Automated Recommendation of Software Refactorings based on Feature Requests

Ally S. Nyamawe[1], Hui Liu[1,*], Nan Niu[2], Qasim Umer[1], and Zhendong Niu[1]
[1]School of Computer Science and Technology, Beijing Institute of Technology, China
nyamawe@udom.ac.tz,{liuhui08, zniu}@bit.edu.cn, qasimumer667@hotmail.com
[2]Department of Electrical Engineering and Computer Science, University of Cincinnati, USA
nan.niu@uc.edu

Abstract—**During software evolution, developers often receive new requirements expressed as feature requests. To implement the requested features, developers have to perform necessary modifications (refactorings) to prepare for new adaptation that accommodates the new requirements. Software refactoring is a well-known technique that has been extensively used to improve software quality such as maintainability and extensibility. However, it is often challenging to determine which kind of refactorings should be applied. Consequently, several approaches based on various heuristics have been proposed to recommend refactorings. However, there is still lack of automated support to recommend refactorings given a feature request. To this end, in this paper, we propose a novel approach that recommends refactorings based on the history of the previously requested features and applied refactorings. First, we exploit the state-of-the-art refactoring detection tools to identify the previous refactorings applied to implement the past feature requests. Second, we train a machine classifier with the history data of the feature requests and refactorings applied on the commits that implemented the corresponding feature requests. The machine classifier is then used to predict refactorings for new feature requests. We evaluate the proposed approach on the dataset of 43 open source Java projects and the results suggest that the proposed approach can accurately recommend refactorings (average precision 73%).**

Index Terms—**Feature Requests, Machine Learning, Refactorings Recommendation, Software Refactoring.**

I. Introduction

Software systems continuously change and evolve to adapt new requirements and accommodate new components. Change of requirements is inevitable as the business and stakeholder demands continuously evolve. As a result, software systems constantly need to be maintained in order to continue satisfying their intended objectives. During software evolution, developers often receive new requirements expressed as feature requests. To implement the requested features, developers often perform necessary modifications (refactorings) to prepare their systems to accommodate the new requirements [1]. Software refactoring is a well-known technique that has been extensively used to improve software quality by applying changes on internal structure that do not alter its external behaviors [2].

Soares *et al.* [3] state that refactorings are most commonly applied for a particular reason such as implementing a feature or bug fixing, than in the dedicated refactoring sessions aiming at evolving the software design. The recently conducted empirical study to determine the motivation behind refactoring found that refactoring activity is mainly motivated by the changes in the requirements and much less by code smell resolution [4]. Moreover, Silva *et al.* [4] suggest that there is a need for the refactoring recommendation systems to recommend suitable solutions to facilitate maintenance tasks especially the implementation of the feature or bug fix requests.

Usually, in order to improve the next releases of software, developers of most software systems, particularly open source projects, allow users to report the issues (i.e., new feature and bug fix requests). The common and dominant way to track and manage the reported issues is the use of issue tracking systems e.g JIRA [5], Bugzilla [6], and GitHub Issue Tracker [7]. Issue trackers allow for discussing a feature request, assigning request to developers, and tracking the status of the request [8]. Given the requirements expressed in such feature requests, developers often need to locate the source code that should be modified to allow the implementation of the requested feature. As a result, several techniques have been proposed to leverage feature requests to allow locating (e.g., based on requirements traceability and text similarity) and recommending software entities (e.g., API methods) that can be used to implement the feature [1], [9], [10]. 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 machine-learning-based approach that recommends refactorings based on the history of the previously requested features and applied refactorings. The proposed approach learns from the training dataset associated with a set of applications and can be used to suggest refactorings 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 classes only and it does not point to the locations in the codebase for the recommended refactoring. In practice, it could be integrated with other approaches/tools that could suggest such

[*]Corresponding author: Hui Liu (liuhui08@bit.edu.cn)

2332-6441/19/$31.00 ©2019 IEEE
DOI 10.1109/RE.2019.00029

locations, and thus makes complete refactoring suggestions. The proposed approach also helps developers pick up proper refactoring tools by suggesting refactoring classes.

The past feature requests and their associated commits (from which refactorings are detected) are retrieved from the corresponding issue trackers and software repositories respectively. Normally, each feature request can be linked to the corresponding source code commits to identify the types of refactorings applied on such commits. The previously applied refactorings are recovered from the corresponding software repositories by using the state-of-the-art refactoring detection tools, e.g., `Ref-Finder` [11], `RefactoringCrawler` [12], `RefDiff` [13], and `RMINER` [14].

The proposed approach is evaluated on the dataset of 43 open source Java projects altogether consisting of $13,550$ commits from GitHub repository and $13,367$ feature requests from JIRA issue tracker. The evaluation results suggest that, the proposed approach can accurately recommend refactorings and attain an average precision of 73%.

The major contributions of this paper include:

(1) A new automated approach to recommend refactoring solutions given a feature request based on the history of the previous feature requests and applied refactorings from a set of applications. Such applications could be different from the one where refactorings are recommended.

