NAIST-IS-DD83519751

Doctoral Dissertation

Improving Automation in Bug Report Categorization and Defect Prediction

Nachai Limsettho

August NA, 2016

Department of Information Science
Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Nachai Limsettho

Thesis Committee:

Professor Kenichi Matsumoto	(Supervisor)
Professor Yasuhiko Nakashima	
Assistant Professor Hideaki Hata	(Co-supervisor)
Professor Akito Monden	Okayama University
Assistant Professor Jacky Keung	City University of Hong Kong

Improving Automation in Bug Report Categorization and Defect Prediction*

Nachai Limsettho

Abstract

Many automated software engineering techniques have been proposed to help the process of bug report categorization and defect prediction. While the outcome of these techniques can be rewarding, the process of deploying them is often difficult and labor intensive. This dissertation focuses on improving the ease of deployment for bug report categorization and defect prediction with less human resources. This dissertation investigates and proposes the solutions for three aspects of automated software engineering techniques: nonparametric preprocessing of natural language, cross-project prediction, and unsupervised categorization. In preprocessing, a new approach is proposed to extract feature vectors from natural language in the bug reports and conducted experiments. The experimental results showed that the new features still retain the pattern which can easily be categorized by classifier algorithm. The cross-project prediction and unsupervised categorization tackle the same problem, that is, the unavailability of historical training dataset. When a similar dataset from another project is available, cross-project approach can be used. This dissertation proposes a technique for improving the cross-project performance by taking the distribution of the target unlabeled project into account. Compared with conventional techniques, the experimental results showed that the prediction performances were significantly improved. Lastly, the unsupervised categorization framework is proposed for the situations where the similar dataset is unavailable. Using clustering and cluster labeling techniques, the proposed framework could automatically categorize bug

*Doctoral Dissertation, Department of Information Science, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD83519751, August NA, 2016.

reports with comparable performance to the supervised approach. The conclusion is that the proposed techniques and framework could reduce human efforts required for the deployment of bug report categorization and defect prediction techniques, while still retain their performances compared to conventional techniques.

Keywords:

automated knowledge extraction, bug report classification, defect prediction, machine learning, classification, clustering

Acknowledgements

This dissertation would not have been completed without the people who had supported me. The same people who are still guiding me in pursuing my endeavors. Please allow me to take this opportunity to show my appreciation to everyone:

First and foremost, to Professor Kenichi Matsumoto, who granted me the opportunity to come to Japan and allowed me to be the part of his laboratory, I am sincerely grateful. Without the opportunity he gave me on that day, this thesis research would not have existed.

I must also sincerely thank Professor Akito Monden, for always be there when I don't know how to proceed. His invaluable comments and ideas helped me in countless situations.

I am sincerely grateful to Assistant Professor Hideaki Hata, for his guidance and patience to work with me, I know I am not an easy person to work with, so thank you for putting up with me. Without his help, I would not be able to accomplish what I did here at NAIST.

To Assistant Professor Akinori Ihara, who gave me many comments and suggestions to improve my research quality.

To Assistant Professor Jacky Keung, whom I am truly grateful for the opportunity to intern under his laboratory.

To Associate Professor Kitsana Waiyamai and Dr. Pattara Leelaprute, who introduced me to this program. Without their suggestion that day, I would not be here where I am.

To Associate Professor Arnon Rungsawang from Kasetsart University. He always comes to visit me here at NAIST, I have learned a lot from their advises.

To all members of SE Lab in NAIST and all my friends here, thank you all for all their friendship and supports. Special thanks to Papon Yongpisanpop who looked after me when I came to Japan and helped me adjusted to my new environment, the time we spent here is really fun and interesting. To Withawat Tangtrongpairoj thanks you for your supports and encouragements. Special thanks go to Christopher Michael Yap and Damien Rompapas who light up my boring days, I truly enjoy when we talked and laughed together.

This research was made possible by the generous support from the Japanese

Ministry of Education, Culture, Sports, Science and Technology (MEXT), for which I am exceedingly grateful.

Finally, I dedicate this dissertation to my family and my girlfriend, their encouragement, understanding and trust in me reminded me that for every chapter of life. I love you all.

List of Publications

Peer review journal paper

1. Nachai Limsettho, Hideaki Hata, Akito Monden, and Kenichi Matsumoto, "Unsupervised Bug Report Categorization using Clustering and Labeling Algorithm," International Journal of Software Engineering and Knowledge Engineering, 2016 (in IJSEKE) *(related to Chapter 6)*

2. Nachai Limsettho, Kwabena Ebo Bennin, Jacky W. Keung, Hideaki Hata, and Kenichi Matsumoto, "CDE-SMOTE: Cross Defect Prediction Using Quantification and Over-Sampling," (to be submitted) *(related to Chapter 5)*

Peer review international conference

1. Nachai Limsettho, Hideaki Hata, Akito Monden, and Kenichi Matsumoto, "Comparing hierarchical dirichlet process with latent dirichlet allocation in bug report multiclass classification," In Proceedings of the 2014 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), July 2014. (Las Vegas, U.S.A.) *(related to Chapter 4)*

2. Nachai Limsettho, Hideaki Hata, Akito Monden, and Kenichi Matsumoto, "Automatic Unsupervised Bug Report Categorization," In Proceedings of 2014 6th International Workshop on Empirical Software Engineering in Practice (IWESEP), November 2014. (Osaka, Japan) *(related to Chapter 6)*

Contents

Acknowledgements iii

List of Figures **x**

List of Tables **xii**

1 Introduction: Human Effort and Knowledge Discovery in Software Quality Assurance **1**

1.1. Motivation 2

1.2. Contributions 4

1.3. Thesis Layout 4

2 Background: Automated System in Software Quality Assurance **7**

2.1. Bug Report Categorization 7

2.2. Defect Prediction 9

3 Preliminaries: NLP Preprocessing & Machine Learning **11**

3.1. NLP Preprocessing 11

3.1.1 General Preprocessing 11

3.1.2 Bag of Words 12

3.1.3 N-grams 13

3.1.4 Topic Modeling 13

3.2. Machine Learning 14

3.2.1 Classification 14

3.2.2 Quantification 15

3.2.3 Clustering 15

4 Improving Automation: Bug Report Categorization (Classification) 17
4.1. Background . . . 17
4.2. Proposed Algorithm . . . 19
4.3. Experimental-Design . . . 23
4.3.1 Datasets . . . 23
4.4. Results . . . 24
4.4.1 Experiment 1: Comparing LDA and HDP Performance . . 24
4.4.2 Experiment 2: Characteristics of LDA and its Optimization 26
4.5. Threats to Validity . . . 29
4.6. Conclusion . . . 29

5 Improving Automation: Improving Cross-Project Defect Prediction 31
5.1. Background . . . 31
5.1.1 Synthetic Minority Oversampling Technique (SMOTE) . . 33
5.1.2 Class Distribution Estimation (CDE) . . . 34
5.2. Proposed Algorithm . . . 35
5.2.1 CDE-SMOTE Principles . . . 35
5.2.2 Steps and Procedures . . . 35
5.3. Experimental-Design . . . 38
5.3.1 Experimental Setup . . . 38
5.4. Result . . . 46
5.4.1 Experiment 1: Oracle and Original Classifiers Comparison 46
5.4.2 Experiment 2: Performance of the Class Distribution Estimator . . . 48
5.4.3 Experiment 3: CDE-SMOTE and Origianl Classifiers Comparison . . . 51
5.4.4 Experiment 4: CDE-SMOTE and Related Works Comparison 52
5.5. Discussion . . . 58
5.5.1 Implications . . . 58
5.5.2 Validity . . . 59
5.6. Conclusion . . . 60

6 Improving Automation: Bug Report Categorization with Shortage of Historical Data **63**
6.1. Background . . . 63
6.1.1 Supervised Learning Approach . . . 66
6.1.2 Unsupervised Learning Approach . . . 66
6.1.3 Topic Modeling . . . 67
6.2. Proposed Algorithm . . . 68
6.2.1 Topic Modeling Phase . . . 68
6.2.2 Clustering Phase . . . 70
6.2.3 Cluster Labeling Phase . . . 70
6.3. Experimental Design . . . 76
6.3.1 Measurements . . . 76
6.3.2 Datasets . . . 78
6.3.3 Design For Each Experiment . . . 78
6.4. Results . . . 82
6.4.1 Experiment 1: Comparison Between Bag-of-Words and Topic Modeling . . . 82
6.4.2 Experiment 2: Categorizing Bug Reports from Different Projects . . . 84
6.4.3 Experiment 3: Categorizing Bug Reports in one Project into Bug and Other Requests (In-Project Classification) . . . 85
6.4.4 Experiment 4: Comparison Between Our Method and Cross-Project Classification . . . 86
6.4.5 Experiment 5: Cluster Labeling Results . . . 87
6.5. Threats to Validity . . . 90
6.5.1 Measurements used . . . 90
6.5.2 The categories of bug reports . . . 90
6.5.3 Experimented Datasets . . . 90
6.5.4 Heuristics parameters . . . 91
6.5.5 Coverage of other possible parameters and approaches . . . 91
6.6. Conclusion . . . 91

7 Conclusion **93**
7.1. Future Work . . . 95

Bibliography **97**

List of Figures

2.1 Research Areas of this Dissertation 7

4.1 Features Extraction: Diagram of the Topic Modeling phase 21
4.2 Features Extraction: Diagram of the Classification phase 23

5.1 Diagram showing how SMOTE works (k=3) 34
5.2 CDE-SMOTE Diagram: Class Distribution Estimation 36
5.3 CDE-SMOTE Diagram: Class Distribution Modification 37
5.4 CDE-SMOTE Diagram: Prediction Model Building 38
5.5 CDE-SMOTE: Oracle comparison to the Original Classifier: Wilcoxon Win-Tie-Loss 47
5.6 CDE-SMOTE: Actual class distribution mismatch ($PredictedMismatch_{Value}$) between train and unlabeled datasets in the actual value 49
5.7 CDE-SMOTE: Class Distribution Mismatch Compared to the original Training Data: Percentage differrent betwen actual test data error and CDE estimation ($PredictedMismatch_{\%Diff}$) 50
5.8 CDE-SMOTE: comparison to the Original Classifier: Wilcoxon Win-Tie-Loss 52
5.9 CDE-SMOTE: Box plots Perfomance (Actual value) comparison between CLAMI and CDE-SMOTE (VOTE 2) - When the Cross-Project training dataset is chosen randomly 55
5.10 CDE-SMOTE: Percentage of cases that CDE-SMOTE shows Significant improve over CLAMI 57

6.1 Unsupervised: Diagram of the Topic Modeling phase 69
6.2 Unsupervised: Diagram of the Clustering phase 71

6.3 Unsupervised: Diagram of Adjusted Jensen-Shannon Divergence: Cluster Labeling 72
6.4 Unsupervised: Diagram of NLP Chunk: Cluster Labeling 73
6.5 Unsupervised: F-measure comparison between Our Unsupervised Method and Cross-Project Classification 87

List of Tables

4.1 Features Extraction: Class Distribution of Combined Dataset . . . 24
4.2 Features Extraction: Comparison Between LDA and HDP 25
4.3 Features Extraction: Accuracy of the LDA Topic-based Classifier 27
4.4 Features Extraction: F-Measure of the LDA Topic-based Classifier 28
4.5 Features Extraction: ROC of the LDA Topic-based Classifier . . . 28

5.1 CDE-SMOTE: Datasets Class Distribution (Pr) 40
5.2 CDE-SMOTE: Dimensions (Metrics) of Datasets 40
5.3 CDE-SMOTE: Oracle increase performance (%) compared to original classifier [Averaged from 14 x 13 = 182 combinations of cross-project pairs] 48
5.4 CDE-SMOTE: increase performance compared to original classifier (Percentage) 53
5.5 CDE-SMOTE: cross-project defect prediction performance in terms of Balance, G-measure, and F-measure 53
5.6 CDE-SMOTE: Increase performance (Percentage) comparison between Burak Filtered dataset and Burak Filtered dataset with CDE-SMOTE applied 54
5.7 CDE-SMOTE: Increase performance (Percentage) comparison with CLAMI - When the Cross-Project training dataset is chosen randomly 54
5.8 CDE-SMOTE: Increase performance (Percentage) comparison with CLAMI - When the Cross-Project training dataset is selected . . 57

6.1 Unsupervised: Class distribution of the experimented datasets . . 79

6.2 Unsupervised: Experiment 1 - Comparison between different Dataset Dimension 83
6.3 Unsupervised: Experiment 2 - Categorizing bug reports from different projects 84
6.4 Unsupervised: Experiment 3 - Categorizing bug reports in one project into Bug and Other Requests 85
6.5 Unsupervised: Experiment 5 - Cluster Labeling Results 88

Chapter 1

Introduction: Human Effort and Knowledge Discovery in Software Quality Assurance

Nowadays, software development infrastructure is complicated and contains a lot of connecting components. Bug tracking system, also known as issue tracking system, is undeniably a fundamental part of any software development infrastructure. It is a database that records the information of the known bug and contains a collection of bug reports submitted by either its developer themselves or by its end-user. Aside from its usual benefits, it also provides a good quantity of information which can be extracted and analyzed [1–7], to offer important insight into the quality of the target software project.

Bug report categorization is often used to extract meaningful information. This technique can be used in many applications such as: to detect duplicated bug reports [2, 3, 7], to estimate bug fixing time [1], or to correct bug report type [8]. Machine learning approach is usually employed for this task; most of them rely on classification [4–8], a supervised learning approach. These approaches construct a prediction model from the training data of the labeled bug reports that can later be used to automatically categorize new incoming data into predetermined labels. The advantage of this approach is that it can greatly reduce the amount of human effort required after a classification model is built. However, building the said model is, sometimes, quite difficult. The supervised learning approach

requires a lot of labeled bug reports to construct its model, but these reports are often unavailable in many software engineering projects. In most cases, obtaining this dataset itself is not an easy task, since preparing a training data needs human inspections and in order for supervised learning to work properly, a large amount of data is initially required. While the long existing project that already manually processed its data, could bypass this problem easily; the same could not be said for many other projects.

While analyzing a bug tracking system has many possible applications, the major problem is that most of bug report's information is in the unstructured form: such as natural language in the title, description , and comment sections. Traditionally, these reports are inspected by humans [8]; this approach has good accuracy and flexibility. However, the time it takes to understand each individual report combined with the numerous numbers of reports makes manually reading through them impractical or even impossible in many situations [8]. It is clear that in order to use this natural language information in a practical environment, the automated translating process is needed. Natural language processing (NLP) techniques are commonly employed [2,5,9,10] to solve this problem; the raw text is converted into a more processable form, often to create a set of features vector from bug reports.

Another essential technique in quality assurance of software engineering is defect prediction. It is the process of predicting the defect-proneness of software modules, and helps in organizing and allocating testing resources [11, 12], by indicating modules with high defective risks. Defect prediction is known to work well when the prediction model is built using its own historical data [11, 12]. However, given a lack of historical data for a new project, the ability to build an effective defect prediction model becomes a very difficult task, as specifically noted by He et al. [13] and Turhan et al [14].

1.1. Motivation

While many quality assurance processes do, indeed, reduce the human effort required tremendously, many of their parts are still required human experts to process and oversee the procedures. The effort which could be spent elsewhere if

the process becomes automatic. The example of such scenario are:

Example situation 1: The number of topics for topic modeling. The number of topics is one of the most important parameters required for topic modeling and often required as an input parameter from an expert in the Natural Language Processing (NLP) field. However, the appropriate number of topics largely depends on the target bug report repository, which means randomly assigning the number of topics will often lead to performance degradation of the software quality assurance technique that builds upon it. To account for this problem experts usually run several experiments to find the suitable number of topics for their application, this lead to increasing amount of time and resource that required for the deployment of the target software quality assurance technique. Furthermore, even when the number of topics is carefully tuned, the increasing amount of text from the new incoming bug reports will, sooner or later, lead to a change in the suitable number of topics resulting in the needed for parameter tuning again.

The stream of text in the natural language form is very hard to understand and interpret by machine learning algorithm. To account for this problem, the natural language processing (NLP) techniques are commonly employed [2,5,9,10] to transform the text into matrix form; for example, bag-of-words, N-grams or Topic modeling. These processes often required input parameters from experts, i.e. the number of topics for topic modeling, and without carefully tuning, the performance of the model build upon will usually degrade.

