Empirical Software Engineering
https://doi.org/10.1007/s10664-020-09871-2

Feature requests-based recommendation of software refactorings

Ally S. Nyamawe[1] · Hui Liu[1] · Nan Niu[2] · Qasim Umer[1] · Zhendong Niu[1]

Published online: 28 August 2020

Abstract
Software requirements are ever-changing which often leads to software evolution. Consequently, throughout software lifetime, developers receive new requirements often expressed as feature requests. To implement the requested features, developers sometimes apply refactorings to make their systems adapt to the new requirements. However, deciding what refactorings to apply is often challenging and there is still lack of automated support to recommend refactorings given a feature request. To this end, we propose a learning-based approach that recommends refactorings based on the history of the previously requested features, applied refactorings, and code smells information. First, the state-of-the-art refactoring detection tools are leveraged to identify the previous refactorings applied to implement the past feature requests. Second, a machine classifier is trained with the history data of the feature requests, code smells, and refactorings applied on the respective commits. Consequently, the machine classifier is used to predict refactorings for new feature requests. The proposed approach is evaluated on the dataset of 55 open source Java projects and the results suggest that it can accurately recommend refactorings (accuracy is up to 83.19%).

Keywords Feature requests · Code smells · Machine learning · Recommendation · Software refactoring

1 Introduction

Requirements change is inevitable as the business, technologies, and stakeholder demands continuously evolve (Jayatilleke et al. 2018). The adaptation to ever-changing software requirements is one of the key factors for the evolution of software systems. During software evolution, developers often receive new requirements expressed as feature requests which demand for the implementation of a new functionality or enhancement of an existing feature. The most common and dominant means to track and manage feature requests is the

Communicated by: Kelly Blincoe, Daniela Damian, and Anna Perini

This article belongs to the Topical Collection: *Requirements Engineering*

✉ Hui Liu
liuhui08@bit.edu.cn

Extended author information available on the last page of the article.

use of issue tracking systems e.g JIRA (2019), Bugzilla (2019), and GitHub Issue Tracker (2019). Through issue tracker, a feature request can be discussed, assigned to a developer, and keep track of its status (Heck and Zaidman 2013). To implement the requested feature, first, developers usually need to locate the source code that should be modified. As a result, several techniques have been proposed to leverage feature requests to locate (e.g., based on requirements traceability and text similarity) and recommend software entities (e.g., API methods) that can be used to implement the requested feature (Thung et al. 2013; Niu et al. 2014; Palomba et al. 2017). Second, developers often apply refactorings on the located source code to make their systems adapt to the new requirements (Niu et al. 2014; Ratzinger et al. 2007). However, deciding what refactorings to apply is often challenging.

Software refactoring is a state-of-the-art practice that has been extensively used to improve software quality by applying changes on internal structure without altering its external behavior (Fowler 1999). Currently, most of the existing refactorings recommendation approaches focus at resolving design flaws in source code commonly known as code smells (Xu et al. 2017; Liu et al. 2013; Ouni et al. 2017). However, the recently conducted case study to investigate the motivation behind refactoring found that refactoring activity is mainly motivated by the changes in the requirements and much less by code smell resolution (Silva et al. 2016). Furthermore, the empirical analysis of refactorings from software repositories found that refactorings are most commonly applied by developers for a specific goal such as implementing a feature or bug fixing, than in the refactoring sessions dedicated for evolving the software design (Soares et al. 2011; Murphy-Hill et al. 2012). This finding was further confirmed by Palomba et al. (2017) in the exploratory study on the relationship between changes and refactoring. Silva et al. (2016) advocate the need for refactorings recommender systems that focus on facilitating maintenance tasks (i.e., implementing feature requests or fixing bugs). However, to the best of our knowledge, there is still lack of automated support to recommend refactorings during the implementation of feature requests.

To this end, in this paper we propose a learning-based approach that recommends refactoring types based on the history data of the previously requested features, applied refactorings, and code smells information. The proposed approach learns from the training dataset associated with a set of applications and can be used to suggest refactoring types for feature requests associated with other applications (or new feature requests associated with the training applications). Our approach involves two classification tasks: first a binary classification that suggests whether refactoring is needed or not for a given feature request, and then a multi-label classification that suggests the type of refactoring. Notably, the proposed approach suggests refactoring types only and it does not point to the locations in the codebase for the recommended refactoring. However, it could be integrated with other approaches/tools that could suggest such locations, and thus makes more complete refactoring suggestions. The proposed approach also helps developers pick up proper refactoring tools by suggesting refactoring types. The past feature requests and their associated commits are retrieved from the corresponding issue trackers and software repositories respectively. Usually, each feature request can be linked to the respective commits that addressed the request through a unique feature request identifier which is often added in the commits' messages. In practice, the previously applied refactorings on such commits can be recovered by using the state-of-the-art refactoring detection tools, e.g., `RefDiff` (Silva and Valente 2017), `RMINER` (Tsantalis et al. 2018), and `Ref-Finder` (Kim et al. 2010). The proposed approach is evaluated on the dataset of 55 open source Java projects altogether consisting of 18,899 feature requests from JIRA issue tracker. The evaluation results suggest that, the proposed approach can accurately recommend refactoring types and attain an accuracy of up to 83.19%.

This paper is an expanded version of our previous conference paper (Nyamawe et al. 2019). Compared to the conference version (Nyamawe et al. 2019), in this paper we make the following expansion:

- We revise the proposed approach to leverage code smells in source code besides feature requests. With such additional information, the proposed approach is improved significantly. The precision and recall for predicting the need for refactoring are improved from 66.36% to 80.75%, and from 85.32% to 93.76%, respectively. In addition, the accuracy of refactorings recommendation has improved from 70.75% to 83.19%.
- We evaluate the proposed approach with larger dataset. The number of involved feature requests has been increased significantly by 41%.
- We investigate additional research question(RQ3) about the impact of code smells on refactorings recommendation.
- We implement our approach on three additional classifiers (i.e., Random Forest, Decision Tree, and Convolutional Neural Network) that were not implemented in our previous paper (Nyamawe et al. 2019).

The rest of the paper is organized as follows. In Section 2 we review the work related with our research. Section 3 presents our recommendation approach. We evaluate the proposed approach and discuss threats to validity of our results in Section 4. We finally conclude our paper and state the future work in Section 5.

2 Related Work

2.1 Refactoring Practice

Software refactoring (Fowler 1999; Mens and Tourwé 2004) has received increased attention from the research community and a lot of approaches have been proposed that strive to improve the overall quality of software systems. Developers often apply refactorings to improve source code maintainability, comprehensibility, and prepare their systems to adapt to new requirements (Xu et al. 2017; Liu et al. 2013). Usually, software refactoring involves identifying refactoring opportunities (e.g., code smells) and selecting proper refactoring operations to alleviate them. However, such task is often challenging especially in large and non-trivial software systems (Bavota et al. 2014a). Consequently, a majority of the effort has been devoted in devising refactoring recommenders that can (semi-) automatically detect code smells, suggest refactoring solutions, and apply them. Such great tools as JMove (Terra et al. 2018), JDeodorant Fokaefs et al. (2007, 2011), and DECOR (Moha et al. 2010), make software refactoring efficient and less error-prone.

Understanding the rationale driving the application of refactorings can be useful in evolving refactoring recommenders that are tailored to the actual needs of software developers (Silva et al. 2016). As a result, several research (Kim et al. 2012; Silva et al. 2016; Palomba et al. 2017) have invested in empirically investigating the developers' motivations behind applying refactorings, which found that refactoring is mainly motivated by the changes in the requirements. However, among the prevalent techniques that have been used in recommending refactorings include source code metrics (Chaparro et al. 2014; Tsantalis and Chatzigeorgiou 2009), search-based and software change history (Kessentini et al. 2017; Tsantalis and Chatzigeorgiou 2011; Ouni et al. 2016; Lin et al. 2016), requirements traceability (Niu et al. 2014; Nyamawe et al. 2018) and machine learning techniques (Xu et al. 2017; Liu et al. 2018; Ratzinger et al. 2007; Pantiuchina et al. 2018). To the best of our knowledge, none of these approaches have attempted to automate refactoring recommendation based on past feature requests. In the following, we briefly highlight some of the state-of-the-art refactorings recommendation techniques.

2.2 Code-Metrics-Based Recommendation of Refactorings

One of the most common ways to recommend refactoring solutions is based on the computation of source code metrics (Chaparro et al. 2014; Tsantalis and Chatzigeorgiou 2009). Such approaches take the assumption that, refactorings that lead to the improved source code metrics (e.g., cohesion and coupling) are the best ones. For example, Bavota et al. (2014b) proposed an approach that recommends extract class refactorings based on the analysis of structural and semantic relationships between methods of a class to identify chains of strongly related methods. Consequently, the chains are used to create new classes with improved cohesion than the original class. Simon et al. (2001) proposed a metrics based refactoring approach to facilitate developers in deciding where to apply which refactoring. They defined a metrical distance measure between the members of a class (i.e., attributes and methods) to identify how they are closely related. Consequently, a defined metric is used to measure cohesion of a class. For the two entities of a class, say x and y, the metric is formalized as:

$$distance(x, y) = 1 - \frac{|p(x) \cap p(y)|}{|p(x) \cup p(y)|} \tag{1}$$

where $p(x)$ and $p(y)$ are the set of properties possessed by x and y respectively. Therefore, entities with low distances are considered more cohesive than those with higher distances. In this case, for instance, a move method refactoring is recommended if a method is closer to the entities in another class than those of the class it is currently in. Similar approaches in recommending refactorings based on code design metrics are also proposed in Fokaefs et al. (2007, 2012) and Terra et al. (2018).

2.3 Search-Based Recommendation of Refactorings

Ouni et al. (2016) proposed a multi-criteria refactoring recommender which suggests an optimal sequence of refactorings that, among other criteria, targets at maximizing consistency with the previously applied refactorings. In their work they contended that, the history of code changes is essential and can increase confidence in recommending new refactorings. To ensure consistency with past refactorings, they defined the following fitness function:

$$Sim\_refactoring\_history(RO) = \sum_{j=1}^{n} e_j \tag{2}$$

where n is the number of previously applied refactorings, and e_j is a refactoring weight that measures the similarity between the recommended refactoring operation (RO) and the past refactoring operation j. A detailed survey of search-based refactorings recommendation approaches is recently conducted in Mohan and Greer (2018).

2.4 Software Change History-Based Recommendation of Refactorings

Kessentini et al. (2017) proposed an approach that recommends refactorings based on the analysis of bug reports and history of change. The assumption of this approach is that, a class which is recently modified or listed in previous bug reports is more likely to demand refactoring. Additionally, the previously applied refactorings were also considered to deduce possible potential refactorings for current release. Tsantalis and Chatzigeorgiou (2011) proposed a tool (Eclipse plugin) that exploits past source code changes to rank refactoring suggestions. The philosophy of this approach is that, a piece of code that has undergone several changes in the past, is more likely to demand refactoring in the future. Consequently, a refactoring involving such code should receive a higher priority. Similar approaches to recommend refactorings based on development history are proposed in Ouni et al. (2015) and Ouni et al. (2013). Generally, these approaches Kessentini et al. (2017), Tsantalis and Chatzigeorgiou (2011), Ouni et al. (2013, 2016, 2015) suggest that past source code change history is useful in recommending new refactorings. In addition, historical data is essential in producing quality code to evolve software systems (Mei and Zhang 2018). However, such approaches do not consider feature requests in their recommendation.