(2) Evaluation results of the proposed approach on the refactorings and feature requests history data suggest that the proposed approach can accurately recommend refactorings given a new feature request.

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

II. Related Work

Software development and maintenance activities often involve changing the internal structure of the software systems without affecting their observable behaviours. Such kind of source code transformation is called refactoring [2], [15]. Software refactoring aims at remedying software design flaws and improving software quality, especially reusability, maintainability, and extensibility [16]. Consequently, refactoring makes the source code easier to maintain, understand and adapt to new requirements [17]. Refactoring has long been practiced as a technique to resolve design flaws in source code commonly known as code smells. Detecting code smells and applying relevant refactorings especially in large and non-trivial software systems is often challenging [18]. Consequently, extensive research has been devoted to develop tools and techniques to automatically detect code smells, recommend refactorings, and apply them. Such tools as JDeodorant [19], iPlasma [20] and DECOR [21], make software refactoring efficient and less error-prone. Refactoring recommendation aims at facilitating developers in identifying refactoring opportunities (i.e., code smells), selecting and executing optimal refactorings promptly. Refactoring recommendation systems should be tailored to address the real needs of software developers and help to effectively promote the practice of refactoring by recommending refactoring solutions that facilitate maintenance-oriented tasks [4]. Among the key techniques that have been used in recommending refactorings include source code metrics [22], [23], software change history [24], [25], search-based approaches [26], [27], requirements traceability [1], [28] and machine learning techniques [17], [29]. However, to the best of our knowledge there is still lack of automated support to recommend refactoring solutions based on past feature requests. Historical data is essential in producing quality code to evolve software systems [30]. Next, we review some research related to our work.

Our work is inspired by the earlier work proposed by Niu *et al.* [1]. The authors in [1] 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, and then semantic characterization is leveraged to detect code smells that may hinder the fulfillment of the requirements. Consequently, refactorings are recommended to resolve the identified code smells. The key difference of the approach in [1] with our proposed approach is on how to recommend refactoring solutions, though both approaches rely on feature requests (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, whereas Niu *et al.* [1] only work with current feature request as input and recommend refactorings that ensure full implementation of the requested feature. Moreover, the proposed technique in [1] is based on manual analysis of requirements action themes which are consequently used to recommend refactoring. Action theme refers to the intended action (e.g., Add, Enhance, Remove) to be taken to implement a feature such as enhancing a quality attribute. Manual analysis is often tedious and error-prone. Our work implements an automated recommendation approach.

Nyamawe *et al.* [28] proposed a refactoring recommendation approach which is based on requirements traceability and source code metrics. The approach in [28] leverages the traceability between requirements and source code to determine how best source code elements can be arranged and consequently recommend refactorings that will lead to improved traceability and source code metrics e.g., cohesion and coupling. Traceability has a great role to play in delivering a quality software. An empirical study conducted by Rempel *et al.* [31] suggests that, requirements traceability completeness greatly decreases the expected rate of defects in the developed software. Both approaches in [1] and [28] do not take into consideration the past applied refactorings during their recommendation process. Moreover, the proposed approach differs from [28] 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.* [9] 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.

Moreover, machine learning techniques have demonstrated promising contribution in the field of software refactoring. Several approaches have been proposed to leverage machine learning to detect refactoring opportunities and recommend refactorings. For example, Liu *et al.* [29] recently proposed a deep learning based approach for feature envy code smell detection. Their approach trains a neural network based classifier with the training samples (generated automatically) consisting of the methods with or without feature envy. The classifier outputs whether the input method from a given class envies another class. Furthermore, the resulting neural network is used to predict whether a method should be moved to any of the identified class. In addition, Xu *et al.* [17] 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.* [17] 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. These approaches differ from our proposed approach as they do not use feature requests in their recommendation.

III. Approach

The framework of the proposed approach is analyzed in Fig. 1. First, we extract the feature requests from the issue tracker and their corresponding commits from a software repository. Second, by using the state-of-the-art refactoring detection tools

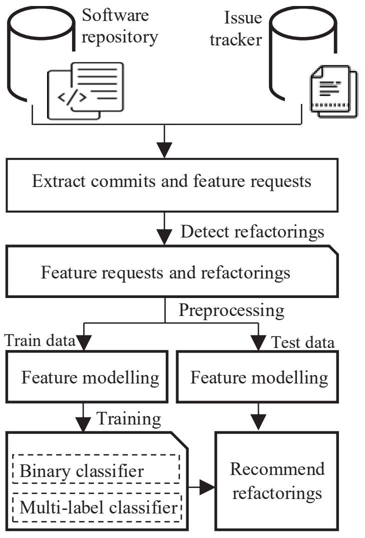

Fig. 1. **Refactoring Recommendation Approach.**

we recover refactorings applied in the retrieved commits. Next, we apply the preprocessing on the retrieved feature requests. Finally, we train the machine-learning-based classifier which is then used to predict refactorings for a new feature request. In the following we elaborate each of these steps in detail.