Example situation 2: The preparation of the labeled training dataset. Many automated quality assurance process in software engineering employed classification, a machine learning technique. Even though this process offers a lot of benefits and utilities, it also has limitation. In order to build a classification model a set of labeled historical data in required; however, obtaining the said training dataset is not an easy task, especially if the project is newly started, a lot of human effort will be required to gather and correct that dataset [8, 15]. While cross-project classification [13,14,16,17], the technique which trains a model using labeled data from similar projects, can be used to solve this problem. its performance is still not quite as good as the within project approach. Further-

more, there are many situations where cross-project approach can not be applied.

This dissertation aims to reduce the human resource spent in such scenarios and improving the efficiency of software quality assurance.

1.2. Contributions

This dissertation aims to reduce the amount of human resource needed to deploy this automation process and offer better solutions that are more automated and still retain the performance comparable to those of the previous approaches. The followings are the contributions of this dissertation:

1. Improving an automation process of converting natural language into topic membership vector by introducing the use of nonparametric topic modeling instead of the commonly used topic model. This eliminates the need for parameter tuning, thus further reduces time and effort needed to processing bug reports.

2. Improving the cross-project prediction performance to improve it feasibility when it can be applied. Our technique solves the problems associated with cross-project prediction by using quantification and oversampling

3. Create an automated framework for bug report categorization that is capable even when the cross-project prediction cannot be applied. Using clustering and cluster labeling methods, our framework provides the definitive benefit of not requiring any training dataset, while still having comparable performance to the supervised approach.

1.3. Thesis Layout

This dissertation is structured into seven chapters. The first chapter is this section which introduces the topic of human effort and knowledge discovery in software quality assurance.

Chapter 2 and 3 explain the background of this dissertation in the software quality assurance, NLP preprocessing, and machine learning, respectively.

Chapter 4 presents a study of improving automation in the conversion from text to feature vectors. In the context of bug report classification, we make a comparison between the feature vectors extracted from the conventional parametric topic model, Latent Dirichlet Allocation (LDA) [18], and its non-parametric cousin, Hierarchical Dirichlet Process (HDP) [19]. Empirical evaluation is conducted with manually labeled bug reports from open-source software projects.

Chapter 5 presents a study of cross-project categorization where the historical dataset of the target project is not available but there is a historical dataset from another similar project. The proposed algorithm, CDE-SMOTE, focuses on the problem of imbalance class distribution, where the amount of skewness is uncertain due to the cross-project scenario. Empirical evaluation is conducted with code complexity information from open-source software projects.

Chapter 6 presents a study of unsupervised categorization where the historical dataset is not available at all, for both within and without the target project. This chapter present the framework for using in such scenario utilizing clustering and cluster labeling techniques. Empirical evaluation is conducted with manually label bug reports from open-source software projects.

Finally, Chapter 7 concludes the dissertation with a summary and directions for future work.

Chapter 2

Background: Automated System in Software Quality Assurance

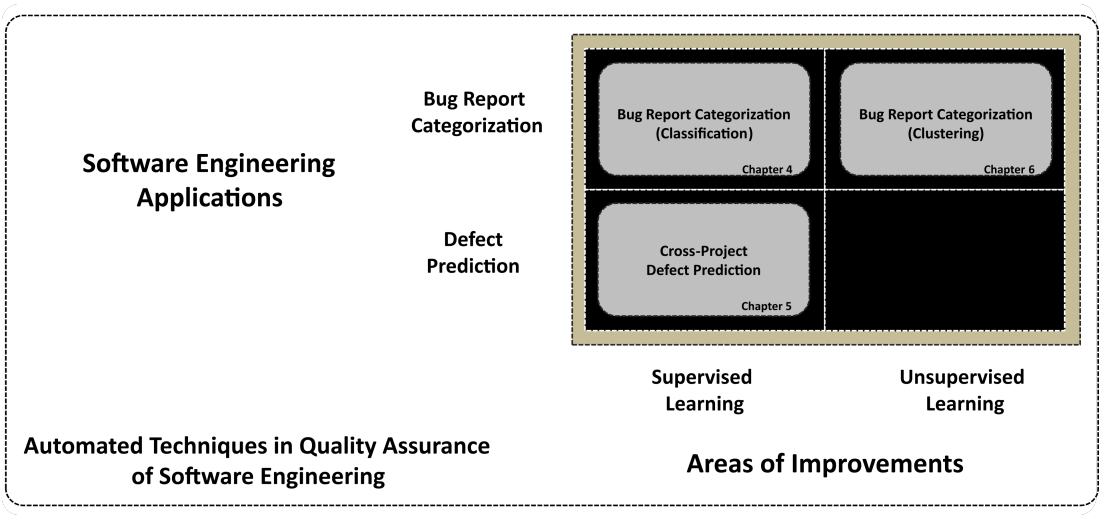

Figure 2.1. Research Areas of this Dissertation

2.1. Bug Report Categorization

The aims of bug report categorization are to categorize unlabeled bug reports into desired categories. Several different goals can be achieved by this catego-

rization depending on utilized categories; for example, if categories are Bug and Others [5], bug report classification can be used to help the assigners decide how their project resource should be allocated. Or if the categories are whether the bug report is Duplicated or Not Duplicated [2,3,7], the classification process can help identifying whether the submitting bug report is the same as any previously submitted bug reports or not, in order to reduce efforts and save time for developers in fixing the same issues. There are several kinds of research related to bug reports categorization. Some [8, 15] use a manual inspection to classify bug reports according to their interested categories. While using a traditional approach like this has its own benefits and is necessary for some situations, it usually requires more time and human resources than what most projects can actually afford.

To solve this problem, the previous research proposes an automated system for categorizing bug reports. Most researches use classification [4–8], a supervised learning approach, to learn concepts from a labeled dataset.As the classification technique required the input in a form of feature vectors, the bug reports raw text parts which contain the most information it cannot be used directly. As such natural language processing (NLP) techniques are usually applied to convert the raw text parts of each bug report into a feature vector format. Depending on the approach different kind of vector is utilized, there are mainly three main approaches usually used in the conversion: Bag of Words [20], N-grams [7], and Topic Modeling [2,5,9,10]. These NLP processes will be discussed in more detail in the following NLP section.

While these classification approaches already reduce the development effort drastically, there are still several areas where they can be improved to be more automated. For instance, so far, in the previous research employing topic modeling [2,5,9,10], the appropriate number of topics for each project is hard to determine and can be widely different depending on the project. Furthermore, up to this point, there is yet to be any automated approach to estimate the number of topics required for accurate bug report type classification. In this dissertation, we propose a nonparametric approach to automatically classify bug reports with, another topic modeling method, Hierarchical Dirichlet Process (HDP) [19]. The result indicates that our nonparametric approach performance is comparable to

the parametric one.

Another example of the case where the automation can be improved is the requirement for a dataset of labeled historical bug reports. In order to create a bug report classification model, a number of labeled bug reports is needed. The major problem is that, unless the labeled dataset is readily available, a huge amount of labor will require to process and categorized these bug report [8,15]. There are two major solutions for this problem: Cross-project classification and unsupervised learning, both with advantages and disadvantage of its own. This dissertation proposes an approach for improving the current cross-project prediction and provide a new unsupervised alternative for when the cross-project approach cannot be applied, due to the absent of compatible cross-project dataset.

2.2. Defect Prediction

Defect prediction is the process of identifying fault-prone modules and aids in the effective allocation and prioritization of scarce testing resources [21]. Several defect prediction models have been proposed in the past decade employing various machine learning, statistical and manual approaches. Conventional methods such as neural networks [22], support vector machine [23], bayesian classifier [24] and so on [25] have been used. Menzies et al. [26] recommends the application of Naive Bayes with logNums preprocessing for better defect prediction. These proposed defect prediction models are however studied in within-project scenarios. The models are open to projects with historical datasets available hence restricting their applicability on new projects without any historical dataset available. To avoid this problem, the cross-project prediction is often employed [13, 14, 16, 17], it circumvents the problem by using the historical datasets from other projects. Over the years, several studies [13,14,27–30] have been proposed to solve the lack of historical dataset problem, which can be divided into two main approaches: the unsupervised and the cross-project defect prediction approaches.

With the availability of open source datasets, the feasibility of cross-project defect prediction where datasets of other projects are used to train prediction models has been investigated in recent years but with an inconclusive result. The first to attempt on cross-project defect prediction was Zimmermann et al. [27].

Conducting a large-scale experiment on 12 real-world datasets, 622 cross-project prediction models were analyzed and investigated for the feasibility of cross-project defect prediction. Observing a low success rate of 3.4%, they concluded cross-project defect prediction was still a challenge. In their work, Turhan et al. [14] proposed a practical defect prediction approach for organizations aiming to employ defect prediction but lacks historical data. Applying the principles of analogy-based learning, they use the k-nearest neighbor algorithm which selects 10 nearest data instances for every unlabeled test instance for cross-company defect prediction. They demonstrate that small data samples acquired using their approach could be used to build effective defect predictors. Similarly, Peters et al. [17] proposed a new filter which outperformed the Burak filter proposed in the work by Turhan et al. [14]. The Peters filter selects training data considering the structure of the other projects and could select as few as one data instance for each test instance. Conducting large-scale cross-project defect prediction experiments on 34 data sets extracted from 10 open source projects, He et al. [13] observes that carefully selecting training data from different projects is very vital for constructing defect prediction models for new projects. They also support conclusions that cross-project defect prediction works in few cases as previously reported in studies by Zimmermann et al. [27] and Turhan et al. [14]. Jureczko and Madeyski [31] applied clustering techniques to partition various projects into distinct groups with the assumption that projects in the same group have the similar characteristics. They argue that a defect prediction model trained on datasets in the same group is reusable for new projects, which tend to have the characteristics of the group hence no need for datasets to have historical datasets before defect prediction model could be constructed. Zhang et al. [32] studies the performance of the ensemble approach, which combines multiple classifiers together, in the cross-project scenario. Their results indicate that several ensemble algorithms can outperform the CODEP, a defect prediction algorithm proposed by Panichella et al. [9].

Chapter 3

Preliminaries: NLP Preprocessing & Machine Learning

3.1. NLP Preprocessing

Most information in bug reports is in the raw text format, which is unstructured and cannot be directly processed by machine learning algorithms. In order to convert the raw text into a more processable format, natural language processing (NLP) techniques are commonly employed often to create a set of features vector from bug reports. The NLP techniques involve in this dissertation can be categorized into the following: General Preprocessing, Bag of Words, N-grams, and Topic modeling.

3.1.1 General Preprocessing

General preprocessing are simple and heuristic techniques that often utilize by other NLP techniques in order to preprocess the raw text input. While these techniques are simple, they usually are necessary for improving the quality of the latter process down the pipeline; although depending on the final goal, a certain technique could be excluded. The processes that fall into the general category are Parsing, Tokenization, Stemming, and Stop Words Removal.

The parsing process goal is to extract the textual information from incoming bug reports. As most of the time the extracted bug reports are in XML format, a lot of non- textual information such as tags, attributes, and declarations are also included. Thus, the parsing process in required to selecting the intended parts for further preprocessing.

Tokenization is to break the stream of text into terms and removal of unnecessary punctuations. Most of the times the space, comma, and dot punctuations are used as an indicator for the tokenized location, although depending on the approach, other punctuations can be used together as well. The output of this process is a list of words containing the input raw text

The stemming process is used to convert the tokenized terms back into their root form [33], often as a basic method for grouping words with a similar meaning. Note that, this stemmed root is not necessary to be the same as the morphological root of the word, just that the related words share the same stemmed root. There are several methods for stemming ranging from a simple lookup table to more complicated method such as Stochastic algorithm [34] or Lemmatization [35], which involves determining the part of speech of the word.

Stop Words Removal [36] is a process used to remove a list of certain words from processed text. These stop words are terms that contain very little information when alone, for instance: is, an, or and. Such word only conveys its meaning when placed in its correct position, they will not provide useful information when transformed into a feature vector space that disregards their position. Note that, the content of the stop words list depends on the context of the target project; even if the word normally has its meaning it can still be included in the list given the situation.

3.1.2 Bag of Words

Bag of words is a process used to represent a raw text in a simplified model. Each document, i.e. bug report, is transformed into a feature vector, with each dimension represents a word. An input raw bug report will be processed by several general preprocessing techniques, such as parsing, tokenization, stemming, and stop words removal; followed by quantifying the proportion of each word in that bug report. The method for quantifying is vary depending on each particular

research, ranging from zero/one to term weighting.

So far, most software engineering researches that utilizing the bag of word [37] rely on term weighting , specifically, they often rely on term frequencyinverse document frequency (tfidf) for quantifying the proportion of words in bug reports.

3.1.3 N-grams

N-grams is, similar to the bag of words, also a process used to represent a document through a simplified model. However, instead of representing a word in each dimension, a dimension of n-grams vector will consist of n words that are a contiguous sequence. These sequences are used to represent a document through their quantified proportion. Normally the length of sequences are fixed at n, some approach, however, offers varying length sequences ranging from one to n.

The n-grams approach is commonly used in many software engineering researches [7, 38], compared to the bag of words, n-grams offer much less sparse dimensions for the feature vectors as its dimension is a lot more specific.

3.1.4 Topic Modeling

Topic modeling [18, 19] is an unsupervised learning technique that captures the underlying structure of the document repository by grouping co-occurrence words into the same topic. The result is a set of topics, a cluster of words that likely to share the same meaning. A document can be associated with topics using a topic proportion vector that indicates what topics that document is associated with. The more the document relates to the topic, the more proportion is assigned to that topic. Compared to the bag-of-words, topic modeling can greatly reduce the effect of data sparseness, which is one of the main problems of the word-level approach. In addition, this approach also help reduces synonymy and polysemy problems by grouping the co-occurrence words together. This generally makes documents much easier to distinguish and it reduces the computation time for both supervised and unsupervised learning.

Projecting bug reports into topic vector space can be advantageous in many ways. When comparing to the bag of words approach, its performance is definitely better [5]. Plingclasai et.al [5] has shown that this area could benefit from topic

modeling and could significantly improve classification performance of bug reports by just adopting it instead of using a word-level model. This is mainly due to two reasons. First, by projecting documents into topic vector space, we can greatly reduce the effect of data sparseness which is one of the main problems of word-level approach. Second, by grouping words that frequently co-occur in document corpus into a single topic, we can also reduce the problem of synonymy and polysemy. This generally makes documents easier to distinguish as well as reduce the computation time.

While topic modeling is certainly very useful, using this approach alone is often not enough to understand the underlying structure of bug reports; even with topics proportion demonstrated, comprehending the similarity of each bug report in high dimensional data space is far from easy.

3.2. Machine Learning

3.2.1 Classification

Supervised learning is widely used in the area of software engineering; the most prevalent method is a classification that trains a classification model with training dataset to later be used to classified new incoming data [4]. Each instance in the training dataset is labeled with its actual class; these classes are pre-determined and act as prior knowledge which the model will try to learn.

While this approach is widely used and clearly has its own advantages, its major problem lies in its absolute requirement for the prior knowledge; without this information the classification model simply can not be built and obtaining this knowledge is far from easy. Since to obtain a good classification model, a large amount of training data is needed, human inspection is required in order to prepare this dataset. In Herzig et.al [8], a large amount of time and effort are spent to reclassify bug report categories.

One way for supervised learning to mitigate this problem is cross-project classification [13,16,39]. This method builds a classification model from a dataset from another project instead of using its own. This generally makes obtaining training data become easier, and it also makes the concept that the model learns

become much more general.

3.2.2 Quantification

Class distribution estimation (CDE) or Quantification is a technique in a machine learning [40]. Unlike classification that is interested in the actual label of each instance, quantification is more interested in the distribution of each class; given an unlabeled dataset, a quantification will estimate the proportion of each class in that dataset. This approach has many possible applications; while it has yet to be utilized in the software engineering field, it has been adopted in many other fields such as in text mining [41], sentiment analysis [42] and epidemiology [43]. Our research uses this estimated class distribution to approximate the amount of oversampling needed for the target unlabeled project.

3.2.3 Clustering

Some research in this area uses unsupervised learning to find hidden structures within their data. It is commonly used in bug triaging [10], duplicate bug report detection [2] and in topic modeling [9,44]. The main advantage of this approach is the non-requirement of a training dataset. This greatly helps reduce the amount of effort required to obtain and process prior knowledge which would otherwise be needed for the supervised learning.

