Rochester Institute of Technology
RIT Scholar Works

Theses

4-2021

Can feature requests reveal the refactoring types?

Sultan Fahad Almassari
sa2553@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation
Almassari, Sultan Fahad, "Can feature requests reveal the refactoring types?" (2021). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Can feature requests reveal the refactoring types?

by

Sultan Fahad Almassari

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Software Engineering

Supervised by
Dr. Mohamed Wiem Mkaouer

Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

April 2021

The thesis “Can feature requests reveal the refactoring types?” by Sultan Fahad Almassari has been examined and approved by the following Examination Committee:

Dr. Mohamed Wiem Mkaouer
Assistant Professor, RIT
Thesis Committee Chair

Dr. Ikram Chaabane
Assistant Professor, FSEGS
Thesis Committee Member

Dr. J. Scott Hawker
Associate Professor
Graduate Program Director, RIT
Thesis Committee Member

Dedication

To my parents, brother, sisters and friends for providing endless love, support, and faith.

Acknowledgments

First and foremost, I would like to express my deep gratitude and praise to the Almighty God for entitling me to successfully achieve this master research and for His showers of blessings throughout my research work.

My sincere gratitude must go to my advisor, Dr. Mohamed Wiem Mkaouer, for his assistance, guidance, fathomless inspiration, and motivational speech. His motivation words and meticulous scrutiny have immeasurably empowered me to successfully accomplish this research work. His faith enabled me to see a path of being unstoppable to accomplish a goal. I would like to extend my thanks to my thesis committee member, Dr. Ikram Chaabane for all of her constructive feedback to my thesis work.

I am thankful and appreciative to my academic adviser and thesis committee member, Dr. J. Scott Hawker, for his invaluable guidance and support, and I like to extend my deep thanks to all faculty members of the Department of Software Engineering at Rochester Institute of Technology for revealing all the secret recipes of software engineering domains throughout my period of my master's program.

I am forever indebted to my parents for their continuous support and unparalleled love. I would not be here, achieving my dream, without their faith in me and experiences that have made me who I am. I would like to extend my deep gratitude to my siblings for their sincere encouragement. My journey would have been impossible if not for my lovely family, and I devote this milestone to them.

These acknowledgments would not be complete without mentioning QA specialists, Hashem Alhariri, and Ibrahim Aldayel, for giving me an opportunity to develop my interest in software engineering during my work experience. Without their unique enthusiastic support, I would not have developed my interest in software engineering. For financial

aid, I am grateful for the Saudi Arabia for providing me with this wonderful opportunity to pursue master degree in the United States.

Last but not least, I am so thankful to everyone who has supported me emotionally during this educational journey, and I want to express my apologies for those whom I unintentionally forget to mention their names.

Abstract

Can feature requests reveal the refactoring types?

Sultan Fahad Almassari

Supervising Professor: Dr. Mohamed Wiem Mkaouer

Software refactoring is the process of improving the design of a software system while preserving its external behavior. In recent years, refactoring research has been growing as a response to the degradation of software quality. Recent studies performed an in-depth investigation in (1) how refactoring practices are taking place during the software evolution, (2) how to recommend refactoring to improve the design of software, and (3) what type of refactoring operations can be implemented. However, there is a lack of support when it comes to developers' typical programming tasks, including feature updates and bug fixes. The goal of this thesis is to investigate whether it is possible to support the developer through recommending appropriate refactoring types to be performed when the developer is assigned a given issue to handle. Our proposed solution will take as input the text of the issue along with the source code and tries to protect the appropriate refactoring type that would help in adapting efficiently the existing source code to the given feature request. To do so, we rely on the use of supervised learning. We start with collecting various issues that were handled using refactoring. This data will be used to train a model that will be able to predict the appropriate refactoring, given as input an Open issue description. We design a classification model that inputs a feature request and suggests a method-level refactoring. The classification model was trained with a total of 4008 feature request examples of four

refactoring types.

Our initial results show that this solution suffers from several challenges including the class imbalance: not all refactoring types are equally used to handle issues. Another challenge we detected is related to the description of the issue itself which typically does not explicitly mention any potential refactoring. Therefore, there will be a need for a large set of issues to be able to appropriately learn any patterns among them that would discriminate towards a given refactoring type.

Contents

Dedication . . . . **iii**

Acknowledgments . . . . **iv**

Abstract . . . . **vi**

1 Introduction **1**
1.1 Overview 1
1.2 Research Objectives 2
1.3 Proposed Approach 3

2 Background **5**
2.1 Software Refactoring 5
2.2 Software Requirement 5
2.3 Classification of Text 6
2.3.1 Machine Learning 6
2.4 Evaluation Metrics 7
2.4.1 Accuracy 7
2.4.2 Precision 8
2.4.3 Recall 8
2.4.4 F-score 8
2.5 Resampling Techniques 9
2.5.1 Random Oversampling 9
2.5.2 Random Undersampling 9
2.5.3 Synthetic Minority Oversampling Technique 9
2.5.4 NearMiss 9

3 Related Work **10**
3.1 Software Refactoring 10
3.2 Software Requirement 13

3.3 Classification of Text Documents 13

4 Approach 15

4.1 Overall Feature Request Framework 15

4.2 Feature Request Classification 16

4.2.1 Dataset 16

4.2.2 Data Pre-processing Techniques and Feature Modelling 18

4.2.3 Training and Prediction 20

5 Experimental Results 22

5.1 RQ1: To what extent our supervised learning could effectively recommend a software refactoring for new feature requests? 22

5.2 RQ2: What is the impact of resampling techniques on the classification performance ? 25

5.3 RQ3: What is the best performing resampling technique? 26

5.4 RQ4: How effective is our machine learning in recommending refactoring based on our baseline resampling approach? 27

6 Threats to Validity and Future Work 29

6.1 Threats to Validity 29

6.2 Future Work 30

7 Conclusion 31

Bibliography 32

List of Tables

3.1 Refactoring types. 12

4.1 The distribution of refactoring types [32] 17
4.2 The distribution of refactoring types in our dataset 17

5.1 Performance of multiclass classification without balancing techniques. . . . 23
5.2 explainable confusion matrix 24
5.3 The performance of balancing training set 26
5.4 comparison between multi-balance techniques on LSVC 27

List of Figures

4.1 Overall Classification Framework. 16
4.2 TF-IDF. 20

5.1 Confusion Matrix 24

Chapter 1

Introduction

1.1 Overview

Quality of software system has been considered as the main practice during software evolution to improve the software quality attributes i.g. the maintainability and performance. In order to improve software quality, software refactoring activities take place to optimize the intra/inter design of a software system with respecting its external functionally behavior [16]. Recent studies [11, 21, 42, 41, 8] have designed automated refactoring detection tools to provide a rich domain where researchers can learn (1) when software developers perform refactorings on the source code, (2) what type of refactorings are performed, (3) which class contains the performed refactoring operations by mining open source project repositories.