A. 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 I.

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. Then, 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 [32] we retrieve all commits identifiers of the commits that contain in their log 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 such that:

$$fr = < frID, summary, description, commitID > \quad (1)$$

where $frID$ represents a unique feature request identifier, $summary$ is the title of a feature request, $description$ 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. However, in this study we only focus on the cases where the links between the feature requests and commits are known explicitly. In the future the state-of-the-art techniques can be leveraged to generate missing links. For example, recently, Rath *et al.* [33] proposed a machine-learning based approach that trains a classifier to identify the missing issue tags in commit messages and consequently generate the missing links.

B. 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 [13] 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 Tsantatlis *et al.* [14] 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 proved to be effective in detecting applied refactorings by comparing subsequent versions of the program. 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:

$$fr =< frID, summary, description, commitID, ref > \tag{2}$$

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.

C. Text Pre-Processing

Text pre-processing is the key step which involves the cleaning and preparing the data for classification, which consequently improves the classification performance [34]. We leveraged text preprocessing techniques 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) [35]. 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 [36] which implements suffix stripping algorithm for lemmatization. Porter's stemmer has been extensively used in various software engineering researches [10], [37].

TABLE I
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

D. Vector Space Model

Vector space model is the representation of a set of documents as vectors in a common vector space [38]. In this step, the preprocessed feature requests are converted into a feature vector space model 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:

$$IDF(t, D) = \frac{1}{DF(t, D)} \tag{3}$$

$$w_{\text{t,fr,D}} = TF(t, fr) \times IDF(t, D) \tag{4}$$

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

E. Training and Recommendation

The feature vectors obtained in the previous step are then subject to the classifiers for training and prediction (i.e., recommending refactorings). The proposed approach leverages the Logistic Regression (LR), Multinomial Naïve Bayes (MNB), Support Vector Machine (SVM), and Random Forest (RF) classifiers. The classifiers have been implemented by a well-known machine learning library based on python called scikit-learn [39]. We have specifically selected such machine learning algorithms because they are widely used and have shown to be effective in text classification [40], [41]. We generally model our text classification problem such that, we have a description $fr \in FR$ of a feature request, where FR is the feature requests space; and a fixed set of classes $R = \{r_1, r_2, ..., r_{\mathrm{m}}\}$. In our case here, classes are also referred to as labels or refactorings. 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., recommending refactorings): $f : FR \rightarrow R$.

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 the feature requests that required refactoring and those which did not require refactoring. Then, the classifier categorizes a feature request fr into a class c as function f such that:
$$c = f(fr), \ \ c \in \{0, 1\}, \ \ fr \in FR \tag{5}$$
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 save as input to the next stage to identify the types of refactorings required.
- **Multi-label classification**, after identifying the feature requests that require refactoring, 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) and the applied refactorings will save as input. Consequently, the classifier categorizes a feature request fr into a class c as function f such that:
$$c = f(fr), \ \ c \subseteq R, \ \ fr \in FR \tag{6}$$
where c is the set of one or more refactorings.

Therefore, the classifiers are generally trained to determine whether refactoring is required, if yes then they should predict (i.e., recommend) the types of refactorings required for the unseen feature requests.

IV. Evaluation

In this section we present the evaluation of the proposed approach which 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 with the state-of-the-art approach proposed by Niu *et al.* [1]. In the following, we first highlight the research questions that this study is addressing. 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.

A. 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 refactorings?
- **RQ3**: How accurate is *FR-Refactor* in predicting the need for refactoring compared to the state-of-the-art baseline approach?
- **RQ4**: How accurate is *FR-Refactor* in recommending required refactorings compared to the state-of-the-art baseline approach?
- **RQ5**: 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 concern the performance evaluation of different machine learning algorithms in identifying whether refactoring is needed or not and in identifying which refactoring type is needed. **RQ3** investigates the performance of *FR-Refactor* in predicting if a given feature request would demand refactoring. To answer **RQ3** we compare *FR-Refactor* to the traceability-enabled approach proposed by Niu *et al.* [1] which is based on manual analysis of requirements semantics to recommend refactorings. We selected this approach because, to the best of our knowledge, it is the only existing approach which is based on requirements to drive refactoring. The research question **RQ4** evaluates the accuracy of *FR-Refactor* compared against the baseline approach [1] in recommending refactorings. The accuracy of recommendation is essential in providing useful (true positives) refactorings to the developers rather than overloading developers with irrelevant refactorings. **RQ5** investigates how the proposed approach works when it is applied to new applications that are different from those involved in the training.

B. Dataset

We note that there exist a few publicly available and manually validated datasets of refactorings mined from software repositories [13], [14], [42]. However, these oracles contain few representative refactorings and feature requests. For example, the evaluation oracles used in [13] and [14] consist of 448

TABLE II
THE DISTRIBUTION OF FEATURE REQUESTS (FR) AND COMMITS (COM.) IN OUR DATASET.