Aside from this, the amount of knowledge obtainable from supervised learning is also limited by its prior knowledge; supervised learning cannot comprehend anything beyond which it is specifically taught. This means that supervised learning will always categorize bug report to the predetermined class which, sometimes, is not the best approach since a certain class might be better represented as two or more in certain situations.

Chapter 4

Improving Automation: Bug Report Categorization (Classification)

4.1. Background

In data mining, good quality of data is a valuable asset. This also applies to empirical software engineering as well. Since nowadays, mining data from changes and bug databases had become common. As bug database is built from bug reports, quality of bug reports are crucial to data quality [45]. Correctly classified bug reports will greatly help in both research validity and modeling performance. More detail bug report will also contain more information which could help in understanding data. On the contrary, inadequate information and misclassified bug reports lead to misleading research and misrepresenting model. However, Antoniol et al. [46] found that a significant number of bug reports are incorrectly classified; many reports labeled as bug are not actually a bug. They are, in fact, referring to other things such as a request for a new feature, an improvement, or an update to documentation. These errors happen mostly due to reporters misunderstanding and the complicating nature of Bug Tracking System used for reporting a number of other requests besides bug [46].

In order to correct these miss-classification, large amount of effort is required, especially for manual inspection [8, 15, 46, 47]. For example, Herzig et al. [8] spent

totaling 725 hours, 90 days, to classify over 7,000 bug reports. For this very reason, a technique to automatically classify bug reports is desired.

Several studies have been proposed to tackle this problem. A word-based automatic classification technique [46] , by Antoniol et al., creates classification model base on word corpus and got a decent result. A recent study proposed a binary classification based on topic modeling approach [5]. This method tries to improve bug report classification process by substitute word-level document corpus with Latent Dirichlet Allocation (LDA) [18] topic membership vectors. The experiments show that topic-based model outperforms word-based in almost of the evaluated cases.

Nevertheless, some problems still remain. First is that the LDA approach [5] requires a parameter tuning in order to work optimally. This means a certain amount of effort is needed. Second, only one dimension of topic modeling, LDA, has been explored. Other dimensions of topic modeling are left uncharted. Third, it only works on binary classification this abandons some useful information, which otherwise would be obtainable with multiclass classification.

The motivations for this work are as followed:

1. **Quality of Data:** For most of statistical and data mining tasks, good quality of data is essential. The mistakes in data, aka noise, can lead to misleading and poor performance model. More specifically in bug classification task, decision boundary between each class can be significant affect, thus, make the classification model to be unsuitable for real world data [45]. Another aspect of data quality is how detail it is. For bug report data, while binary class data can contain a good amount of information and suitable for many tasks. The absence of some important information that otherwise obtainable with multiple classes data could be a problem. Using multiple classes data allows exact pinpoints of bug report purpose. It also enables research into many directions which would be impossible with binary one.

2. **Topic Modeling Approach:** Using topic modeling to preprocess documents for bug report classification is advantageous in many ways. Compared with manual categorization, it definitely saves a huge amount of effort and time. Its performance is also better than the bag of words approach [5].

There are two reasons for this. First, since words that frequently occur together in document corpus are grouped into a topic, problems of synonymy and polysemy are diminished. Second, it projects a very sparse vector space model into a more compact and meaningful form. This generally helps classification in both computation time and classifying accuracy.

Topic modeling can be done in many different ways; in this chapter, two of popular methods are experimented on. Latent Dirichlet Allocation (LDA) [18] is a Bayesian approach to topic modeling bug reports. This method views each document as a mixture of various topics and assumes that topic distribution has a Dirichlet prior. Another way is Hierarchical Dirichlet Process (HDP) [19]. It is a nonparametric Bayesian model which assumes the number of topics from Dirichlet process and allows mixture components to be shared between groups. Both approaches have its merit. While LDA is easier to apply since many tools and libraries implement it, the nonparametric nature of HDP is also very appealing.

4.2. Proposed Algorithm

Our bug report classification process is divided into two main phases. First is Topic Modeling phase, which converts bug reports into topics membership vectors. In the second phase, Classification, data from the previous phase are combined with its classes then preprocessed and used to build a classification model. More detail will be described in following subsections.

1. **Topic Modeling Phase:** This phase is consist of five steps.

   (a) **Parsing:** Our bug reports come in XML format. In order to get a more meaningful data from these bug reports, we extract three textual sections: title, description, and comments. These sections are combined into a single text file per bug report.

   (b) **Tokenization:** After parsed, the stream of text from bug reports are tokenized, broken into terms, and unnecessary punctuations are removed.

(c) **Stemming:** In this step, the tokenized terms are mapped and converted to their root form. Porter Stemming algorithm [33] is used for experiments in this chapter.

(d) **Removing Stop Words:** Some words in English hold little to no meaning alone. As such, they are removed. We use stopwords from mallet 2.0.7 stoplist [36]. The examples of these words are a, both, but, by, can, and the.

(e) **Topic Modeling:** Topic modeling is applied in this step in order to automatically extract topics from a text corpus. Two of well-known topic modeling methods are applied in this research as we want to experiment on which approach is more suitable for topic modeling bug reports.

First is LDA, which is commonly used and implemented. This method is a probabilistic generative model and it is required for a user to specify the number of topics (N). Since the best perform N depends on the dataset, parameter tuning on topic's number is needed. Therefore, for any experiment that uses LDA topic, several numbers of topics are examined.

Second is HDP, which can assume the number of the topic by itself, hence reducing tuning effort. However, as the second level of HDP drawn samples from already drawn the subset of the first level (assuming it is 2-levels HDP), topics drawn from HDP are overlapped. This is different from LDA since LDAs topics are drawn separately from base distribution, thus making it less likely to overlap. Though the overlapping nature of HDP can be advantageous in many situations, it can also make data be harder to separate.

After topic modeling process, the output of both approaches is a set of topic membership vectors. Each vector represents a bug report and consists of a set of topics with its proportion. These topics comprise of co-occurring words throughout the bug report textual corpus and their proportion indicates what topics such bug report are related.

Figure 4.1 summarizes these steps in topic modeling phase.

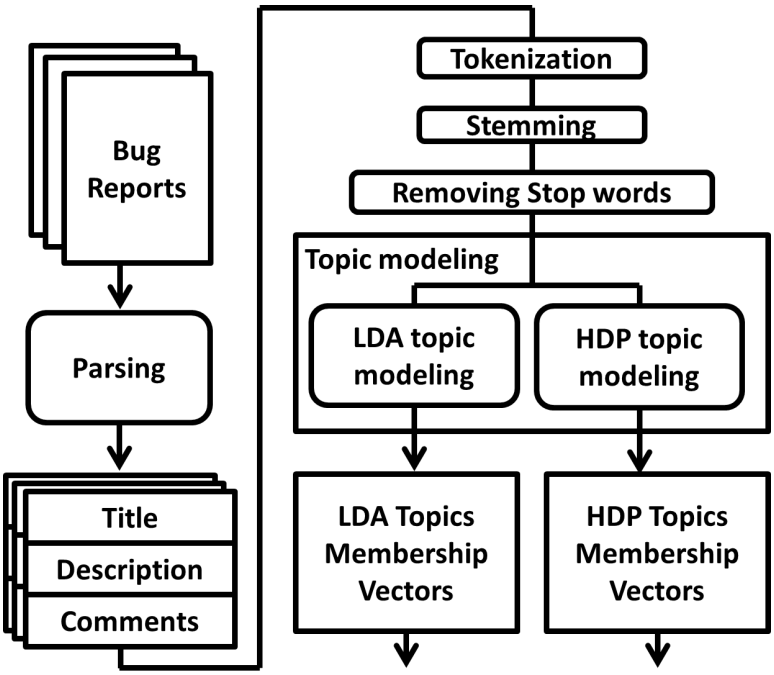

Figure 4.1. Features Extraction: Diagram of the Topic Modeling phase

2. **Classification Phase:** In this phase, HDP and LDA data are processed separately. Each topic membership vectors dataset from the previous phase is combined with the corrected dataset containing bug reports actual classes. This process is done to prepare these data for the classification task. Data from LDA and HDP then proceed independently to the following three steps.

   (a) **Preprocessing:** This step will be described later in each experiment results subsection as the process is different for each experiment.

   (b) **Spilting Data:** Preprocessed data are then spilt randomly into two datasets: train and test dataset. The test data are reserved for evaluation, while train data are passed to the next step. To validate our result, we employ 10-fold cross-validation to all experiments in this chapter and report average value of 10 runs as our result.

   (c) **Build Classifiers:** Classification models, aka classifiers, are built from train data. All classifiers built in this research utilize multi-

class ensemble classifier all-against-all, a wrapper based type classifier. The wrapped classifiers depend on the experiment. For Experiment 1, the base classifier is a logistic regression. While in Experiment 2, three types of classification technique are wrapped: Alternating Decision Tree (ADTree), Naive Bayes and Logistic Regression [48]. These classification techniques are chosen based on previous researches [5, 49]. Their details are described below.

i. **All-against-All:** This is a wrapper based type classification technique that enables binary classifiers to handle multiclass datasets. The wrapper based nature of this classifier means it cannot work by itself and needs a base classifier to wrap on. Specifically, this classifier transforms a K classes classification problem into K(K-1)/2 binary classification problems of separating between each pair of classes, while ignoring the rest of them. Results from these classifiers then are combined via voting. We choose this method to handle multiclass problem since it is intuitive and has a good performance.

ii. **Alternating Decision Tree (ADTree):** It is a set of generalized decision trees that employ boosting algorithm. A number of boosting iterations is user specific. For each iteration, the weight for each instance will be given differently according to the previous iteration results. The correctly classified instances are given reduced weight while the misclassified are given a larger weight.

iii. **Naive Bayes:** This classifier applying Bayes' theorem with an assumption that each other features aside from class are independent.

iv. **Logistic Regression:** A regression analysis that uses probability scores to measure the relationship between class and features.

Figure 4.2 summarizes these steps in the classification phase.

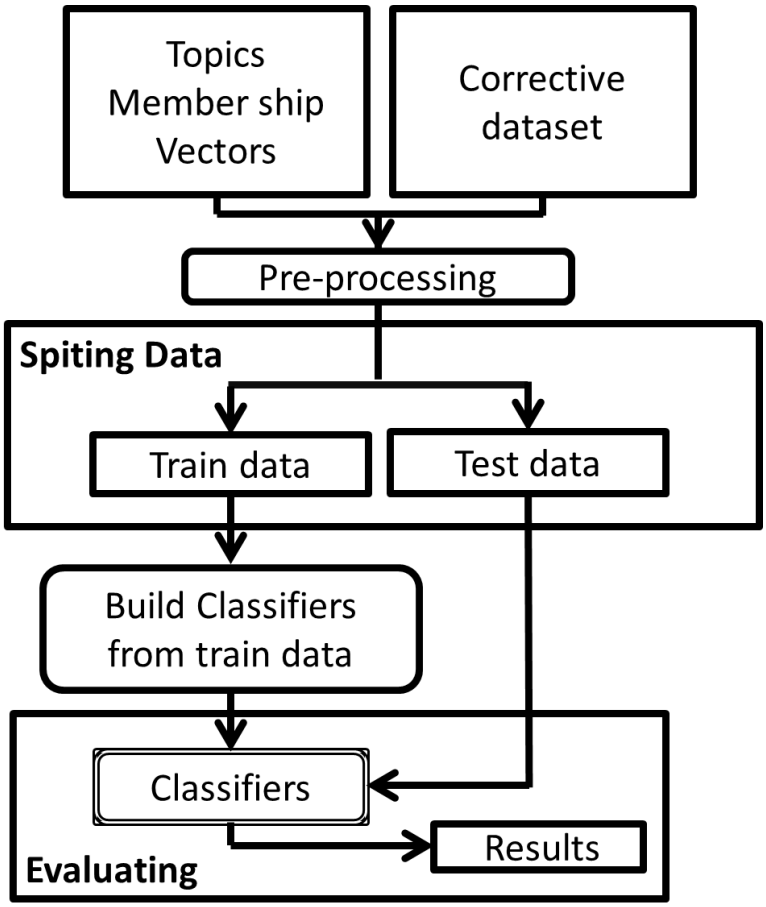

Figure 4.2. Features Extraction: Diagram of the Classification phase

4.3. Experimental-Design

Our experiments are divided into two parts: LDA-HDP comparison in Experiment 1, then optimization of LDA in Experiment 2. All experiments in this paper are evaluated by accuracy, F-measure and receiver operating characteristic (ROC).

4.3.1 Datasets

Our study use combined data from three datasets in previous study [8]. The bug report from three open-source software projects from Apache: HTTPClient,

Jackrabbit, and Lucene, are combined. The reason we used combined dataset is because we want to evaluate each method in cross-project learning environment since it can greatly reduce the amount of effort required in preparing training data in real world implementation. The Class distribution of combined dataset is shown in Table 4.1.

Table 4.1. Features Extraction: Class Distribution of Combined Dataset

Bug Report types (Classes)	Number of Reports	Percentage of Reports
Bug	2,718	49.96%
Improvement (IMPR)	2,092	38.46%
Request For Enchantment (RFE)	337	6.19%
Test	79	1.45%
Total	5,440	5,440%

4.4. Results

4.4.1 Experiment 1: Comparing LDA and HDP Performance

We want to compare LDA and HDP performance in this experiment, so aside from using LDA and HDP, all other parameters here are identical.

LDA and HDP topic membership vectors from Topic Modeling phase are independently processed. After combined with the corrected dataset, both datasets are preprocessed. Here, the values in topic dimensions of each instance are the occurrence of words from that topic divided by the sum of word occurrences from all topics of that instance. Note that this sum is not the same as a total word occurrence count for that instance as two different topics may have some identical words.

The preprocessed datasets are then spilt into train and test dataset. Then train datasets are used to build a multiclass classification model; in this experiment, All-against-all classifiers are built with Logistic Regression as a base.

The result of Experiment 1 is shown in Table 4.2 where overall performance is presented in the upper table. The first column of the upper table is evaluation measures. Other consecutive columns are grouped into results from LDA and HDP. For LDA, the three columns represent different numbers of topics which are 25, 50 and 100 respectively. As for HDP, since the number of topics for each run is not identical, we report the result from 3 runs of HDP in the following columns.

For the lower table, F-measure in each class is shown here. Bug report types or classes are in the first column while the other columns are identical to the one in the upper table.

Table 4.2. Features Extraction: Comparison Between LDA and HDP

	Overall Performance					
	LDA			HDP		
	25 topics	50 topics	100 topics	1 run	2 run	3 run
Accuracy	62.94%	64.38%	67.04%	63.57%	63.99%	63.62%
F-measure	0.591	0.611	0.641	0.599	0.604	0.601
Roc	0.766	0.785	0.807	0.764	0.767	0.768
Number of Topics	25	50	100	42	46	47
	F-measure for each Type of Bug Report					
BUG	0.729	0.743	0.767	0.733	0.734	0.732
IMPR	0.584	0.605	0.639	0.598	0.607	0.603
RFE	0.006	0.034	0.090	0.006	0.000	0.006
TASK	0.026	0.100	0.091	0.052	0.077	0.066
TEST	0.024	0.100	0.229	0.065	0.068	0.022

As we see in Table 4.2, LDA with 50 and 100 topics perform better than HDP and the trend seems to go up as the number of topics is increased. This can be interpreted in two ways. First, the performance of classifier built from LDA topics will increase more and more as the number of topics increased; or second, the performance will increase until the number of topics reaches a certain point

then it will start to drop. This question is answered in Experiment 2, which indicates that second interpretation is right. Therefore, we can summarize that with a proper number of topics tuning, LDA performance is better than HDP. While this make HDP seems unsuitable for this task, it still has its use, as its performance is still comparable with classifier built from LDA and it requires no parameter.

The F-measure in each class from Table 4.2 demonstrates that for this dataset both LDA and HDP suffer from lack of data and imbalance dataset problems. The F-measure for the three minority classes are terrible. So measure for handling this problem is needed, the interesting approaches are sampling and cost-sensitive technique.

4.4.2 Experiment 2: Characteristics of LDA and its Optimization

As for Experiment 2, we want to search for an answer for two questions. First is whether LDA performance can increase unlimitedly with the increasing number of topics or it will start to drop at some point. The second question is how to optimize LDA for the optimum performance. This is achieved by varying numbers of topics (N), preprocessing methods and classifiers.

