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

On the Use of Textual Feature Extraction Techniques to Support the Automated Detection of Refactoring Documentation

Licelot Marmolejos · Eman Abdullah AlOmar · Mohamed Wiem Mkaouer · Christian Newman · Ali Ouni

Received: date / Accepted: date

Abstract Refactoring is the art of improving the internal structure of a program without altering its external behavior, and it is an important task when it comes to software maintainability. While existing studies have focused on the detection of refactoring operations by mining software repositories, little was done to understand how developers document their refactoring activities. Therefore, there is recent trend trying to detect developers documentation of refactoring, by manually analyzing their internal and external software documentation. However, these techniques are limited by their manual process, which hinders their scalability.

Hence, in this study, we tackle the detection of refactoring documentation as binary classification problem. We focus on the automatic detection of refactoring activities in commit messages by relying on text-mining, natural language preprocessing, and supervised machine learning techniques. We design our tool to overcome the limitation of the manual process, previously proposed by existing studies, through exploring the transformation of commit messages into features that are used to train various models. For our evaluation, we use and compare five different binary classification algorithms, and we test the effectiveness of these models using an existing dataset of manually curated messages that are known to be documenting refactoring activities in the source code. The experiments are carried out with different data sizes and number of bits. As per our results, the combination of Chi-Squared with Bayes Point Machine (BPM) and Fisher Score with Bayes Point Machine could be the most efficient when it comes to automatically identifying refactoring text patterns in commit messages, with an accuracy of 0.96, and an F-Score of 0.96.

Keywords— machine learning, refactoring, software quality.

Licelot Marmolejos · Eman Abdullah AlOmar · Mohamed Wiem Mkaouer · Christian Newman
Rochester Institute of Technology
E-mail: {lmarmolejos,eaa6167,mwmvse,cdnvse}@rit.edu

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

1 Introduction

During the lifecycle of a software system, developers need to make some changes to the source code of that system to facilitate its understandability and maintainability. To make that possible, they need to write comprehensive code and refactor very often. The term refactoring was first introduced by Opdyke in 1992 [30], who defined it as a group of operations that improve the structure of a system to make its evolution and design less complex and preserve its behavior.

A key factor when it comes to refactoring is that only the internal structure of a program is modified and its external behavior should stay the same. Some examples of refactoring operations are: renaming, extracting or moving any source code element, and they can be applied at different levels within the program; for instance, at package, class or method level.

Previous studies found that the motivation behind refactoring can be attributed to multiple reasons, e.g. to simplify code readability and maintainability [27, 4], improve internal quality attributes of a system [10, 3,5], minimize code duplication [38], to address defects [32], to avoid merge conflicts [35], to ease the code review process [25], and to quickly get other developers on track with their projects after being absent for a period of time [29].

While automating the detection of refactoring operations, from the source code, has reached a high level of accuracy [42], the need for close analysis of how developers document these refactoring activities. This can reveal more insights on the developers intent behind the application of these refactorings, which can support the on going research on refactoring automation. Therefore, recent studies [13,2] have manually investigated developers commit messages to extract a taxonomy on how developers actually document their refactoring activities. Refactoring documentation is coined as *Self-Admitted* or *Self-Affirmed* refactoring. As an illustrative example, we refer to the simplified example extracted from the `bekvon/residence` project [1] (last checked 2020/06/20) 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 whose type is *Rename Method*. Documenting refactoring, similarly to any type of code change documentation, is useful to decipher the rationale behind any applied change, and it can help future developers in various engineering tasks, such as program comprehension, design reverse-engineering, and debugging. However, the detection of such refactoring documentation was hardly manual and limited. There is a need for automating the detection of such documentation activities, with an acceptable level of accuracy.

Fig. 1: An example of a refactoring, and its corresponding documentation [1].

AlOmar et al. [2] presented a text-mining approach, in which they collected and analyzed thousands of commit messages from a dataset containing more than 400,000 commit messages from several open-source Java projects, generated using RefactoringMiner [42], a state-of-the-art refactoring detection tool, capable of identifying 38 refactoring types.

Further, they proposed the concept of Self-Affirmed Refactoring (SAR) patterns, which are text keywords that developers use to describe their refactoring activities in commit messages. They were able to identify around 87 SAR patterns, such as: "use better name", "move method", "core remodularization", "code cleanup", "minor simplification", etc. Nevertheless, they carried out the SAR identification manually, which is time-consuming, and subjective. This problem seems to be solvable using a simple string matching approach, however, recent studies have shown that developers misuse refactoring keywords, and therefore, a naive keyword matching will trigger a potential number of false positives. To address this challenge, we aim to design a solution that uses supervised learning to learn the context of the keywords, when properly used in the context of refactoring, and so accurately identifies these patterns, in an automatic way.

1.1 Research Goal

Therefore, we also propose a text-mining approach to automatically detect refactoring activities in commit messages by searching for a list of SAR patterns, i.e. if a commit message contains one of the SAR patterns, it is considered as "refactoring", otherwise as "non-refactoring". So, to conduct our study, we first extract 3,000 commit messages from the same dataset used in [2], to prepare a set of commit messages, which some are tagged as containing refactoring patterns (i.e., SAR) and those who do not. Next, we preprocess the content of the commit messages and build our refactoring classification model using the Feature Hashing technique to transform our data into numbers so that the used machine learning algorithms can operate on them.

To better learn the context of the keywords, we explore different feature selection methods, such as Chi-Squared (CHI) and Fisher Score (FS). These feature selections techniques were used to extract the learning knowledge for five different two-class machine learning algorithms, namely: Bayes Point Machine (BPM), Neural Network (NN), Logistic Regression (LR), Boosted Decision Tree (BDT), and Averaged Perceptron (AP), part of the Azure Machine Learning environment. After that, we evaluate and compare the results and discuss what combination of algorithms performs better in predicting SAR patterns or refactoring activities from commit messages.

1.2 Innovation highlights

After obtaining the results, we could observe that the accuracy of our classifier tends to be higher when filtering the features than when using all of them. In addition, we also tried using a different data transformation technique, called Latent Dirichlet Transformation or LDA, using the same dataset, and scoring method as well as machine learning algorithm but based on our results Feature Hashing gives higher accuracy and F-Score.

In this study, we were able to obtain an accuracy of up to 0.97 by using a combination of Feature Hashing as our data transformation technique, plus Chi-Squared as feature selection method and Bayes Point Machine (BPM) as the machine learning model.

We understand that recognizing refactoring paths in a software project relying only on commit messages can be a very challenging and a complex task due to a large amount of refactoring types, and the way developers document their refactoring-related changes. Yet; this solution can be used as a complementary technique to existing refactoring detection tools, as well as a quick solution for personal projects to effectively identify refactoring activities with minimal manual effort. Also, we would like to motivate other researchers to build on this approach and collect more refactoring keywords or SAR patterns.

- Testing various features selection techniques with various binary classifiers, for the problem of detecting refactoring documentation (Section 3.3).
- A new dataset of detected refactoring textual patterns (Table 3).
- A replication package of our work is publicly available for researchers and practitioners to replicate and extend[1].

This paper is structured as follows: **Section 2** talks about existing refactoring detection tools and their advantages and disadvantages. **Section 3** provides information about the technology used and the steps involved to carry out this study. **Section 4** discusses the results and presents the answers to our research questions. **Section 5** indicates the research implications of our model. **Section 6** discusses the model limitations while **Section 7** captures any threats to our study validity. and in **Section 8** this paper comes to its end.

2 Related Work

Due to the importance of refactoring in a software system, a number of tools have been built to make its detection possible based on different methodologies, namely, rules, metrics, graph matching, search and dynamic analysis. Details of some of the most relevant studies are provided in the following subsections.