Project	FR	COM.	Project	FR	COM.	Project	FR	COM.	Project	FR	COM.
Accumulo	474	479	Drill	446	451	Lens	172	176	Stanbol	154	157
Archiva	193	198	Flink	555	562	Ode	102	106	Storm	302	305
Aries	314	319	Geronimo	38	44	Oodt	26	26	Struts 2	214	219
Atlas	70	74	Giraph	252	254	PDFBox	482	489	Synapse	79	82
Axis2-Java	204	208	Gora	31	34	Pivot	185	189	Systemml	34	36
Beam	295	298	Groovy	385	389	Sentry	87	89	Tajo	444	451
Bookkeeper	132	132	Hbase	1389	1396	Sling	2369	2387	Tapestry-5	381	385
Calcite	74	77	Impala	178	182	Spring-Datamongo	261	264	Velocity	56	58
Carbondata	226	230	Jclouds	122	127	Spring-Integration	517	523	Wicket	237	242
Cayenne	390	393	Jena	202	209	Spring-ROO	797	800	Zookeeper	80	80
Curator	24	26	Kafka	307	314	Spring-Security	87	90			
TOTAL:	Projects: 43, Feature Requests: $13,367$, Commits: $13,550$										

and $3,188$ known refactoring operations respectively. In addition to that, not all refactorings in such oracles are associated with the implementation of feature requests. Therefore, to attain the reasonable amount of feature requests and refactorings to effectively train our classifiers, we exploit `RefDiff` and `RMINER` which were recently validated manually and used in creating an oracle of refactorings proposed by Tsantatlis *et al.* [14]. The tools are publicly available and effective in detecting applied refactorings by comparing subsequent versions of the program.

Table II highlights the distribution of the feature requests and commits of the subject applications used in building our dataset. First, we extracted the feature requests (Request ID, Summary, and Description fields) of the subject applications from JIRA issue tracker which have been addressed to completion (i.e., marked as "Closed" or "Resolved"). JIRA explicitly links the feature requests to their corresponding commits in a repository through request ID. Second, we retrieved the relevant commits from the repository that was used to implement the retrieved feature requests. We only selected Java open source projects from GitHub repository whose commits explicitly specify in their log messages the issue (i.e feature request identifier) which is addressed. Thus our selection contains a reliable link between a feature request in an issue tracker and the corresponding commit in a software repository. Third, the retrieved commits were subject to the refactoring detection tools to recover the applied refactorings. Finally, we created a dataset of 43 open source Java projects altogether consisting of $13,550$ commits from GitHub repository and $13,367$ feature requests from JIRA issue tracker. Out of $13,367$ feature requests, a total of $7,943$ ($59\% = 7,943/13,367$) feature requests are associated with one or more refactorings, whereas the remaining 41% ($= 5,424/13,367$) of the feature requests do not have any refactoring. The feature requests and their associated refactorings used are available online at http://doi.org/10.5281/zenodo.3335978. Such subject applications were selected because they cover a wide-range of domains, publicly available, developed by different developers, and have long evolution history. Consequently, it is likely that they will have varieties of feature requests and refactorings. Further, Table III highlights the distribution of refactoring types in our dataset.

TABLE III
THE DISTRIBUTION OF REFACTORINGS IN OUR DATASET.

Refactorings	Number	Refactorings	Number
Extract Interface	141	Move Method	1441
Extract Method	4225	Pull Up Attribute	186
Extract Superclass	132	Pull Up Method	255
Inline Method	784	Push Down Attribute	102
Move And Rename Class	268	Push Down Method	112
Move Attribute	957	Rename Class	801
Move Class	784	Rename Method	2940
Total			13,128

C. Process and Metrics

The training and prediction involve two steps. First, we train the classifiers to predict whether the given feature requests would require refactoring or not. Second, for the feature requests identified to require refactoring, we train the classifiers to predict the refactoring types. To evaluate the classifiers for predicting whether refactoring is required or not (i.e., binary classification), we use the traditional accuracy, precision, recall, and F-measure metrics. Further, since each feature request can be associated with one or more refactorings, 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 [43]. To evaluate the performance of the multi-label classifiers we leveraged the common and widely used metrics including hamming loss, hamming score, and subset accuracy [43]–[45].

To answer the research questions (RQ1-RQ4), we conduct 10-fold cross validation where the collected data (as specified in the preceding section) are randomly partitioned into ten

TABLE IV
CLASSIFIERS' PERFORMANCE FOR PREDICTING THE NEED FOR REFACTORING (%).

Classifier	Accuracy	Precision	Recall	F-measure
SVM	64.13	**69.49**	71.22	70.32
MNB	**65.40**	66.36	**85.32**	**74.63**
LR	**65.35**	69.30	75.35	72.16
RF	63.59	69.16	70.59	69.79

subsets. On each fold of the evaluation, one subset is employed as testing data whereas others are taken as training data.

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 refactorings 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 refactorings for a feature request fr_i.