2.5 Requirements-Traceability-Based Recommendation of Refactorings

Our approach is inspired by the earlier work proposed by Niu et al. (2014). The authors in Niu et al. (2014) proposed a traceability-based refactoring recommendation approach to ensure that the requested requirements are fully implemented. Their approach leverages requirements traceability between the requirements under development and the implementing source code to accurately locate where the software should be refactored. To determine what types of refactorings should be applied, the authors developed a new scheme that examines the requirements semantics as they relate to the implementation. Requirements semantics involves manual analysis of requirements action themes which refer to the intended action (e.g., Add, Enhance, Remove) to be taken to implement a feature such as enhancing a quality attribute. Consequently, semantic characterization is leveraged to detect code smells that may hinder the fulfillment of such actions. In summary, this approach proposed a novel scheme that maps requirements action themes and code smells to refactorings. This approach was qualitatively evaluated based on asking opinions from the developers of the involved subject application to rank the recommended refactorings. Generally, the approach scored 3.8 in the 5 point scale which suggests that the recommended refactorings were somewhat appropriate. The key difference of the approach in Niu et al. (2014) with our proposed approach is on how to recommend refactoring solutions. Although both approaches rely on feature requests (i.e., requirements) to recommend refactorings, the proposed approach leverages previous feature requests and their associated applied refactorings to predict refactorings for the implementation of the current feature request. On the other hand, Niu et al. (2014) only work with current feature request as input and recommend refactorings that ensure full implementation of the requested feature. Additionally, the approach in Niu et al. (2014) is based on manual analysis of requirements which is often tedious and error-prone, whereas our work implements an automated recommendation approach.

Nyamawe et al. (2018) proposed an approach that leverages requirements traceability and code metrics to recommend refactoring solutions. The assumption of this approach is that, the traceability between requirements (i.e., use cases) and source code is useful in inferring how code elements relate functionally and how well they should be grouped. Besides traceability, authors employed code metrics (i.e., cohesion and coupling) to establish a tradeoff between traceability and code design improvement. To quantify the quality of traceability, authors used traceability entropy to identify how classes are traced to use cases and vice versa. However, the approaches in Niu et al. (2014) and Nyamawe et al.

(2018) do not consider the history of applied refactorings during their recommendation processes. Additionally, our approach differs from Nyamawe et al. (2018) in that the former is learning-based whereas the latter is not.

In line with facilitating developers during maintenance task especially when implementing feature requests, Thung et al. (2013) proposed an approach to recommend API methods given a feature request. Their approach takes as input the textual description of a new feature request and recommends methods from API library that a developer can use to implement a feature. The proposed approach learns from the training dataset of the past resolved or closed feature requests and changes made to a software system recorded in issue trackers and software repositories respectively. Then, the past similar feature requests are retrieved along with the relevant methods used to implement them. The approach then learns a ranking function and consequently recommends the potential and relevant library methods to the developer. This approach is different from ours in the sense that, based on feature request they recommend API methods to the developer, whereas the proposed approach recommends refactoring solutions.

2.6 Machine Learning-Based Recommendation of Refactorings

Furthermore, several approaches have been proposed to employ machine learning techniques to detect refactoring opportunities and recommend refactorings. For example, Ratzinger et al. (2007) leveraged change history mining to extract features that can be used to predict the need for refactoring in the next two months by using machine learning classifiers. Besides extracting features from evolution data, they also identified the changes applied as either refactoring or not based on the commit messages. However, their prediction models do not distinguish different types of refactorings (e.g., rename class, extract method, etc.). In contrast, our approach explicitly suggests the classes of refactorings required.

Liu et al. (2018) proposed an approach that leverages deep learning techniques to automatically generate labeled training set consisting of methods with or without feature envy. Consequently, such training set is used to train the neural network classifier to predict whether a given method envies another class. Besides that, the classifier also predicts the potential target class where the envy method should be moved to. The structural and textual information were used to decide whether a given method should be moved to another class. Structural information computes how close a method is to the target classes, whereas textual information reveals the semantic relationship between methods and classes. Recently, Pantiuchina et al. (2018) developed a refactoring recommender that targets at preventing the introduction of code smells into the codebase. Their approach relies on machine learning techniques to train the classifier that predicts classes which are likely to be affected by a particular smell in near future. The approach takes a change history of a system with its latest version as an input and deduces the historical trend of 14 predefined class quality metrics. Consequently, they compute regression slope line fitting the values of the metrics for each class that would be used to infer whether the quality is degrading or not. For example, a high positive slope for the WMC (Weighted Methods per Class) metrics indicates that the complexity of a given class is strongly increasing.

In addition, Xu et al. (2017) proposed a machine learning based approach that learns a probabilistic model to recommend *Extract Method* refactorings. The proposed approach extracts structural and functional features from software repositories which encode the concepts of complexity, coupling, and cohesion. Based on these features the approach learns to extract appropriate code fragments from a source of a given method. The proposed approach called GEMS is developed as an Eclipse plug-in for Java programs. Xu et al. (2017) contend that, usually human involvement is required in identifying true refactorings which often leads to the use of small-sized datasets for efficiency. However, to allow working on large datasets and ensure correct recommendations, the deployment of machine learning based approaches is inevitable. Yue et al. (2018) proposed a learning-based approach that recommends code clones (i.e., duplicated code) for refactoring by training the machine learning models with features extracted from refactored and non-refactored clones mined from software repositories. The authors defined 5 categories of key features that characterize clones' content, evolution history, co-evolution relationship, as well as spatial locations of clone peers and syntactic difference between clones. The findings suggest that, history-based features are effective in recommending refactorings than the features extracted from the present version of a software. This leads to a very interesting observation that, when applying refactoring, developers mostly consider the past history than the existing version. These approaches, however, differ from our proposed approach as they do not consider feature requests in their recommendation.

3 Approach

3.1 Overview

As depicted in Fig. 1, the proposed approach follows the following six key steps to predict the need for refactoring and recommend the required refactorings. First, we extract the feature requests from the issue tracker (JIRA) and their respective commits from a software repository (GitHub). Usually, the two artifacts (feature requests and commits) can be linked through a unique feature request identifier. Second, we recover the previously applied refactorings on the retrieved commits. Third, by using the state-of-the-art tool we identify the code smells associated with the source code in each of the retrieved commits. The output from the previous two steps is the file containing feature requests with their associating refactorings and code smells. Fourth, we apply the text preprocessing on the contents of the file to prepare the textual data suitable for the next steps. Fifth, feature modelling is applied to convert the textual data into a numerical representation (feature vectors) for training the classifiers. Finally, we train the machine-learning-based classifier which gives a prediction model for predicting and recommending refactorings for new feature requests. Next, we elaborate each of these key steps in detail.

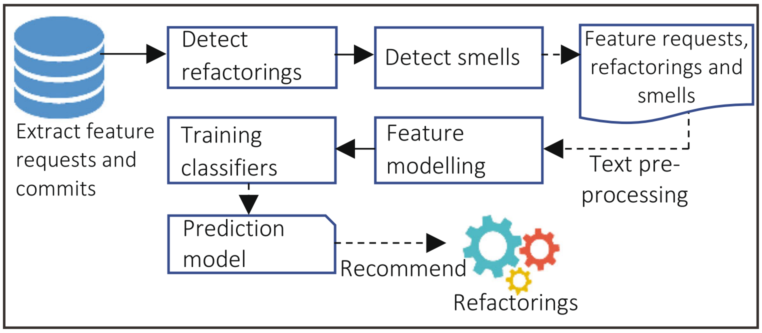


Fig. 1 The framework of the proposed approach

3.2 Feature Requests and Commits Extraction

Software users are usually allowed to request for the new feature or enhancement of the existing feature by submitting a feature request. A feature request often requires some new source code to implement the requirements that cannot be satisfied by the current codebase. Generally, a feature request contains several data fields including: unique request ID, summary, description, resolution, etc. This study is concerned with the data fields which are listed in Table 1.

The feature requests for each of the subject applications that have been addressed to completion (i.e., marked as "Closed" or "Resolved") are retrieved from the issue tracker. The details (ID, Summary, and Description) of the retrieved feature requests which are generally the free-form texts are stored for further processing. Such feature requests were then used to formulate our dataset. The dataset was split into training and testing set. The testing set of feature requests is the one that was used to evaluate the performance of the approach in predicting and recommending refactorings. All feature requests had to undergo the NLP steps (e.g., stop word removal, lemmatization and vector space modelling) prior to training and testing the classifiers. Therefore, once the classifiers are trained with existing feature requests (say original FRs) then it can be readily used to predict and recommend refactorings for the new FRs (after preprocessing and vector space modelling). Next, the repository of each subject application (comprising of several commits) is cloned from Git repository to a local computer by using Eclipse. To speed up the process of detecting refactorings the repositories of the subject applications were first cloned to a local machine rather than being cloned during the refactoring detection process. Next, by using Git bash commands (2018) we retrieve all commits that contain in their commit messages the specified feature request identifiers of the feature requests we retrieved earlier. Finally, at this stage for each feature request fr from the set of all feature requests FR is formalized as:

$$fr = < frID, summ, desc, commitID > \tag{3}$$

where $frID$ represents a unique feature request identifier, $summ$ is the summary or title of a feature request, $desc$ is the detailed description of a feature request and $commitID$ is the unique identifier of a commit used to implement a feature request. The identified commits are then inputted in the next step to detect the applied refactorings. Note that, all commits associated with a given feature request are retrieved in order to identify all types of refactorings applied for such feature request. That means a one-to-many association between feature requests and commits is taken into consideration. In this study we only focus on the cases where the links between the feature requests and commits are known explicitly. Notably, the cases with missing links accounts around 30% of the total links of the applications forming up our dataset. In the future the state-of-the-art techniques can be leveraged to generate missing links. For example, recently, Rath et al. (2018) proposed an

Table 1 Data fields of a feature request

Attribute	Description
ID	A number which uniquely identifies a feature request
Summary	The summary or title of a request
Description	The detailed description of a request
Status	The current status of a request
Resolution	The implementation status of a request

approach that trains a classifier to identify the missing issue tags in commit messages and generate the missing links.

3.3 Detection of Refactorings

To detect refactorings applied in the commits we leveraged the state-of-the-art refactoring detection tools (`RefDiff` and `RMINER`). `RefDiff` is an automated approach introduced by Silva and Valente (2017) to detect refactorings applied between two source code revisions archived in Git repository. The tool uses the combination of heuristics based on static analysis and code similarity to detect 13 common refactoring types. Moreover, `RMINER` is a novel technique recently proposed by Tsantalis et al. (2018) to mine refactorings from software repositories. `RMINER` runs an AST-based statement matching algorithm to detect 15 representative refactoring types. These tools are selected because they can easily be used as Eclipse plugins and they are effective in detecting applied refactorings by comparing subsequent versions of the program. Such tools were recently applied on manually validated refactorings oracle and reported the precision and recall of 98% and 87% for `RMINER` and 76% and 86% for `RefDiff` respectively (Tsantalis et al. 2018). Note that, the results from the two detectors were combined to form a union set. Consequently, the set was manually checked to identify the overlapping cases and eliminate duplication. To identify refactorings, the text file with the list of all commits identified in the previous step is inputted to the refactoring detection tool. For each $commitID$ the tool detects refactorings applied and then outputs the *txt* file with the list of $commitID$ and the associated refactorings. If the inputted $commitID$ is not returned in the output file, then such commit is considered not to have any refactorings. At the end of this step for each feature request fr from the set of all feature requests FR is such that:

$$fr =< frID, summ, desc, commitID, ref > \tag{4}$$

where ref represents the set of refactorings detected in a commit. Note that, if no refactorings are detected then the value of ref is set to null. Therefore, we identify if a given feature request fr_i demands refactorings or not based on the following condition:

$$fr_i = \begin{cases} \text{no refactoring,} & \text{if } ref = \emptyset \\ \text{need refactoring,} & \text{if } ref \neq \emptyset \end{cases} \tag{5}$$

3.4 Detection of Code Smells

Code smells are symptoms of poor design and implementation choices (Fowler 1999) that suggest for possibility of refactoring. For example, one of the classical smells is Long Method (i.e., a method is excessively long and it is doing more than its name suggests). Usually, this smell can be resolved by Extract Method refactoring which extracts parts of the method that can well go together to form a new method (Fowler 1999). To uncover code smells we leveraged an automated code review tool (i.e., Codacy) Codacy (2019) which applies static analysis to identify design issues in software repositories. The tool is powered by PMD (2019) to implement the set of rules which are the basis for identification of design issues in the codebase. Therefore, to determine how code smells associate with a feature implementation, we first establish a link between a feature request and the commit through feature request identifier. This aspect is well described in Section 3.2. Next, by using Codacy, we detect code smells that were introduced by such particular commit. We employed this tool because it supports a variety of languages and it is publicly available for reviewing open source projects. Besides that, the tool can be easily integrated with GitHub repositories and provide quality analysis on the source code files which are to be modified to implement the requested feature. It follows that, for the past feature requests, we identified the list of smells that were fixed in each commit that implemented such feature requests. Thus, a feature request that has been identified as needing refactoring in Formula (4) will further be presented as:

$$fr =< frID, summ, desc, commitID, ref, smells > \quad (6)$$

where $smells$ is the set of code smells related to a commit $commitID$.