This refactoring activity is be necessary when there is a need for either a new software requirement or a functional enhancement termed as *feature request*(FR). The aim of the feature request is to support the continuous software evolution[20] for the enhancement purpose of projects. Open source projects need tracking systems to issue feature request and to maintain its software evolution; the following are tracking systems: GitHub Issue Tracker [1], Bugzilla [2], and JIRA [3]. The issued feature request demands developers to adapt the feature request in source code, leading to an immediate remedy request to perform software refactoring operations to enhance the internal software design.

---

[1]https://github.com/features
[2]https://www.bugzilla.org
[3]https://www.atlassian.com/software/jira

1.2 Research Objectives

The major concern for software evolution that feature requests threaten software internal deign and result software deficiency when developers receive feature requests to adapt them in source code, without taking into account the current design of the system. Thus, there are several challenges associated with the remedy of the deficiency. One of the challenges is that it subjectively requires a human effort to opt the software refactoring operations that can cure the deficiency introduced with the feature request adaptation. This human effort depends on manual inspection of software artifacts to locate where the portion of code should be refactored in order to adapt a feature request. However this manual process can be significantly time-consuming, especially when selecting what type of refactoring operation and where it should be performed. Feature requests target typically various files, and therefore, without having a concise view of the design, performing refactoring can have adversarial effects on design quality.

One potential solution to mitigate the human efforts would be to semi automatically recommend what type of refactoring should be applied for a given feature request. Since there is an Armada of previously performed refactorings for input feature requests, this represents a rich space for us to learn from it. Thus, supervised learning represents a potential solution for this problem. Therefore in this thesis, we consider the recommendation of the refactoring for a given per request as a supervised learning problem. We start with using an existing data set of various refactorings applied to the response to feature requests. We design a model that is trained and tested on this data set and we report the performance of our solution by breaking down its accuracy across all the classes (refactoring types). To better understand how this works, let us consider the following example:

> *"**split** PDFont encode As discussed in the dev@pdfbox.apache.org (thread : Questions about to Unicode Cmap).We need to **split** PDFont encode to get one **method** providing the string and one providing the cid."*

in this example, there is an intention to to split a method called *PDFontencod* resulting

a new method having a string variable and *cid* variable. This means that developers will perform *Extract Method* for the *PDFontencod* method, there fore we can teach our model with keywords e.g. split and method.

Due to the lack of just-in-time suggestion, the prime ultimate object of this thesis is to deign and build our machine learning model in order to recommend a refactoring operation for the given feature request. In our research work, we propose a different approach from our baseline work [32] by tackling the severe imbalance between classes with several resampling techniques. The effort of our study benefits quality assurance domains and increases the software quality.

To design our research, we define the following research questions:

Rereach Questions:

RQ1: To what extend our supervised learning could effectively suggest a refactoring operation based on feature request? We performed a comparative study between several classifiers to learn from feature request and solve multi-classes classification problem.

RQ2: Can resampling techniques help to improve the classification models? We applied oversampling balance technique on the imbalance data sets for our competitor classification algorithms to compare and evaluate them.

RQ3: What is the optimal resampling technique to improve the model performance? We performed several balance techniques on the optimal classifier to recommend a refactoring operation from learning feature request keywords.

RQ4: How effective is our machine learning in recommending refactoring based on our baseline resampling approach? We adapted our baseline model's balance technique in our approach of feature extraction.

1.3 Proposed Approach

In order to reduce human efforts in identifying the refactoring operations during software evolution, we combine Natural Language Process(NLP) and supervised learning to formulate the suggestion of refactoring operation as a multi-class classification problem. This

approach consumes a set of feature requests, labeled with 4 classes, i.e., method level refactorings, collected by our baseline work [32] as input to learn several classifiers. One benefit of using a multi-class classifier is that at least the model can be taught with the most common refactoring operations (*Extract Method, Inline Method, Move Method, and Rename Method*). These refactoring operations are recognized with a the purpose of providing an improvement in software design quality. The dataset will be processed to obtain only single label that is associated with each feature request. Then, it will undergo data preprocessing and feature extraction techniques in order to propose a collection of features for each class. This operations will optimize our model ability to learn from keywords. Then the dataset is splited into training and testing sets where the training set will be balanced to tackle the imbalance issue between classes. The evaluation of model relies on the most popular performance indicators: Recall, Precision, F1-score and Accuracy.

The structure of this thesis is presented as follows. chapter 2 introduces key information related to refactoring activity, requirement, and text classification as well as an additional important background knowledge to provide better understandability for this work. Chapter 3 explains our related work that motivated this work. Chapter 4 demonstrates our approach and its data preparation, balancing technique, and our model setup with its classifiers' selection and evaluation. In chapter 5, our experiment and findings are discussed. We provide an insight of threats to the validity of this work and our recommendation in chapter 6, and then we conclude our work in 7.

Chapter 2

Background

2.1 Software Refactoring

Software system is structured from a combination of elements: classes, methods, variables and attributes. Although if the syntactic design of software structure is poorly implemented, software flaws would be introduced as these flaws identified in [16], which 72 software refactoring operations helps to improve the software internal structure. As an example of one the common software flaws is that God class having enormous amount of code placed in a single class, caused due to the merge several classes unintentionally. The proper solution to fix this code smell could be done by *Extract Class or/and Move Methods, Rename Package/Class* [16]. Recently most of studies have designed different approaches and tools to fix software flaws by refactoring operation, one of which JDeodorant [15, 14] helps to detect software flaws then suggest a proper solution to fix the given software flaws. Their approach, a given single class, is analyzed by a hierarchical agglomerative clustering algorithm and Jaccard distance as the distance metric. However, most of the proposed approaches of refactoring solutions have a lack of suggesting refactoring operation just-in-time, which leads research to discover optimal solutions.