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

$$HammingLoss(H) = \frac{1}{|D|}\sum_{i=1}^{|D|}\frac{|z_i \Delta r_i|}{|R|} \tag{7}$$

where Δ stands for the symmetric difference between the two sets (i.e., the set of predicted refactorings and the set of true refactorings for the feature request f_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 refactorings (z_i) is to the true set of refactorings (r_i) for the given feature request f_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:

$$HammingScore(H) = \frac{|r_i \cap z_i|}{|r_i \cup z_i|} \tag{8}$$

- **Subset accuracy**, measures the fraction of examples classified correctly, i.e., the predicted set of refactorings (z_i) is similar to the true set of refactorings (r_i) for the given feature request f_i. Subset accuracy can be intuitively considered as the traditional accuracy metric [43]. It is formally defined as:

$$SubsetAccuracy(H) = \frac{1}{|D|}\sum_{i=1}^{|D|}[\![z_i = r_i]\!] \tag{9}$$

where $[\![z_i = r_i]\!]$ 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.

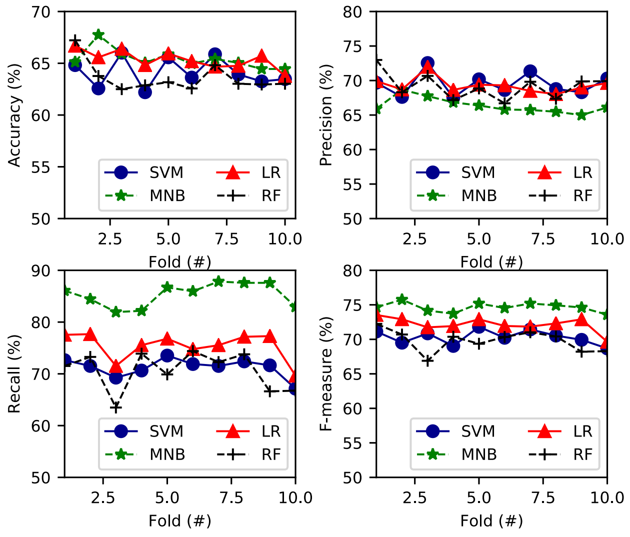

Fig. 2. **Performance of the Classifiers in Predicting Refactoring.**

D. *RQ1: Performance of Different Classifiers in Identifying the Need for Refactoring*

Table IV highlights the effectiveness of the employed classifiers in predicting whether refactoring is required to implement a given feature request. Note that, the best recorded results for each metric is highlighted in bold. From the table, it is observed that the precision of predicting whether refactoring would be required ranges between 66.36% and 69.49%. Therefore, on average of up to 68.58% the need for refactoring can be accurately predicted. Furthermore, the results generally suggest that, on average, MNB and LR classifiers outweigh all other classifiers. MNB classifier achieved an F-measure of 74.63% and an accuracy of 65.40%, whereas, LR classifier achieved an F-measure of 72.16% and can accurately predict the need for refactoring (average precision 69.30%). In addition to that, Fig. 2 compares the performance of the classifiers in terms of accuracy, precision, recall, and F-measure in ten-fold cross validation. As depicted in Fig. 2, it is evident that there is no significant difference in classifiers' individual performance across different folds. Note that, our binary classification problem 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. In this case Naïve Bayes classifier performed better than other classifiers. Naïve Bayes classifier is the widely used generative classifier that can easily accommodate any domain-specific knowledge and also performs better with hierarchical classification scenario [40]. MNB has shown to be effective in binary text classification in various studies including enhancement requests approval prediction [41] and spam emails detection [46]. Because 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.

TABLE V
CLASSIFIERS' PERFORMANCE FOR REFACTORINGS RECOMMENDATION (%).

Classifier	Subset accuracy	Hamming score	Hamming loss
SVM	**70.75**	**70.85**	**0.027**
MNB	14.98	15.05	0.062
LR	43.86	43.86	0.043

*E. **RQ2:** Performance of Different Classifiers in Recommending Refactorings*

As pointed out earlier, the refactorings recommendation is cast here as a multi-label classification problem, hence Table V depicts the results in terms of subset accuracy, hamming score, and hamming loss which are widely used metrics for evaluating multi-label classifiers. We only selected three representative classifiers (SVM, MNB, and LR) because they can adapt popular learning techniques and directly work on multi-label data without transforming multi-label learning problem into other classification problems such as single-label or binary classification [43], [47]. Note that, the best recorded results for each metric is highlighted in bold. As shown in Table V, the results indicate that SVM classifier outperforms MNB and LR by the difference of $56\%(= 70.75\% - 14.98\%)$ and $27\%(= 70.75\% - 43.86\%)$ in terms of subset accuracy respectively. Subset accuracy (also called $classification\ accuracy$) returns the percentage of instances where the set of labels (i.e., refactorings) predicted by the classifier is exactly the same with their corresponding truth set [47]. Therefore, the results suggest that the needed refactorings can be accurately recommended on up to 71% accuracy. Moreover, compared to other classifiers, SVM achieved the lowest value (i.e., 0.027) on hamming loss which identifies to what extent the classifier predicts the irrelevant refactorings and omits relevant refactorings. This metric is normalized between 0 and 1. The value of the metric closes to 0 indicates better performance of the classification. Furthermore, Table VI presents the performance of SVM classifier in recommending individual refactorings. The results generally suggest that, the classifier achieves an average precision of 73% in recommending refactorings. However, 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 refactorings, including Push Down Method, are very few in the dataset which can consequently affect their prediction. In future we will consider balancing the dataset to include enough number of each refactoring type. From these findings, we therefore conclude that the proposed approach is accurate in recommending refactorings.