The numbers of topics in this experiment start from 50 topics and increase by 50 until it reaches 200. After that, we examine on every 100 topics until it reaches 600.

Three preprocessing methods are experimented on. First, a simple count of words occurrences in each topic is used as a value in topic dimensions. Second, we use the existence of words in the topic, the value will be 1 if for that bug report there is an occurrence of words in that topic and will be 0 if a word from that topic is not found. Third, the preprocessing method used in Experiment 1 is used.

The results for Experiment 2 are shown in Table 4.3, 4.4 and 4.5 The Table 4.3 shows accuracy, Table 4.4 shows weight F-measure and Table 4.5 show Roc. Each table aligns in a similar way. The first column, N is the Number of LDA topics; the rest of columns are grouped base on their preprocessing. The count is

a simple count of words occurrence, Exist uses the existence of word and Ratio is the preprocessing method used in Experiment 1. Each preprocessing column consists of three sub-columns that indicate used classification techniques, ADTree for Alternating Decision Tree, NB for Naive Bayes and LR for Logistic Regression. The black cell means that experiment on that cell position takes too long to finish, thus left blank. The best result for each sub-column is marked with gray color.

Table 4.3. Features Extraction: Accuracy of the LDA Topic-based Classifier

	Count [integer,0-N]			Exist [boolean,0/1]			Ratio [double,0-1]		
N	ADTree	NB	LR	ADTree	NB	LR	ADTree	NB	LR
50	0.62	0.24	0.61	0.60	0.53	0.61	0.62	0.34	0.64
100	0.62	0.32	0.63	0.61	0.52	0.63	0.64	0.35	0.67
150	0.62	0.50	0.63	0.59	0.51	0.62	0.63	0.39	0.67
200	0.61	0.50	0.63	0.60	0.50	0.63	0.62	0.40	0.67
300	0.62	0.51	0.62	0.60	0.50	0.61	0.61	0.42	0.63
400	0.61	0.52	0.62	0.61	0.51		0.62	0.46	0.63
500	0.61	0.51		0.60	0.51		0.60	0.49	0.60
600	0.60	0.52		0.60	0.50		0.61	0.49	
AVG	0.61	0.45	0.62	0.60	0.51	0.62	0.62	0.43	0.65

From ADTree and Logistic Regression columns of these tables, we can see that the performance of LDA topic-based classification model does not increase along with the number of LDA topics. Instead, the performance will increase until reach the certain point depending on the dataset, then it starts to drop. This is because too much increase in a number of topics will lead to too sparse dataset and a lot of uninformative features. As for some Naive Bayes columns that the best perform a number of topics are 500 and 600, this is due to slower increase trend of Naive Bayes, which means that they have not yet reached the optimum performance.

When comparing performance by varying preprocessing methods, the Ratio method is the most promising one. Both ADTree and Logistic Regression perform best with this preprocessing while Naive Bayes, on the other hand, prefers Exist

Table 4.4. Features Extraction: F-Measure of the LDA Topic-based Classifier

	Count [integer,0-N]			Exist [boolean,0/1]			Ratio [double,0-1]		
N	ADTree	NB	LR	ADTree	NB	LR	ADTree	NB	LR
50	0.59	0.29	0.56	0.56	0.53	0.58	0.58	0.40	0.61
100	0.58	0.36	0.60	0.57	0.52	0.60	0.60	0.41	0.64
150	0.59	0.43	0.61	0.56	0.52	0.60	0.60	0.44	0.65
200	0.57	0.44	0.61	0.57	0.51	0.62	0.59	0.45	0.65
300	0.59	0.45	0.59	0.56	0.51	0.60	0.58	0.47	0.62
400	0.58	0.46	0.62	0.57	0.51		0.59	0.49	0.63
500	0.58	0.45		0.56	0.53		0.57	0.52	0.61
600	0.56	0.46		0.56	0.51		0.58	0.52	
AVG	0.58	0.42	0.60	0.56	0.52	0.60	0.59	0.47	0.63

Table 4.5. Features Extraction: ROC of the LDA Topic-based Classifier

	Count [integer,0-N]			Exist [boolean,0/1]			Ratio [double,0-1]		
N	ADTree	NB	LR	ADTree	NB	LR	ADTree	NB	LR
50	0.74	0.59	0.75	0.72	0.69	0.74	0.74	0.68	0.79
100	0.75	0.59	0.78	0.73	0.70	0.76	0.76	0.66	0.81
150	0.75	0.59	0.77	0.72	0.70	0.77	0.76	0.69	0.81
200	0.74	0.59	0.77	0.72	0.71	0.78	0.75	0.66	0.81
300	0.74	0.59	0.74	0.72	0.71	0.76	0.73	0.67	0.78
400	0.74	0.59	0.76	0.73	0.71		0.75	0.66	0.79
500	0.73	0.60		0.72	0.73		0.73	0.68	0.81
600	0.72	0.60		0.72	0.72		0.73	0.68	
AVG	0.74	0.59	0.76	0.72	0.71	0.76	0.74	0.67	0.80

method.

From these three classification techniques, Logistic Regression achieves the best evaluation scores in all three measurements: accuracy, F-measure, and Roc.

Though when the number of topics is exceed 300, its run time increases tremendously. In this regard, ADTree runtime is doing a lot better. Its runtime is more reasonable with a large number of topics and its classifying performance is still comparable to the best performs logistic regression. As for Naive Bayes, although it got the fastest run time, its classifying performances is significantly worse than the other two techniques.

Therefore, for these mentioned reasons. We recommend using Ratio for preprocessing, using all-against-all with Logistic Regression as the base for classification when the size of the dataset is small while using ADTree with a bigger dataset. As for the appropriate number of topics, it depends on the dataset but starts from smaller N value is generally better for both Logistic Regression and ADTree.

4.5. Threats to Validity

This research experiments on published dataset from the previous study. Although data we use are manually inspected with a fixed set of rules, some errors might still occur. The rules for manual inspection also depends on an individual perspective which could be different for each person. These might cause data to change thus cause our classifier to produce different results.

Some of the processes in our research involve random value. For example, HDP and 10-fold cross-validation are both random process. Thus, although we try to repeat our experiment as much as possible to ensure the validity of our results, we cannot guarantee that our results are optimal.

Experiments are done on the limited research subject. All bug reports in our combined dataset come from projects written in Java and using JIRA bug tracker which might not be representative for other programming language or bug tracker system.

4.6. Conclusion

In this chapter, we propose a method for automatically classify bug reports base on its textual information without the need to do a parameter tuning. This fur-

ther reduces time and effort need to process these bug report. In section 4.4.1, the result from our experiment demonstrates that this nonparametric method performance is comparable, though lowers, to the parametric one. In section 4.4.2, we also experiment on how to optimize the bug report classification process that uses parametric method to topic modeling bug reports. The experiments are done on varying topic numbers, preprocessing methods and classification technique. The result could serve as a guideline to efficiently employ this bug report classification process.

Chapter 5

Improving Automation: Improving Cross-Project Defect Prediction

5.1. Background

Defect prediction is a process of predicting the defect-proneness of software modules and is of great importance in organizing and managing scarce testing resources [11, 12]. Defect prediction is known to work well when the prediction model is built using its own historical data [11, 12]. However, given a lack of historical data for a new project, the ability to build an effective defect prediction model becomes a very difficult task, as specifically noted by He et al. [13] and Turhan et al [14].

Several software engineering studies have been working on improving the performance of defect prediction models in the absence of historical data. One of them is cross-project defect prediction [13, 14, 27, 28], which selects and utilizes historical data from other similar projects to fill in the gap. It enables the construction of a prediction model in the otherwise not possible scenario; however, the inherited problem from the classification model has so far been overlooked. Most classification algorithms are developed based on the assumption that all instances are equally important, which means it will usually try to maximize the number of correctly classified instances. Generally, defect prediction dataset

contains more non-defective examples than the defective ones [50]. This class imbalanced issue causes a trained prediction model to be biased toward the majority and thus shifts the decision boundary toward the non-defective class. Software quality teams and researchers are however interested in the defective or minority class [50].

A common preprocessing technique adopted by researchers [51–53] for enhanced defect prediction performance is the application of sampling techniques such as over and under sampling. These techniques are applied to alleviate the negative effects of highly skewed datasets or the imbalanced distribution nature of defect prediction datasets [54]. However, oversampling techniques have been shown to perform better than undersampling techniques in several empirical studies [51–53] whereby more minority or defective examples are added to the dataset. Nevertheless, determining in advance the amount of oversampling required is still a key challenge during the training of a prediction model. This issue is especially more important in the cross-project scenario where the distribution of the defective examples, the percentage of defective modules in a dataset, cannot be easily assumed as in the within project situation. The exact amount of the skewness is varied depending on each project; while some might only have a very low number of defective modules (i.e. 0.081% in NetBSD 0.081), half of the module in another project might be defective (i.e. 0.491 in XFree86). None of the existing cross-project defect prediction studies have also considered this issue when oversampling is applied. Having prior knowledge of the class distribution of the unlabeled data and how this information could be applied to improve the performance of the prediction model would be of great benefit in addressing this challenge.

There are a few related works aiming to solve these problems, such as Ryu et al. [55] and Ryu et al. [56] works which use TCSBoost and Boosting-SVM [57], respectively. However, what has been missing in previous studies of the within and cross-project defect prediction is the possibility of knowing beforehand the expected percentage of defect-prone modules from the unlabeled data without any prior knowledge, and how this information could be applied to improve the prediction performance of the prediction model. This problem is especially more important in the cross-project scenario where the distribution of the defective

modules cannot be easily assumed to be similar to its training dataset as in the within project situation.

This chapter proposes Class Distribution Estimation with Synthetic Minority Oversampling Technique (CDE-SMOTE), a technique for cross-project defect prediction that modifies the distribution of the training dataset according to the estimated distribution of the unlabeled dataset. Whilst it is not practical to assume that the true distribution of the unlabeled data can be obtained, it can be estimated using a quantification approach [40] from the machine learning field. By leveraging this estimated distribution, we can approximate the amount of oversampling required for each dataset and prevent excessive oversampling, which can degrade the prediction performance. The hypotheses and performance of CDE-SMOTE are validated and evaluated through four experiments together with Wilcoxon signed-rank tests. We conduct extensive empirical studies on 14 open-source projects considering all of their possible cross-project pairs for a total of $14 \times 13 = 182$ cross-project pairs, and 7 defect prediction models comprising of 5 base classifiers and 2 ensemble classifiers.

In this study, CDE-SMOTE significantly improved the cross-project defect prediction performance, offering significant improvement in 63% of the cross-project pairs according to Wilcoxon test: with 16.422%, 29.687% and 20.259% improvement for Balance, G-measure, and F-measure, respectively. It also improved the prediction performance compared to the CLAMI [30] and Burak [14] approaches.

5.1.1 Synthetic Minority Oversampling Technique (SMOTE)

These techniques are applied to alleviate the negative effects of highly skewed datasets or the imbalanced distribution nature of defect prediction datasets [54]. However, oversampling techniques have been shown to perform better than undersampling techniques in several empirical studies [51–53, 58] whereby more minority or defective instances are added to the dataset.

Among the oversampling techniques, SMOTE is one of the most prevalent techniques for synthetic data generation [51]. Proposed by Chawla et al. [59], it aims to alleviate the imbalance in the original dataset by synthetically generating new data instances in the region of the minority class so as to shift the classifier

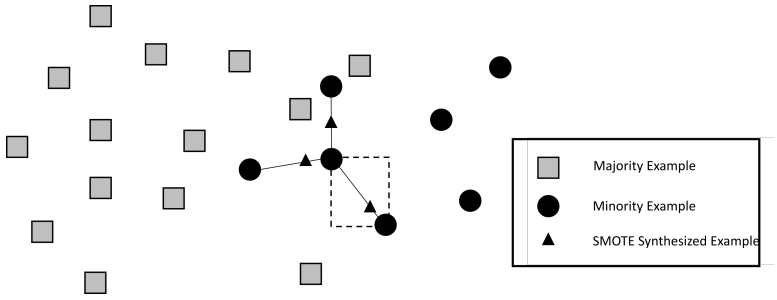

Figure 5.1. Diagram showing how SMOTE works (k=3)

learning bias towards the minority class. Figure 5.1 shows how SMOTE works.

SMOTE generates synthetic examples by:

1. Choose an example from the minority class. The middle circle in Figure 5.1.

2. Find it k-nearest neighbors, they must belong to the minority. In Figure 5.1 these are the three surrounding circles (k=3).

3. Create synthetic examples, between the chosen example and its neighbors, each synthetic example will have each of its features randomized in the value between the chosen and its neighbor. In Figure 5.1, this is represented by the three triangles. The possible location for the synthetic example in the right corner is within the dashed box.

   Then repeats these steps until the target amount of oversampling is reached.

5.1.2 Class Distribution Estimation (CDE)

Class distribution estimation (CDE) or Quantification is a technique in a machine learning [40]. Unlike classification that is interested in the actual label of each instance, quantification is more interested in the distribution of each class; given an unlabeled dataset, a quantification will estimate the proportion of each class in that dataset. This approach has many possible applications; while it has yet to be utilized in the software engineering field, it has been adopted in many other fields [41–43]. Our research uses this estimated class distribution to approximate the amount of oversampling needed for the target unlabeled project.

5.2. Proposed Algorithm

5.2.1 CDE-SMOTE Principles

Our proposed approach, Class Distribution Estimation with Synthetic Minority Oversampling Technique (CDE-SMOTE) aims to reduce the negative effects of a highly skewed dataset in the cross-project defect prediction by using CDE and SMOTE oversampling. The amount of oversampling is decided by the unlabeled dataset estimated distribution in order to prevent excessive oversampling which cause the prediction performance to become lower.

1. **Hypotheses of CDE-SMOTE** The base hypotheses of CDE-SMOTE are:

   (a) **First** The training dataset could be modified to better suit the class distribution of the target unlabeled dataset

   (b) **Second** Without knowing any actual label of the unlabeled dataset, the class distribution of the unlabeled dataset can be estimated.

   The first hypothesis is based on the previous work by [60], though with modification. While the second hypothesis is base on the quantification field [40] in machine learning.

2. **Steps and Procedures** CDE-SMOTE consists of three main steps: class distribution estimation, class distribution modification, and prediction model building.

5.2.2 Steps and Procedures

The first step, class distribution estimation, is shown in Figure 5.2. Our approach starts by building the first classification model from the training dataset, this is an estimator classifier for approximating the class distribution of the unlabeled dataset. The training dataset is a historical data from another software archive that already labeled and modified using SMOTE oversampling to have an equal number of the Defective and the Clean classes. Following that, the unlabeled dataset from the target software project is then labeled by the estimator classifier; this will yield a machine labeled result of the target software. The estimated

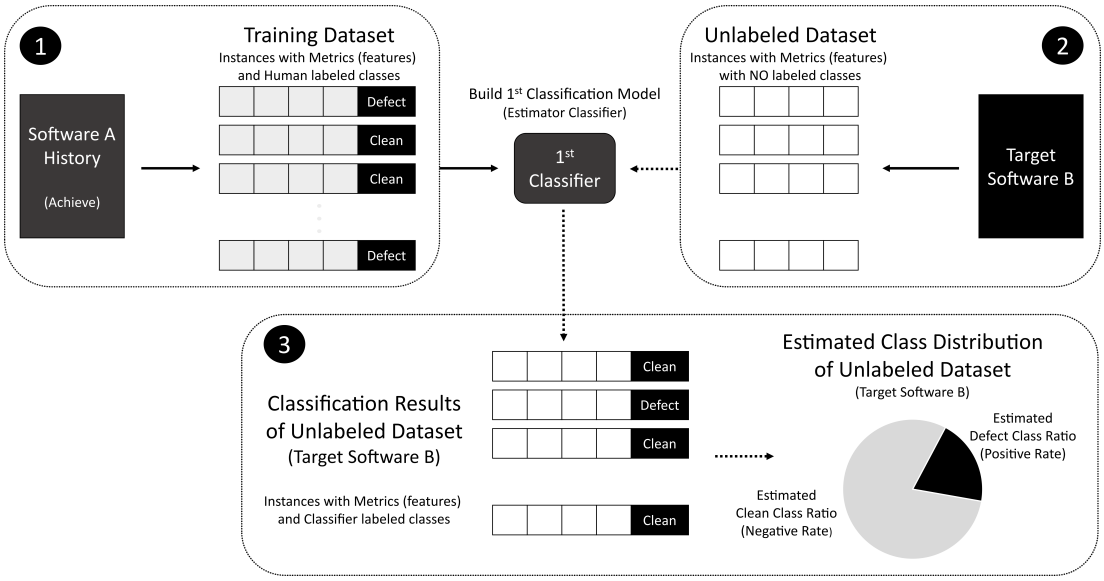

Figure 5.2. CDE-SMOTE Diagram: Class Distribution Estimation

distribution of target software is then obtained by the classification and count (CC) technique [40], simply count the number of machine-labeled instances for each class. Our assumption is that, while there might be some classification mistakes in the first labeled result, the overall distribution should still remain quite accurate. This assumption is investigated in our results related to experiment 2.