2.2 Software Requirement

In the world of software evolution, the identification of the software requirement specification aims to deliver a sound software function and provide a clear understandability for software developers, being either pre-requirements or post-requirements. Requirement

engineering (RE) has been considered as a main part of software evolution to success in delivering feature request. While the success of FR delivery for software project requires a requirements traceability system, Open source projects have started to use requirements traceability system like GitHub [1], Bugzilla [2], and JIRA [3]. The requirements traceability system has shown to be beneficial in software engineering domains e.g. software refactoring [30]. Since software evolution has a thrive of increase the quality of FR delivery, Several works [5, 17, 10, 25] have discovered the common issues about the practice of requirement process. One study [17] found that there are the lack of common requirements definition and the lack of misunderstandability among developers because they have different perception. Hence, the final delivery of feature request could introduce software flaws which will cost human effort and time to cure it. This issue can be ideally cured by recommending refactoring operation to developers before transfer the document requirement into functional/none-functional requirement in the software system, facilitating software engineer to opt the optimal solution.

2.3 Classification of Text

In this section, we revise various classification techniques that can be applied on textual data to suggest their types.

2.3.1 Machine Learning

Machine learning is an approach to learn automatically from a collection of data i.g. image, video or text in order to serve different purposes. In terms of learning from data set, there are approaches grouped as supervised and unsupervised learning, where supervised learning or classification learns from information that maps the information as an input with a label representation as an output. In unsupervised technique, learning mechanism attempts

[1] https://github.com/features
[2] https://www.bugzilla.org
[3] https://www.atlassian.com/software/jira

to learn and understand the given data by cluster them into several groups based on a certain similarity. Both learning techniques works for different types of data i.g. image, number, video and text. Machine learning algorithms are now widely shown a provement to solve problems in several domains such as speech recognition, self-driven and text classification [7]. As text classification in the supervised learning, the classification model receives a collection of labeled *training* texts to develop a classification model's rules in order to increase its ability of recommendation. This collection of labeled data can be treated as binary/multi-class problem, where in binary problem (i.g. the label would be like 0 or 1). For multi-class problem, the set of data can have more than 2 classes—— each class represents a large number of text.

In this thesis, We focus on supervised technique to learn multi-class by utilizing the benefit of Natural language process(NLP), used for text classification. Specifically, we are developing an automated classifier to suggest a refactoring type for the given feature request.

2.4 Evaluation Metrics

For the purpose of evaluating the classifications performance, we are looking at several classification measurements needed to be reviewed.

2.4.1 Accuracy

It is a metrics that help to measure the overall performance of a classification model. The following formula shows how accuracy of a classification model can be calculated.

$$Accuracy = \frac{tp + tn}{tp + fp + fn + tn} \tag{2.1}$$

Where TP, TP, FP, and FN are described as following:

- True positives (tp): help to sum the number of elements correctly classified for the positive class

- True negatives (tn): count correct classifications for the negative class.

- False positives (fp): help to sum the number of unpredictable items for the positive class

- False negatives (fn): count the number of unpredictable items for the negative class.

As equation, the sum of correct predicted items are divided by the sum of all classes — correct and incorrect.

2.4.2 Precision

It is a metrics that help to measure accuracy of the positive class predictions in the classification model, showing the ratio of correct classifications out of the positive class predictions. The following formula shows how precision of a classification model can be calculated.

$$Precision = \frac{tp}{tp + fp} \tag{2.2}$$

2.4.3 Recall

It is a metrics that help to calculates accuracy of model predictions for true positive cases, showing the ratio of correct classifications as positive out of the positive class predictions. Recall equation are shown as a following.

$$Recall = \frac{tp}{tp + fn} \tag{2.3}$$

2.4.4 F-score

It is a metrics that help to find the ratio of harmonic among precision and recall, known as as F-measure or F1-score. This equation provides a trade-off between precision and recall, the equation of F1-score shown as following:

$$F - score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{2.4}$$

2.5 Resampling Techniques

The behavior of machine learning techniques could impact on performance of the model when the classification dataset has imbalanced class distribution. The reason is that several machine learning algorithms are implemented to learn from an equal number of distribution for each class. When there is imbalance distribution, the performance of the model often mislead the algorithms by learning from the majority of classes and ignoring the importance of minority. Thus, several study have proposed balancing techniques in order to provide an equal priority to each class. Some of widely used balancing methods are shown as following:

2.5.1 Random Oversampling

This Method duplicates examples from the minority class to have an equal distribution class of majority by oversampling the data set.

2.5.2 Random Undersampling

This approach of balance technique is to randomly delete examples from the majority class in order to balance it with the minority of class distribution.

2.5.3 Synthetic Minority Oversampling Technique

Synthetic Minority Oversampling Technique (SMOTE) is another approach of oversampling technique; SMOTE generate examples for the minority class by selecting examples closed in the feature space.

2.5.4 NearMiss

Near Miss is a method that selects examples from the majority class by determining the most near distance on average to the closest examples from the minority one.

Chapter 3

Related Work

3.1 Software Refactoring

Several research papers [42, 41, 8, 21, 11, 43] designed an automatic approach to identify refactoring operations that have been implemented during coding activity, which inspires quality community to investigate more about developers' practices in refactoring activities to add value in the improvement of the software quality. In the work of RefactoringCrawler [11], their an automated tool detects refactoring operation in java languages which is designed as an Eclipse plugin called RefactoringCrawler. Their approach detects refactoring based on similar fragments in source files between two version. In Ref-finder tool [21], their work has produced an Eclipse plugin called to determine refactoring occurred in two version of java projects by utilizing the benefit of s the syntax tree to compute structural change-facts. However, these tools limits researchers from being free in terms of selecting their preference of platform since they are an Eclipse plugins.

An easy way to classify most refactoring papers is by which refactoring method they focus on. Five main categories were identified: Extract Class, Extract Method, History, Move Method, and Combination. Out of the five, Extract Class, Extract Method, and Move Method are the most similar, as they are all traditional refactoring implementations. History is unique as it focuses on using version history to identify refactoring opportunities, making its papers more closely related to detection than correction. Combination papers touch on several refactoring methods.

The papers centered around Extract Class and Extract Method were the most grounded.

They tended to build upon the existing Extract frameworks, and proposed improvements to established approaches that seemed genuinely well-researched and tested, if occasionally somewhat situational. Move Method papers emulated this practice as well, but had the widest range of improvements of the three [18]. While one paper might focus on efficiency, the other might propose a way to rank refactoring candidates. History papers had an interesting similarity in that almost all of them utilized HIST in some way, shape, or form [38, 33]. This worked in their favor, providing a common background for any reader interested in that particular domain, as well as a starting point for personal experimentation or development.