The information of code smells is leveraged to further enrich the feature set that could improve the discrimination of feature requests and consequently improve the prediction performance. Therefore, the feature requests' summary $summ$, description $desc$, and code smells $smells$ will serve as the primary source of feature set that will be used to train our classifiers that target to predict refactorings ref. The usage of feature requests as presented in Formula (6) is described in detail in Section 3.7. Furthermore, it is worth noting that, Formula (3), (4), and (6) do not imply that, a given a feature request can only be related to only one commit. But rather, they just show a tuple that contains a feature request and related commit. However, one feature request could have several tuples relating it to different commits.

3.5 Text Pre-processing

To clean and prepare the feature requests for classification we employed text pre-processing. Such process is essential in improving the classification performance (Uysal and Günal 2014). Text preprocessing is purposely used to transform the feature requests (which are written in natural languages) into a form suitable for textual analysis by using Python Natural Language Processing Toolkit (NLTK) (Loper and Bird 2002). The texts which are considered here are those from the summary and description fields of the feature requests. The applied NLP techniques include tokenization, stop word removal, and lemmatization. First, tokenization involves breaking up a document into a lists of individual words (i.e., tokens). In this step some characters such as numbers and punctuation are excluded as they do not contain any useful information. Second, stop word removal is applied, the common and frequently used words such as "a", "an", "the", "in", and "is" are eliminated as they do not carry any useful information and just introduce noise to NLP activities. Finally, lemmatization is applied to convert the words as they appear in the document back into their common base form. This base form is usually referred to as *Lemma*. This process reduces the number of tokens and hence the complexity of NLP activities. In this study we use Porter's stemming (Porter 2006) which implements suffix stripping algorithm for lemmatization. Porter's stemmer has been extensively used in various software engineering studies (Mahmoud and Niu 2014; Palomba et al. 2017).

3.6 Feature Modelling

The classification of feature requests which are free-form texts written in natural language involves some key natural language processing steps which include vector space representation (Runeson et al. 2007; Sun et al. 2010). Therefore, we selected the vector space model for representation of word features extracted from feature requests into numerical representation which is suitable for machine learning. Vector space model is among the mostly used model mainly because of its conceptual simplicity (Manning and Schütze 2001) and it has been widely used in texts clustering, categorization and classification (Thung et al. 2014; Sun et al. 2010; Nizamani et al. 2018). Consequently, in this step, the preprocessed feature requests are converted into a feature vector space model (Manning et al. 2008) which represents the bag of words extracted from feature requests as a vector of weights. The weight of a word represents its importance in a document. To quantify how important a word is to a document in a corpus the term frequency (TF) and inverse document frequency (IDF) are often used. We therefore use TF-IDF to represent features in a feature vector. Suppose in our corpus D we have a term t and a document fr, then Term Frequency $TF(t, fr)$ defines the number of times the term t appears in a document fr, whereas Document Frequency $DF(t, D)$ defines the number of documents in the corpus that contain the term t. Here, a corpus refers to the collection of all feature requests, whereas a document and term refer to a single feature request (fr) and a word (i.e., a token) respectively. Note that, Inverse Document Frequency (IDF) is the reciprocal of the Document Frequency (DF). Therefore, TF-IDF computes the weight w of a term t in a document fr from corpus D as follows:

$$IDF(t, D) = \frac{1}{DF(t, D)} \tag{7}$$

$$w_{\text{t,fr,D}} = TF(t, fr) \times IDF(t, D) \tag{8}$$

The higher the value of the weight, the more important the term is and has higher discriminating power between documents.

3.7 Training and Recommendation

The feature vectors obtained in the previous step are then subject to the classifiers for training and prediction (i.e., recommending refactoring types). The proposed approach leverages six widely-used machine learning classifiers: Logistic Regression (LR), Multinomial Naïve Bayes (MNB), Support Vector Machine (SVM), Decision Tree (DT), Random Forest (RF) and Convolutional Neural Network (CNN). The classifiers have been implemented by a well-known machine learning library based on Python called scikit-learn (Scikit-learn). We have specifically selected such machine learning algorithms for various reasons including the nature of our classification problem and effectiveness of the individual classifier. Usually, text classification is attributed with high dimensional input space which often leads to a lot of features (Chen et al. 2009; Khan et al. 2010). For example, SVM classifiers are effective in managing high dimensional feature space and removing irrelevant features (Khan et al. 2010). Besides that, SVM classifiers are effective in handling sparsity problem and well recognized as accurate text classifiers (Godbole and Sarawagi 2004). Furthermore, Jiang et al. (2013) suggest that, Naïve Bayes classifiers are widely-used to address the classification problem in the domains with large number of attributes such as text classification. Naïve Bayes classifiers are relatively effective, fast and easy to implement (Jiang et al. 2013). Given the assumption of attribute conditional independence of Naïve Bayes classifiers, the parameters for each attribute can be estimated easily and separately which consequently simplifies learning (Jiang et al. 2013). In summary, these classifiers can handle the problem of sparseness and high dimensionality. Such properties of the classifiers apply to our dataset as well which consequently make them optimal for our approach. Generally, the employed classifiers have been widely used and shown to be effective in text classification (Aggarwal and Zhai 2012; Nizamani et al. 2018).

We generally model our text classification problem such that, we have a feature request fr from the feature requests space FR such that $fr \in FR$. Each feature request fr may belong to one or more classes from the set of fixed classes R, where $R = \{r_1, r_2, ..., r_{\text{m}}\}$.

In our case here, classes are also referred to as labels or refactoring types. Suppose we have a training set T of labeled feature requests (fr, r), where $(fr, r) \in FR \times R$. Our goal is to train a classifier (or a classification function) f that maps feature requests to classes (i.e., recommended refactoring types):

$$f : FR \rightarrow R \tag{9}$$

To boost the training and recommendation performance, our approach involves two classification tasks, i.e., binary and multi-label classification. In the following we describe these two tasks in details.

– **Binary classification**, at this first stage the classifier is trained to determine whether refactoring is needed or not. The input to the classifier is all feature requests (noted as $allReq$) and the goal is to classify them into two categories: those that deserve refactoring (noted as $desReq$) and those that do not deserve refactoring (noted as $otherReq$). The training set contains the texts (i.e., summary and description) of the feature requests, code smells information, along with their labels. At this stage, the labels i.e., ref (see Formula (6)) are presented as 1 if refactoring is needed or 0 otherwise. The test data contains feature requests for which we need to predict if they would need refactoring or not. The classifier categorizes a feature request fr into a class c as function f such that:

$$c = f(fr), \quad c \in \{0, 1\}, \quad fr \in FR \tag{10}$$

  where c represents the classification result: 0 implies a feature request fr does not require refactoring whereas 1 implies that refactoring is needed.

  Consequently, the feature requests which are identified to need refactoring will serve as input to the next stage to identify the types of refactorings required.
– **Multi-label classification**, after identifying the feature requests that require refactoring (noted as $desReq$), the multi-label classification is performed to predict the specific types of refactorings which are required. Multi-label classifiers are leveraged because a given feature request may involve more than one type of refactoring. To train the classifier, past feature requests (i.e., those identified to need refactoring), code smells information, and the applied refactorings will serve as input (see Formula (6)). The classifier will then be tested with data containing feature requests and their associating code smells to recommend the types of refactorings that would be needed. The classifier is therefore trained to categorize a feature request fr into a class c as function f such that:

$$c = f(fr), \quad c \subseteq R, \quad fr \in FR \tag{11}$$

  where c is the set of one or more refactoring types.

In summary, the classifiers are generally trained to determine whether refactoring is required or not, if yes then they should predict (i.e., recommend) the types of refactoring required for the unseen feature requests.

4 Evaluation

In this section we present the evaluation of our approach that we refer to as *FR-Refactor* (Feature-Request-based Refactoring). To evaluate the performance of *FR-Refactor* in predicting the need for refactoring and recommending required refactoring types, we compared it against the state-of-the-art approach proposed by Niu et al. (2014). In the following, we first highlight the research questions that this study is investigating. We then describe the dataset used in our experiments. Next, the process and metrics used as the basis of our evaluation are described. Finally, we present and analyse the experimental results and conclude the section by highlighting the threats to validity of our results.

4.1 Research Questions

The evaluation investigates the following research questions:

- **RQ1**: How accurate are different machine learning classifiers in predicting the need for refactoring?
- **RQ2**: How accurate are different machine learning classifiers in recommending required refactoring types?
- **RQ3**: How does the code smells information influence the performance of the classifiers?
- **RQ4**: How accurate is *FR-Refactor* in predicting the need for refactoring compared to the state-of-the-art baseline approach?
- **RQ5**: How accurate is *FR-Refactor* in recommending required refactorings compared to the state-of-the-art baseline approach?
- **RQ6**: Can the proposed approach still obtain good results on applications that are different from those involved in the training?

The research questions **RQ1** and **RQ2** respectively investigate the performance of different machine learning classifiers in predicting the need for refactoring and identifying which refactoring types are required. Furthermore, **RQ3** aims at exploring how the addition of code smells information influences the classification performance of the proposed approach. **RQ4** concerns the performance of *FR-Refactor* in predicting if a given feature request would demand refactoring or not. The research question **RQ5** evaluates the accuracy of *FR-Refactor* in recommending refactoring types. The accuracy of recommendation is essential to determine at what extent the approach suggests useful refactoring types (true positives) to the developers rather than just overloading developers with irrelevant refactoring types (false positives). Finally, **RQ6** investigates the performance of the proposed approach when it is applied to predict and recommend refactoring types to new applications that were not involved in the training.

To answer **RQ4** and **RQ5** we compare *FR-Refactor* with the traceability-enabled approach proposed by Niu et al. (2014) which is based on manual analysis of requirements semantics to recommend refactoring types. Authors performed such analysis by analyzing the description of each feature request to identify terms similar or related to the action (e.g., Add, Enhance, Remove) that has to be taken to implement a feature. Given the identified action themes along with the code smells information, the required refactoring types can be identified based on the novel scheme proposed by Niu et al. (2014). We note that, the baseline approach (Niu et al. 2014) was implemented as a separate approach but was evaluated on the same dataset applied for evaluating our proposed approach. To avoid missing out the feature requests with relevant themes, we also considered the themes (i.e., verbs) in their different forms. For example, Add, Addition and Adding were considered to have the same intention such as adding a functionality. The other key consideration that we took into account is to identify synonyms of each of the key themes. Therefore, a feature request containing a key verb or its synonym were considered to be in the same action theme category. This baseline approach is selected because, to the best of our knowledge, it is the only existing approach which is based on requirements to drive refactoring.