The second step, class distribution modification, is shown in Figure 5.3. This part takes the estimated distribution and the training dataset as inputs then output a modified training dataset. The estimated positive rate, the ratio of the number of defective instances to the number of overall instances, is used to dictate the amount of oversampling required. The modification is done by oversampling the original unmodified training dataset, adding synthetic examples to the training dataset until the class distribution of the training data becomes the reverse of the estimated distribution of the unlabeled dataset. To achieve this, synthetic examples will be added to the minority class in the training data, in order to shift the defect prediction model decision boundary toward that minority class. For example, if the distribution of the training is are 6:4 and the estimated distribution of the unlabeled datasets is 8:2, synthetic examples will be added to the class with 4 ratio until the distribution of the training dataset changed

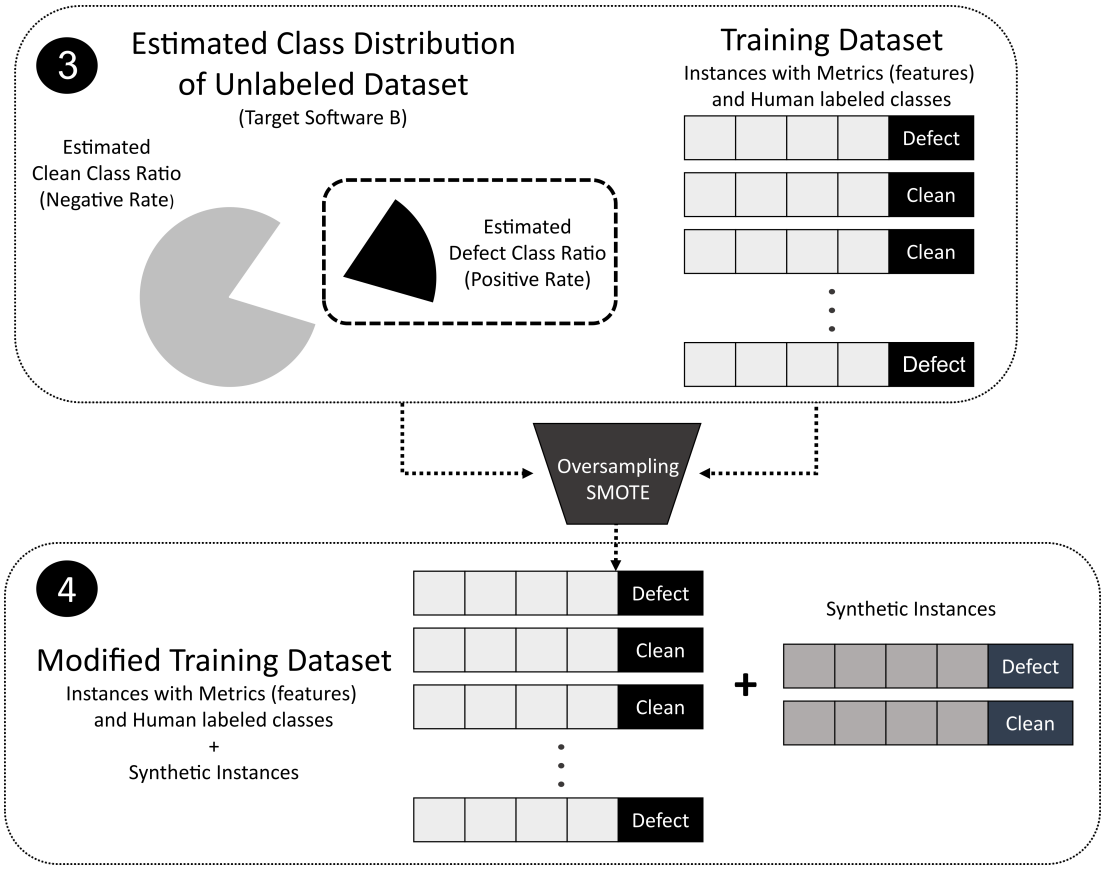

Figure 5.3. CDE-SMOTE Diagram: Class Distribution Modification

to 2:8. These synthetic examples are generated by SMOTE [59], a well-known oversampling technique. Our aim is to improve the prediction performance whilst avoiding the excessive oversampling of the minority class.

The last step, prediction model building, is shown in Figure 5.4, the modified training dataset from the second part is used to create the second classification model, CDE-SMOTE prediction model. The unlabeled dataset from the target software is then labeled by this classifier; its result is final classification results of the target software which is evaluated in our results related to experiment 3 and 4.

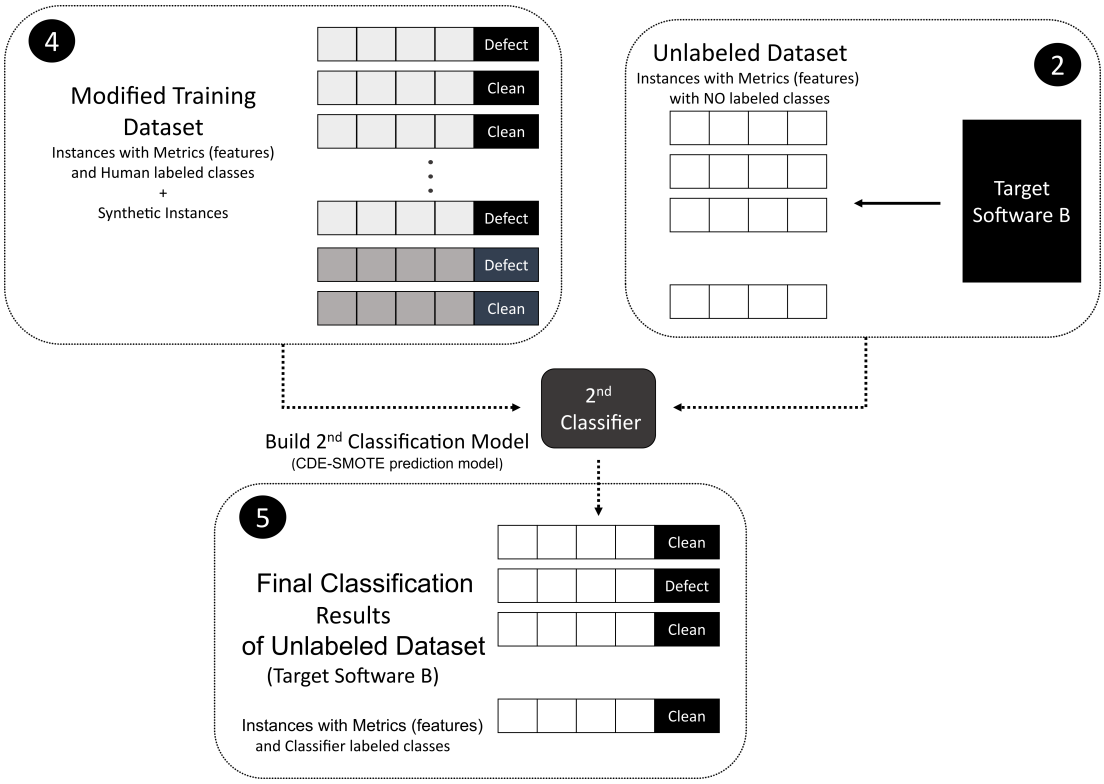

Figure 5.4. CDE-SMOTE Diagram: Prediction Model Building

5.3. Experimental-Design

5.3.1 Experimental Setup

Experimented datasets, evaluation measures, and validation procedure for our base hypotheses are explained in this section.

The cross-project defect prediction experiments are conducted with the following seven classification algorithms:

- J48 Decision Tree (Quinlan 2014)
- Random Forrest (10 trees) (Breiman 2001)
- Nave Bayes (NB) (John et al. 1995)
- Logistic Regression (Le Cessie et al. 1992)

- kNN (k = 3) (Aha et al.. 1991)
- Vote ensemble: Average of Probability (J48+NB) (Kuncheva 2004)
- Vote ensemble: Average of Probability (J48+NB+kNN(3)) (Kuncheva 2004)

Five of them are very well known and commonly used in the defect prediction area, while the remaining two are the ensemble classifiers created from the combinations of these classification algorithms. The classification algorithms used in this chapter are all implemented in WEKA Machine Learning Toolkit, version 3.6.3. [48].

1. **Datasets** The cross-project defect prediction experiments are conducted using 14 datasets, with each dataset extracted from a different open source software project. We deliberately extracted 14 single release version of different open source software engineering projects for the experiment, each with different class distribution as presented in Table 5.1. Our aim is to experiment on a wide variety of class distribution to see whether adjusting the training distribution can help mitigate the negative effect of the difference in class distribution between the training and the unlabeled datasets has on prediction performance. The metrics of each software repository are collected from its commit logs using Git/CVS version control tools to extract seven common process metrics as recommended by Moser et al. [61]. Module labeled as "error" in its commit logs, having error density more than zero, are thus labeled as Defective in our datasets.

   These datasets consist of following nine metrics commonly used in defect prediction [25]; they are shown in Table 5.2:

2. **Evaluation Criteria**

   We have two set of measurements, one for measuring the prediction performance of the defect prediction model and second for measuring the mismatch in the class distribution estimation process.

   For the first set of measures, the evaluation measures used are: probability of detection (PD), probability of false alarm (PF), balance (Bal), G-measure

Table 5.1. CDE-SMOTE: Datasets Class Distribution (Pr)

Datasets	%Defective Module	Datasets	%Defective Module
Clam	0.058	GANYMEDE	0.158
NetBSD	0.081	OpenBSD	0.161
Scilab	0.088	Squid	0.236
OpenNMS	0.102	WineHQ	0.359
Samba	0.113	XFree86	0.491
Helma	0.122	Hylafax	0.511
Spring	0.136	Iipnetfilter	0.616

Table 5.2. CDE-SMOTE: Dimensions (Metrics) of Datasets

Name	Type	Description
CODECHURN	Integer	The total number of lines of code added and deleted from the module.
LOCADDED	Integer	The total number of lines of code added to the module.
LOCDELETED	Integer	The total number of lines of code deleted from the module.
REVISIONS	Integer	Number of revision made to the module.
AGE	Integer	Age of the module.
BUGFIXES	Integer	Number of bug fixed in the module.
REFACTORINGS	Integer	Number of code refactoring made to the module.
LOC	Integer	Number of lines of code in the module.
BUGGINESS	Boolean	Indicate the defect proneness of the module. Defective or Clean.

and F-measure. These measures are widely used in the defect prediction field, which emphasizes the importance of the defective class.

- *Precision*: How accurate is the prediction of the defective class, this is used to calculate the F-measure:

  $$Precision = \frac{\#Correctly\,Predicted\,Defective\,Modules}{\#Predicted\,Defective\,Modules}$$

- *Probability of Detection (PD)*: Recall of the defective class:

  $$PD = \frac{\#Correctly\,Predicted\,Defective\,Modules}{\#Actual\,Defective\,Modules}$$

- *Probability of False Alarm (PF)*: Rate of misprediction of Non-Defective module:

  $$PF = \frac{\#Incorrectly\,Predicted\,NonDefective\,Modules}{\#Actual\,NonDefective\,Modules}$$

- *Balance (Bal)*: The Euclidean distance between (0,1) and (PF, PD) points:

  $$Bal = 1 - \frac{\sqrt{(1-PD)^2+(0-PF)^2}}{\sqrt{2}}$$

- G-measure: The harmonic mean of PD and (1-PF):

  $$G - measure = \frac{2\times PD\times(1-PF)}{PD+(1-PF)}$$

- F-measure (F_1): The harmonic mean of precision and recall. In this chapter, only the F-measure of the Defective-class is evaluated:

  $$F_1 = \frac{2\times Precision\times Recall}{Precision+Recall}$$

The predicted results are compared with the original classifier, classification model built from the unmodified training data, then Wilcoxon signed rank tests are performed. The Wilcoxon Win-Tie-Loss across all the five measures as well as the percentage improvements are shown for evaluation in experiment 1.

The second set of measures aims to measure the performance of class distribution estimation, the measures of this set are: Predicted class distribution mismatch in the actual value and in percentage difference compared to the difference in the actual training and the unlabeled datasets.

- *Positive Rate*: is the distribution of the defective module in a term of the ratio between the number of defectives and the total number of modules:

$PositiveRate = \frac{\#Defective\,Modules}{\#Modules}$

Three *Positive Rate* are used in our experiment:

- $PositiveRate_{Train}$, True distribution of the defective module in the actual unmodified training dataset.
- $PositiveRate_{Unlabeled}$, True distribution of the defective module in the unlabeled dataset.
- $PositiveRate_{Predicted}$, Predicted distribution of the defective module in the unlabeled dataset.

- $ActualMismatch_{Value}$: is the value of positive rate difference between the train and the unlabeled datasets:

$$ActualMismatch_{Value} = |PositiveRate_{Train} - PositiveRate_{Unlabeled}|$$

- $PredictedMismatch_{ActualValue}$: is the value of positive rate difference between the estimation (predicted) and the actual distribution of the unlabeled datasets:

$$PredictedMismatch_{Value} = |PositiveRate_{Predicted} - PositiveRate_{Unlabeled}|$$

- $PredictedMismatch_{\%Diff}$: is the percentage of mismatch difference between the estimation, $PredictedMismatch_{Value}$, and the actual mismatch, $ActualMismatch_{Value}$. Its negative value indicates that the estimated positive rate is closer to the true distribution, on the other hand, the positive value means the estimation is more misleading than the training dataset distribution.

$$PredictedMismatch_{\%Diff} = \frac{PredictedMismatch_{Value} - ActualMismatch_{Value}}{ActualMismatch_{Value}} \times 100$$

3. **Validation Procedure**

Experiment 1: Oracle and Original Classifiers Comparison

The first experiment aims are to validate the first hypothesis and investigate whether the prediction model can be improved if the true distribution

of unlabeled dataset, $PositiveRate_{Unlabeled}$, is know beforehand. By confirming our first hypothesis, we demonstrate the possibility of improving the cross-project prediction model by the modification of training dataset and the danger of applying training data of one project for another project without considering their class distributions.

In this experiment, we assume that the distribution of the unlabeled data is known beforehand; which should be noted that it is not practical in most cross-project defect prediction scenario. This knowledge is used to adjust a number of training instances from each class in the training dataset to make the training dataset more suitable to the current unlabeled dataset. The adjustment is done by adding synthetic examples to the training dataset until the class distribution of the training data becomes the reverse of the actual unlabeled dataset.

This modified training dataset is used to build a classification model to predict the defective modules in the unlabeled dataset. This model is called "Oracle classifier" as it obtained information that would not have been possible to obtain in a normal circumstance. We then examine the prediction results of this model against the true labels of the unlabeled dataset and evaluate its performance. The evaluating measures used are probability of detection (PD), probability of false alarm (PF), balance (Bal), G-measure and F-measure. We then compare the prediction results with the original classifier, that is, the classification model built from the unmodified training data, and lastly applied a statistical test, specifically the Wilcoxon signed rank test to compare the significant difference in the performance of the models. Experiments are performed 14 times, each time one project is selected as a training project to train a classification model. This model is then used to predict the defect in the remaining 13 projects, for the total of $14 \times 13 = 182$ cross-project pairs. Each cross-project pair is studied with the above classification algorithms, for the total of $182 \times 5 = 1,274$ runs. The Wilcoxon test is done for each training dataset selected across all of its cross-project pairs in each measure and reported in terms of Win, Tie or Loss depending on its significance at p ¡ 0.05 two-tailed test. Five Win-Tie-Loss values are reported for each training dataset, as such, the total

runs of Win-Tie-Loss for each classification algorithm on the 14 datasets is $14 \times 5 = 70$.