However, the Combination papers were the most unique. They benefited from not pigeonholing themselves into a single refactoring method. This lack of a singular focus did not make them too generic or large, because they tended to focus on something other than approaches to the established refactoring methods; they focused on prioritization of recommendations. One paper suggested techniques for recommending refactoring methods in large systems [6]. Another paper in particular presented an approach to prioritizing code smells for refactoring, somehow managing to provide significant information relevant to the domain of code smells in a refactoring methods paper [44]. This was one paper that remained relevant in both literature reviews. These papers provide a valuable service in establishing common researching ground between the domains of code smell detection and refactoring methods.

As more as software community become interested in improving the quality aspects by learning from the detected refactoring operations, it produces more competitive tools. One independent tool called RefDiff in the work of has proposed their approach of detecting java refactoring operations [42] then extended their approach to collect refactoring activities in two well-known program languages (C and JavaScript) [41], then including Go parser to understand the operated refactoring in Go language by developers [8]. One well engineered tool called Refactoring Miner from the work of [43], they has proposed method to detect refactoring activities from two commits pushed in repository, which their

tool can be used as API (An application programming interface) for java projects. Their tool provides us confidence in the tool's detection since it has high scores in precision and recall compared with other state-of-the-art refactoring detection tools. Their tools detects 14 types of refactoring operations that are provided in the table 3.1.

Although many research papers mainly participated in improving the criteria of a refactoring recommendation with the aim of increasing the quality of detection, the refactoring recommenders a deficiency in their recommendation approach. This deficiency motivated Niu et al. [30] participate in refactoring area to provide a sound detector with a new approach (i) to recommend which type of refactoring should be applied on source code and (ii) to locate where the refactoring should be refactored by using requirements traceability. The interest of recommending refactoring based on feature request among researchers has been increased, which leads to design multi-class/multilabel learner from feature request in order to recommend one of 14 common refactoring operations [32]. This work is our baseline work where we built different model to solve a single-label/ multi-class problem.

Table 3.1: Refactoring types.

Refactoring Type	**Description**
Extract Interface	To Create an interface class with common operations from the current class
Extract Method	To move a portion of code from a method to initial a new method fit the moved method
Extract Superclass	To Create a shared superclass that contains all the identical fields and methods .
Inline Method	To replace method invocation with the method's content.
Move And Rename Class	Two operations implemented: to move a class to different package and update the class
Move Attribute	To move an attribute from one class to another
Move Class	To move a class from one package to another
Move Method	To move a method from one class to another
Pull Up Attribute	To pull the identical attributes from sub-classes to superclass
Pull Up Method	To pull the identical methods from sub-classes to superclass
Push Down Attribute	To push an attribute used by only one class from superclass to its sub-class
Push Down Method	To push the method used by only one class from superclass to its sub-class
Rename Class	To change the name of a class identification
Rename Method	To change the name of a method identification

3.2 Software Requirement

One earlier work [30] has proposed an approach to ensure that feature request are efficiently adapted in the source code by designing traceability-based refactoring recommendation. Their approach (1) identify the portion of code that should be refactoring by utilizing the benefit of the developing requirements and the source code to accurately locate where the software should be refactored. (2) determine the candidate refactoring operation in order to operate it to prevent software flaws from being introduced. Another study[31] inspired by [30] has proposed a recommendation approach that utilizes requirements traceability and code metrics (i.e., cohesion and coupling) to provide refactoring operations in order to enhance the software design. These [31, 30] both works have shown a proven the important role of understandability of the candidate refactoring solution during the requirements, leading to reduce the human effort and time-consuming in fixing software flaws. The main different between our work and our baseline work is that our work leverages feature requests as well to automatically suggest refactoring solution with a help of multi-class approach. We feed our machine model with single-label/multi-class dateset as suggested refactoring operations in the table 4.1.

We opt these types because mining tools i.g. [43] to extract refactoring activity has an ability to identify only 14 types used by our baseline to collect their dataset. Despite of our model different from our baseline model [32], we utilize their dataset for a seek to compare our findings.

3.3 Classification of Text Documents

Recent works [23, 4, 1] have proposed an automated approaches on the commits to discover the possibility of optimizing the refactoring activity in order to reduce the human effort and improve the quality aspects of the software system by a help of classification model. In the work of Alomar et al. [2], their motivation was to recommend a refactoring

operations for method level by learning from the commit message. Their work was challenged in different classifiers (Random Forest (RF), Gradient Boosting Machine(GBM), Logistic Regression (LR), One-vs-All strategy, Support Vector Machine (SVM), Locally Deep SVM (LD-SVM), Averaged Perceptron Method (APM), and Bayes Point Machine (BPM),Logistic Regression(LR), Random Forest (RF), Decision Jungle (DJ), and Neural Network(NN)). Their findings in thier best model has shown that their model has an ability to predict the candidate refactoring types based on learning from terms frequency in GBM the highest average F-score of 0.59.

Another work [32] has utilized the benefit of NLP to improve their previous work [31] by approaching just-in-time method that could suggest one of 14 refactoring operation based on feature request. Their approach take feature request as input to recommend a refactoring operation i.g. Move Method or Extract Class. These refactoring was extracted from open source projects by a help of Refactoring Miner tool [43], then they minied requirements traceability to extract feature requests from the given open source projects. Their solution was to solve multi-label/mulit-calss problem in order to increase the accuracy of refactoring predection, shown that their recommending model scored with accuracy of 0.83. The main different between our work and our baseline work is that our work leverages feature requests as well to automatically suggest refactoring solution with a help of multi-class approach. We feed our machine model with single-label/multi-class dateset.

Chapter 4

Approach

In this chapter, we first present the overall framework of the feature request-based refactoring recommendation, then we detail each step of this process.

4.1 Overall Feature Request Framework

Our approach addresses a multi-class classification problem where the goal is to predict the appropriate refactoring type given a feature request. The proposed solution to automate the refactoring recommendation follows a series of successive steps as depicted in figure 4.1. The input data represents a collection of feature requests labeled with 14 refactoring types, which was publicly available on GitHub [https://github.com/nyamawe/FR-Refactor]. It is worth noting that the labeling process is based on efficient mining tools (i.e., RefDiff [42] and RMINER [43]) providing high scores of precision and recall reaching 98% and 87% for RMINER, and 76% and 86% for RefDiff respectively. Before splitting our input data into train and test sets, the first step to follow is to pre-process the natural language texts of the feature requests so as to clean the learning data. The following step concerns the feature extraction allowing the conversion of textual data into a numerical representation suitable for training classifiers. Then, being aware of the class imbalanced learning data, we applied sampling techniques to inhibit bias of classifiers towards the most representative classes. The final step trains machine learning classifiers based on the sampled data and evaluates the resulting learners on regarding new requirements. We detail each step of the presented workflow in the next section.