4.2 Dataset

We note that there exist a few publicly available and manually validated datasets of refactorings mined from software repositories (Hegedüs et al. 2018; Silva and Valente 2017; Tsantalis et al. 2018). However, these oracles contain few representative refactorings and feature requests. For example, the evaluation oracles used in Silva and Valente (2017) and Tsantalis et al. (2018) consist of 448 and 3, 188 known refactoring operations respectively. Additionally, not all refactorings in such oracles are associated with the implementation of feature requests. Besides that, the code smells associated with the commits in these oracles are not explicitly specified. Therefore, to attain the reasonable amount of feature requests, code smells and refactorings to effectively train our classifiers, we exploited the state-of-the-art tools to uncover refactorings and code smells in the commits retrieved from software repositories. To recover previously applied refactorings we employed `RefDiff` and `RMINER` tools which were recently validated manually and used in creating an oracle of refactorings proposed by Tsantalis et al. (2018). The tools are publicly available and effective in detecting applied refactorings by comparing subsequent versions of the program. Moreover, the code smells associated with each commit were identified by an automated code review tool (i.e., Codacy) Codacy (2019) which is powered by PMD (2019) and applies static analysis to detect design issues in software repositories. The tool is specifically selected because it is publicly available for open source projects and can be easily integrated with GitHub repository.

Table 2 highlights the distribution of the feature requests of the subject applications used in creating our dataset. First, we extracted the feature requests (Request ID, Summary, and Description fields) of the subject applications from the Apache JIRA issue tracker accessed on (2020). Note that, the feature requests are those that have been addressed to completion (i.e., marked as "Closed" or "Resolved"). Normally, JIRA explicitly links the feature requests to their respective commits in a repository by using unique request/issue identifier. Second, we retrieved the relevant commits from the repository that were used to implement the feature requests retrieved earlier. Our subject applications selection was based on

Table 2 The distribution of feature requests (FR) in our dataset

Project	FR	Project	FR	Project	FR	Project	FR	Project	FR
Accumulo	474	Cayenne	390	Jena	202	Opennlp	392	Stanbol	154
ActiveMQ	753	Curator	24	Kafka	307	PDFBox	482	Storm	302
Ambari	391	Drill	446	Kylin	680	Pig	577	Struts 2	214
Archiva	193	Flink	555	Lens	172	Pivot	185	Synapse	79
Aries	314	Geronimo	38	Maven	359	Ranger	278	Syncope	599
Atlas	70	Giraph	252	Myfaces	407	Sentry	87	Systemml	34
Axis2-Java	204	Gora	31	Nifi	430	Sling	2369	Tajo	444
Beam	295	Groovy	385	Nutch	348	Spring-Datamongo	261	Tapestry-5	381
Bookkeeper	132	Hbase	1389	Ode	102	Spring-Integration	517	Velocity	56
Calcite	74	Impala	178	Oodt	26	Spring-ROO	797	Wicket	237
Carbondata	226	Jclouds	122	Oozie	318	Spring-Security	87	Zookeeper	80
TOTAL:	Projects: 55, Feature Requests: 18, 899								

Java open source projects from GitHub repository whose commits messages explicitly contain the identifier of the feature requests that were addressed. In other words, our selection contains a reliable link between a feature request in an issue tracker and the corresponding commit in a software repository. Third, by using the tools mentioned in the preceding paragraph, we recovered the previously applied refactorings and the respective code smells on the previous versions of the source code in the retrieved commits. Finally, we created a dataset of 55 open source Java projects consisting of $18,899$ feature requests from JIRA issue tracker. Out of $18,899$ feature requests, a total of $9,915$ ($52\% = 9,915/18,899$) feature requests are associated with one or more refactorings, whereas the remaining 48% ($= 8,984/18,899$) of the feature requests do not have any refactorings. The dataset is publicly available on GitHub (https://github.com/nyamawe/FR-Refactor). The subject applications forming up our dataset were selected because they cover a wide range of domains, publicly available, developed by different developers, and have long evolution history. Consequently, it is more likely that they will have a variety of feature requests, code smells, and refactorings.

Furthermore, Table 3 summarizes the distribution of refactoring types in our dataset. Additionally, Fig. 2 presents the distribution of the number of refactoring types (i.e., all labels excluding no-refactoring category) per each data point (i.e., feature request). From the figure we observed that, in our dataset, most ($63\% = 6,221/9,915$) of the feature requests are associated with only one refactoring type, whereas only 37% ($= 3,694/9,915$) are associated with more than one refactoring type. However, such amount is significant which is why we casted the refactoring recommendation problem as multi-label classification problem. We further noted that, there was not feature request that had more than 10 refactoring types.

4.3 Process and Metrics

The evaluation of the proposed approach follows two key steps. First, the classifiers are trained to predict whether the given feature requests would require refactoring or not.

Table 3 The distribution of refactoring types in our dataset

Refactoring types	Number
Extract Interface	192
Extract Method	5,310
Extract Superclass	176
Inline Method	997
Move And Rename Class	357
Move Attribute	1,244
Move Class	975
Move Method	1,884
Pull Up Attribute	248
Pull Up Method	339
Push Down Attribute	136
Push Down Method	159
Rename Class	1,316
Rename Method	3,664
Total	16,997

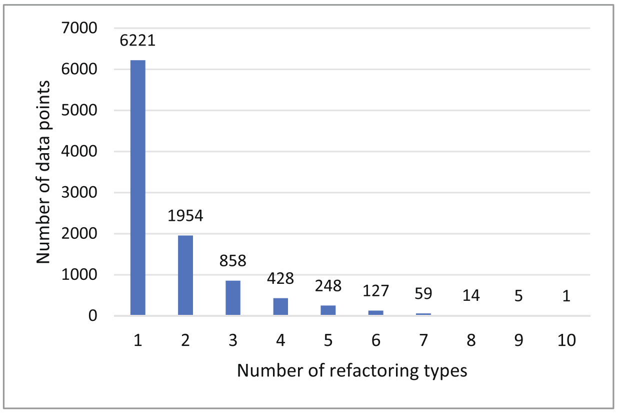


Fig. 2 Number of refactoring types per data points

Second, for the feature requests identified to require refactoring, the classifiers are trained to predict the required refactoring types. To answer the research questions (RQ1–RQ5), we conduct 10-fold cross validation where the dataset is randomly partitioned into ten subsets. On each fold of the evaluation, one subset is employed as testing data whereas others are taken as training data. Furthermore, we cast the problem of predicting the need for refactoring as a binary classification problem. To evaluate the binary classifiers we leveraged the traditional accuracy, precision, recall, and F-measure metrics. Moreover, since in the second step each feature request can be associated with one or more refactoring types, we cast our refactoring recommendation problem as the multi-label classification problem. In multi-label classification each example can be associated with several labels simultaneously, hence its performance evaluation is much more complicated than in the traditional single-label classification (Zhang and Zhou 2014). To evaluate the performance of the multi-label classifiers we use the common and widely-used metrics including hamming loss, hamming score, and subset accuracy (Zhang and Zhou 2014; Godbole and Sarawagi 2004; Schapire and Singer 2000), as defined in the following.

Suppose $FR = \{fr_1, fr_2, ..., fr_m\}$ denotes the feature requests space, and $R = \{r_1, r_2, ..., r_n\}$ denotes the refactoring space with possible n different types of refactorings. The task of multi-label learning is to train a classifier with an evaluation dataset $D = \{(fr_i, r_i)|1 \leq i \leq m\}$, where $r_i \subseteq R$ is the set of refactoring types associated with a feature request fr_i. For any unseen feature request $fr_i \in FR$, the classifier H predicts $H(fr_i) \subseteq R$ denoted as z_i as the set of possible refactoring types for a feature request fr_i. It follows that:

– **Hamming loss**, computes the fraction of labels (refactoring types) incorrectly predicted, i.e., a relevant refactoring is missed or an irrelevant refactoring is predicted. Hamming loss is formally defined as:

$$HammingLoss(H) = \frac{1}{|D|}\sum_{i=1}^{|D|}\frac{|z_i \Delta r_i|}{|R|} \tag{12}$$

where Δ stands for the symmetric difference between the two sets (i.e., the set of predicted refactoring types and the set of true refactoring types for the feature request fr_i). Note that, as the value of the metric closes to 0 the better the classifier's performance.

– **Hamming score**, symmetrically computes how close the set of the predicted refactoring types (z_{i}) is to the true set of refactoring types (r_{i}) for the given feature request fr_{i}. Note that, the larger the value of the metric (with optimal value of 1) the better the classifier's performance. Hamming score can be formally defined as:

$$HammingScore(H) = \frac{|r_{\mathrm{i}} \cap z_{\mathrm{i}}|}{|r_{\mathrm{i}} \cup z_{\mathrm{i}}|} \tag{13}$$

– **Subset accuracy**, measures the fraction of examples classified correctly, i.e., the predicted set of refactoring types (z_{i}) is similar to the true set of refactoring types (r_{i}) for the given feature request fr_{i}. Subset accuracy can be intuitively considered as the traditional accuracy metric (Zhang and Zhou 2014). It is formally defined as:

$$SubsetAccuracy(H) = \frac{1}{|D|}\sum_{i=1}^{|D|}[\![z_{\mathrm{i}} = r_{\mathrm{i}}]\!] \tag{14}$$

where $[\![z_{\mathrm{i}} = r_{\mathrm{i}}]\!]$ returns 1 if the two sets are identical or 0 otherwise. Note that, the larger the value of the metric (with optimal value of 1) the better the classifier's performance.

4.4 RQ1: Performance of Different Classifiers in Identifying the Need for Refactoring

To investigate which classifier will be effective in predicting the need for refactoring when implementing a given feature request, we evaluated the performance of six machine learning algorithms: SVM, MNB, LR, RF, DT, and CNN. Table 4 presents the effectiveness of each classifier in terms of accuracy, precision, recall, and F-measure. Note that, the best recorded result for each metric is highlighted in bold. From the table, it is observed that the accuracy of predicting whether refactoring would be required ranges from 70.44% to 76.01%. The results further suggest that, on average precision (see column 2) of up to 78.12% the need for refactoring can be accurately predicted. Furthermore, the results generally indicate that, on average, MNB and LR classifiers outperform all other classifiers. MNB classifier achieved an F-measure of 86.77% and an accuracy of 76.01%, whereas, LR classifier achieved an F-measure of 82.75% and can accurately predict the need for refactoring with up to 79.37% precision. Furthermore, from the table we can observe that, a convolutional neural network (CNN) classifier has performed slightly lower than the rest of the classifiers. One possible reason for that is, deep learning classifiers generally require significantly large datasets to achieve a competitive performance. We, therefore, conclude the preceding analysis that Multinomial Naïve Bayes (MNB) classifier turned out to be the best classifier in this case. MNB has shown to be effective in binary text classification in various studies including enhancement requests approval prediction (Nizamani et al. 2018) and spam emails detection (Feng et al. 2016). We note that, our binary classification

Table 4 Classifiers' performance for predicting the need for refactoring (%)

Classifier	Accuracy	Precision	Recall	F-measure
SVM	74.08	79.58	86.14	82.73
MNB	**76.01**	**80.75**	**93.76**	**86.77**
LR	74.00	79.37	86.43	82.75
RF	73.78	79.51	82.79	81.12
DT	73.21	75.58	82.35	78.82
CNN	70.44	73.90	83.06	78.21

problem also involves classifying texts which assigns feature request to different classes (i.e., refactoring or non-refactoring) based on the words features present in the document. MNB classifier is the widely used generative classifier that can easily accommodate any domain-specific knowledge and also performs better with hierarchical classification scenario (Aggarwal and Zhai 2012). Since the evaluation results suggest that MNB works best in predicting whether refactoring is required to implement a given feature request (noted as binary classification), MNB would be used in the rest of the paper for the binary classification.

The possible potential challenges that may have hindered higher prediction accuracy include: 1) for the same feature request, developers may propose different solutions (implementation strategies), which has significant influence on the refactoring activities. Consequently, predicting the refactoring while implementation strategies are not specified is natively challenging. 2) Refactoring activities are not indispensable even if they are helpful for the given programming task. Consequently, many factors, e.g., the preference/experiences of the developers, the schedule of the team, and the employed QA measures may help in accurate prediction of refactorings. However, it is challenging to get such information in advance, and thus the prediction of refactoring is challenging. Although precision is relatively low (76.5%), recall is higher (89.37%). Consequently, based on our prediction, developers can apply refactoring tools for suggested feature requests only and thus save significant cost.