Experiment 2: Performance of the Class Distribution Estimator

The second experiment aims are to validate the second hypothesis and investigate the performance of quantification technique in the cross-project environment. Since the actual distribution of the testing or unlabeled data, in most cases, is assumed to be unobtainable, the following question is can it be estimated from the unlabeled instances of the testing data? By only using the unlabeled data, the same data used as the input for the already built classification model, the quantification performance of the classification and count (CC) technique in the cross-project environment is examined.

To evaluate the estimation performance, quantification experiments are run on 182 cross-project pairs; each prediction model is built from the labeled historical data from 1 project then used to estimate the 13 remaining unlabeled projects. Estimation performances are is evaluated in term of $PredictedMismatch_{ActualValue}$ and $PredictedMismatch_{\%Diff}$.

Experiment 3: CDE-SMOTE and Original Classifiers Comparison

Given that we can reliably estimate the unlabeled data distribution, can we build a better cross-project defect prediction model based on this estimated value? This experiment presents the practical use of this chapter, to estimate and adjust the training data according to each set of unlabeled data.

The performance of the CDE-SMOTE prediction model is validated in terms of PD, PF, Bal, G-measure and F-measure. The results are compared to the performance of the original classifier using Wilcoxon Win-Tie-Loss and percentage improvement in the same manner as in Experiment 1; the results are shown in Section 5.4.3. In contrast to the experiment conducted in Results related to experiment 1, the actual distribution of the

unlabeled dataset is never used in this experiment as a measure of avoiding contaminating the trained classifier.

Experiment 4: CDE-SMOTE and Related Works Comparison

The fourth experiment aims are to compare the predictive performance of CDE-SMOTE with its related works, to this end, two state of the art filtering techniques discussed in our related works: Burak filter [14] and CLAMI [30], are used as comparisons.

The Burak filter is an approach proposed for selecting the right training examples for the target unlabeled project. This approach filters large quantity of labeled instances, usually consisting of several software engineering projects, and selects only a subset of these combined projects to be used as a training dataset. Each instance in the filtered dataset is selected according to its similarity to the unlabeled instance; for each unlabeled instance, the closest k labeled instances are selected and added to the training dataset.

Our Burak filter experiments consist of 14 runs. As Burak filter assumes that there is a large amount of training dataset composed of historical data from several software engineering projects, for each run, one dataset is selected as the testing/unlabeled whilst the rest (13 datasets) are combined to create a composited labeled dataset with which the Burak filter is applied to. The number of closest instances, k, is set to 10 and the similarity is measured using the Euclidean distance metric. We then applied our CDE-SMOTE to the filtered training dataset and compared its performance to just using the Burak filter alone. The performance is evaluated in terms of increased Balance, G-measure, and F-measure.

The second approach, CLAMI, is the more recent approach. It is an unsupervised threshold approach for identifying the defect-prone modules from an unlabeled dataset. CLAMI starts by calculating the median of each feature or metrics from the unlabeled dataset. The median value for each feature is used as a threshold, values which exceed the corresponding median are identified and marked. CLAMI then counts the number of marked values for each instance and clusters the same number together. The instances are then separated into two big groups: the group with higher and lower

number of marked values, which are labeled as defect-prone and not defect-prone, respectively. After this labeling, CLAMI then performs metrics and instance selections to further refine its labeled dataset. The final labeled dataset is then used as a training dataset for building a defect prediction models.

While the CLAMI and its assumptions are very different from ours, the final goal is the same: to identify the defect-prone modules. To demonstrate the effective performance of our proposed approach, we compare the performance between ours and CLAMI. Different from the Burak experiments where we filter the training dataset using Burak's technique before applying CDE-SMOTE, the CLAMI experiments are not performed on the CLAMI labeled dataset, as our pilot experiment shows that applying CDE-SMOTE to CLAMI labeled dataset often result in either insignificant or detrimental performance. Instead, the performances are compared between only using CDE-SMOTE and only using CLAMI. The other experiment setups experiments are similar to the one in Results related to experiment 3, performed 14 times for the total of $14 \times 13 = 182$ cross-project pairs, we just substitute the original classifier with CLAMI and compared them. In our experiment, the Logistic Regression Classifier is used for CLAMI from its author claim that it performs the best for CLAMI [30].

The results of Both Burak and CLAMI experiment are shown in Section 5.4.4.

5.4. Result

5.4.1 Experiment 1: Oracle and Original Classifiers Comparison

For this experiment, we compared the cross-project defect prediction performance of the oracle classifier with the performance of the classifier built from the original unmodified training dataset. Across 14 extracted datasets, 182 cross-project experiments are conducted and the summarized results in terms of Wilcoxon Win-Tie-Loss comparison between the oracle and the original classifiers for each

select training dataset. The barplot in Figure 5.5 shows the summarized result, the Y axis denotes the number of Win, Tie or Loss while the X axis denotes the classification algorithm used.

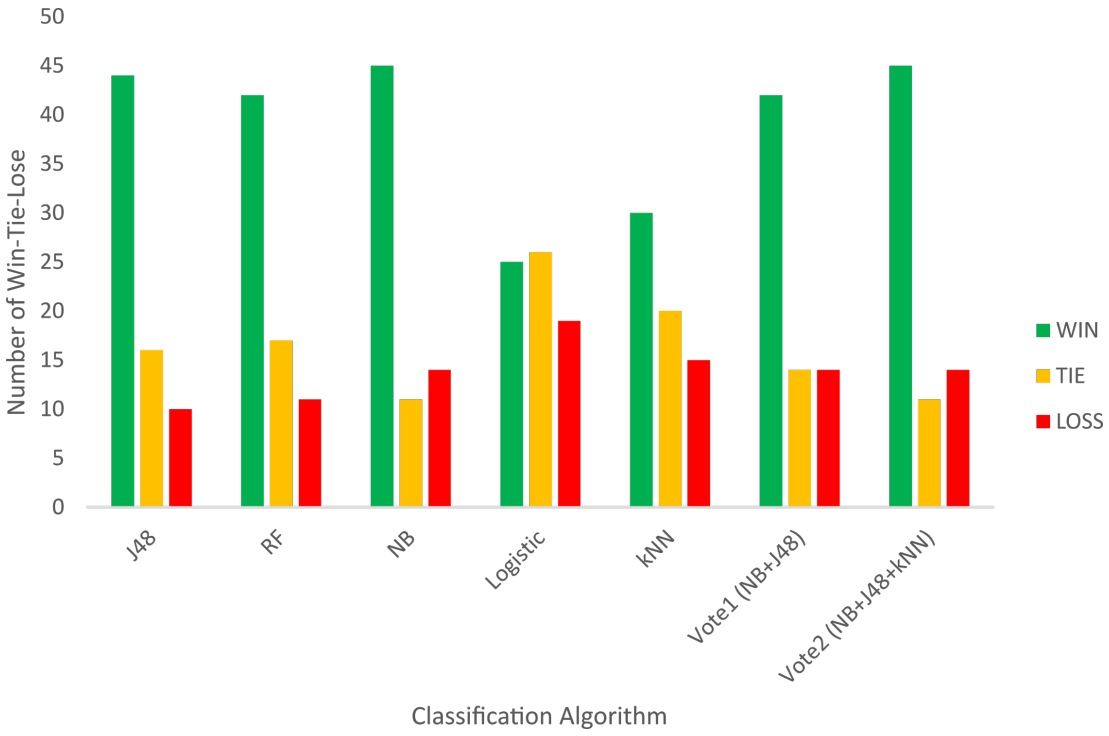

Figure 5.5. CDE-SMOTE: Oracle comparison to the Original Classifier: Wilcoxon Win-Tie-Loss

We observe from Figure 5.5 that modifying the training dataset to be the reverse of the class distribution of the unlabeled dataset can help improve the performance of the cross-project defect prediction models. Out of the 490 Win-Tie-Loss comparison, 70 from each of the 7 classifiers, this approach performed significantly better than the original, 64.286% of the time according to the statistical Wilcoxon test.

The average increase in performance of each classifier is shown in Table 5.3. The table demonstrates the average performance increases, compared to the original predictor, evaluated using the following three measures: Balance, G-measure and F-measure. All measures are the means across 182 cross-project pairs.

Across these seven classification algorithms, we can see the increase in per-

Table 5.3. CDE-SMOTE: Oracle increase performance (%) compared to original classifier [Averaged from 14 x 13 = 182 combinations of cross-project pairs]

	Balance	G-measure	F-measure
J48	23.016	47.296	40.082
RF (10 Trees)	15.628	22.861	22.861
Naive Bayes	10.143	15.726	10.045
Logistic	0.920	-1.380	24.110
kNN (k=3)	11.649	21.262	20.910
Vote 1 (J48+RF)	12.979	20.874	15.490
Vote 2 (J48+RF+kNN)	22.637	39.157	26.224
Averaged	13.853	24.406	22.817

formance across all measures; G-measure and F-measures increase by 24.406% and 22.817%, respectively. Furthermore, the performances of all algorithms have shown at least some improvements. Out of the seven experimented algorithms, aside from a slight F-measure decrease in Logistic Regression, none of the algorithms experienced a performance degradation in the results.

5.4.2 Experiment 2: Performance of the Class Distribution Estimator

As it is not practical to assume that the distribution of the unlabeled data is known beforehand, this experiment aims to investigate the practicality of estimating the distribution of the unlabeled dataset. First, the average mismatch class distributions in term of positive rate (PR) between the estimation and the true value ($PredictedMismatch_{Value}$), averaged from all cross-project pairs for each training dataset, are shown in Figure 5.6. From the graph, the Y-axis denotes the positive rate (PR) mismatch while the X-axis denotes the various training datasets used for training the prediction model. The Logistic Regression and kNN results are excluded from Figure 5.6 as well as our further experiments, since we found in this experiment that these two algorithms could not accurately estimate the distribution of the unlabeled dataset. Their estimations were even

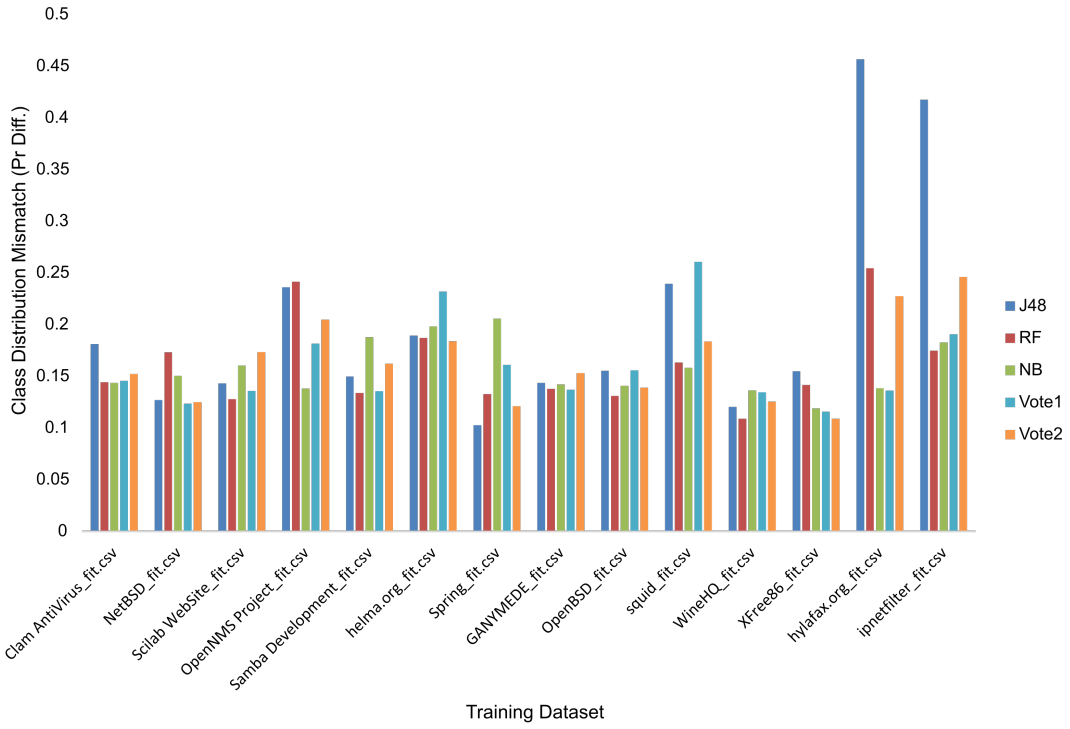

Figure 5.6. CDE-SMOTE: Actual class distribution mismatch ($PredictedMismatch_{Value}$) between train and unlabeled datasets in the actual value

more mismatched than using the original training dataset as it is, generating 160% and 190% more error at maximum compared to just using the unmodified training dataset.

From Figure 5.6, we observe that the remaining five classifiers can estimate the class distribution of the unlabeled dataset. Without any prior information or knowledge about the unlabeled dataset, the class distribution can be estimated with a positive rate (PR) mismatch value of 0.1689 averaged across all classifiers.

Comparing the estimation result with the mismatch of the original training dataset, we show the increase/decrease percentage mismatch difference ($PredictedMismatch_{\%Diff}$) in Figure 5.7. The X-axis displays the training dataset used, each result is averaged across all of its unlabeled datasets. The Y-axis shows the increase and decrease in percentage error of the estimated distribution,

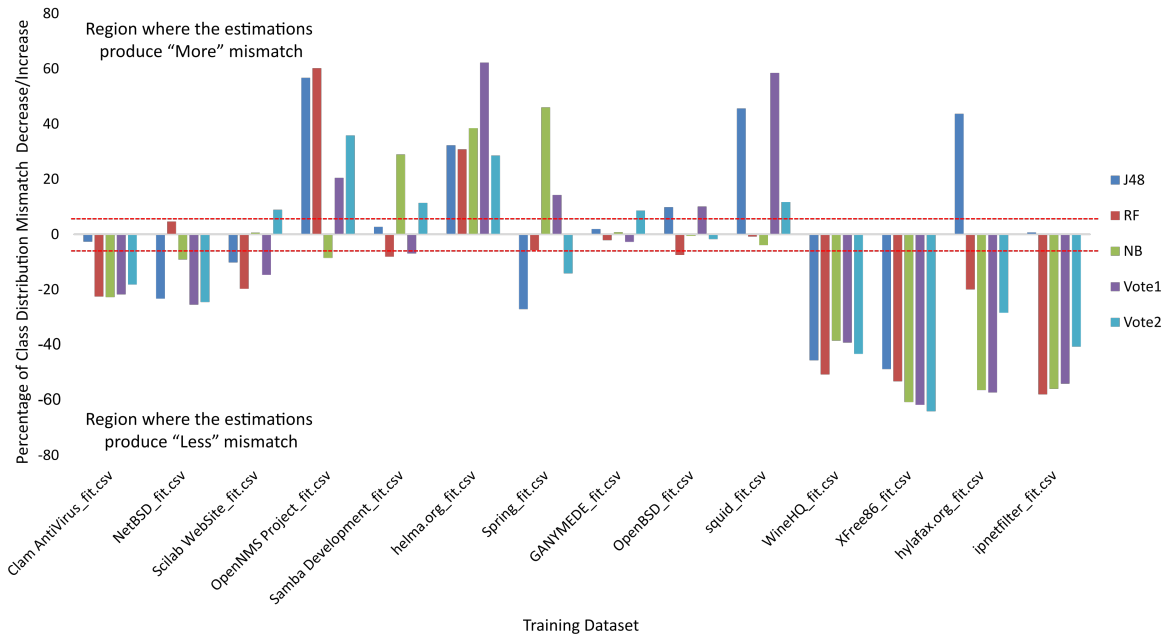

Figure 5.7. CDE-SMOTE: Class Distribution Mismatch Compared to the original Training Data: Percentage differrent betwen actual test data error and CDE estimation ($PredictedMismatch_{\%Diff}$)

$PredictedMismatch_{Value}$, compared to $ActualMismatch_{Value}$. The decrease in error, when the value is less than zero, indicates that the estimated class distribution is more accurate than assuming that the distribution of the unlabeled is the same as the training dataset. While on the contrary, the increase in error implies that the actual performances for each selected training estimation is more misleading than the original unmodified training data. The best possible estimation will decrease the error by 100% maximum (-100%) where the prediction is exactly the same as the true distribution, On the other hand, there is no upper limit for the increase in estimation error ($+\infty$). The two red dotted horizontal lines in the figure show the lines where there are 5% increase and decrease in class distribution mismatch.