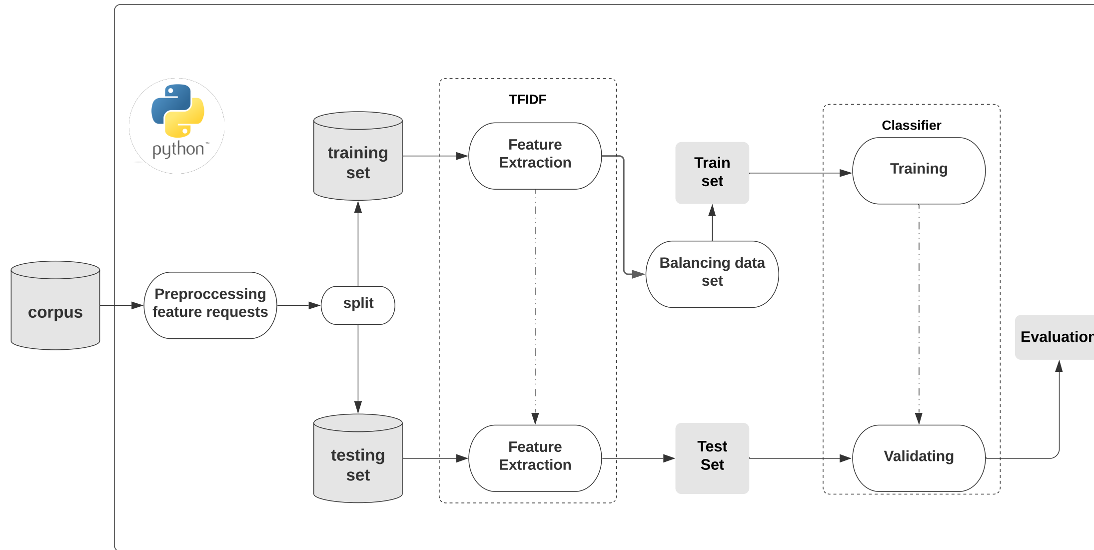

Figure 4.1: Overall Classification Framework.

4.2 Feature Request Classification

4.2.1 Dataset

Our learning process relied on the large public dataset reported by Nyamawe et al. [32]. It is a rich source of data collected from 55 open source Java projects gathering altogether 18,899 feature requests. Thus, our choice for such data relies on the variety of the covered domains, the public availability, the large number of training instances allowing an improved learning, and the data integrity. In fact, the included feature requests were annotated by two refactoring detection tools among the best performing state-of-the-art ones (i.e., RefDiff [42] and RMINER [43]). At first, this dataset was introduced in the context of multi-class and multi-label classification when refactoring may be one of 14 values and one feature request may be associated with more than one type of refactoring. The data distribution was displayed in Table 4.2, where rows are grouped by the refactoring subject (method, attribute, class or interface) and sorted in the descending order of the class size. In our study, we focus on the most common classes, which are mainly related to the method-based refactoring as the most frequent subject. Thus, our learning data cover

feature requests which have been resolved through one of the following refactoring types: Extract method, Rename method, Move method or Inline method. Under each category, it is worth noting that each feature is used once so as to handle a single label classification task. Our data distribution is summarized in Table 4.1. Excluding the multi-labeled feature requests justifies the differences in sizes of the same class shown in both Tables 4.2 and 4.1.

Table 4.1: The distribution of refactoring types [32]

Refactoring Type	**No.**
Extract Method	3067
Rename Method	1592
Move Method	303
Inline Method	298
Pull Up Method	50
Push Down Method	17
Move Attribute	84
Pull up Attribute	28
Rename Class	310
Extract Super-class	46
Move And Rename Class	59
Move Class	367
Extract Interface	47
Push down Attribute	9

Table 4.2: The distribution of refactoring types in our dataset

Refactoring Type	**No.**
Extract Method	2463
Rename Method	1111
Move Method	206
Inline Method	228

4.2.2 Data Pre-processing Techniques and Feature Modelling

Pre-processing learning data is an important step to enhance the classification performance. To this end, we perform common preprocessing techniques on the introduced feature requests using Python Natural Language Processing Toolkit (NLTK) of the scikit-learn API *scikit-learn [34].*

Stop-word Removal

Meaningless common English words such as *is, am, are, if, for, the, etc.* are removed to clean data from noise. In this study, we use the stop-word list [39].

Lemmatization

Lemmatization in linguistic is the process of mapping a word to its root form. For example, the words ”studying”,”studies”, and ”studied” are mapped to the word ”study” by WordNet for mappings[36, 35, 13]. Such strategy treats similarly different words having the same base, which reduces the complexity of the natural language processing activities. In our study, we rely on lemmatization as it would know that the word better is derived from the word good, and hence, the lemme is good. However, in stemming, this is not possible. So, it suffers from over-stemming or under-stemming. For example, the word ”better” could be reduced to either ”bet”, or ”bett”, or just retained as better. Therefore, we use lemmatization to process the text of the issue. This reduces the search space to only fewer keywords that can be later used to indicate a given refactoring type over another.

Lowercasing

Lowercasing is the process of making each letter lowercased so as to handle similarly the same words with different case. For example, the three words ”Query”, ”QUERY”, and ”QUery” map to the same lowercase form ”query”. Such strategy solves the sparsity issue and allows the removal of all the stop-words.

Noise Removal

Special and punctuation characters like '!', '?', '*', etc. are removed before training with the natural language.

N-Gram Feature Selection

An N-gram means a set of N words in an textual example, extensively utilized in natural language processing tasks. The N-gram is represented as a set of co-occurring words to help a machine a better understand by looking surrounding the context and moving one word forward. As an example for 2-gram (bigram), if we have an example "read this book soon", the bag of word would have ["read this","this book","book soon"]. The window of the grams can be extended to be tri-grams or more.