Rule/heuristic-based technique

Tsantalis et al. [42] proposed RrefactoringMiner, also known as RMiner, an open-source tool with the potential of identifying 38 refactoring patterns at multiple granularity levels: package, type, method, considering only potential refactoring-data such as deleted, added and changed files. It is based on AST (Abstract Syntax Tree) statement matching algorithm, and predefined refactoring rules with no need for similarity threshold. Taneja et al. [39] created RefactLib, which analyzes two different versions of an Application Programming Interface (API) and detects and extracts refactoring instances depending on their textual similarity, the similarities, and size of the entities, and information regarding obsolete entities by using a set of change-metrics and syntactic analysis. Moreover, Weissgerber [44] presented an algorithm focused on detecting refactorings at different levels, to detect refactoring applied between two versions of a software system by parsing deltas and comparing entities based on their names and bodies similarities.

Xing and Strolia [45] created UMLDiff, to automatically detect changes in the structure of the design of subsequent versions of an object-oriented software system and capable of tracking changes among different classes with a focus on the interface level. Dig et al. [14] developed an Eclipse plugin, RefactoringCrawler, which can find seven high-level refactoring types from Java projects by creating a tree of a program in which each node of the tree is the representation of a source code entity (package, class, method, field), and using Shingle algorithm to find similarity between entities under evaluation.

Moreover, Kim et al. [22] build Ref-finder that works within the Eclipse IDE, which can detect 63 refactoring types. It takes two program versions as input, and decompose them as a database of logic facts and identifies refactorings via logic queries. Silva and Valente [35] came up with RefDiff, a tool that can detect 13 different refactoring types in Version Histories of Java programs based on static analysis and code similarity.

Metric-based

Demeyer et al. [12] tried to identify refactoring activities by using change metrics between two subsequent software releases of three general categories: method size, class size, and inheritance computed for each class. Chidamber et al. [11] proposed a suite of class-level metrics namely CK-metrics, in honor of the initial letter of their last names, to find refactorings in object-oriented programs. Mahouachi et al. [24] were able to

[1] https://drive.google.com/drive/folders/1h-ek4lc3O2XLCdDTQpMQjos5MM7uECif?usp=sharing

identify refactorings by combining software metrics and using a search-based process to reduce the difference in metrics.

Graph-matching based

Soetens et al. [37] implemented a tool to identify floss refactorings of renamed and moved methods by matching operation histories. Kehrer et al. [21] presented a tool able to identify complex differences from two versions of a software system using graph transformation rules, representing their operations to match them between the versions.

Search-based

Some other studies have shown that it is possible to detect refactorings in a software system by representing the structural changes between two of its versions as a graph search; for instance, Hayashi et al. [17], used two commit messages (previous and revised commits) as versions of a program to search paths between them. Thangthumachit et al. [40] detected refactoring operations at different levels of a program based on the similarity of its source code elements including refactoring types such as rename, move and extract.

Dynamic analysis-based

The objective of this technique is to analyze the behavior of a system after any modification through test cases and if the behavior was not the same, the change was not refactoring otherwise it was. For instance, SafeRefactor [36] generates unit tests to analyze the behavior of a system by comparing an original version and a modified one.

However, most of those previous studies rely on similarity thresholds and are only intended to be used on a specific programming language, mainly on Java projects. Additionally, most of those tools have to go through infinite loops to understand the design of a system; and therefore, demand large computational time, so in that case, performance becomes a concern. Additionally, it might be significantly hard to use them in real-world scenarios, particularly the ones that require much more semantic information on the source code, such as program dependency graphs, class models, meta-models or control flow graphs, which is very hard if not impossible to obtain in many languages (such as JavaScript or C), or when considering only program fragments (e.g., plug-ins).

Some additional studies have contemplated the size of commit messages as their main and only element to detect refactorings, like Hindle et al. [19], but it was later discussed by Hattori et al. [16] who indicated that the results of the previous study can be very ambiguous since there is not a consistent definition of what a large or small commit is.

Nevertheless, previous studies have shown that it is possible to detect refactorings by merely analyzing the commit messages since there are some words and phrases that specify refactoring operations (self-affirmed refactoring). For instance, Stroggylos et al. [38], identified refactorings from the source code version logs of different projects by searching for the word stemming from "refactor".

Further, AlOmar et al. [2] came up with a text mining-based study, in which they extracted refactoring commit messages by RMiner [42], and manually analyzed them to find patterns that developers use to indicate their refactoring-related changes. In their work, they came up with a list of keywords and phrases that could be very helpful to detect refactorings.

Despite some limitations that commit messages present due to the way developers document their activities, we believe that the study by AlOmar et al. [2] can be very useful when it comes to identifying refactorings in a general way as it can be applied to different programming languages and can be used as a complement to other approaches.

Thus, to overcome their limitations, as they performed a manual process, labor-intensive, time-consuming and error-prone, we propose an automatic way of identifying refactoring activities or SAR from commit messages by using Natural Language Processing (NLP), and training machine learning algorithms using a dataset containing the refactoring keywords that they suggest.

3 Methodology

In a nutshell, the goal of our work is to automatically identify then classify commit messages containing refactoring documentation, i.e., Self-Affirmed Refactoring. Our proposed framework can be seen in Figure 2, consisting of four main parts. In the first one, the data is prepared and the content of the commit messages is preprocessed to remove unnecessary and irrelevant information for the classification and to normalize the data. Whereas, in the second one, the data is converted into hash values, in which each of them represents one of more features in the commit messages. Moreover, in the third part, the features are filtered to only select the most important ones from the dataset, depending on the indicated number of desired features, and in the fourth part, the machine learning algorithms are trained and tested based on the selected features. The resulting two-class classifier is able to operate over unlabeled texts.

Figure 3, shows a general overview of our experimental procedure. It can be seen that it consists of five

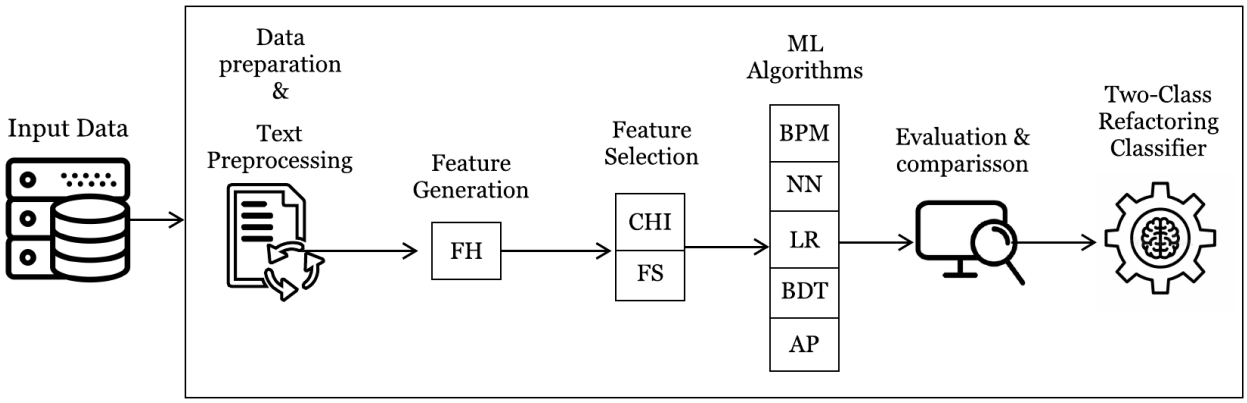


Fig. 2: Our framework for the detection of refactoring documentation.

main stages, i.e., (1) data preparation and text preprocessing, (2) feature generation, (3) feature selection, (4) training two-class refactoring classifier using different machine learning algorithms, and (5) evaluating and comparing the performances of the algorithms.

In this section, we provide details about the technology that we used, our data collection, and information with respect to all the methodologies applied in our work to build our machine learning solutions for refactoring classification and that would help us to answer our research questions. Further, we also talk about our experimental setup.