We note that, SVM performed better than other classifiers in refactorings recommendation. 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 spaces. SVM employs overfitting protection and has the ability to learn which can be independent from the dimensionality of the feature space. Second, few irrelevant features, in text classification irrelevant features are very few and 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 facts, SVM is shown to be effective for text classification and well recognized to be accurate [44].

TABLE VI
RESULTS FOR INDIVIDUAL REFACTORINGS RECOMMENDATION.

Refactoring type	Precision	Recall	F-measure
Extract Interface	0.63	0.58	0.61
Extract Method	0.66	0.74	0.70
Extract Superclass	0.85	0.43	0.58
Inline Method	0.83	0.38	0.52
Move And Rename Class	0.89	0.54	0.67
Move Attribute	0.86	0.34	0.49
Move Class	0.79	0.42	0.55
Move Method	0.82	0.42	0.56
Pull Up Attribute	0.69	0.60	0.64
Pull Up Method	0.67	0.42	0.52
Push Down Attribute	0.84	0.38	0.52
Push Down Method	0.83	0.33	0.47
Rename Class	0.72	0.39	0.50
Rename Method	0.65	0.51	0.58
Micro avg	0.71	0.52	0.60
Macro avg	0.77	0.46	0.56
Weighted avg	0.73	0.52	0.59

Because the evaluation results suggest 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.

*F. **RQ3:** FR-Refactor vs State-of-the-art Baseline Approach in Predicting the Need for Refactoring*

To evaluate the performance of *FR-Refactor* in predicting the need for refactoring, we compared it (based on MNB) with the state-of-the-art approach proposed by Niu *et al.* [1]. As depicted in Fig. 3, *FR-Refactor* (based on MNB classifier) significantly outperforms the state-of-the-art approach. We note that, *FR-Refactor* improves F-measure by $32.8\%(74.6\% - 41.8\%)$, and attains better performance in terms of recall (85.3%) 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 (32.1%). In addition to that, *FR-Refactor* improves accuracy and precision by 19% and 6.6% respectively. The results lead us to the conclusion that, *FR-Refactor* can accurately predict the need for refactoring. Whereas the baseline approach may fail to explicitly identify the requirements semantics and consequently predict the need for refactoring, *FR-Refactor*

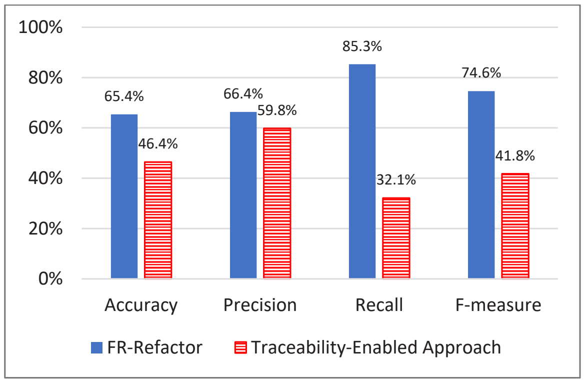

Fig. 3. **Performance in Predicting the Necessity of Refactoring.**

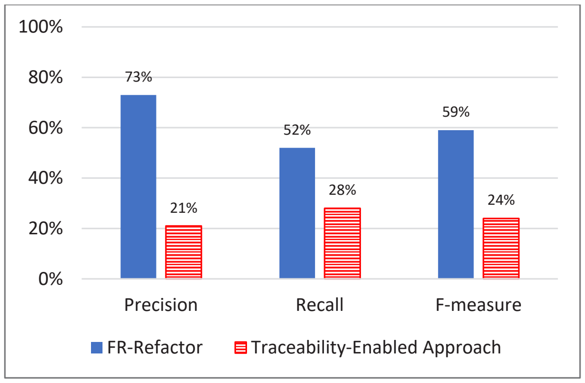

Fig. 4. **Performance in Refactorings Recommendation.**

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 past features like (`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.

G. ***RQ4:*** *FR-Refactor vs State-of-the-art Baseline Approach in Recommending Refactorings*

Fig. 4 compares the performance of *FR-Refactor* (based on SVM classifier) against the state-of-the-art approach [1] in recommending refactorings. Generally, the results suggest that, on average *FR-Refactor* outperforms the baseline approach in accurately recommending refactorings. We observe that *FR-Refactor* can accurately recommend relevant refactorings with an average precision of 73% which is equivalent to an improvement of precision by $52\%(73\%-21\%)$ compared against the baseline approach. Furthermore, *FR-Refactor* significantly improves recall and F-measure by $24\%(52\% - 28\%)$ and $35\%(59\% - 24\%)$ respectively.

We note that, *FR-Refactor* achieves better results than the baseline approach. The baseline approach only relies on the predefined requirements semantics and also ignores the possibility that a given feature request may also belong to different categories of requirements semantics. Consequently, the approach may miss out some relevant refactorings. 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 refactorings. First, a *Rename Method* and *Extract Method* refactorings. 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* 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 refactorings 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 refactorings.