From Figure 5.7, we observe that on several instances, our proposed approach accurately estimates the class distribution of the unlabeled dataset. Out of 70 test cases, the estimations reduced the class distribution mismatch more than 5%

in 50% of the cases compared to the 30% significant error when our approach was not applied.

5.4.3 Experiment 3: CDE-SMOTE and Origianl Classifiers Comparison

As demonstrated in Experiment 1, the modified training dataset produces better prediction performance, and in Experiment 2, the class distribution of an unlabeled dataset can be estimated. This experiment aims to investigate the practicality of using this estimated class distribution. The experiment setup is very similar to that in experiment 1, with the only difference being that, we have no prior knowledge about the class distribution of the unlabeled datasets. Rather than use the actual class distribution of the unlabeled dataset which is unknown to us, the estimated class distribution is used. Additionally, as mentioned previously, the Logistic Regression and kNN algorithms are not included in this experiment since they could not accurately estimate the distribution of the unlabeled datasets.

Figure 5.8, shows the Wilcoxon Win-Tie-Loss comparison between our CDE-SMOTE and the original classifier, the Y-axis denotes the number of Win, Tie or Loss while the X-axis denotes the classification algorithm used.

The results in Figure 5.8 shows that the CDE-SMOTE performed much better than the original classifier. Considering the 350 Win-Tie-Loss comparisons, the prediction performances significantly improved in 62.857% of the cases. The J48 and the ensemble Vote 2 (J48+RF+kNN) models accomplished 44% and 50% improvement, respectively.

The increase in performances for the remaining classifiers are shown in Table 5.4 summarized with respect to their Balance, G-measure and F-measure values which are averaged across 182 cross-project pairs.

From Table 5.4, we observe the increase in performances for all measures and all classifier algorithms, especially for J48 and Vote 2 (J48+RF+kNN) cases which exhibited major improvements.

The actual performances for each selected training data are shown in Table 5.5. The first column presents the training dataset, the remaining columns are

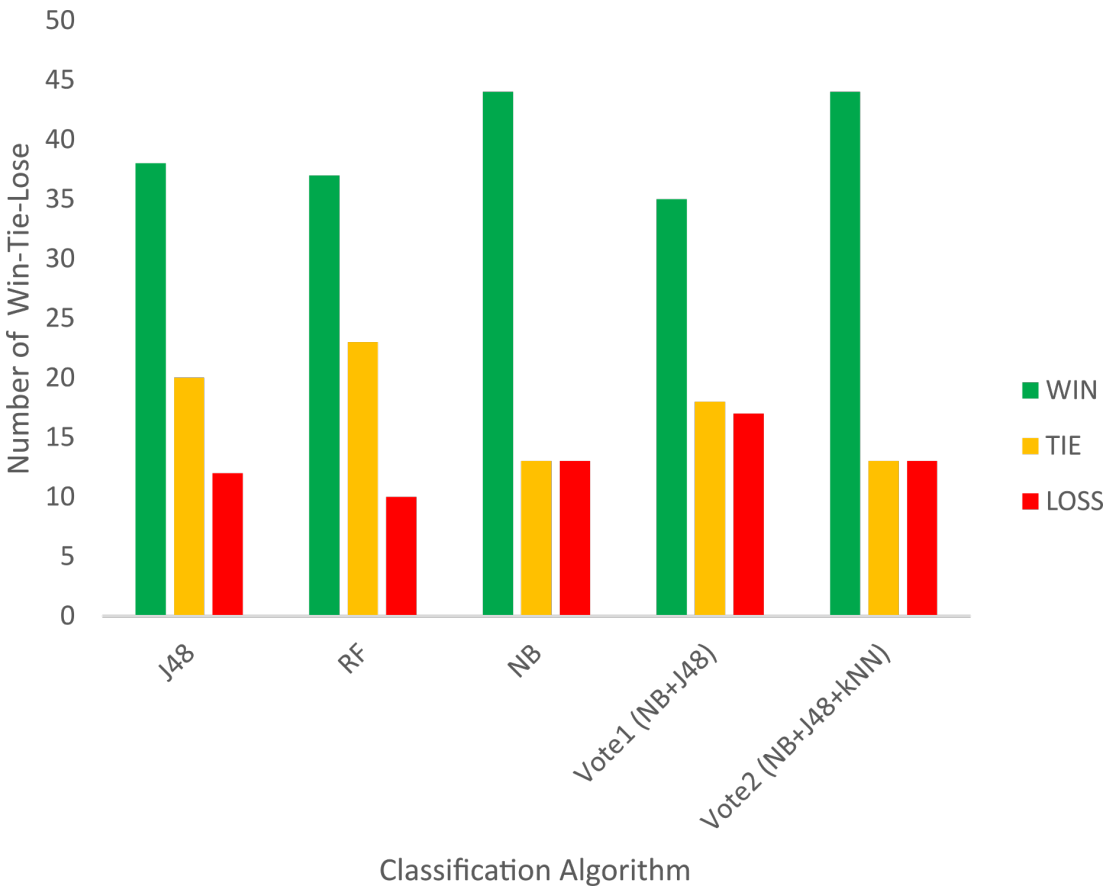

Figure 5.8. CDE-SMOTE: comparison to the Original Classifier: Wilcoxon Win-Tie-Loss

the Balance, G-measure, and F-measure, respectively. The performances shown for each training dataset are the averaged from all 13 cross-project pairs.

5.4.4 Experiment 4: CDE-SMOTE and Related Works Comparison

Aiming to compare our proposed method to other related works, two well-known defect prediction approaches are implemented: Burak filter and CLAMI.

In the Burak filter, experiments were performed across 14 extracted datasets, the average increase performance for each classifier after CDE-SMOTE is applied to the training dataset selected by Burak filter is shown in Table 5.6.

Table 5.4. CDE-SMOTE: increase performance compared to original classifier (Percentage)

	Balance	G-measure	F-measure
J48	20.471	43.455	34.049
RF (10 Trees)	15.487	27.806	21.261
Naive Bayes	11.018	17.467	10.082
Vote 1 (J48+RF)	12.038	19.718	12.163
Vote 2 (J48+RF+kNN)	23.094	39.988	23.741
Averaged	16.422	29.687	20.259

Table 5.5. CDE-SMOTE: cross-project defect prediction performance in terms of Balance, G-measure, and F-measure

Training Dataset	Balance					G-measure					F-measure				
	J48	RF	NB	Vote 1	Vote 2	J48	RF	NB	Vote 1	Vote 2	J48	RF	NB	Vote 1	Vote 2
Clam	0.632	0.655	0.586	0.651	0.631	0.629	0.653	0.569	0.649	0.629	0.448	0.467	0.414	0.458	0.423
NetBSD	0.465	0.446	0.658	0.591	0.623	0.384	0.344	0.657	0.578	0.618	0.302	0.282	0.441	0.419	0.414
Scilab	0.646	0.618	0.583	0.638	0.669	0.644	0.609	0.562	0.633	0.670	0.435	0.421	0.398	0.439	0.459
OpenNMS	0.623	0.630	0.619	0.634	0.634	0.614	0.624	0.602	0.631	0.627	0.430	0.417	0.421	0.419	0.404
Samba	0.602	0.579	0.494	0.591	0.579	0.597	0.562	0.434	0.585	0.568	0.406	0.376	0.327	0.416	0.377
Helma	0.573	0.575	0.643	0.625	0.655	0.548	0.561	0.633	0.617	0.651	0.414	0.382	0.423	0.423	0.435
Spring	0.532	0.508	0.653	0.645	0.603	0.485	0.450	0.644	0.636	0.589	0.378	0.350	0.410	0.422	0.397
GANYMEDE	0.573	0.508	0.639	0.566	0.648	0.546	0.448	0.634	0.540	0.643	0.364	0.302	0.396	0.350	0.405
OpenBSD	0.552	0.535	0.625	0.579	0.616	0.527	0.493	0.620	0.559	0.604	0.356	0.377	0.422	0.368	0.404
Squid	0.563	0.579	0.628	0.598	0.613	0.551	0.563	0.627	0.590	0.611	0.341	0.374	0.397	0.376	0.374
WineHQ	0.578	0.603	0.537	0.573	0.632	0.559	0.592	0.498	0.552	0.626	0.384	0.413	0.356	0.391	0.421
XFree86	0.652	0.671	0.569	0.655	0.672	0.643	0.669	0.544	0.648	0.670	0.426	0.442	0.382	0.426	0.425
Hylafax	0.560	0.620	0.626	0.531	0.631	0.512	0.615	0.612	0.479	0.617	0.353	0.401	0.399	0.312	0.405
Ipnetfilter	0.618	0.666	0.675	0.678	0.653	0.598	0.666	0.674	0.673	0.652	0.402	0.434	0.428	0.427	0.429
Averaged	0.584	0.585	0.610	0.611	0.633	0.560	0.561	0.594	0.598	0.627	0.389	0.388	0.401	0.403	0.412

The results in Table 5.6 indicates that, by taking into account of the distribution difference, the prediction performance could be significantly improved. Compared to just using Burak's filter alone, according to Wilcoxon signed-rank tests at $p \leq 0.05$, our CDE-SMOTE combined with Burak's filter significantly enhanced the prediction performances in four measures: probability of detection (PD), balance (Bal), G-measure and F-measure. This demonstrates that CDE-SMOTE can be used in conjunction with Burak filter and it does provide a significant improvement in prediction performance.

In contrast to the Burak experiments, the CLAMI algorithm was directly

Table 5.6. CDE-SMOTE: Increase performance (Percentage) comparison between Burak Filtered dataset and Burak Filtered dataset with CDE-SMOTE applied

	Balance	G-measure	F-measure
J48	18.852	35.566	24.469
RF (10 Trees)	16.855	32.499	21.044
Naive Bayes	17.262	35.596	21.502
Vote 1 (J48+RF)	22.327	41.538	20.440
Vote 2 (J48+RF+kNN)	28.684	52.873	22.904
Averaged	20.796	39.615	22.072

trained on a single dataset, which was selected as the unlabeled dataset as CLAMI required no training dataset. The results from the 14 trained CLAMI datasets was then compared to the CDE-SMOTE results from Experiment 3. Table 5.7 displays the averaged prediction performances of CLAMI across 14 unlabeled datasets and across 182 cross-project pairs for CDE-SMOTE.

Table 5.7. CDE-SMOTE: Increase performance (Percentage) comparison with CLAMI - When the Cross-Project training dataset is chosen randomly

	Balance	G-measure	F-measure
J48	-7.277	-10.734	-2.033
RF (10 Trees)	-7.013	-10.585	-2.079
Naive Bayes	-3.107	-5.344	1.110
Vote 1 (J48+RF)	-2.899	-4.673	1.708
Vote 2 (J48+RF+kNN)	0.559	-0.041	3.933
Averaged	-3.948	-6.275	0.528

As shown in Table 5.6, performances of the CLAMI approach were really promising. With the ensemble classification model Vote 2 (J48+RF+kNN) being the only model that demonstrated some slight improvement in prediction performance, CLAMI outperformed the other classification models trained with CDE-SMOTE. Whilst the results shows that CLAMI is a very efficient defect pre-

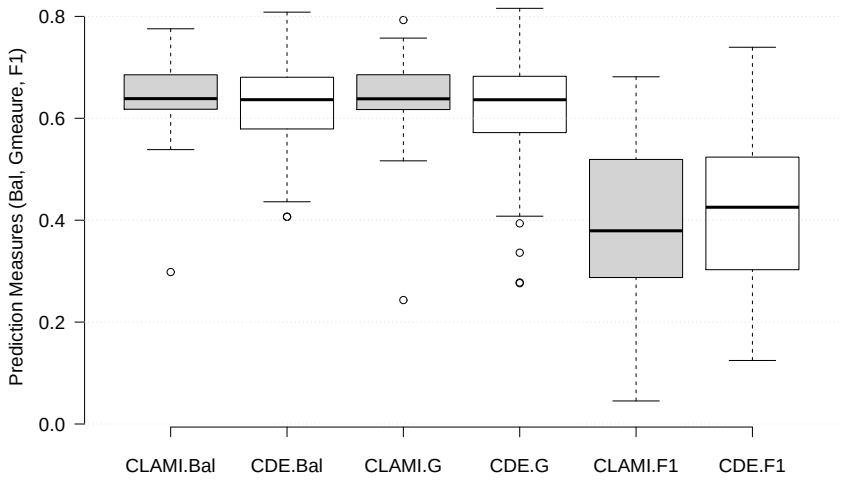

Figure 5.9. CDE-SMOTE: Box plots Perfomance (Actual value) comparison between CLAMI and CDE-SMOTE (VOTE 2) - When the Cross-Project training dataset is chosen randomly

diction approach, it should, however, be noted that, the results computed for the CDE-SMOTE models were trained on cross-project training datasets randomly chosen without considering the similarity between the cross-project pairs projects in contrast to the CLAMI algorithm.

Figure 5.9 shows the performance comparison between CLAMI and CDE-SMOTE Vote 2 (J48+RF+kNN) in terms of Balance (Bal), G-measure (G), and F-measure (F1). The Y-axis denotes the actual value of these measures. Results from CLAMI come from 14 experiments, as it runs on only unlabeled data, while CDE-SMOTE results are from 182 cross-project pairs.

From Figure 5.9, in term of Balance and G-measure there is almost no different between CLAMI and CDE-SMOTE; their medians are exactly the same (0.640); with slightly larger ranges for CDE-SMOTE which suggest lower consistent in these two measures. On the other hand according to the F-measure, CDE-SMOTE offer an improvement over CLAMI with 13.16% increased in median.

In the real-world scenario, the main advantage the Cross-Project defect prediction approach holds over the unsupervised method such as CLAMI, is the ability to select the training dataset that is similar to the target unlabeled project. Aiming to investigate this question, in the three measurements: Balance, G-measure, and F-measure, we do another Win-Tie-Loss comparison for each selected cross-project pair. Win is defined as the case where CDE-SMOTE offers more than 5% improvement than CLAMI, Loss when CLAMI offers more than 5% improvement than ours, and Tie when neither of the cases is true. Each Win-Tie-Loss, contributes 1, 0, and -1 to the selected cross-project pair, the case where the summation of scores is more than 0 is deemed as Success and the rest is considered as No improvement.

Figure 5.10 presents the ratio of Success and No improvement. The 14 unlabeled datasets in total are represented by a barplot represented on the x-axis and each unlabeled dataset consists of 13 scores in percentages distributed among the two results (ratio).

Overall we observe that the overall percentage of the Success cases is 39.010% despite the fact we randomly selected cross-project pairs. Moreover, out of 14 randomly chosen datasets, 12 (85.7%) of them contains at least one case where CDE-SMOTE Success compared to CLAMI, offering the better prediction performances. This shows that CDE-SMOTE could achieve better performance results than the CLAMI algorithm when the training dataset is carefully selected.

In Table 5.8, we present the overall increase in prediction performance when only the Success cross-project pairs are selected. Results for OpenNMS and WineHQ datasets were thus omitted in the table since they were regarded as no Success projects.

Similar to Figure 5.10, the results in Table 5.8 also indicates that when the training datasets are carefully selected, CDE-SMOTE approach could perform significantly better than CLAMI. We observe how CLAMI performs very bad for the NetBSD dataset. Should we try to exclude the results of NetBSD from the table, the average performance improvement is still quite significant, with CDE-SMOTE gaining 7.742406%, 8.045064%, and 18.70076% increments, respectively for Balance, G-measure, and F-measure.