3.1 Data Collection

The dataset utilized in this study contains 1,208,970 refactoring operations, originally extracted by using RefactoringMiner [42], a state-of-the-art refactoring detection tool, from 3,795 open-source Java projects. This dataset is the same one used by AlOmar et al. [2], of which they analyzed 58,131 out of a total of 433,833 commit messages in the dataset, across 3,795 projects. Moreover, we took their list of SAR patterns to create our refactoring dataset, which can be seen in Table 1, and we also made some modifications to a few SAR patterns to obtain a higher number of refactoring operations indicated in commit messages. In Table 2 those modifications are shown.

Once we had collected the SAR patterns, we proceeded to manually extract 1,500 commit messages containing those patterns and 1,500 not containing SAR patterns using MySQL queries. Then, we manually labeled a total of 3,000 commit messages into two different categories; for instance, '0' for non-refactorings and '1' for refactorings. Next, we performed manual inspection of the labeled data to avoid adding false positives to our model. Therefore, if there was a commit message with the word "re-factoring" in the group of non-refactoring messages, we deleted that instance and did not consider it as part of that group.

During the manual classification of commit messages, we encountered some other patterns that developers use to indicate their refactoring-related activities. We understand that the more SAR patterns we pass to our model, the more it will learn and the higher the resulting recall for our solution will be. Therefore, in Table 3 we can see those new SAR patterns.

After all, we ended up incorporating all the SAR patterns, including the ones that we found, to our final set of refactoring commit messages, with another group of non-refactoring commit messages, which do not contain any of those SAR patterns.

3.2 Data Preparation and Text Preprocessing.

To prepare our data, we first detected the languages in the commit messages and split the dataset into two groups, one containing only English words and another one with words in different languages. For our study, we selected the English group since that is our target language.

Further, we preprocessed the content of the commit messages to normalize the data, and remove irrelevant words to boost the performance of our classifier, similar to [4]. This process involved the following steps:

- **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 [23]. In our work, we tokenize each commit by splitting the text into its constituent set of words. We also

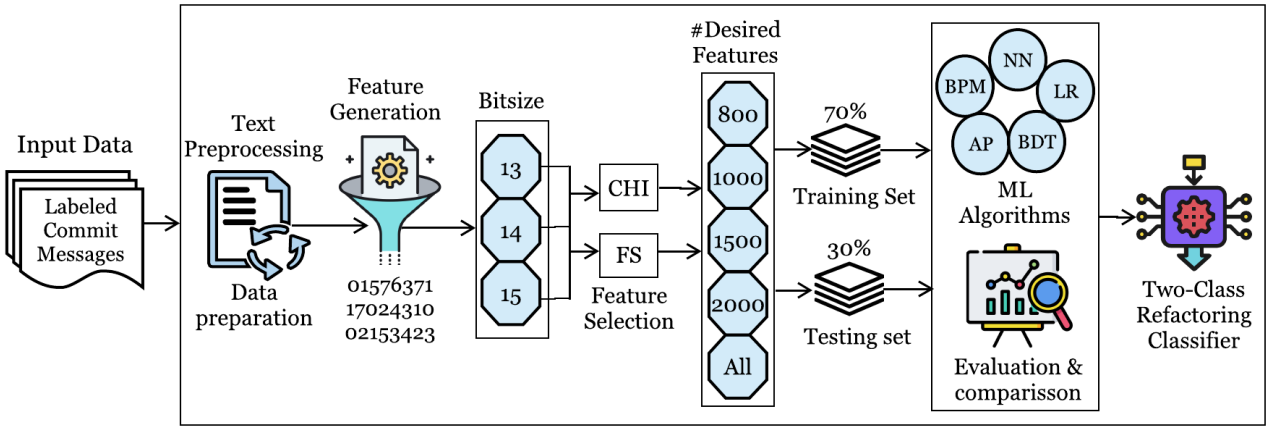


Fig. 3: Experimental procedure on comparing different Feature Hashing bitsize, scoring methods, numbers of desired features, and machine learning algorithms.

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). By lemmatizing the words, we obtained the base or canonical form of the verbs presented in the commit messages. For instance, the terms "cleaned, cleaning" would be changed to "clean", and the terms "is, are, were" would be changed to "be", and so on.
- **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 [33].
- **Case 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.

After preprocessing the commit messages, we manually verified that the text had been correctly preprocessed by randomly selecting some of them from our corpus, and we also validated to have 1,500 refactoring instances and 1,500 of non-refactoring.

3.3 Feature Extraction Using Feature Hashing

Since the machine learning algorithms cannot directly identify text, the data was converted into hashes based on the terms. Hence, given a bag-of-words (BOW) in different commit messages, the machine learning algorithms are expected to learn features to predict the output. The features that the algorithms used to train and predict in this study are derived using the Feature Hashing technique, also known as hashing trick, in which various words with varying word length are mapped to different features based on their hash values.

- **Feature Hashing (FH):** This is a technique that operates on high-dimensional text documents used as input in a machine learning model, to map string values directly into encoded features and represent them as integers [(Shi et al. [34], Weinberger et al. [43]]. This technique helps to reduce dimensionality and to make the feature weights lookup more efficient. In order to execute Feature Hashing, we need to specify the target column(s) that will be transformed and those columns need to be string data type, we also need to specify the number of bits o bitsize, to indicate the number of features to be generated, and the number of N-gram terms or sequence of n words treated as a unique unit.
  Furthermore, Feature Hashing uses a fast learning machine framework, namely: Vowpal Wabbit, which in turn uses Murmurhash3, a non-cryptographic and open-source hash function created by Austin Appleby. Some previous papers indicate how good the performance of Murmurhash3 is, particularly in [31]. That is a 32-bit algorithm and it determines the number of features to be created based on the following

Table 1: Original list of Self-Affirmed Refactoring (SAR) patterns

(1) Add*	(51) Fix technical debt
(2) Fix*	(52) Use a safer method
(3) Mov*	(53) Getting code out of
(4) Merg*	(54) Naming improvements
(5) Chang*	(55) Simplify the design
(6) Creat*	(56) Enhanced code beauty
(7) Inlin*	(57) Fix module structure
(8) Reduc*	(58) Improve code quality
(9) Remov*	(59) Minor simplification
(10) Renam*	(60) More easily extended
(11) Split*	(61) Pull up some code up
(12) Cleanup	(62) Replace it with word
(13) Enhanc*	(63) Many cosmetic changes
(14) Extend*	(64) Modularize this class
(15) Improv*	(65) Move unused file away
(16) Modify*	(66) Remove redundant code
(17) Replac*	(67) Avoid future confusion
(18) Rework*	(68) Get rid of unused code
(19) Rewrit*	(69) Moved more code out of
(20) Extract*	(70) Make maintenance easier
(21) Decompos*	(71) Removing unused classes
(22) Introduc*	(72) Renamed for consistency
(23) Redesign*	(73) Change package structure
(24) Refactor*	(74) Fixing naming convention
(25) Reformat*	(75) Major structural changes
(26) Simplify*	(76) Minor structural changes
(27) Nicer code	(77) Clean up unnecessary code
(28) Reorganiz*	(78) Improve naming consistency
(29) Encapsulat*	(79) Nonfunctional code cleanup
(30) Restructur*	(80) Refactor bad designed code
(31) Code cleanup	(81) Simplify code redundancies
(32) Change design	(82) Deleting a lot of old stuff
(33) Code revision	(83) Inlined unnecessary classes
(34) Use less code	(84) Removed poor coding practice
(35) Code cleansing	(85) Reorganize project structures
(36) Code cosmetics	(86) Code reformatting
(37) Fix code smell	(87) Moved and gave clearer names to
(38) Polishing code	(88) Antipattern bad for performances
(39) Code reordering	(89) Code maintenance for refactoring
(40) Nicer formatted	(90) Added more checks for quality factors
(41) Nicer Structure	(91) Refactoring towards nicer name analysis
(42) Use better name	
(43) Code improvements	
(44) Code optimization	
(45) Fix a design flaw	
(46) Fix quality flaws	
(47) Fix quality issue	
(48) Minor enhancement	
(49) Remove dependency	
(50) Simplify the code	