4.5 RQ2: Performance of Different Classifiers in Recommending Refactoring Types

As mentioned earlier, the refactoring types recommendation is cast here as a multi-label classification problem, hence Table 5 summarizes the results in terms of subset accuracy, hamming score, and hamming loss which are widely used metrics for evaluating multi-label classifiers. The best recorded result for each metric is presented in bold. Note that, subset accuracy (also called $classification\ accuracy$) returns the percentage of instances where the set of labels (i.e., refactoring types) predicted by the classifier is exactly the same with their corresponding truth set (Gibaja and Ventura 2015). Generally, as shown in Table 5, SVM turned out to be the best performing classifier with the recorded subset accuracy and hamming score of 83.19% and 83.03% respectively. Therefore, this implies that, the needed refactoring types can be accurately recommended with up to 83.19% accuracy. Furthermore, from Table 5, we make the following observations:

– In the case of multi-label classification, SVM classifier has significantly outperformed MNB, RF, and DT classifiers by the difference of 42.66, 19.47, and 22.88 percentage

Table 5 Classifiers' performance for refactoring types recommendation(%)

Classifier	Subset accuracy	Hamming score	Hamming loss
SVM	**83.19**	**83.03**	**0.031**
MNB	40.53	39.92	0.059
LR	81.88	81.26	0.033
RF	63.72	63.84	0.041
DT	60.31	59.99	0.040
CNN	75.44	75.81	0.047

point on subset accuracy respectively. On the other hand, SVM has only outperformed LR marginally in both metrics.

– Compared to other classifiers, SVM achieved the lowest value (i.e., 0.031) on hamming loss which identifies to what extent the classifier predicts the irrelevant refactoring types and omits relevant refactoring types. This metric is normalized between 0 and 1. As the value of the metric closes to 0, it indicates better performance of the classification.

SVM often performs better due to the following reasons. First, high dimensional input space texts produce a lot of features which consequently lead to a very large feature space. SVM employs overfitting protection and has the ability to learn which can be independent from the dimensionality of the feature space. Second, in text classification, sometimes documents are represented by some amount of features which could be irrelevant, therefore a classifier should be able to combine several features (i.e., dense concept). Third, sparsity of document vectors, usually each document contains a document vector with a lot of entries which are zeros. SVM based classifiers have shown to be effective in handling problems with sparse instances and dense concepts. Based on these factors, SVM is shown to be effective for text classification and well recognized to be accurate (Godbole and Sarawagi 2004). Since the preceding results analysis suggests that SVM works best in suggesting refactoring classes (noted as multi-label classification), SVM would be used in the rest of the paper for the multi-label classification.

Furthermore, we investigated the performance of SVM in recommending individual refactoring types. As depicted in Table 6, generally the results suggest that the classifier achieves an average precision of 76% in recommending refactoring types. We further observed that, the classifier achieved the best results (72%) in terms of F-measure for the Extract Method refactoring. The possible reason for this best performance comparing to the F-measure results recorded for other refactorings could be the amount of such refactorings in the dataset. If we refer to Table 3, Extract Method refactoring is the one with the most (31%) representative examples in the dataset. This is due to the fact that, in practice,

Table 6 Results for individual refactoring types recommendation

Refactoring type	Precision	Recall	F-measure
Extract Interface	0.63	0.61	0.62
Extract Method	0.68	0.77	0.72
Extract Superclass	0.85	0.41	0.55
Inline Method	0.84	0.39	0.53
Move And Rename Class	0.89	0.54	0.67
Move Attribute	0.86	0.37	0.52
Move Class	0.78	0.43	0.55
Move Method	0.85	0.47	0.67
Pull Up Attribute	0.69	0.60	0.64
Pull Up Method	0.67	0.40	0.50
Push Down Attribute	0.81	0.36	0.50
Push Down Method	0.83	0.35	0.49
Rename Class	0.75	0.42	0.54
Rename Method	0.69	0.55	0.61
Weighted avg	0.76	0.54	0.61

certain refactoring types are more prevalent than the others. For example, the recently conducted empirical study by Silva et al. (2016) on the refactorings performed by developers on 539 commits of 185 java projects retrieved from GitHub found that, Extract Method is the most popular high-level refactoring. On the other hand, in some few cases (e.g., Push Down Method) the classifier achieved a recall and F-measure below 50%. This can be justified by the fact that some of the refactoring types, including Push Down Method and Push Down Attribute have very few instances in the dataset (see Table 3) which can consequently affect their prediction. The same study by Silva et al. (2016) also found that Push Down refactorings were among the least popular refactorings. It is therefore challenging to create a dataset with sufficient number of refactorings and yet balance the number of representative examples for each refactoring type. In view of that, we implemented the classifier which is based on the distribution of refactorings in our dataset. The classifier predicts a given feature request fr will deserve a refactoring type r at a chance of $fr(r)$ such that:

$$fr(r) = \frac{|FR(r)|}{|allFR|} \tag{15}$$

where, $FR(r)$ are all feature requests that are associated with refactoring r in the training set, and $allFR$ are all feature requests in the training set.

Ideally, the classifier is assessed based on the accuracy of recommending individual refactoring types. We established the chances for a given feature request to deserve certain refactoring based on Formula (15). Consequently, from the training set we compute the number of instances that were correctly and incorrectly classified for each individual refactoring types. On average, the classifier recorded the precision, recall and F-measure of 79%, 63%, and 70% respectively for recommending individual refactoring types. Such performance is slightly higher than that reported earlier in Table 6 where a classifier (based on SVM) attained an average precision, recall and F-measure of 76%, 54%, and 61% respectively. The preceding analysis suggests that, the proposed approach is accurate in recommending refactoring types.

It is worth noting that, in general binary classification should be more accurate than multi-label classification if they are applied to the same dataset. However, in our case, they are applied to different datasets, and thus there is no obvious relation between their performances. We also note that, the multi-label classification achieves higher accuracy than the binary classification. That is because of the difference in their testing data. As mentioned earlier in Section 3.7, we first apply the binary classifier to all of the feature requests (noted as $allReq$), and such feature requests are classified into two categories: those that deserve refactoring (noted as $desReq$) and those that do not deserve refactoring (noted as $otherReq$). The multi-label classifier is then applied to the $desReq$ only, and $otherReq$ is not involved.

4.6 RQ3: The Influence of Code Smells Information

This evaluation explores the influence of code smells information on the classifiers' performance in predicting refactoring and recommending the required refactoring types. In contrast to other binary and multi-label classification experiments reported in this study, in this case we trained our classifiers with only feature requests and refactoring types. That means we excluded code smells information. Consequently, the classifiers were tested to predict refactoring and recommend refactoring types for new feature requests. The results for predicting refactoring and recommending refactoring types are respectively reported

Table 7 Classifiers' performance for predicting the need for refactoring (without code smells) (%)

Classifier	Accuracy	Precision	Recall	F-measure
SVM	71.96	77.09	80.63	78.82
MNB	73.10	76.51	89.37	82.44
LR	72.17	77.74	80.70	79.19
RF	69.49	77.82	78.77	78.29
DT	68.23	76.68	78.01	77.34
CNN	64.36	69.92	78.45	73.94

in Tables 7 and 8. Note that, the results reported in these tables are based on the previous approach proposed in Nyamawe et al. (2019) which do not consider code smells in its recommendation. Consequently, to analyze the influence of code smells on the classifiers' performance we compare the results obtained by the previous approach (Nyamawe et al. 2019) reported in Tables 7 and 8 (noted as without code smells) with that obtained by our proposed approach reported earlier in Tables 4 and 5 (noted as with code smells) respectively.

To investigate the influence of code smells information on classifiers' performance in predicting the need for refactoring we compare the results reported in Tables 4 and 7. We note that, this comparison includes two more additional machine learning classifiers (i.e., Decision Tree and Neural Network) that were not implemented by Nyamawe et al. (2019) for binary classification. To clearly visualize the change in performance, we graphically depict such comparison in Fig. 3 in terms of accuracy. From the figure we observe that, code smells information slightly increased the performance of the classifiers. The noticeable increase in performance of around 5 percentage point is recorded for RF, DT, and CNN classifiers. Such a slight change can be justified by the fact that, it is not necessarily that only feature requests that associate with code smells will demand refactoring.

Furthermore, to visualize the influence of code smells on refactoring recommendation we compare the results reported in Tables 5 and 8. Also note that, this comparison includes three more additional machine learning classifiers (i.e., Random Forest, Decision Tree and Neural Network) which were not implemented by Nyamawe et al. (2019) for multi-label classification. For clarity, the graphical representation of that comparison is depicted in Fig. 4 in terms of subset accuracy. From the figure we observe that, code smells information positively influenced the performance of the classifiers. For example, SVM classifier recorded a performance increase of 14 ($= 83.2\% - 69.2\%$) percentage point when including

Table 8 Classifiers' performance for refactoring types recommendation (without code smells) (%)

Classifier	Subset accuracy	Hamming score	Hamming loss
SVM	69.21	70.46	0.026
MNB	24.73	24.79	0.059
LR	69.21	69.93	0.026
RF	27.55	27.63	0.036
DT	22.14	22.16	0.039
CNN	66.53	66.97	0.042

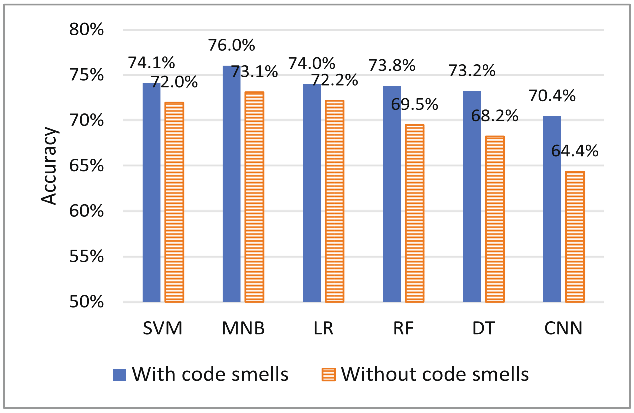


Fig. 3 The influence of code smells information on refactoring prediction performance

code smells information as input to the classifier along with feature requests and refactoring types during training. Such trend can also be observed in the rest of the classifiers where MNB, LR, RF, DT, and CNN respectively recorded the change of 15.8, 12.7, 36.1, 38.2, and 8.9 percentage point by including code smells information. Notably, only 58% of the feature requests and refactorings in our dataset were associated with code smells. This implies therefore that, the refactorings recommendation is somewhat not biased. Therefore, the possible reason for such improvement is that, code smells have been proven useful in accessing (e.g., based on smells catalogs) the special kinds of software refactorings required to alleviate them (Vidal et al. 2014). Consequently, understanding the specific smell type along with a given feature request can lead to accurate prediction of a required refactoring type. That is because, often a given code smell could have multiple refactoring solutions (Fokaefs et al. 2011; Niu et al. 2014). We, therefore, conclude the preceding analysis that inclusion of code smells information boosts the accuracy of refactoring types recommendation.