Length-Frequency

It is an approach that studies the length of words by counting its letters[28]. It was found that shortest words were the most frequent. We remove all long words, resulting in reducing all function name since they would be large i.g "*AddressingSubmissionInHandler*". We found that using this technique when processing messages from commits and issues allows the removal

Feature Modelling

The preprocessed features, written in natural language, need to be converted into numerical representation so that they can be involved easily into the classification process. Therefore, each bag of words composing a feature request is associated to a vector of weights referring the importance of the corresponding token in the corpus. In our study, weights are computed using one of the most common techniques used for vector space modelling, which is the TF-IDF quantifier [40]. As shown in Listing 4.2, this measure joins the term frequency (TF) and inverse document frequency (IDF), where $TF(t, d)$ defines the number of times the term t appears in a document d, and $IDF(t, d)$ defines the number of documents in the

corpus D that contains the term t [32].

$$\mathbf{tf}(t, d) = \frac{f_d(t)}{\max\limits_{w \in d} f_d(w)}$$

$$\mathbf{idf}(t, D) = \ln\left(\frac{|D|}{|\{d \in D : t \in d\}|}\right)$$

$$\mathbf{tfidf}(t, d, D) = \mathbf{tf}(t, d) \cdot \mathbf{idf}(t, D)$$

$$\mathbf{tfidf}'(t, d, D) = \frac{\mathbf{idf}(t, D)}{|D|} + \mathbf{tfidf}(t, d, D)$$

$$f_d(t) := \text{frequency of term t in document d}$$

$$D := \text{corpus of documents}$$

Figure 4.2: TF-IDF.

4.2.3 Training and Prediction

Once the data is cleaned and well represented in a vector space model, it is therefore subject to machine learning algorithms for training and prediction. Our classifier handles a multi-class problem which predicts one of the four method-based refactoring mentioned above (i.e., Extract method, Rename method, Move method and Inline method), given a feature vector. To train the classifier, we split the whole dataset into training and testing sets with a 70% and 30% distribution, respectively. Since the class distribution is still imbalanced as shown in Table 4.1, we perform sampling techniques on the training data so that to avoid the bias of the classifier towards the majority classes (i.e., Extract Method and Rename Method). Therefore, all the classes are equally represented through one of four sampling strategies : two under-sampling methods (RUS and NearMiss [12, 26]) and two over-sampling ones (ROS and SMOTE [9, 12]). RUS and ROS stand for Random Under-Sampling and Random Over-Sampling, respectively; whereas SMOTE signifies Synthetic

Minority Oversampling Technique. Each technique could have an impact or effectiveness on the model performance. Several research have explored the performance of balancing techniques; the work of [12] showed the under-sampling approach reduced the misclassification costs while the over-sampling approach in our baseline work [32] improved their model ability to recommend the refactoring type. Based on balanced training data, different candidate learning algorithms may be used. There is no a winner learner which works best for every problem. In fact, its performance depends on many factors such as the size and the structure of the data, the relevance and the integrity of the learning instances, the linearity of the problem, etc. Hence, we perform a comparative study of six common learning algorithms suitable for handling a multi-class NLP task. The candidate algorithms are Linear Support Vector Machine (LSVM)[45], Logistic Regression (LR)[22, 3], Neural Network (NN)[19], Random Forest Decision Tree (RFDT)[37], Multinomial Naive Bayes (MNB)[27], and Stochastic Gradient Descent(SGD)[29]. The performance of each algorithm is evaluated on real hold-out set of feature requests according to common statistical measures: *Precision* 2.2, *Recall* 2.3, *Accuracy* 2.1, and *F1-Score* 2.4. The experimental results of the comparative study are presented in chapter 5.

Chapter 5

Experimental Results

In this chapter, we evaluate the effectiveness of our approach to recommend the appropriate refactoring type for new users' requirements. All experiments were performed using Python scikit-learn and NLTK API. The evaluation of the already mentioned classifiers (i.e., Linear Support Vector Machine(LSVM), Logistic Regression(LR), and Neural Network(NN), Random Forest Decision Tree (RFDT), Multinomial Naive Bayes(MNB) was in response to four research questions:

RQ1: To what extent our supervised learning could effectively recommend a software refactoring for new feature requests?

RQ2: What is the impact of resampling techniques on the classification performance ?

RQ3: What is the best performing resampling technique?

RQ4: How effective is our machine learning in recommending refactoring based on our baseline resampling approach?

We present our experimental results to study each RQ in the following subsections.

5.1 RQ1: To what extent our supervised learning could effectively recommend a software refactoring for new feature requests?

To answer this question, we trained the above mentioned classifiers by using the parameter setting as employed by [32] based on their source code for SGD while relying on the default parameter setting of scikit-learn API for the rest of learners (LR, RFDT, MNB and

LSVM). For NN, we set the hyperparameters with[solver='adam',alpha=0.0001,hidden-layer-sizes=(35), activation="tanh",shuffle=True] after testing a sum of parameters.

The input data represents 4.008 feature requests which were cleaned using feature engineering processes. Reminding that the class label belongs to four method-based categories as shown in Table 4.1. The learned classifiers are therefore evaluated using average accuracy, precision, recall and F1-measure. We present the performance evaluation results in Table 5.1. As it can be shown, RFDT, LR and MNB are the best-performing classifiers in terms of Accuracy and Recall with an average reaching 60%. In contrast , the remaining classifiers showed the best performance in terms of F-measure. Overall, performances of the 6 used classifiers are close. We opt for LSVM for further analysis since it shows the best performance in terms of F-measure firstly and Accuracy secondly. Thus, we draw the resulting LSVM's confusion matrix as depicted in Table 5.2. It presents the percentage of True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN) for each class. TP are listed on the diagonal. As it can be shown, the Extract Method class was the best predicted class (TP Rate reaching 62%) whereas the Move Method was the worst predicted one (null TPR). From Table 4.2, it can be seen that the best performed class, Extract Method, corresponds to the majority one (61% of the whole dataset) whereas Move Method is associated to the least represented class (only 5% of the whole dataset). In fact, standard learning algorithms are always shown to be biased towards the majority class because of the lack of data to train efficiently the classifiers.

Table 5.1: Performance of multiclass classification without balancing techniques.

Classifier	**Accuracy**	**Precision**	**Recall**	**F1-measure**
Logistic Regression	0.60	0.47	0.60	0.46
Random Forest Decision Tree	0.60	0.47	0.60	0.47
Multinomial Naive Bayes	0.60	0.36	0.60	0.45
Neural Network	0.56	0.48	0.56	0.50
Linear Support Vector Machine	0.57	0.48	0.57	0.50
Stochastic Gradient Descent	0.54	0.47	0.54	0.50

To address this class imbalance problem, we performed sampling techniques looking for balancing the class distribution either by under-sampling the majority class or by over-sampling the minority one. We study the impact of such strategies in the following section.