Table 2: List of modified SAR patterns

(1) Modif*	(15) Fix* quality flaw
(2) Simplif*	(16) Remov* dependency
(3) Polish* code	(17) Code improvement*
(4) Chang* design	(18) Fix* quality issue
(5) Us* less code	(19) Renam* consistency
(6) Simplif* code	(20) Reorganiz* structure
(7) Pull* some code	(21) Fix* technical debt
(8) Us* better name	(22) Remov* unused classes
(9) Code cosmetic*	(23) Remov* redundant code
(10) Delet* old stuff	(24) Improv* code quality
(11) Simplif* design	(25) Mov* more code out of
(12) Fix* code smell	(26) Fix* naming convention
(13) Nam* improvement	(27) Chang* package structure
(14) Modulariz* class	(28) Improv* naming

Table 3: List of new SAR patterns

(1) Typo	(15) Formatted
(2) Tidy*	(16) Cleaned up
(3) Spell* code	(17) Code clean
(4) Tidied	(18) Get rid of
(5) Polish*	(19) Getting rid of
(6) Clarif*	(20) Meaningful
(7) Separat*	(21) Modulariz*
(8) Optimiz*	(22) Pulled out
(9) Organiz*	(23) Cleaning up
(10) Clean-up	(24) Better name
(11) Pull out	(25) Pulling out
(12) Structur*	(26) New structure
(13) Correct*	(27) Duplicate code
(14) Normaliz*	

formula: (2^k)-1, being k the number of bits, so, if we specify a bitsize of 10, then (2^10)-1 = 1,023. So, the resulting number indicates the number of features to be generated after executing Feature Hashing. It is recommended to increase the number of bitsize depending on the size of N-grams vocabulary that will be used to train the model to avoid collisions; currently, the maximum number for bitsize that it can accept is 31; for instance, (2^31)-1.

Each hash value represents one or more N-gram features, depending on the specified number of bits, also called bitsize. When working with a large dataset, it is recommended to increase the number of bits to avoid data collision and have one hash value representing only one N-gram feature. Consequently, for us to know what could be a good bitsize to train our model, we first preprocessed our data and defined the N-gram as bigram since this one can contain more domain-specific semantic content than a singular word. Next, we counted the number of unique bigram terms in our preprocessed data to have an idea of how many features our dictionary of bigram terms contained.

Considering that the number of unique bigram terms in our preprocessed dataset was 16,897, we tried different number of bits based on that, for example, (1) 13, generating a total of 8,191 features, (2) 14, which generates a total of 16,383 features, and (3) 15, for the cre-

ation of 32,767 features. We then obtained the results after applying different numbers of bits to evaluate and compare the effectiveness of our model.

- **Feature Selection:** After obtaining a group of bigram terms as features, it is convenient to select only the predominant ones based on their frequency importance of words, i.e. good-candidate features, to reduce the dimensionality of the original feature space, and possibly improve the classification efficiency. Thus, to identify those bigram key terms with the greatest predictive power to train machine learning algorithms, and remove redundant features from our model, we tried two different feature selection methods, namely: Chi-Square (CHI) and Fisher Score (FS), which rely on statistical or information-theoretical measures.

- Chi-Squared (CHI): A commonly used statistic to analyze categorical data with more than one variable. This statistical method gives a score to each attribute of the dataset to then determine its relevance and correlation with the class [46]. The resulting assigned score is an indicator of how far observed attributes are from expected random attributes. So, in other words, it measures the lack of independence between a variable and a category.
- Fisher Score (FS): With this mechanism, a subset of features is selected to score the distance between them. Features in different classes should be as large as possible, whereas the distance between features from the same class should be as small as possible. This method also combines features to remove redundant ones.

We compared the performance of our machine learning model by trying different feature set sizes: (1) 800, (2) 1,000, (3) 1,500, and (4) 2,000, to further evaluate the performance of the algorithms using those fixed set of features. In addition, we also tried using all the features generated by the Feature Hashing technique to compare the results when using all the features or filtering them by using any scoring method.

3.4 Model Training & Building

To train and test the two-class machine learning algorithms contemplated in this study, we used a stratified train-test split methodology to divide the rows of the transformed dataset from the selected features into two different sets: 70% for training purposes while the remaining data 30% was used to measure its error rate.

Additionally, as we mentioned before, to improve the accuracy of our model, we increased the number of feature creation or "bitsize" to avoid data collision when adding more new data, and we also tried different scoring methods, e.g. Chi-Squared, and Fisher Score from the Azure ML environment, and different feature set sizes.

3.5 Classifier Selection & Evaluation

Below is a brief description of each of the classification algorithms used in this study.

- **Bayes Point Machine (BPM):** A linear classifier based on the Bayesian approach, originally proposed by Herbrich et al. [18]. This algorithm is a generative classifier, which means that it tries to understand the differences between the classes to further be able to classify them and even generate them. Moreover, it focuses on finding and returning the average of all the possible good decision straight-line boundaries between two classes and returns the average classifier in kernel space, which approximates the Bayesian classification strategy.
- **Averaged Perceptron (AP):** A supervised machine learning algorithm created by Frank Rosenblatt. Similar to Bayes Point Machine, it is also a linear classifier, i.e. it takes advantage of linearly separated data with large margins [15]. The name of "Perceptron" is because it combines a set of weights with the feature vector.
- **Logistic Regression (LR):** A linear statistical technique that uses a logistic function as its core to transform probability predictions into binary values, in other words, each feature in the dataset will be assigned a probability between 0 and 1. Contrary to Bayes Point Machine, this one is a discriminative classifier; thus, it only tries to understand whether any attribute belongs to a specific class without learning much in detail about it.
- **Boosted Decision Tree (BDT):** An ensemble learning technique that combines several weak learners, i.e. trees containing a set of input features, into stronger classifiers, then the predictions are based on that group of trees. Additionally, trees are generated iteratively, and a weight is assigned to the output of each tree, regarding its accuracy [20].
- **Neural Network (NN):** A set of interconnected layers of nodes, designed to recognize patterns and underlying relationships in an input set of data similar to the way the human brain works; e.g. learning to perform some tasks by examining training examples or historical data, which need to be previously labeled. In this type of algorithm, the data moves through the nodes in only one direction, and each

node will assign a weight to every different incoming item.

After training and testing the models, the effectiveness of each combination of algorithms is then evaluated based on the accuracy, precision, recall and error rate. A higher accuracy indicates that our refactoring classifier is able to correctly predict commit messages as refactoring or not. Moreover, precision refers to how many commit messages predicted as refactoring were valid refactoring, while recall shows how often those refactoring messages were captured by the model. Thus, the higher the recall, the more refactoring elements can be identified and the higher the precision, the more predictions are correct. We used the Formula 4 and 3 to obtain the accuracy and the F-Score, while Formula 1 for the precision, and Formula 2 for the recall of our model.

On the other hand, if the test set presents a high error rate and a low error rate in the training set, it is an indication of high variance as the model failed to correctly classify new commit messages, but if both training and test sets show a high error rate, that is high bias.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F - Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4}$$

For this study, we used a PC operating on Windows 10 (64-bit), with Intel(R) Core(™) i7-3630QM CPU, and 8GB RAM. However, we used Azure Machine Learning Azure Machine Learning Studio [28], a cloud-based tool, and Google Chrome as our browser, to conduct the experiments and to build, train, test, and evaluate our models with the previously mentioned algorithms, and further deployed a predictive experiment with the combination of algorithms with the highest accuracy..

Table 4: Performance of different classifiers (Binary Classification).