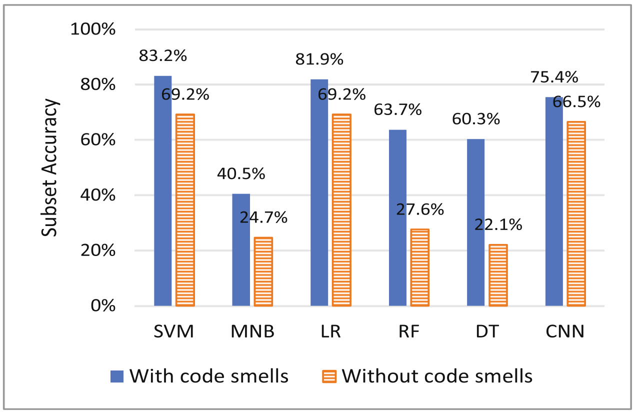


Fig. 4 The influence of code smells information on refactoring types recommendation performance

4.7 RQ4: FR-Refactor vs State-of-the-Art Approach in Predicting the Need for Refactoring

To investigate the performance of *FR-Refactor* in predicting the need for refactoring, we compared it (based on MNB classifier) with the state-of-the-art traceability-enabled approach proposed by Niu et al. (2014). The results of the comparison are depicted in Fig. 5. Generally, results suggest that *FR-Refactor* (based on MNB classifier) significantly outperforms the state-of-the-art approach. From the figure we observe that, *FR-Refactor* significantly achieved an increase in F-measure of 43.3 (82.4% − 39.1%) percentage point. Moreover, *FR-Refactor* achieves better performance in terms of recall (89.4%) because of its ability to learn from past feature requests and predict accordingly, compared to the state-of-the-art approach which attains lower recall (28.5%). In addition to that, *FR-Refactor* improves accuracy and precision by 27.6 (= 73.1% − 45.5%) percentage point and 12.1 (= 76.5% − 64.4%) percentage point respectively. The results lead us to the conclusion that, *FR-Refactor* can accurately predict the need for refactoring. As the baseline approach may fail to explicitly identify the requirements semantics and consequently predict the need for refactoring, *FR-Refactor* leverages past history to predict the need for refactoring of a new feature request. For example, the baseline approach failed to identify if a feature request `CAY-1350:` "*Implement memorized sorting of modeler columns*" will need refactoring. However, *FR-Refactor* predicted the need for refactoring for such feature request. This feature request suggests for additional functionality that would allow users to sort items in the table and their preferences should be memorized. Analysis on the history data revealed that, the feature request `CAY-1350` for the additional functionality is similar to some past features (e.g., `CAY-1251:`"*Memorize user-selected column widths in preferences*") and the later feature request (`CAY-1350`) is an improvement request related to `CAY-1251`. Moreover, the two feature requests were implemented in two different commits but the same type of refactoring (i.e., Extract Method) was applied.

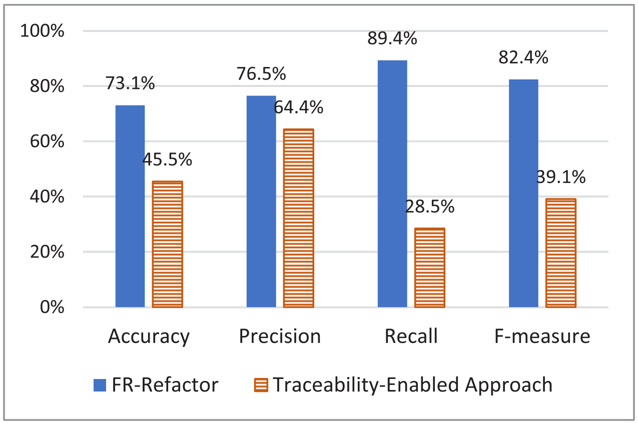


Fig. 5 Performance in predicting the necessity of refactoring

4.8 RQ5: FR-Refactor vs State-of-the-Art Approach in Recommending Refactoring Types

To answer research question **RQ5** we compared the performance of *FR-Refactor* (based on SVM classifier) and the state-of-the-art traceability-enabled approach (Niu et al. 2014) in recommending refactoring types. Evaluation results are presented in Fig. 6. From the figure we make the following observations:

– First, *FR-Refactor* can accurately recommend relevant refactoring types for most of the feature requests. Its average precision is up to 76%. Compared to the state-of-the-art baseline approach, it improves precision significantly by 56 (= 76% − 20%) percentage point.
– Second, *FR-Refactor* significantly outperforms the state-of-the-art in terms of F-measure. It achieved an average F-measure of 61%. This is equivalent to an improvement of 36 (= 61% − 25%) percentage point compared to the baseline approach which recorded an F-measure of 25%.
– Third, the recall (54%) of *FR-Refactor* is significantly higher than that of the baseline approach (34%).

Based on the preceding analysis, we conclude that *FR-Refactor* achieves better results than the baseline approach. One of the possible reasons for the underperformance of the baseline approach could be due to the fact that it relies on manual analysis of predefined requirements semantics. Additionally, it also ignores the possibility that a given feature request may belong to different categories of requirements semantics. Consequently, that may lead to miss out some relevant refactoring types. For example, consider the following part of a feature request. "*Simplify SDK API interfaces. Current SDK API interfaces are not simpler and don't follow builder pattern. If new features are added, it will become more complex*". This feature request suggests for enhancing the quality attributes that will lead to the reduction of interface complexity. One way of getting rid of complexity is to allow for separation of concerns and reduce unneeded code. *FR-Refactor* recommended the following refactoring types. First, a *Rename Method* and *Extract Method* refactoring types. The later refactoring is recommended to decompose long and complex methods. Second, an *Inline Method* refactoring for the calls instances of unneeded methods. Ideally, *Inline Method*

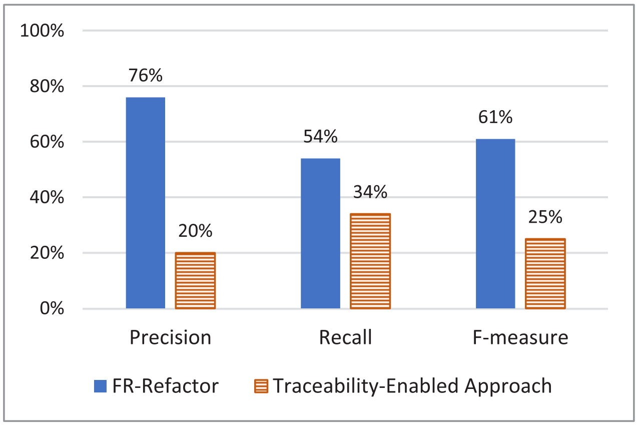


Fig. 6 Performance in refactoring types recommendation

refactoring involves replacing calls to the method with its content and delete the method itself. On the other hand, the baseline approach (based on its scheme) only recommended *Substitute Algorithm* refactoring to alleviate *Long Method* flaw which is considered to cause complexity. Furthermore, consider the following part of another feature request. "*Improve broadcast table cache. Currently, broadcast implementation keep a tuples on scan operator and It creates a duplicated table cache in memory*". This feature request highlights the problem of duplicated feature. The proposed refactoring types are *Extract Method*, *Move Method*, and *Rename Class*. The *Extract Method* refactoring is applied to remove code duplication, whereas *Move Method* refactoring to move a method to class which is more functionally related to it. In addition, *Rename Class* is applied to rename the class from which a method was removed to properly reflect its responsibility. In this case, the baseline approach failed to explicitly uncover the requirement's action theme of such feature request and hence was unable to identify the required refactoring types.

4.9 RQ6: Cross Project Evaluation

This evaluation aims to explore the effectiveness of the proposed approach in predicting and recommending refactoring solutions for the feature requests that their applications were not involved in the training of classifiers. Basically, there might exist a case where some applications have not accumulated enough history, hence it is crucial to understand if the history of other applications can be useful to recommend refactorings for such applications with insufficient history. Note that, in the preceding experiments we take all the dataset from different subject applications as a whole, and conduct 10-fold evaluation on the resulting dataset. To answer **RQ6**, in this section we conduct a cross project evaluation. We use 50 subject applications for the evaluation, and divide them into 10 groups (each contains 5 applications). On the 10 groups of data, we conduct 10-fold evaluation where testing projects are different from training projects. Evaluation results suggest that the proposed approach performs well even if the testing applications are different from training applications. Its precision, recall, and F-measure on binary classification are 72.86%, 81.92%, and 77.12%, respectively. Whereas, its precision, recall, and F-measure on multi-label classification are 81.54%, 59.37%, and 68.71%, respectively. Comparing such results against those reported in Sections 4.7 and 4.8, we conclude that the proposed approach still obtains good results when testing projects are different from training projects.

One possible reason for the success is that the classifiers can learn some generic rules applicable to different applications, e.g., the presence of special code smells may suggest a specific type of refactoring actions, and some special words or phrases in feature requests may suggest the necessity of refactorings. To investigate the latter, we slightly adopted the comprehensive list of patterns which are potential in inferring refactoring-related activities proposed recently by AlOmar et al. (2019). We note that, such patterns were drawn from commit messages and shown to have significant correlation with refactoring-related changes applied on their associated commits. Consequently, from the list of patterns we identified 24 unique and representative keywords which we used to evaluate the feature requests. For each keyword w we computed how likely (denoted as $RefactoringRate(w)$) the feature requests containing such keyword can be associated with refactoring. $RefactoringRate(w)$ can be formalized as follows:

$$RefactoringRate(w) = \frac{|refactoredFR(w)|}{|allFR(w)|} \tag{16}$$

where, $refactoredFR(w)$ are feature requests that contain the keyword w and deserve refactoring, and $allFR(w)$ are all feature requests that contain the keyword w.

For clarity, we only list the top ten keywords with the highest $RefactoringRate$ in Table 9. Other leveraged keywords not listed include: $Replace$, $Improve$, $Extract$, $Reduce$, $Move$, $Change$, $Import$, $Enhance$, $Modify$, $Remove$, $Better$, $Avoid$, Add, and Fix. The $RefactoringRate$ of these keywords ranges from 54% to 65%. We noted that, the feature requests with such keywords tend to have more significant refactoring rate than those without. Generally, the keywords clearly indicate the intention of a feature request such as improving or optimizing a quality attribute. Additionally, we observed that, certain feature requests explicitly mention the word *refactor* to infer the need for improving a specified functionality. For example, the description of the feature request with identifier $CARBONDATA - 1838$ reads as "*Refactor and optimize 'SortRowStepUtil' to make it efficient and more readable*". Furthermore, it is worth noting that, feature requests are often stated in solution-space terms contrary to the classical requirements (Alspaugh and Scacchi 2013; Niu et al. 2014). Consequently, that facilitates the mapping between feature requests and refactorings. The classifiers can therefore learn the combination of such terms as features from the feature requests and use them as the basis for prediction. We notice that the mentioned keywords are generic terms that may appear in any feature request independent of the application domains.

To further investigate **RQ6**, we train a decision tree (DT) based classifier and investigate the rules that the decision tree learns. We used keywords from feature requests whose $RefactoringRate$ is not less than 50% and code smells associated with the feature requests. A given feature request fr is represented as two vectors, K_fr and S_fr. The first vector is such that, $K_fr = \{v_1, v_2, ..., v_n\}$, where $v_i \in \{0, 1\}$ represents whether a keyword k_i appears in a feature request fr or not. The second vector, $S_fr = \{s_1, s_2, ..., s_m\}$, where s_k represents whether the kth code smell appears in the associated source code. The value of S_fr is 0 if a feature request fr does not associate to any code smell. Table 10 presents the performance of the decision tree classifier. As it can be observed from the table, on average, the classifier can accurately predict the need for refactoring on up to 68% precision.

