, E.A.; Cinnéide, M.Ó.; Mkaouer, M.W. An Empirical Study on the Impact of Refactoring on Quality Metrics in Android Applications. In Proceedings of the 2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft), Madrid, Spain, 17–19 May 2021; pp. 28–39.
53. Silva, D.; Tsantalis, N.; Valente, M.T. Why We Refactor? Confessions of GitHub Contributors. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; ACM: New York, NY, USA, 2016; pp. 858–870. doi:10.1145/2950290.2950305.