Predict \ Target	Extract	Inline	Move	Rename
Extract	692 62%	2 40%	1 33%	103 53%
Inline	65 6%	2 40%	1 33%	4 2%
Move	73 7%	0 0%	0 0%	21 11%
Rename	290 26%	1 20%	1 33%	67 34%

Figure 5.1: Confusion Matrix

Table 5.2: explainable confusion matrix

	Extract Method	**Inline Method**	**Rename Method**	**Move Method**
TP	692	2	67	0
TN	97	987	695	1236
FP	106	70	94	292
FN	428	3	128	3

5.2 RQ2: What is the impact of resampling techniques on the classification performance ?

Due to the severe imbalance dataset, as observed in RQ1, we performed further investigation to balance only the training set for all classifiers with a well-known technique called Oversampling. This oversampling approach was introduced in our baseline work [32] showing a signification improvement in their model's ability by balancing entire dataset. Unlikely, our experiment in balancing only training set shown in the table 5.3. The result obtained no effect on improving the performance of RFDT, LR , and LSVC with F1-score of 0.50, 0.50, and 0.51 on average respectively. The scores of NN and SGD had a slight improvement on average except their F1-score having 0.50. Unlikely, the prediction performance of MNB to classify the refactoring type slightly decreased with accuracy, recall, and f1-measure of 0.40, 0.40, and 0.42 , respectively except precision having an improvement with 0.46. Due to a slight improvement on average for classifiers, we validated their feature selection to understand and to explain how the model behavior works. We leveraged LSVC to extract top 10 features from each refactoring type. We found that our bag of words had 24 common texts occurred in multiple refactoring type, 5 of which are overlapped among all refactoring operations. As recognized from the common terms (*"use","add","creat","support","new"*) that shared in labels(Extract method, Rename Method, Inline method and move method),"use" term; for instance, was mentioned in Extract method 1634 times, leading to confuse the model with a strong enforcement. This phenomenal changed the direction of an optimal performance due to relying on Extract Method's features.Overall, since LSVC was the only classifier shown an improvement on its results with balancing technique among its competitor classifiers, we leveraged LSVC to examine its prediction of refactoring type based on feature requests with additional balance techniques.

Table 5.3: The performance of balancing training set

Classifier	Accuracy	Precision	Recall	F1-measure
Logistic Regression	0.52	0.48	0.52	0.50
Random Forest Decision Tree	0.58	0.49	0.58	0.50
Multinomial Naive Bayes	0.40	0.46	0.40	0.42
Neural Network	0.57	0.48	0.57	0.50
Linear Support Vector Machine	0.55	0.49	0.51	0.51
Stochastic Gradient Descent	0.55	0.48	0.55	0.50

5.3 RQ3: What is the best performing resampling technique?

Apparently, our dataset has imbalanced class instances, causing a minclassification for minority instances. As desired to rule the behavior of our model, we performed four popular balance techniques to reveal its effectiveness with class-imbalance problems in order to validate and select the optimal one with a better chance to learn about all refactoring types, not only those with high target frequency. This investigation was motivated to understand whether the model could obtain the ability to identify the refactoring operation from the balanced feature request or not. We utilized of the following balance techniques (oversampling, undersampling, SMOTE, and NearMiss) on LSVC in order to evaluate their effectiveness. As table 5.4 revealed it behavior in the all candidate techniques, we noticed that when the minority of the class instances is increased to match the number of the majority ones, the ability of the model to determine how to predict a class from a given feature request increased as oversampling and SMOTE techniques shown. The approach of undersampling and NearMiss techniques introduced bias to the model. This provided a clue that the majority of instances can tackle somehow the model's understandability as shown in the study of [24]. They conducted a study that explored the effectiveness of the distraction of class instances for training and testing data sets——their findings stated that the model performance can be extended to obtain the capability of learning from spliting the dataset

with ratio of 80 for training and 20 for testing. In our experiment, when we performed a similar investigation to determine the optimal class distribution for balancing training set, we found that the distribution ratio of 90 for training can increase the learning ability of our model from 0.51 to 0.53 for F1-score.

Table 5.4: comparison between multi-balance techniques on LSVC

Balance Technique	**Accuracy**	**Precision**	**Recall**	**F1-measure**
Oversampling	0.55	0.49	0.51	0.51
Undersampling	0.29	0.47	0.29	0.34
SMOTE	0.54	0.49	0.50	0.50
NearMiss	0.25	0.47	0.25	0.29

5.4 RQ4: How effective is our machine learning in recommending refactoring based on our baseline resampling approach?

Our baseline work [32] showed that the optimization in their classification model with their balance technique(oversample the training and testing data sets as one entity) improves accuracy to 0.83 for SVM (LSVC)—– our LSVC model obtained the highest result among its competitor algorithms as our baseline work found. We hence adapted their optimization approach of balance technique in our LSVC model that it has its processed and feature extraction approach to understand how effective their approach. Our findings from the combined techniques stated that the classification of refactoring type from our model has an ability to behave precisely with F1-score and accuracy of 0.90 on average. Our phases in preparing the features before feeding them into the model declares that we had a sound optimization in the model capability to extract and clean the data sets. Despite of the remarkable effectiveness with overall high F1-score,the reflection of the actual model quality is uncertain due to the dataset overbalancing causing overfitting. This explains that we only

need to balance the training set to prevent the model from having en error introduced.

Summary. In terms of suggesting refactoring operation for a given feature request, our LSVC model performance with the distribution ratio of 90 for balanced trained set had a capability to recommend a refactoring type for developers in a favor of just-in-time with F1-score of 0.53.

Chapter 6

Threats to Validity and Future Work

This chapter introduces the potential internal and external threats to the validity of our study. We also address our tentative future plan that could increase the performance of our model.

6.1 Threats to Validity

One **internal threat** is that our study used a subset of dataset that was collected by our baseline work [32], which was explored to verify how the NLP could recommend one of the four refactoring types in the method level for the given feature request. This extraction of subset from dataset could mislead the classification prediction since they were a few examples. Another possible internal threat could be in our approach that excluded cross-validation algorithm. This algorithm could minimize the error variation of the classification learning evaluation by splitting the distribution into several folds. Additional threat is that we omitted stratify approach that helps to equally distribute the quantity of examples, although we ensured that our data distribution was equally split without stratify approach. Finally, the obtained dataset includes a large number of feature requests having unintentional need to introduce optimization for the design structure resulting a poor description for feature requests. As **external threat**, our model heavily relies on NLP libraries (i.g. NLTK and scikit-learn), although they are well-known and the most popular APIs for NLP.