H. ***RQ5:*** *Cross Project Evaluation*

In the preceding evaluation, we take all dataset from different applications as a whole, and conduct 10-fold evaluation on the resulting dataset. To answer RQ5, in this section we conduct cross project evaluation. We use 40 subject applications for the evaluation, and divide them into 10 groups (each contains 4 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 works well even if the testing projects are different from training projects. Its precision, recall and F-measure on binary classification are 65%, 73%, and 68.8%, respectively. Its precision, recall, and F-measure on multi-label classification are 77%, 50%, and 55.4%, respectively. Comparing such results against those in Section IV-F and Section IV-G, we conclude that the proposed approach still obtains good results when testing projects are different from training projects.

I. Threats to Validity

A threat to construct validity is concerned with the implementation of the approach. The major threat relates to the correctness of the recovered refactorings. That is because the leveraged refactorings detection tools are not 100% on both recall and precision. The inaccuracy of the refactorings oracle may be accelerated by the fact that the refactoring detection tools may be unable to detect all of the past applied refactorings. Moreover, the tools may suggest irrelevant or incorrect refactorings (false positives) and may miss out some true refactorings (false negatives). Consequently, that may threaten the accuracy of the refactorings oracle. However, to reduce the threat we checked the dataset for possible errors, but still there could be some errors slipped in unnoticed. 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 and some of them have long development history. Threats 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 varied 43 Java open source projects and 14 common refactoring types. To further reduce this threat, 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. Finally, the internal threats 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.

V. Conclusion and Future Work

During software evolution, developers often receive feature requests that demand for the implementation of the new feature or extension of an existing feature. To implement the requested features, developers usually 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. In this paper we propose a machine-learning-based approach to recommend refactorings based on the history of previous feature requests and applied refactorings. The proposed approach learns from the training dataset associated with a set of applications and can be used to suggest refactorings for feature requests associated with other applications or that associated with the training applications. The proposed approach is evaluated on the dataset of 43 open source Java projects altogether consisting of $13,550$ commits from GitHub repository and $13,367$ feature requests from JIRA issue tracker and their associated refactorings recovered by using the state-of-the-art refactoring detection tools. The experimental results suggest that, the proposed approach can accurately recommend refactorings and attains an average precision of 73%.

Although the proposed approach suggests refactoring classes 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 classes) and 'where' (where the refactoring should be applied). Consequently, the baseline approach proposed by Niu et al. [1] 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 [1] (that suggests refactoring classes) because the evaluation results suggest that it is more accurate than the baseline approach in suggesting refactoring classes. 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 [48] 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 [49] suggests that it is often up to developers to pick up the proper tools to identify different classes of refactoring opportunities, e.g., *GEMS* [17] for *extract method* refactoring opportunities, and *JMove* [50] for *move method* refactoring opportunities. Suggesting refactoring classes (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 interference with each other [49], knowing the required refactoring classes of different features help in deciding the order of refactoring classes and hence deciding the order of implementing features.

Our future research plan in this direction includes the following. First, it would be interesting to investigate how to locate where recommended refactorings should be conducted by mining the feature requests and analyzing the related source code. Second, we would investigate how to improve our approach by leveraging word embedding and deep learning techniques. Finally, although the experimental results suggest that the proposed approach is accurate, we plan in the future to conduct a qualitative evaluation that will involve the actual application of the approach by developers. Human evaluators will further reveal the applicability and usefulness of the approach.

Acknowledgment

The authors would like to say thanks to the anonymous reviewers 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

[1] N. Niu, T. Bhowmik, H. Liu, and Z. Niu, "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, 2014.

[2] M. Fowler, *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series, Addison-Wesley, 1999.

[3] G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, and T. Massoni, "Analyzing refactorings on software repositories," in *25th Brazilian Symposium on Software Engineering, SBES 2011, Sao Paulo, Brazil, September 28-30, 2011*, pp. 164–173, 2011.

[4] D. Silva, N. Tsantalis, and M. T. Valente, "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, 2016.

[5] JIRA. https://www.atlassian.com/software/jira.

[6] Bugzilla. https://www.bugzilla.org/.

[7] GitHub. https://github.com/features.

[8] P. Heck and A. Zaidman, "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, Russia*, pp. 43–52, 2013.

[9] F. Thung, S. Wang, D. Lo, and J. L. Lawall, "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, 2013.

[10] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. C. Gall, F. Ferrucci, and A. D. Lucia, "Recommending and localizing change requests for mobile apps based on user reviews," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017* (S. Uchitel, A. Orso, and M. P. Robillard, eds.), pp. 106–117, IEEE / ACM, 2017.

[11] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "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, 2010.

[12] D. Dig, C. Comertoglu, D. Marinov, and R. E. Johnson, "Automated detection of refactorings in evolving components," in *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, pp. 404–428, 2006.

[13] D. Silva and M. T. Valente, "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, 2017.

[14] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "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, 2018.

[15] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[16] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1112–1126, 2013.

[17] S. Xu, A. Sivaraman, S. Khoo, and J. Xu, "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, 2017.