For space limitation, the learned rules and the resulting full decision tree are available as online appendix at https://github.com/nyamawe/FR-Refactor. A part of the tree (learned rules) is presented in Fig. 7. An example of the learned rules is presented in the first top block of Fig. 7. This rule (noted as $Rule1$) suggests that **if** a feature request does not

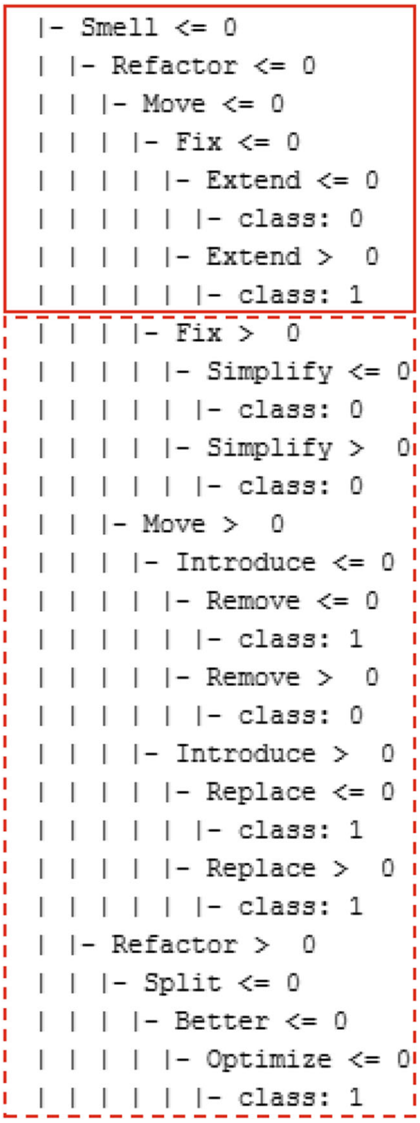
Table 9 Top-10 Keywords with highest refactoring rate

Keyword	Refactoring rate (%)
Refactor	83
Restructure	80
Rewrite	78
Rename	76
Introduce	75
Simplify	72
Extend	70
Optimize	67
Split	66
Cleanup	66

Table 10 Performance of Decision Tree classifier

Class	Precision	Recall	F-measure
Non-refactoring	0.75	0.71	0.73
Refactoring	0.55	0.67	0.58
Weighted avg.	0.68	0.67	0.67

associate with any smell and does not contain the keywords $Refactor$, $Move$, Fix, $Extend$, **then** such a feature request does not need any refactoring. However, **if** it contains the keyword $Extend$ **then** it needs refactoring. Notably, the keywords involved in $Rule1$,



```
|- Smell <= 0
| |- Refactor <= 0
| | |- Move <= 0
| | | |- Fix <= 0
| | | | |- Extend <= 0
| | | | | |- class: 0
| | | | |- Extend >  0
| | | | | |- class: 1
| | | |- Fix >  0
| | | | |- Simplify <= 0
| | | | | |- class: 0
| | | | |- Simplify >  0
| | | | | |- class: 0
| | |- Move >  0
| | | |- Introduce <= 0
| | | | |- Remove <= 0
| | | | | |- class: 1
| | | | |- Remove >  0
| | | | | |- class: 0
| | | |- Introduce >  0
| | | | |- Replace <= 0
| | | | | |- class: 1
| | | | |- Replace >  0
| | | | | |- class: 1
| |- Refactor >  0
| | |- Split <= 0
| | | |- Better <= 0
| | | | |- Optimize <= 0
| | | | | |- class: 1
```

Fig. 7 Part of the learned rules

e.g., $Refactor$, $Move$, Fix, $Extend$, are generic terms (not application specific). Consequently, the learned decision rule $Rule1$ could be applied to feature requests of different applications.

4.10 Threats to Validity

First, the threat to external validity is concerned with the generalizability of the proposed approach. To address this threat, in this study we have considered several feature requests from different 55 Java open source projects and 14 common refactoring types. To further reduce this threat, we also conducted a cross-project validation where the projects involved in the training set were different from those in the testing set. This is the standard practice to evaluate how the proposed approach would perform on an independent dataset. In future we plan to work on more feature requests and leverage other refactorings detection tools to enhance the recommendation of a wide range of refactoring types.

Second, the internal threat may stem from the classification models that we leveraged in our approach. The classifiers have been implemented by the well-known Python-based library for machine learning called scikit-learn. To reduce the threat, the implementation of the models and the classification results were carefully examined, however there could be some errors slipped in unnoticed.

Finally, a threat to construct validity is related to the implementation of the approach. The major threat relates to the correctness of the recovered refactorings and code smells that constituted our dataset. That is because the leveraged detection tools are not 100% on both recall and precision. The inaccuracy of the dataset could be accelerated by the fact that the detection tools may be unable to identify all of the past applied refactorings and the respective smells. Moreover, the tools may suggest irrelevant refactorings or smells (false positives) and may miss out some true refactorings or smells (false negatives). Consequently, that may threaten the accuracy of our dataset. However, in the future, the more improved tools after applying necessary fix as suggested in Tan and Bockisch (2019) can be leveraged to boost the accuracy of refactorings detection. In addition, another threat may stem from the possibility that a commit could have some additional refactorings that are not related to the implementation of feature requests. To reduce such threats we checked the dataset for possible errors, but still there could be some unnoticed errors. That would be due to the lack of the systems knowledge as the process did not involve original developers. Finding original developers is challenging considering the number of the subject applications we used and some of them have long evolution history.

5 Conclusion and Future Work

Empirical investigations on developers' motivations behind applying refactorings on their software systems suggest that, refactoring is mostly motivated by the changes in the requirements. However, most of the existing refactorings recommenders solely focus on identifying refactorings opportunities for the sake of resolving code smells and ignore other refactoring motivations such as adding or extending features in software systems. To implement the requested features, developers sometimes apply refactorings to prepare for new adaptation that accommodates the new requirements. However, it is often challenging to determine which types of refactorings should be applied. Consequently, in this paper we propose a learning-based approach that recommends refactoring types based on the past history of feature requests, code smells information, and the applied refactorings on the respective commits. The proposed approach learns from the training dataset associated with a set of applications and can be used to suggest refactoring types for feature requests associated with other applications or that associated with the training applications. To demonstrate the efficacy of our approach, we conducted experiments on the dataset of 55 open source Java projects consisting of 18, 899 feature requests retrieved from JIRA issue tracker and their associated refactorings recovered by using the state-of-the-art refactoring detection tools. Experimental results suggest that our approach significantly improves over the state-of-the-art approach. Evaluation analysis on two tasks (i.e., predicting the need for refactoring and recommending refactoring types) indicates that our approach attains the accuracy of up to 76.01% and 83.19% respectively.

Although the proposed approach suggests refactoring types only and does not point to the classes involved in the refactoring, the proposed approach is helpful to implement feature request. To carry out software refactoring, we should know both 'what' (refactoring types) and 'where' (where the refactoring should be applied). Consequently, the baseline approach proposed by Niu et al. (2014) suggests both 'where' and 'what'. However, the proposed approach suggests 'what' only. One of its potential practical usefulness is to replace/improve the second part of the baseline approach (Niu et al. 2014) (that suggests refactoring types) because the evaluation results suggest that it is more accurate than the baseline approach in suggesting refactoring types. The two approaches working together could suggest both 'what' and 'where' to developers. The proposed approach may also be integrated with other approaches/tools that could suggest 'where'. It is interesting in future to investigate how change impact analysis approaches (Angerer et al. 2019) may help to identify which specific source code entity (e.g., class or method) should be refactored. Another potential practical usefulness of the proposed approach is to help developers pick up proper refactoring tools. Existing study (Liu et al. 2012) suggests that it is often up to developers to pick up the proper tools to identify different classes of refactoring opportunities, e.g., *GEMS* (Xu et al. 2017) for *extract method* refactoring opportunities, and *JMove* (Terra et al. 2018) for *move method* refactoring opportunities. Suggesting refactoring types (by the proposed approach) may significantly facilitate the selection. The third potential practical usefulness of the proposed approach is to prioritize the implementation of different features. Because different categories of refactorings may interfere with each other (Liu et al. 2012), knowing the required refactoring types of different features help in deciding the order of refactoring types and hence deciding the order of implementing features.

Our future work includes the following. First, we would like to conduct a qualitative evaluation that involves the actual application of the approach by developers to further reveal its usefulness and applicability. Second, it would be interesting to investigate how to accurately locate where recommended refactoring types should be applied by exploiting the advancement of requirements traceability. Finally, our results encourage further investigation that would improve our approach, for example, leveraging more advanced techniques such as deep learning algorithms.

Acknowledgements The authors would like to say thanks to the anonymous reviewers of this manuscript as well as to the program chairs and the anonymous reviewers of RE'19 conference for their insightful comments and constructive suggestions. This work is partially supported by the National Natural Science Foundation of China (No. 61772071 and No. 61690205), and the U.S. National Science Foundation Award (CCF-1350487).

References

Aggarwal CC, Zhai C (2012) A survey of text classification algorithms. In: Zhai C (ed) Aggarwal CC. Springer, Mining text data, pp 163–222

AlOmar EA, Mkaouer MW, Ouni A (2019) Can refactoring be self-affirmed?: An exploratory study on how developers document their refactoring activities in commit messages. In: Tsantalis N, Cai Y, Demeyer S (eds) Proceedings of the 3rd international workshop on refactoring, IWOR@ICSE 2019, Montreal, QC, Canada, May 28, 2019. IEEE/ACM, pp 51–58

Alspaugh TA, Scacchi W (2013) Ongoing software development without classical requirements. In: 21st IEEE International requirements engineering conference, RE 2013, Rio de Janeiro-RJ, Brazil, July 15–19, 2013. IEEE Computer Society, pp 165–174

Angerer F, Grimmer A, Prähofer H, Grünbacher P (2019) Change impact analysis for maintenance and evolution of variable software systems. Autom Softw Eng 26(2):417–461

Apache Projects Issues (2020) (Retrived on 12th March 2020). In: https://issues.apache.org/jira/projects/

Bavota G, Lucia AD, Marcus A, Oliveto R (2014) Recommending refactoring operations in large software systems. In: Robillard MP, Maalej W, Walker RJ, Zimmermann T (eds) Recommendation systems in software engineering. Springer, pp 387–419

Bavota G, Lucia AD, Marcus A, Oliveto R (2014) Automating extract class refactoring: an improved method and its evaluation. Empir Softw Eng 19(6):1617–1664

Bugzilla (2019) (Accessed October 2019) https://www.bugzilla.org/

Chaparro O, Bavota G, Marcus A, Penta MD (2014) On the impact of refactoring operations on code quality metrics. In: 30th IEEE International conference on software maintenance and evolution, Victoria, BC, Canada, September 29–October 3, 2014, pp 456–460

Chen J, Huang H, Tian S, Qu Y (2009) Feature selection for text classification with naïve bayes. Expert Syst Appl 36(3):5432–5435

Codacy (2019) (Accessed October 2019) https://github.com/marketplace/codacy

Feng W, Sun J, Zhang L, Cao C, Yang Q (2016) A support vector machine based naive bayes algorithm for spam filtering. In: 35th IEEE international performance computing and communications conference, IPCCC 2016, Las Vegas, NV, USA, December 9–11, 2016, pp 1–8

Fokaefs M, Tsantalis N, Chatzigeorgiou A (2007) Jdeodorant: Identification and removal of feature envy bad smells. In: ICSM 2007. IEEE International conference on software maintenance 2007. IEEE, pp 519–520

Fokaefs M, Tsantalis N, Stroulia E, Chatzigeorgiou A (2011) Jdeodorant: identification and application of extract class refactorings. In: Proceedings of the 33rd international conference on software engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21–28, 2011, pp 1037–1039

Fokaefs M, Tsantalis N, Stroulia E, Chatzigeorgiou A (2012) Identification and application of extract class refactorings in object-oriented systems. J Syst Softw 85(10):2241–2260

Fowler M (1999) Refactoring—improving the design of existing code. Addison Wesley object technology series. Addison-Wesley

Gibaja E, Ventura S (2015) A tutorial on multilabel learning. ACM Comput Surv 47(3):52:1–52:38

Git Bash Commands (2018) (Retrived on 28th November 2018). In: https://www.atlassian.com/git

GitHub (2019) (Accessed October 2019) https://github.com/features

Godbole S, Sarawagi S (2004) Discriminative methods for multi-labeled classification. In: Dai H, Srikant R, Zhang C (eds) Advances in knowledge discovery and data mining, 8th Pacific-Asia Conference, PAKDD 2004, Sydney, Australia, May 26-28, 2004, Proceedings, Springer, Lecture Notes in Computer Science, vol 3056, pp 22–30

Heck P, Zaidman A (2013) An analysis of requirements evolution in open source projects: recommendations for issue trackers. In: 13th International workshop on principles of software evolution, IWPSE 2013, Proceedings, August 19–20, 2013, Saint Petersburg, pp 43–52