6.2 Future Work

This study sheds lights on the capability of the text classification model to recommend refactoring operations in method level based on learning from a feature request. Our future work should extend and include all 14 types of refactoring. This would help our tentative plan to cover the quality aspects of development. Additional goal is to cure the severe imbalance dataset by collecting a large number of examples. Then, using more resampling strategies e.g. hybrid resampling techniques with more models to potentially improve the performance accuracy. This will help the classification model when each class has enough unique term frequency to identify themselves among other classes. For better learning, it would be ideal to collect well-descriptive feature requests in order to improve the way of how the model learn. Also, we should consider all overlapping keywords as stop words to help in better discriminating between the classes. Another plan is to included cross-validation algorithm to reduce the error variation. Last major future work would be designing hierarchical classifier to verify whether the feature request needs to be refactored or not and then suggest the refactoring type. In fact, some feature requests can be adapted into code without perform any refactoring operation.

Chapter 7

Conclusion

Software engineer receives new feature requests to adapt them in the source code, hence, some refactoring operations are required to serve the design quality requirement e.g. maintainability. This thesis presents an automated approach with help of just-in-time method to recommend a refactoring type to developers for optimal adaption of feature request. The designed approach is a machine learning model that receives feature request as an input to analyze it then suggest a refactoring type as an output. We perform an evaluation of five different multi-class classifiers. We first compare between classifier models without preforming balance techniques and then balance all classifiers in different experiments to track the optimization of the model performance. Finally, we validate several balancing techniques for Linear Support Vector Machine model. Our findings indicate that our approach with a help of our Linear Support Vector Machine model performance with the distribution ratio of 90 for balanced trained set increase the result of F1-score to reach 0.53.

Bibliography

[1] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 51–58. IEEE, 2019.

[2] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, 171:110821, 2021.

[3] Galen Andrew and Jianfeng Gao. Scalable training of l 1-regularized log-linear models. In *Proceedings of the 24th international conference on Machine learning*, pages 33–40, 2007.

[4] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius Durelli. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, 2020.

[5] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE transactions on software engineering*, 28(10):970–983, 2002.

[6] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*, pages 387–419. Springer, 2014.

[7] Avrim Blum, John Hopcroft, and Ravindran Kannan. *Foundations of data science*. Cambridge University Press, 2020.

[8] Rodrigo Brito and Marco Tulio Valente. Refdiff4go: Detecting refactorings in go. In *Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 101–110, 2020.

[9] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[10] Alexander Delater and Barbara Paech. Tracing requirements and source code during software development: An empirical study. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 25–34. IEEE, 2013.

[11] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, page 404–428, Berlin, Heidelberg, 2006. Springer-Verlag.

[12] Chris Drummond, Robert C Holte, et al. C4. 5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling. In *Workshop on learning from imbalanced datasets II*, volume 11, pages 1–8. Citeseer, 2003.

[13] Christiane Fellbaum. Wordnet. In *Theory and applications of ontology: computer applications*, pages 231–243. Springer, 2010.

[14] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of feature envy bad smells. In *2007 ieee international conference on software maintenance*, pages 519–520. IEEE, 2007.

[15] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1037–1039. IEEE, 2011.

[16] Martin Fowler. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.

[17] Orlena CZ Gotel and CW Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of IEEE International Conference on Requirements Engineering*, pages 94–101. IEEE, 1994.

[18] Ah-Rim Han, Doo-Hwan Bae, and Sungdeok Cha. An efficient approach to identify multiple and independent move method refactoring candidates. *Information and Software Technology*, 59:53–66, 2015.

[19] Lars Kai Hansen and Peter Salamon. Neural network ensembles. *IEEE transactions on pattern analysis and machine intelligence*, 12(10):993–1001, 1990.

[20] Shalinka Jayatilleke, Richard Lai, and Karl Reed. A method of requirements change analysis. *Requirements Engineering*, 23(4):493–508, 2018.

[21] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *in Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, ser. FSE '10*, pages 371–372. Online]. Available: http://doi.acm.org/10.1145/1882291.1882353.

[22] David G Kleinbaum, K Dietz, M Gail, Mitchel Klein, and Mitchell Klein. *Logistic regression*. Springer, 2002.

[23] Rrezarta Krasniqi and Jane Cleland-Huang. Enhancing source code refactoring detection with explanations from commit messages. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 512–516. IEEE, 2020.

[24] D. T. Larose. *Case Study: Modeling Response to Direct Mail Marketing*, pages 265–316. 2006.

[25] Anas Mahmoud and Nan Niu. Supporting requirements traceability through refactoring. In *2013 21st IEEE International Requirements Engineering Conference (RE)*, pages 32–41. IEEE, 2013.

[26] Inderjeet Mani and I Zhang. knn approach to unbalanced data distributions: a case study involving information extraction. In *Proceedings of workshop on learning from imbalanced datasets*, volume 126. ICML United States, 2003.

[27] Andrew McCallum, Kamal Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Citeseer, 1998.

[28] George A Miller, Edwin Broomell Newman, and Elizabeth A Friedman. Length-frequency statistics for written english. *Information and control*, 1(4):370–389, 1958.

[29] Arkadi Nemirovski, Anatoli Juditsky, Guanghui Lan, and Alexander Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization*, 19(4):1574–1609, 2009.

[30] Nan Niu, Tanmay Bhowmik, Hui Liu, and Zhendong Niu. Traceability-enabled refactoring for managing just-in-time requirements. In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, pages 133–142. IEEE, 2014.

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

[32] Ally S Nyamawe, Hui Liu, Nan Niu, Qasim Umer, and Zhendong Niu. Feature requests-based recommendation of software refactorings. *Empirical Software Engineering*, 25(5):4315–4347, 2020.

[33] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, 2015.

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[35] Joël Plisson, Nada Lavrac, Dunja Mladenic, et al. A rule based approach to word lemmatization. In *Proceedings of IS*, volume 3, pages 83–86, 2004.

[36] Martin F Porter. An algorithm for suffix stripping. *Program*, 2006.

[37] Anita Prinzie and Dirk Van den Poel. Random forests for multiclass classification: Random multinomial logit. *Expert systems with Applications*, 34(3):1721–1732, 2008.

[38] D Rapu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, pages 223–232. IEEE, 2004.

[39] Jaideepsinh K Raulji and Jatinderkumar R Saini. Stop-word removal algorithm and its implementation for sanskrit language. *International Journal of Computer Applications*, 150(2):15–17, 2016.

[40] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[41] Danilo Silva, João Silva, Gustavo Jansen De Souza Santos, Ricardo Terra, and Marco Tulio O Valente. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, 2020.

[42] Danilo Silva and Marco Tulio Valente. Refdiff: detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 269–279. IEEE, 2017.

[43] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 483–494. IEEE, 2018.

[44] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23(3):501–532, 2016.

[45] Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, S Yu Philip, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37, 2008.