Classifier	Precision	Recall	Accuracy	F-score
Chi-Squared - Bitsize : 13				
Bayes Point Machine	0.93	0.95	0.94	0.94
Averaged Perceptron Method	0.93	0.91	0.92	0.92
Logistic Regression	0.95	0.83	0.90	0.89
Gradient Boosted Machine	0.93	0.90	0.91	0.91
Neural Network	0.96	0.84	0.90	0.89
Chi-Squared - Bitsize : 14				
Bayes Point Machine	0.95	0.96	0.96	0.96
Averaged Perceptron Method	0.97	0.91	0.94	0.94
Logistic Regression	0.96	0.84	0.90	0.90
Gradient Boosted Machine	0.93	0.89	0.91	0.91
Neural Network	0.97	0.86	0.91	0.91
Chi-Squared - Bitsize : 15				
Bayes Point Machine	**0.96**	**0.96**	**0.96**	**0.96**
Averaged Perceptron Method	0.98	0.93	0.96	0.96
Logistic Regression	0.97	0.84	0.91	0.90
Gradient Boosted Machine	0.94	0.90	0.92	0.92
Neural Network	0.98	0.90	0.94	0.94
Fisher score - Bitsize : 13				
Bayes Point Machine	0.93	0.95	0.94	0.94
Averaged Perceptron Method	0.94	0.92	0.93	0.93
Logistic Regression	0.95	0.94	0.90	0.89
Gradient Boosted Machine	0.94	0.89	0.92	0.91
Neural Network	0.97	0.83	0.90	0.89
Fisher score - Bitsize : 14				
Bayes Point Machine	0.95	0.96	0.95	0.95
Averaged Perceptron Method	0.96	0.91	0.94	0.94
Logistic Regression	0.96	0.84	0.90	0.90
Gradient Boosted Machine	0.94	0.90	0.92	0.92
Neural Network	0.94	0.91	0.93	0.93
Fisher score - Bitsize : 15				
Bayes Point Machine	**0.96**	**0.96**	**0.96**	**0.96**
Averaged Perceptron Method	0.97	0.92	0.95	0.95
Logistic Regression	0.97	0.85	0.91	0.90
Gradient Boosted Machine	0.94	0.90	0.92	0.92
Neural Network	0.98	0.89	0.93	0.93

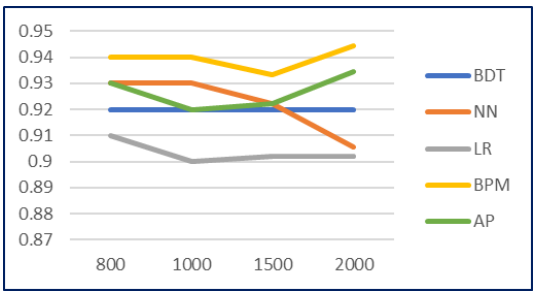
4 Results & Discussion

Our experiments are driven by the following research questions:

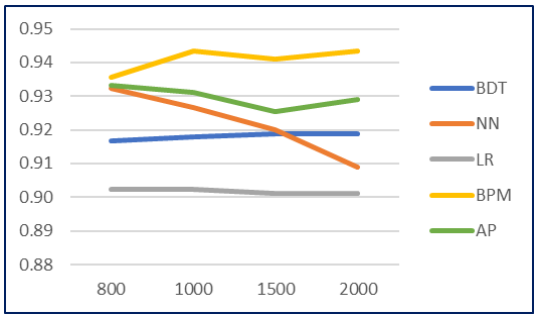
RQ1: How effective is our approach in identifying self-affirmed refactoring (SAR) patterns for refactoring activities from commit messages compared to the baseline [1]?
RQ2: What combination of machine learning techniques and algorithms, from the ones contemplated in our framework, gives the highest accuracy predicting SAR patterns in commit messages?
RQ3: What is the impact of applying Feature Hashing in our SAR classification solution instead of any other data transformation technique?

A total of 3,000 labeled commit messages were used as our dataset in Azure ML, of which 50% contain SAR patterns, while the remaining 50% do not.

A comparison between classification algorithms is reported in Table 4. The accuracy curves of the contemplated machine learning algorithms using Fisher Score as scoring method and different numbers of desired

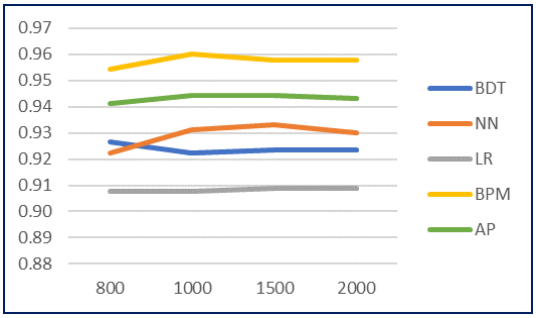


(a) Accuracy curves using a bitsize of 13 and Fisher Score

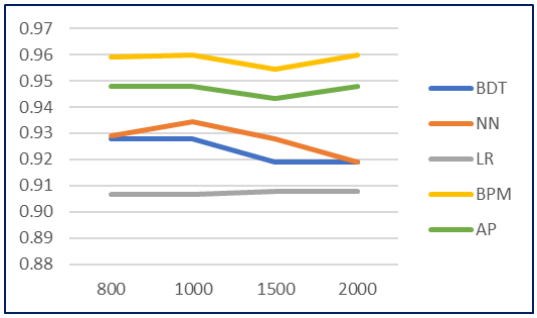


(b) Accuracy curves using a bitsize of 13 and Chi-Squared

Fig. 4: Accuracy using Fisher Score and Chi-Squared, Bitsize:13

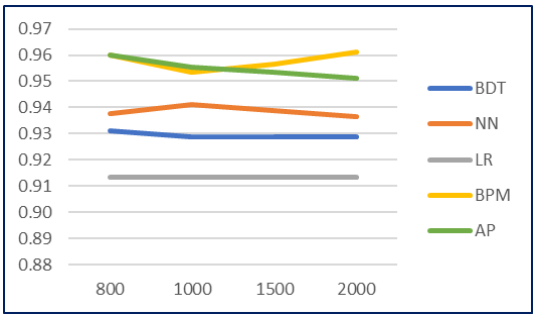


(a) Accuracy curves using a bitsize of 14 and Fisher Score

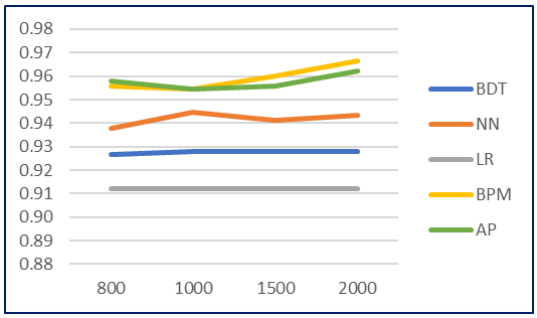


(b) Accuracy curves using a bitsize of 14 and Chi-Squared

Fig. 5: Accuracy using Fisher Score and Chi-Squared, Bitsize:14



(a) Accuracy curves using a bitsize of 15 and Fisher Score



(b) Accuracy curves using a bitsize of 15 and Chi-Squared

Fig. 6: Accuracy using Fisher Score and Chi-Squared, Bitsize:15

features are shown in Figure 4a (with a bitsize of 13), in Figure 5a (with a bitsize of 14), and Figure 6a (with a bitsize of 15). On the other hand, the accuracy curves using Chi-Squared can be seen in Figure 4b (using a bitsize of 13), in Figure 5b (with a bitsize of 14), and in Figure 6b (with a bitsize of 15). In addition, we also obtained the accuracy curves using different number of bits without any scoring method and the results are shown in Figure 8.

Moreover, to simplify the presentation of our results, we decided to show the F-Score, which is the weighted average of precision and recall, which range is between 0 and 1, being '1' the ideal value, instead of showing the results of both metrics: recall and precision. Therefore, the F-Scores of the used machine learning algorithms and using Fisher Score as scoring method can be seen in Figure 9a (with a bitsize of 13), Figure 10a (with a bitsize of 14) and Figure 11a (with a bitsize of 15).