[18] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Recommending refactoring operations in large software systems," in *Recommendation Systems in Software Engineering* (M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, eds.), pp. 387–419, Springer, 2014.

[19] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *ICSM 2007. IEEE International Conference on Software Maintenance, 2007*, pp. 519–520, IEEE, 2007.

[20] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel, "iplasma: An integrated platform for quality assessment of object-oriented design," in *Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume, ICSM 2005, 25-30 September 2005, Budapest, Hungary*, pp. 77–80, 2005.

[21] N. Moha, Y. Guéhéneuc, L. Duchien, and A. L. Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[22] O. Chaparro, G. Bavota, A. Marcus, and M. D. Penta, "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, 2014.

[23] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[24] M. Kessentini, T. J. Dea, and A. Ouni, "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, 2017.

[25] N. Tsantalis and A. Chatzigeorgiou, "Ranking refactoring suggestions based on historical volatility," in *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*, pp. 25–34, 2011.

[26] A. Ouni, M. Kessentini, H. A. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 3, pp. 23:1–23:53, 2016.

[27] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "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, 2016.

[28] A. S. Nyamawe, H. Liu, Z. Niu, W. Wang, and N. Niu, "Recommending refactoring solutions based on traceability and code metrics," *IEEE Access*, vol. 6, pp. 49460–49475, 2018.

[29] H. Liu, Z. Xu, and Y. Zou, "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, 2018.

[30] H. Mei and L. Zhang, "Can big data bring a breakthrough for software automation?," *SCIENCE CHINA Information Sciences*, vol. 61, no. 5, pp. 056101:1–056101:3, 2018.

[31] P. Rempel and P. Mäder, "Preventing defects: The impact of requirements traceability completeness on software quality," *IEEE Transactions on Software Engineering*, vol. 43, no. 8, pp. 777–797, 2017.

[32] "Git bash commands," in *https://www.atlassian.com/git*, Retrived on 28th November 2018.

[33] M. Rath, J. Rendall, J. L. C. Guo, J. Cleland-Huang, and P. Mäder, "Traceability in the wild: automatically augmenting incomplete trace links," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018* (M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, eds.), pp. 834–845, ACM, 2018.

[34] A. K. Uysal and S. Günal, "The impact of preprocessing on text classification," *Information Processing and Management*, vol. 50, no. 1, pp. 104–112, 2014.

[35] E. Loper and S. Bird, "NLTK: the natural language toolkit," in *ACL Workshop Effective Tools Methodol. Teach. Natural Lang. Process. Comput. Linguistics (ETMTNLP)*, pp. 63–70, 2002.

[36] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 40, no. 3, pp. 211–218, 2006.

[37] A. Mahmoud and N. Niu, "Supporting requirements to code traceability through refactoring," *Requirements Engineering*, vol. 19, no. 3, pp. 309–329, 2014.

[38] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge University Press, 2008.

[39] Scikit-learn. https://scikit-learn.org/stable/.

[40] C. C. Aggarwal and C. Zhai, "A survey of text classification algorithms," in *Mining Text Data* (C. C. Aggarwal and C. Zhai, eds.), pp. 163–222, Springer, 2012.

[41] Z. A. Nizamani, H. Liu, D. M. Chen, and Z. Niu, "Automatic approval prediction for software enhancement requests," *Automated Software Engineering*, vol. 25, no. 2, pp. 347–381, 2018.

[42] P. Hegedüs, I. Kádár, R. Ferenc, and T. Gyimóthy, "Empirical evaluation of software maintainability based on a manually validated refactoring dataset," *Information & Software Technology*, vol. 95, pp. 313–327, 2018.

[43] M. Zhang and Z. Zhou, "A review on multi-label learning algorithms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 8, pp. 1819–1837, 2014.

[44] S. Godbole and S. Sarawagi, "Discriminative methods for multi-labeled classification," in *Advances in Knowledge Discovery and Data Mining, 8th Pacific-Asia Conference, PAKDD 2004, Sydney, Australia, May 26-28, 2004, Proceedings* (H. Dai, R. Srikant, and C. Zhang, eds.), vol. 3056 of *Lecture Notes in Computer Science*, pp. 22–30, Springer, 2004.

[45] R. E. Schapire and Y. Singer, "Boostexter: A boosting-based system for text categorization," *Machine Learning*, vol. 39, no. 2/3, pp. 135–168, 2000.

[46] W. Feng, J. Sun, L. Zhang, C. Cao, and Q. Yang, "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, 2016.

[47] E. Gibaja and S. Ventura, "A tutorial on multilabel learning," *ACM Computing Surveys*, vol. 47, no. 3, pp. 52:1–52:38, 2015.

[48] F. Angerer, A. Grimmer, H. Prähofer, and P. Grünbacher, "Change impact analysis for maintenance and evolution of variable software systems," *Autom. Softw. Eng.*, vol. 26, no. 2, pp. 417–461, 2019.

[49] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 220–235, 2012.

[50] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "JMove: A novel heuristic and tool to detect move method refactoring opportunities," *Journal of Systems and Software*, vol. 138, pp. 19–36, 2018.