Hegedüs P, Kádár I, Ferenc R, Gyimóthy T (2018) Empirical evaluation of software maintainability based on a manually validated refactoring dataset. Inf Softw Technol 95:313–327

Jayatilleke S, Lai R, Reed K (2018) A method of requirements change analysis. Requir Eng 23(4):493–508

Jiang L, Cai Z, Zhang H, Wang D (2013) Naive bayes text classifiers: a locally weighted learning approach. J Exp Theor Artif Intell 25(2):273–286

JIRA (2019) (Accessed October 2019) https://www.atlassian.com/software/jira

Kessentini M, Dea TJ, Ouni A (2017) A context-based refactoring recommendation approach using simulated annealing: two industrial case studies. In: Proceedings of the genetic and evolutionary computation conference, GECCO 2017, Berlin, Germany, July 15–19, 2017, pp 1303–1310

Khan A, Baharudin B, Lee LH, Khan K, Tronoh UTP (2010) A review of machine learning algorithms for text-documents classification. J Adv Inf Technol 1(1):4–20

Kim M, Gee M, Loh A, Rachatasumrit N (2010) Ref-finder: a refactoring reconstruction tool based on logic query templates. In: Proceedings of the 18th ACM SIGSOFT international symposium on foundations of software engineering, 2010, Santa Fe, NM, USA, November 7–11, 2010, pp 371–372

Kim M, Zimmermann T, Nagappan N (2012) A field study of refactoring challenges and benefits. In: 20th ACM SIGSOFT symposium on the foundations of software engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA–November 11–16, 2012, p 50

Lin Y, Peng X, Cai Y, Dig D, Zheng D, Zhao W (2016) Interactive and guided architectural refactoring with search-based recommendation. In: Proceedings of the 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2016, Seattle, WA, USA, November 13–18, 2016, pp 535–546

Liu H, Ma Z, Shao W, Niu Z (2012) Schedule of bad smell detection and resolution: a new way to save effort. IEEE Trans Software Eng 38(1):220–235

Liu H, Guo X, Shao W (2013) Monitor-based instant software refactoring. IEEE Trans Softw Eng 39(8):1112–1126

Liu H, Xu Z, Zou Y (2018) Deep learning based feature envy detection. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, ASE 2018, Montpellier, France, September 3–7, 2018, pp 385–396

Loper E, Bird S (2002) NLTK: the natural language toolkit. In: ACL workshop effective tools and methodologies for teaching on natural language processing and computational linguistics (ETMTNLP), pp 63–70

Mahmoud A, Niu N (2014) Supporting requirements to code traceability through refactoring. Requir Eng 19(3):309–329

Manning CD, Schütze H (2001) Foundations of statistical natural language processing. MIT Press, Cambridge

Manning CD, Raghavan P, Schütze H (2008) Introduction to information retrieval. Cambridge University Press, Cambridge

Mei H, Zhang L (2018) Can big data bring a breakthrough for software automation? Science China Inf Sci 61(5):056101:1–056101:3

Mens T, Tourwé T (2004) A survey of software refactoring. IEEE Trans Softw Eng 30(2):126–139

Moha N, Guéhéneuc Y, Duchien L, Meur AL (2010) DECOR: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36(1):20–36

Mohan M, Greer D (2018) A survey of search-based refactoring for software maintenance. J Softw Eng R&D 6:3

Murphy-Hill ER, Parnin C, Black AP (2012) How we refactor, and how we know it. IEEE Trans Software Eng 38(1):5–18

Nizamani ZA, Liu H, Chen DM, Niu Z (2018) Automatic approval prediction for software enhancement requests. Autom Softw Eng 25(2):347–381

Niu N, Bhowmik T, Liu H, Niu Z (2014) Traceability-enabled refactoring for managing just-in-time requirements. In: IEEE 22nd international requirements engineering conference, RE 2014, Karlskrona, Sweden, August, 25–29, 2014, pp 133–142

Nyamawe AS, Liu H, Niu Z, Wang W, Niu N (2018) Recommending refactoring solutions based on traceability and code metrics. IEEE Access 6:49460–49475

Nyamawe AS, Liu H, Niu N, Umer Q, Niu Z (2019) Automated recommendation of software refactorings based on feature requests. In: Damian DE, Perini A, Lee S (eds) 27th IEEE International requirements engineering conference, RE 2019, Jeju Island, Korea (South), September 23–27, 2019. IEEE, pp 187–198

Ouni A, Kessentini M, Sahraoui HA, Hamdi MS (2013) The use of development history in software refactoring using a multi-objective evolutionary algorithm. In: Genetic and evolutionary computation conference, GECCO '13, Amsterdam, The Netherlands, July 6–10, 2013, pp 1461–1468

Ouni A, Kessentini M, Sahraoui HA, Inoue K, Hamdi MS (2015) Improving multi-objective code-smells correction using development history. J Syst Softw 105:18–39

Ouni A, Kessentini M, Sahraoui HA, Inoue K, Deb K (2016) Multi-criteria code refactoring using search-based software engineering: an industrial case study. ACM Trans Softw Eng Methodol 25(3):23:1–23:53

Ouni A, Kessentini M, Cinnéide MÓ, Sahraoui HA, Deb K, Inoue K (2017) MORE: a multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. J Softw: Evol Process 29(5)

Palomba F, Salza P, Ciurumelea A, Panichella S, Gall HC, Ferrucci F, Lucia AD (2017) Recommending and localizing change requests for mobile apps based on user reviews. In: Uchitel S, Orso A, Robillard MP (eds) Proceedings of the 39th international conference on software engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017. IEEE/ACM, pp 106–117

Palomba F, Zaidman A, Oliveto R, Lucia AD (2017) An exploratory study on the relationship between changes and refactoring. In: Proceedings of the 25th international conference on program comprehension, ICPC 2017, Buenos Aires, Argentina, May 22–23, 2017, pp 176–185

Pantiuchina J, Bavota G, Tufano M, Poshyvanyk D (2018) Towards just-in-time refactoring recommenders. In: Proceedings of the 26th conference on program comprehension, ICPC 2018, Gothenburg, Sweden, May 27–28, 2018, pp 312–315

PMDCodacy (2019) (Accessed October 2019) https://github.com/codacy/codacy-pmdjava

Porter MF (2006) An algorithm for suffix stripping. Program 40(3):211–218

Rath M, Rendall J, Guo JLC, Cleland-Huang J, Mäder P (2018) Traceability in the wild: automatically augmenting incomplete trace links. In: Chaudron M, Crnkovic I, Chechik M, Harman M (eds) Proceedings of the 40th international conference on software engineering, ICSE 2018, Gothenburg, Sweden, May 27–June 03, 2018. ACM, pp 834–845

Ratzinger J, Sigmund T, Vorburger P, Gall HC (2007) Mining software evolution to predict refactoring. In: Proceedings of the first international symposium on empirical software engineering and measurement, ESEM 2007, September 20–21, 2007, Madrid, pp 354–363

Runeson P, Alexandersson M, Nyholm O (2007) Detection of duplicate defect reports using natural language processing. In: 29th International conference on software engineering (ICSE 2007), Minneapolis, MN, USA, May 20–26, 2007. IEEE Computer Society, pp 499–510

Schapire RE, Singer Y (2000) Boostexter: a boosting-based system for text categorization. Mach Learn 39(2/3):135–168

Scikit-learn (Accessed October 2019) https://scikit-learn.org/stable/

Silva D, Tsantalis N, Valente MT (2016) Why we refactor? Confessions of GitHub contributors. In: Proceedings of the 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2016, Seattle, WA, USA, November 13–18, 2016, pp 858–870

Silva D, Valente MT (2017) Refdiff: detecting refactorings in version histories. In: Proceedings of the 14th international conference on mining software repositories, MSR 2017, Buenos Aires, Argentina, May 20–28, 2017, pp 269–279

Simon F, Steinbrückner F, Lewerentz C (2001) Metrics based refactoring. In: Fifth conference on software maintenance and reengineering, CSMR 2001, Lisbon, Portugal, March 14–16, 2001, pp 30–38

Soares G, Catao B, Varjao C, Aguiar S, Gheyi R, Massoni T (2011) Analyzing refactorings on software repositories. In: 25th Brazilian symposium on software engineering, SBES 2011, Sao Paulo, Brazil, September 28–30, 2011. IEEE Computer Society, pp 164–173

Sun C, Lo D, Wang X, Jiang J, Khoo S (2010) A discriminative model approach for accurate duplicate bug report retrieval. In: Kramer J, Bishop J, Devanbu PT, Uchitel S (eds) Proceedings of the 32nd ACM/IEEE international conference on software engineering—volume 1, ICSE 2010, Cape Town, South Africa, 1–8 May 2010. ACM, pp 45–54

Tan L, Bockisch C (2019) A survey of refactoring detection tools. In: Krusche S, Schneider K, Kuhrmann M, Heinrich R, Jung R, Konersmann M, Schmieders E, Helke S, Schaefer I, Vogelsang A, Annighöfer B, Schweiger A, Reich M, van Hoorn A (eds) Proceedings of the workshops of the software engineering conference 2019, Stuttgart, Germany, February 19, 2019, CEUR-WS.org, CEUR Workshop Proceedings, vol 2308, pp 100–105

Terra R, Valente MT, Miranda S, Sales V (2018) JMove: a novel heuristic and tool to detect move method refactoring opportunities. J Syst Softw 138:19–36

Thung F, Wang S, Lo D, Lawall JL (2013) Automatic recommendation of API methods from feature requests. In: 2013 28th IEEE/ACM international conference on automated software engineering, ASE 2013, Silicon Valley, CA, USA, November 11–15, 2013, pp 290–300

Thung F, Kochhar PS, Lo D (2014) Dupfinder: integrated tool support for duplicate bug report detection. In: Crnkovic I, Chechik M, Grünbacher P (eds) ACM/IEEE international conference on automated software engineering, ASE ’14, Vasteras, Sweden–September 15–19, 2014. ACM, pp 871–874

Tsantalis N, Chatzigeorgiou A (2009) Identification of move method refactoring opportunities. IEEE Trans Softw Eng 35(3):347–367

Tsantalis N, Chatzigeorgiou A (2011) Ranking refactoring suggestions based on historical volatility. In: 15th European conference on software maintenance and reengineering, CSMR 2011, 1–4 March, 2011, Oldenburg, pp 25–34

Tsantalis N, Mansouri M, Eshkevari LM, Mazinanian D, Dig D (2018) Accurate and efficient refactoring detection in commit history. In: Proceedings of the 40th international conference on software engineering, ICSE 2018, Gothenburg, Sweden, May 27–June 03, 2018, pp 483–494

Uysal AK, Günal S (2014) The impact of preprocessing on text classification. Inf Process Manag 50(1):104–112

Vidal SA, Marcos CA, Pace JAD (2014) An approach to prioritize code smells for refactoring. Autom Softw Eng 23(3):501–532

Xu S, Sivaraman A, Khoo S, Xu J (2017) GEMS: an extract method refactoring recommender. In: 28th IEEE International symposium on software reliability engineering, ISSRE 2017, Toulouse, France, October 23–26, 2017, pp 24–34

Yue R, Gao Z, Meng N, Xiong Y, Wang X, Morgenthaler JD (2018) Automatic clone recommendation for refactoring based on the present and the past. In: 2018 IEEE international conference on software maintenance and evolution, ICSME 2018, Madrid, Spain, September 23–29, 2018, pp 115–126

Zhang M, Zhou Z (2014) A review on multi-label learning algorithms. IEEE Trans Knowl Data Eng 26(8):1819–1837

Affiliations

Ally S. Nyamawe[1] · **Hui Liu**[1] · **Nan Niu**[2] · **Qasim Umer**[1] · **Zhendong Niu**[1]

Ally S. Nyamawe
nyamawe@udom.ac.tz

Nan Niu
nan.niu@uc.edu

Qasim Umer
qasimumer667@hotmail.com

Zhendong Niu
zniu@bit.edu.cn

[1] School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China

[2] Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH, USA