Further, we also present the F-scores obtained after using Chi-Squared as our scoring method; this Figure 9b shows the results with a bitsize of 13, Figure 10b displays the results using a bitsize of 14, and Figure 11b presents the results using a bitsize of 15. Subsequently,

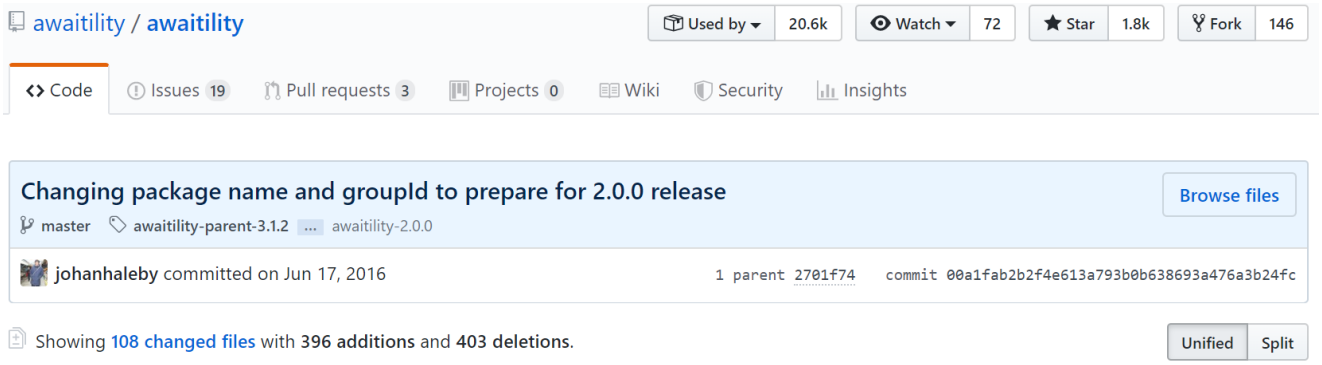


Fig. 7: Example of refactoring commit message that confused the classifiers.

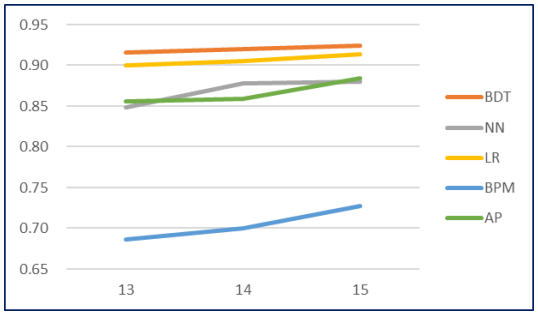


Fig. 8: Accuracy Curves using all the features with no scoring method

the F-scores using different number of bits and without filtering the features is presented in Figure 12.

According to our results, we can infer that the combination of using Bayes Point Machine with a bitsize of 15 and filtering out the data by 2,000 using Chi-Squared and Fisher Score gives the highest accuracy to our solution.

Given those results, we can draw a conclusion that in terms of refactoring classification accuracy, the ranking of the five machine learning algorithms that we used is as follows:BPM>AP>NN>BDT>LR.

To better understand the cases in which our classifiers are not performing well, Figures 7 shows one case of a commit message that confused the classifiers when performing two-class classifications. The commit message contains a pattern (*i.e.*, changing package name) that is a synonym of the patterns "renam*" or "use better name".

4.1 RQ1: How effective is our approach in identifying self-affirmed refactoring (SAR) patterns from commit messages compared to the baseline [2]?

Our resulting machine learning classifier is capable of identifying thousands of refactoring activities in commit messages in an automatic way with less human effort and in less time than the one by AlOmar et al. [2], who took around 7 days to identify some SAR patterns in thousands of commit messages.

4.2 RQ2: What combination of machine learning techniques and algorithms, from the ones contemplated in our framework, gives better accuracy and precision when predicting SAR patterns from commit messages?

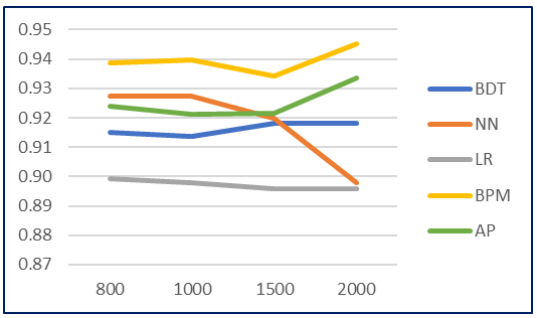
We were able to obtain the highest accuracy by combining Feature Hashing technique with a bitsize of 15, Chi-Squared or Fisher Score as the scoring method, with a number of 2,000 desired features and Bayes Point Machine as the machine learning model, giving an accuracy of 0.96.

4.3 RQ3: What is the impact of applying Feature Hashing in our SAR classification solution instead of any other data transformation technique?

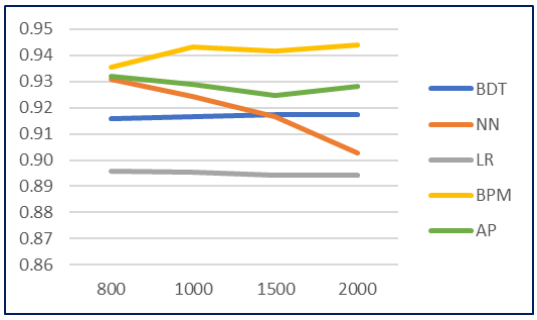
To answer this question, we tried using a different data transformation technique called Latent Dirichlet Allocation or LDA, using Chi-Squared as scoring method and Bayes Point Machine as machine learning algorithm. According to the obtained accuracy scores, Feature Hashing performs better on this type of solution with an accuracy of 0.96 (using the combination with the highest combination of algorithms), while the accuracy using LDA was of 0.67.

5 Research Implications

Our framework is useful to the refactoring community in particular and software engineering in general. Detecting refactoring documentation reveals developers strategies in handling and reducing technical debt

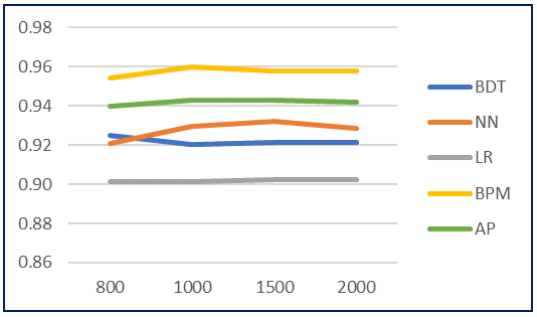


(a) F-Score using a bitsize of 13 and Fisher Score

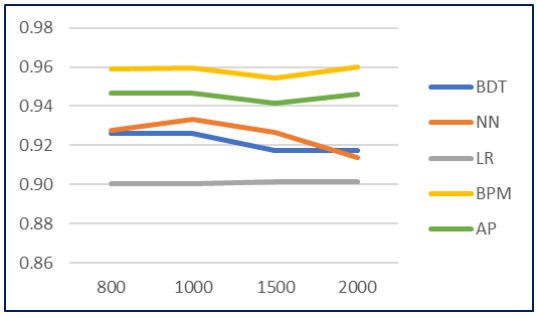


(b) F-Score using a bitsize of 13 and Chi-Squared

Fig. 9: F-Score using Fisher Score and Chi-Squared, Bitsize:13

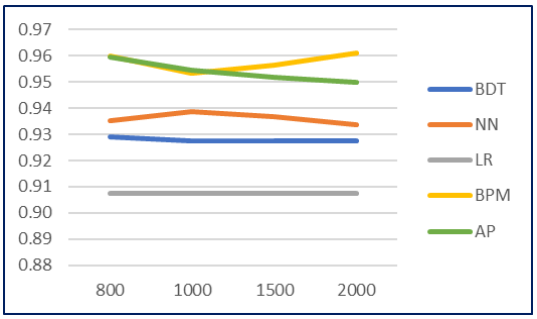


(a) F-Score using a bitsize of 14 and Fisher Score

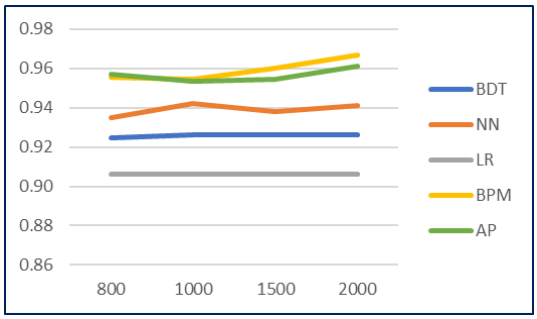


(b) F-Score using a bitsize of 14 and Chi-Squared

Fig. 10: F-Score using Fisher Score and Chi-Squared, Bitsize:14



(a) F-Score using a bitsize of 15 and Fisher Score



(b) F-Score using a bitsize of 15 and Chi-Squared

Fig. 11: F-Score using Fisher Score and Chi-Squared, Bitsize:15

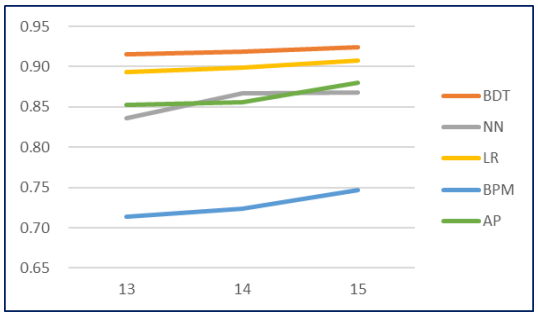


Fig. 12: F-Score using all the features with no scoring method

in their design and source code. Several studies have provided techniques to manage technical debt [41,8], or improving the design structural measurements [26, 9]. But, there is no association of these suggested refactorings with any context, which makes them disconnected from what developers are in the middle of handling. Detecting textual description of refactoring, once associated with the detection of refactorings, will infer the rationale behind the refactoring application.

As we observe the output of our model, we have noticed that developers tend to add a high-level description of their refactoring activity, and occasionally mention their intention behind refactoring (remove duplicate code, improve readability, etc.), along with mentioning the refactoring operations they apply (type migration, inline methods, etc.).

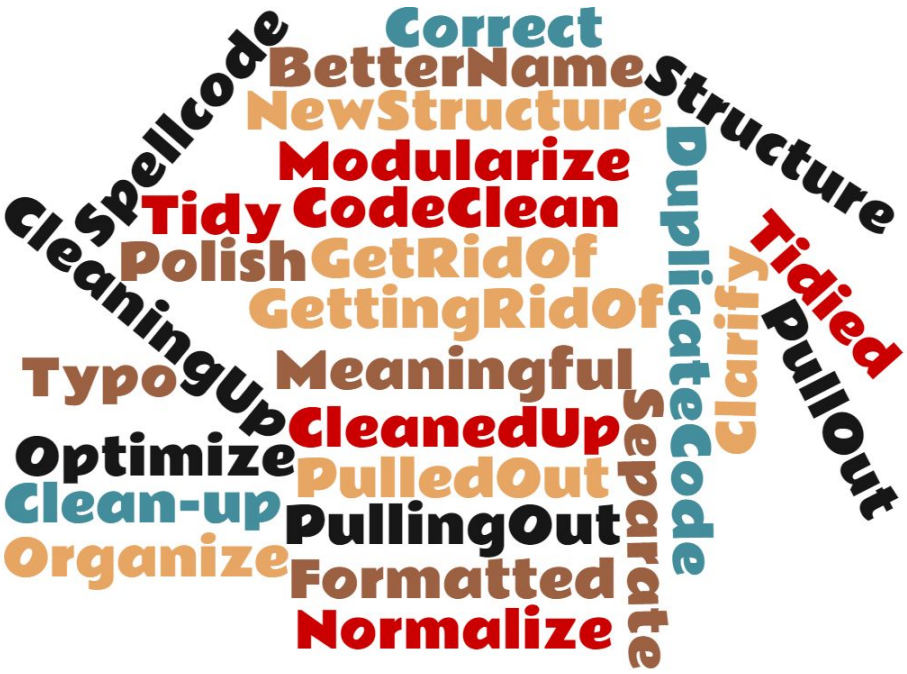


Fig. 13: Sample of patterns identified by automatic classification.

Such information is valuable for researchers and practitioners, as it extracts common refactoring textual patterns, revealing what developers actually care about, and so, it tunes existing research toward designing tools that are most likely to be accepted by developers. For instance, we have noticed that developers explicitly state several refactorings as part of them migrating APIs [7]. yet, there is no refactoring tool that supports such delicate operation, which requires removing all methods from one third-party library, and replacing them with methods from another library [6].

Another interesting finding that we report is the misuse of developers for some refactoring keywords, which can potentially hinder the accuracy of our model. Also, as developers typically focus on describing their functional updates and bug patches, refactoring could be present in these scenarios but without being explicitly mentioned. Moreover, there is no standard or guidelines related to formal refactoring documentation. This makes the documentation process subjective and developer specific. The fine-grained description of refactoring can be time consuming, as typical description should contain indication about the operations performed, refactored code elements, and a hint about the intention behind the refactoring.

Refactoring documentation is a pure reflection the developer's understanding of what has been improved in the source code, which can be different in reality, as the developer may not necessarily adequately estimate the refactoring impact on the quality improvement. In this context, our model helps with capturing refactoring intents, which can allow code review to verify its correctness.

6 Limitation

Our work is currently considered commit messages that are extracted from open source Java projects. We have not explored how our approach behaves across different programming languages. Thus, our future work will focus on building a model that works well across languages as developers might have different ways to document refactorings for each programming language. Another limitation of the study is related to the feature set size. We have evaluated the model using only four feature sizes. As feature size might have an impact on the performance of the model, we plan to try more feature sizes and evaluate the performance of the model. Moreover, our results do not show which exact source code elements were refactored, or how many refactoring operations were performed.

Another intuitive limitation of our work concerns the ambiguity that eventually may exist in commit messages. In other terms, if the committer does not properly describe the code changes, it would be hard for us to properly identify the corresponding code changes we are looking for. This limitation does not only apply to our study, but to any study which relies on analyz-

ing natural language to decipher and understand their corresponding code changes. Eventually, our approach may be useful in the context of checking the consistency between good changes and documentation. In other terms, it is possible to use our model as a Saturday check together the code changes were properly described in a way that an automated model can identify the existence of refactoring operations.

7 Threats to Validity

Internal validity. The evaluation oracle that we used can be related to human errors since we performed a manual task to label the commit messages for training purposes. Nonetheless, to address that issue, we inspected all the commit messages to validate that they were correctly labeled before executing our experiments.

Since our framework relies on commit messages, we used well-engineered Java projects that are most likely to be well documented, when performing our study. Additionally, a well-commented project might not contain SAR as developers might not document refactoring activities in the commit messages. We mitigate this risk by choosing projects that do contain such documentation for our analysis.

External Validity. One of the most important limitations of our approach is that developers do not usually indicate their refactoring-related activities in commit messages as mentioned by Weissgerber [44]. To mitigate this threat, we used well-commented Java projects when performing our study. The evaluation of accuracy, precision, and recall of our model is based on only 3,000 commit messages from different open-source Java projects; therefore, we cannot claim that the result of those metrics will be the same for a different set of commit messages from different types of projects and that the sample size is good enough to yield statistically significant results. Additionally, we were limited to the Azure ML classifiers and the results are based on the default settings as we did not change any parameter value on the used machine learning algorithms. Also, our dataset was originally extracted by using RefactoringMiner [42]; for instance, any limitation or bias from this tool would be reflected in our collected data.

8 Conclusion

Refactoring is a key element when it comes to allowing the evolution of a software system and keeping its quality high. Thus, in this paper, we proposed a framework to identify and then classify refactoring documentation reported in the in commit messages (known as Self-Affirmed Refactorings (SAR) [2]) patterns by using different techniques, such as Feature Hashing and feature selection (Chi-Squared and Fisher Score), and five machine learning algorithms, e.g., Bayes Point Machine (BPM), Averaged Perceptron (AP), Logistic Regression (LR), Boosted Decision Tree (BDT), and Neural Network (NN).

Our main purpose was to find an efficient combination of algorithms that could predict refactorings. Thus, for this, we built, trained, evaluated and compared different refactoring classification models using Azure ML studio, which is a cloud-based environment that consists of multiple machine learning modules to ease the process from building to deploy machine learning solutions.

After we trained different machine learning models and evaluated their results, we selected the one with the best performance regarding predicting SAR patterns. We were able to obtain the highest accuracy among the other ones of 0.96 by combining Feature Hashing with a bitsize of 15 plus filtering the data by 2,000 desired features using Chi-Squared or Fisher Score and Bayes Point Machine.

As an additional contribution, we constructed a predictive analysis model and made it publicly available in the form of a web service to let others access and use our solution to identify refactoring activities in a set of commit messages.

In the future, we intend to explore more commit messages from a more generalized variety of software projects to collect more SAR patterns to keep training our model with new ones. Additionally, we would like to try using a different machine learning environment or connect scripts to our model that could give us the measurement of the execution time and retrieve the text of the top features to better understand the operating logic of the methods and algorithms that are part of our solution.

References

1. https://github.com/bekvon/residence/commit/76c364ea47e5a28b2041a0bb3323cb48bab180c9
2. 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 3nd International Workshop on Refactoring-accepted. IEEE (2019)
3. 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: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–11. IEEE (2019)
4. 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 algo-

rithms to classify refactoring documentation. Expert Systems with Applications p. 114176 (2020)
5. AlOmar, E.A., Rodriguez, P.T., Bowman, J., Wang, T., Adepoju, B., Lopez, K., Newman, C.D., Ouni, A., Mkaouer, M.W.: How do developers refactor code to improve code reusability? In: International Conference on Software and Systems Reuse. Springer (2020)
6. Alrubaye, H., Mkaouer, M.W., Ouni, A.: Migrationminer: An automated detection tool of third-party java library migration at the method level. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 414–417. IEEE (2019)
7. Alrubaye, H., Mkaouer, M.W., Ouni, A.: On the use of information retrieval to automate the detection of third-party java library migration at the method level. In: Proceedings of the 27th International Conference on Program Comprehension, pp. 347–357. IEEE Press (2019)
8. Bavota, G., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A.: An empirical study on the developers' perception of software coupling. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 692–701. IEEE Press (2013)
9. Bavota, G., Panichella, S., Tsantalis, N., Di Penta, M., Oliveto, R., Canfora, G.: Recommending refactorings based on team co-maintenance patterns. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp. 337–342. ACM (2014)
10. Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., Garcia, A.: 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 (2017)
11. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on software engineering **20**(6), 476–493 (1994)
12. Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding refactorings via change metrics. In: ACM SIGPLAN Notices, vol. 35, pp. 166–177. ACM (2000)
13. Di, Z., Li, B., Li, Z., Liang, P.: A preliminary investigation of self-admitted refactorings in open source software (S). In: The 30th International Conference on Software Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, July 1-3, 2018. [13], pp. 165–164. DOI 10.18293/SEKE2018-081. URL https://doi.org/10.18293/SEKE2018-081
14. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: European Conference on Object-Oriented Programming, pp. 404–428. Springer (2006)
15. Freund, Y., Schapire, R.E.: Large margin classification using the perceptron algorithm. Machine learning **37**(3), 277–296 (1999)
16. Hattori, L.P., Lanza, M.: On the nature of commits. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. III–63. IEEE Press (2008)
17. Hayashi, S., Tsuda, Y., Saeki, M.: Search-based refactoring detection from source code revisions. IEICE TRANSACTIONS on Information and Systems **93**(4), 754–762 (2010)
18. Herbrich, R., Graepel, T., Campbell, C.: Bayes point machines. Journal of Machine Learning Research **1**(Aug), 245–279 (2001)
19. 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, pp. 99–108. ACM (2008)
20. Howe, N.R., Rath, T.M., Manmatha, R.: Boosted decision trees for word recognition in handwritten document retrieval. In: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 377–383. ACM (2005)
21. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, pp. 163–172. IEEE Computer Society (2011)
22. Kim, M., Gee, M., Loh, A., Rachatasumrit, N.: Ref-finder: a refactoring reconstruction tool based on logic query templates. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, pp. 371–372. ACM (2010)
23. Kowsari, K., Jafari Meimandi, K., Heidarysafa, M., Mendu, S., Barnes, L., Brown, D.: Text classification algorithms: A survey. Information **10**(4), 150 (2019)
24. Mahouachi, R., Kessentini, M., Cinnéide, M.Ó.: Search-based refactoring detection using software metrics variation. In: International Symposium on Search Based Software Engineering, pp. 126–140. Springer (2013)
25. Mansouri, M.M.: Detection of rename local variable refactoring instances in commit history. Ph.D. thesis, Concordia University (2018)
26. Mkaouer, M.W., Kessentini, M., Bechikh, S., Deb, K., Ó Cinnéide, M.: Recommendation system for software refactoring using innovization and interactive dynamic optimization. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp. 331–336. ACM (2014)
27. 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. Empirical Software Engineering **22**(2), 894–927 (2017)
28. Mund, S.: Microsoft azure machine learning. Packt Publishing Ltd (2015)
29. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Transactions on Software Engineering **38**(1), 5–18 (2011)
30. Opdyke, W.F.: Refactoring object-oriented frameworks. University of Illinois at Urbana-Champaign Champaign, IL, USA (1992)
31. Pan, B., Tian, Y., Zhou, T.S., Wang, F., Li, J.S.: Study on image encryption method in clinical data exchange. In: 2015 7th International Conference on Information Technology in Medicine and Education (ITME), pp. 252–255. IEEE (2015)
32. 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, pp. 35–38. ACM (2008)
33. Saif, H., Fernández, M., He, Y., Alani, H.: On stopwords, filtering and data sparsity for sentiment analysis of twitter (2014)
34. Shi, Q., Petterson, J., Dror, G., Langford, J., Smola, A., Vishwanathan, S.: Hash kernels for structured data. Journal of Machine Learning Research **10**(Nov), 2615–2637 (2009)
35. Silva, D., Valente, M.T.: Refdiff: detecting refactorings in version histories. In: Proceedings of the 14th International Conference on Mining Software Repositories, pp. 269–279. IEEE Press (2017)
36. Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making program refactoring safer. IEEE software **27**(4), 52–57 (2010)
37. Soetens, Q.D., Perez, J., Demeyer, S.: An initial investigation into change-based reconstruction of floss-refactorings.

In: 2013 IEEE International Conference on Software Maintenance, pp. 384–387. IEEE (2013)
38. Stroggylos, K., Spinellis, D.: Refactoring–does it improve software quality? In: Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007), pp. 10–10. IEEE (2007)
39. Taneja, K., Dig, D., Xie, T.: Automated detection of api refactorings in libraries. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 377–380. ACM (2007)
40. Thangthumachit, S., Hayashi, S., Saeki, M.: Understanding source code differences by separating refactoring effects. In: 2011 18th Asia-Pacific Software Engineering Conference, pp. 339–347. IEEE (2011)
41. Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A.: Jdeodorant: Identification and removal of type-checking bad smells. In: 2008 12th European Conference on Software Maintenance and Reengineering, pp. 329–331. IEEE (2008)
42. Tsantalis, N., Mansouri, M., Eshkevari, L.M., Mazinanian, D., Dig, D.: Accurate and efficient refactoring detection in commit history (2018)
43. Weinberger, K., Dasgupta, A., Attenberg, J., Langford, J., Smola, A.: Feature hashing for large scale multitask learning. arXiv preprint arXiv:0902.2206 (2009)
44. Weissgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), pp. 231–240. IEEE (2006)
45. Xing, Z., Stroulia, E.: Umldiff: an algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pp. 54–65. ACM (2005)
46. Yang, Y., Pedersen, J.O.: A comparative study on feature selection in text categorization. In: Icml, vol. 97, p. 35 (1997)