Our results show that although the CLAMI approach is capable of handling

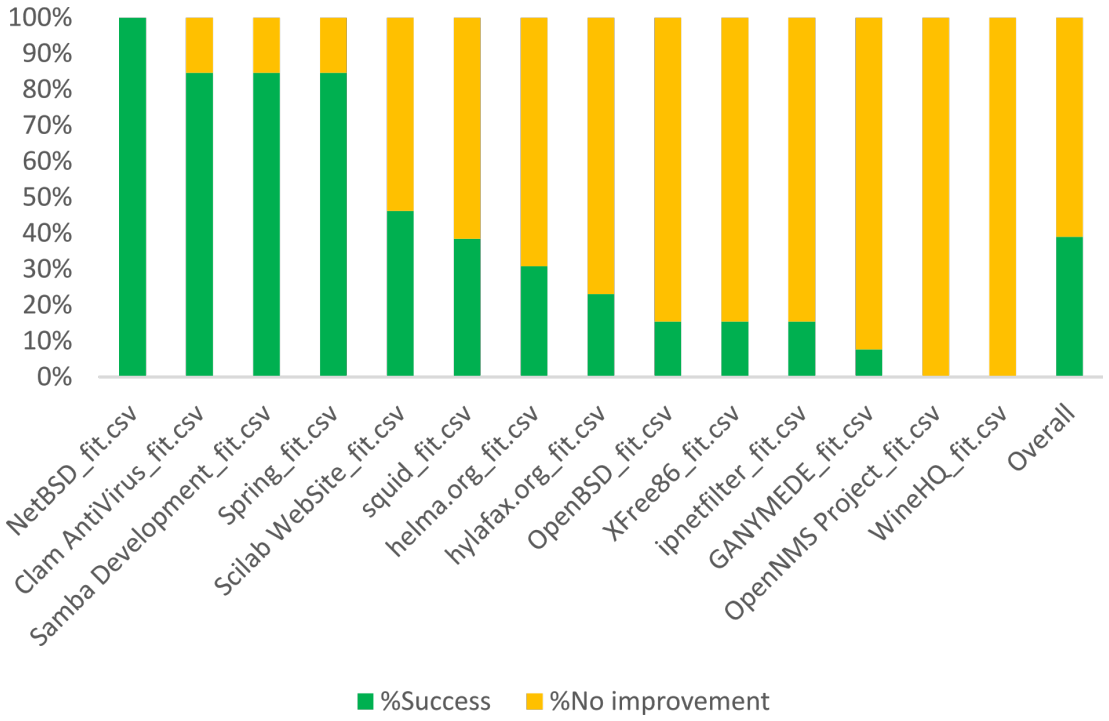

Figure 5.10. CDE-SMOTE: Percentage of cases that CDE-SMOTE shows Significant improve over CLAMI

Table 5.8. CDE-SMOTE: Increase performance (Percentage) comparison with CLAMI - When the Cross-Project training dataset is selected

Training Dataset	Vote 2 (J48+RF+kNN) Increase performance (Percentage)			Training Dataset	Vote 2 (J48+RF+kNN) (Con.) Increase performance (Percentage)		
	Balance	G-measure	F-measure		Balance	G-measure	F-measure
Clam	7.443	7.615	22.223	GANYMEDE	4.122	2.899	6.541
NetBSD	105.686	149.683	452.052	OpenBSD	7.982	8.418	10.416
Scilab	4.996	5.371	21.008	Squid	4.414	4.783	12.256
OpenNMS	-	-	-	WineHQ	-	-	-
Samba	12.330	12.897	38.539	XFree86	4.894	4.923	9.071
Helma	7.865	4.281	24.560	Hylafax	2.408	3.163	6.683
Spring	4.763	4.900	21.778	Ipnetfilter	23.950	29.244	32.635
Averaged	Balance: 15.904		G-measure: 19.848		F-measure: 54.813		

defect prediction very well overall especially for unsupervised prediction, if similar cross-project datasets are available, our proposed approach CDE-SMOTE would offer a significant improvement.

5.5. Discussion

5.5.1 Implications

Our experiments address the practicality of CDE-SMOTE, as well as, our hypotheses and research questions.

Results from Experiment 1 show the danger of applying training data from one project to predict another project without considering their class distributions, and demonstrate that it could be mitigated by modifying the class distribution of training dataset. Using the modified dataset, significantly improvements (increased by at least 5%) can be found in 64% of the test cases according to Wilcoxon signed ranks, and thus validated our first hypothesis.

In Experiment 2, its results demonstrate that the class distribution of an unlabeled dataset can be estimated even in the cross-project scenario, with only 0.1689 positive rate (PR) error on average; compared to just using the unmodified training data, these estimations could significantly reduce the mismatch in 50% of the cases (reduced by at least 5%) which confirms our second hypothesis.

Experiment 3 simulated the practical case of using CDE-SMOTE in real world scenarios. Its results validate our Research Question 3 by confirming that the estimated distribution could be used as a substitute for the actual distribution and could significantly improve (increased by at least 5%) the cross-project defect prediction performance in 63% of the test cases according to Wilcoxon signed ranks.

The fourth Experiment compared CDE-SMOTE with two proposed approaches in literature: Burak filter and CLAMI. According to our results, applying CDE-SMOTE after the Burak filter is applied, can help improve its prediction performance by 27%. When compared to CLAMI: when the training dataset is randomly selected, a slight improvement in F-measure can be expected, while significant performance improvements were observed when similar cross-project

pairs were selected.

Following this, the implications of our results are:

1. When building a cross-project defect prediction model, the class distribution of the training and the intended target projects should be taken into account.

2. The quantification approach from the machine learning field could be applied to the cross-projects scenario.

5.5.2 Validity

The main threats to validity in this study are discussed in this section.

Internal Validity:

As our datasets were extracted from commit logs and faulty modules were labeled based on comments from these logs. Faults that were not reported in such commit logs were thus not included in our dataset and better extraction techniques could be used to ensure all fault data are recorded. As a future study, we will include all possible techniques to record all faulty models aside those in the commit logs.

External Validity:

With our results from the experiments conducted on this limited amount of datasets, we thus cannot guarantee that our results will be able to generalize for every non-experimented projects.

While there are many methods to account for the class distribution such as undersampling, oversampling, resampling and cost-sensitive classifier, only one method of oversampling, SMOTE, is experimented in this chapter. Even if the previous study shows the it is the best approach to handling the imbalanced dataset. There might be a better technique for modifying the training dataset distribution in the cross-project scenario.

In estimating the class distribution of the unlabeled dataset, while there are several way to quantify the class distribution of the unlabeled dataset, only classification and count (CC) technique is investigated. As such we cannot

guarantee that this method is the most suitable approach for estimating the percentage of the defect-prone module in cross-project defect prediction.

Additionally, only several classification algorithms were considered in this study. While these algorithms are widely used in the defect prediction field and we also extend our study to the ensemble techniques, many classification algorithms are still not experimented on. Consideration of more prediction models is left for future studies, in this chapter, only the results from J48 Decision Tree, Random Forests, Nave Bayes, Logistical Regression, kNN, and Vote ensemble: Average of Probability are published.

5.6. Conclusion

This study presents an approach for improving the prediction performance of the defect prediction model. The proposed approach, CDE-SMOTE, alleviates the detrimental effect of class distribution different and highly skew dataset. It can be used by practitioners to predict the defect-proneness of their software-engineering module and could be easily applied to any software engineering project.

In section 5.4.1, results from Experiment 1 show the danger of applying training data from one project to predict another project without considering their class distributions, and demonstrate that it could be mitigated by modifying the class distribution of training dataset. Using the modified dataset, significantly improvements (increased by at least 5%) can be found in 64% of the test cases according to Wilcoxon signed ranks, and thus validated our first hypothesis.

In section 5.4.2, experiment 2 results demonstrate that the class distribution of an unlabeled dataset can be estimated even in the cross-project scenario, with only 0.1689 positive rate (PR) error on average; compared to just using the unmodified training data, these estimations could significantly reduce the mismatch in 50% of the cases (reduced by at least 5%) which confirms our second hypothesis.

In section 5.4.3, experiment 3 simulated the practical case of using CDE-SMOTE in real world scenarios. Its results validate our Research Question 3 by confirming that the estimated distribution could be used as a substitute for the actual distribution and could significantly improve (increased by at least 5%) the

cross-project defect prediction performance in 63% of the test cases according to Wilcoxon signed ranks.

In section 5.4.4, the fourth Experiment compared CDE-SMOTE with two proposed approaches in literature: Burak filter and CLAMI. According to our results, applying CDE-SMOTE after the Burak filter is applied, can help improve its prediction performance by 27%. When compared to CLAMI: when the training dataset is randomly selected, a slight improvement in F-measure can be expected, while significant performance improvements were observed when similar cross-project pairs were selected.

Following this, the implications of our results are:

- When building a cross-project defect prediction model, the class distribution of the training and the intended target projects should be taken into account.

- The quantification approach from the machine learning field could be applied to the cross-projects scenario.

Our approach is validated through four experiments, confirming the validity of each part. Their results demonstrate that CDE-SMOTE could significantly improve the cross-project defect prediction performance. It also supports our underlying theory that the skewness of the unlabeled dataset could be estimated and mitigated by using oversampling to shift decision boundary toward that minority class, hence improve its overall defect prediction performance.

Chapter 6

Improving Automation: Bug Report Categorization with Shortage of Historical Data

6.1. Background

Bug reports offer important insight into the status of the software project; they can be used to estimating the bug-fixing time [1, 32], deciding which bug should be fixed [62], or analyzing the bug type distribution [8, 63]. Categorization is one way to extract meaningful information from bug reports. Traditionally, these reports are inspected and categorized by humans; this approach has good accuracy and flexibility. However, the time it takes to understand each individual report combined with the numerous numbers of reports make manually reading through them impractical or even impossible in many situations [8]. It is clear that in order to use categorization in a practical environment, an automated system is needed.

Many approaches using supervised learning [4,5,44] have been proposed to automate the process of obtaining information from bug reports. These approaches construct a classification model from the training data of the labeled bug reports that can later be used to automatically categorize new incoming data into predetermined labels. The advantage of this approach is that it can greatly reduce the amount of human effort required after a classification model is built. However,

building that said model is, sometimes, quite difficult. The supervised learning approach requires a lot of labeled bug reports to construct its model [4], but these reports are often unavailable in many software engineering projects.

There are mainly two options that can be used to obtain labeled data for the project without historical data. The first approach is manual inspection [8], which as mentioned previously requires a great amount of human effort. The second is a cross-project classification [13, 16, 39], which builds a classification model from the labeled dataset of another project. While this approach has its own advantages, there are also some limitations. Depending on the characteristics between the target and training projects, the same class might not represent the same concept and would be better to differently represent. To illustrate, while both Debian and HttpClient projects have bug reports in a network category, their impacts are starkly different. In an operating system project such as Debian, network bug, while still important, is not as critical as in HttpClient that mainly focus on communicating. Instead of grouping every network bugs in just one class, it would make much more sense for HttpClient to divide this class into several categories.

Another way to obtain knowledge from bug reports is unsupervised learning which extracts information from the underlying structure of unlabeled bug reports. The major advantage of this method is its ability to categorize the bug reports without the need for pre-labeled data. Also, since knowledge obtained by this method is not limited by the pre-determined categories, using this method may allow information to be discovered that might otherwise be easily overlooked by supervised learning.

While using unsupervised learning offers many advantages, this approach has not been applied to classify bug reports. Moreover, this approach still requires some human effort to understand the obtained categories. In other application domains, there have been many attempts to automate a cluster labeling process [4, 64–66] in order to make them easier to comprehend by labeling these clusters with more representative names. Most approaches do this by using either the most prevalent words in that cluster or bigrams of words in the cluster. However, while using these methods makes the cluster result easier to understand to some degree, using the n-gram does not take into account the grammatical meaning

of sentences; its prerequisite is just adjacent terms in a string. Therefore, in many cases, the cluster that is labeled by the verb phrase will be quite counter-intuitive for a human inspector since we normally label a group of things using a noun phrase. For example, between wont connect and connection timeout, the second one is more likely to be a better label candidate.

This chapter extends our workshop paper and gives improvements and further investigation on the performance of our previous categorization framework [67]. In that paper, an almost fully automatic framework was proposed to categorize bug reports, according to their textual contents . That paper also provided a new technique for automatically labeling a cluster using NLP chunking [68] and top words from relevant topics of that cluster.

While the previous paper shows a potential of the proposed framework, there are still many questions left unanswered: Is using topic modeling actually help improve the categorization performance? Is it stable? How well does it perform compared to cross-project classification? These questions are answered in this chapter.

Another problem is that the previous labeling algorithm lacks the variety in its suggesting labels; a few high ranking words dominates almost its entire suggesting labels. For example a word JUnit, although it is representative for its BUG related cluster, appears in 9 out of 10 of the suggesting phrases. This lowers the coverage of the suggesting labels, which is not a good thing. To counter the lack of term variation, this chapter presents a new weighted reduction algorithm to increase the variety of terms in the suggesting labels.

Our result shows that our method can distinguish between different types of bug reports and can categorize them into different groups with a performance result comparable to the supervised learning approach. It also shows that our cluster labeling method can generate representative labels for clusters built in topic vector space.

This chapter is organized as follows. We discuss the Preliminaries in Section II. Section III describes our method. Section IV explains our experimental design. Experiment results are reported in Section V, threats to validity in Section VI, and related works in Section VII. Section VIII concludes our research and future work.

6.1.1 Supervised Learning Approach

Supervised learning is widely used in the area of software engineering; the most prevalent method is a classification that trains a classification model with training dataset to later be used to classified new incoming data [4]. Each instance in the train dataset is labeled with its actual class; these classes are pre-determined and act as prior knowledge which the model will try to learn.

While this approach is widely used and clearly has its own advantages, its major problem lies in its absolute requirement for the prior knowledge; without this information the classification model simply can-not be built and obtaining this knowledge is far from easy. Since to obtain a good classification model, a large amount of training data is needed, human inspection is required in order to prepare this dataset. In Herzig et.al [8], a large amount of time and effort are spent to reclassify bug report categories.

One way for supervised learning to mitigate this problem is cross-project classification [13,16,39]. This method builds a classification model from a dataset from another project instead of using its own. This generally makes obtaining training data become easier, and it also makes the concept that the model learns become much more general. However this generality, sometimes, becomes its own downfall. When the classification model is intended to be used only in one project, the project manager will definitely want a model that best performs on that said project, not one that works best in general. Technically, the decision boundary that the model used to categorize each class will be distorted causing parts of the data to be misclassified.

6.1.2 Unsupervised Learning Approach

Some research in this area uses unsupervised learning to find hidden structures within their data. It is commonly used in bug triaging [10], duplicate bug report detection [2] and in topic modeling [9,44]. The main advantage of this approach is the non-requirement of a training dataset. This greatly helps reduce the amount of effort required to obtain and process prior knowledge which would otherwise be needed for the supervised learning.

Aside from this, the amount of knowledge obtainable from supervised learning

is also limited by its prior knowledge; supervised learning cannot comprehend anything beyond which it is specifically taught. This means that supervised learning will always categorize bug report to the predetermined class which, sometimes, is not the best approach since a certain class might be better represented as two or more in certain situations.

6.1.3 Topic Modeling

Topic modeling is an unsupervised learning technique that captures the underlying structure of the document repository by grouping co-occurrence words into the same topic [18, 19]. The result is a set of topics, a cluster of words that likely to share the same meaning. A document can be associated with topics using a topic proportion vector that indicates what topics that document is associated with. The more the document relates to the topic, the more proportion is assigned to that topic. Compared to the bag-of-words [5, 69], topic modeling can greatly reduce the effect of data sparseness, which is one of the main problems of the word-level approach. In addition, this approach also help reduces synonymy and polysemy problems by grouping the co-occurrence words together. This generally makes documents much easier to distinguish and it reduces the computation time for both supervised and unsupervised learning.

In the software engineering field, there are several research areas that use topic modeling. However most of them are used in bug triaging [10] and duplicate detection [2], while only a few of them are applied to the categorization and knowledge discovery of bug reports. Pingclasai et.al [5] has shown that this area could benefit from topic modeling and could significantly improve classification performance of bug reports by just adopting it instead of using a word-level model [46]. The problem is that using this approach alone is often not enough to understand the underlying structure of bug reports; even with topics proportion demonstrated, comprehending the similarity of each bug report in high dimensional data space is far from easy.

6.2. Proposed Algorithm

We present our methodology in this section. Our framework can be divided into three main phases: Topic Modeling, Clustering and Cluster Labeling. The first phase, Topic Modeling, is to preprocessing raw bug reports textual content. These bug reports are then projected into topic document vectors, a format which can easily be utilized by machine learning algorithms. In the second phase, projected bug reports are categorized according to its textual similarity using clustering algorithm. Finally, in the Cluster Labeling phase, each group of bug reports is labeled by phrases that portray its characteristics.

6.2.1 Topic Modeling Phase

Bug reports are projected onto topic vector space in this phase. This projection processes and transforms bug reports into a more manageable form, and topic modeling is employed instead of the bag-of-words to mitigate data sparseness as well as the effect of word ambiguity. The input of this phase is bug reports in the XML format, and the outputs are topic membership vectors of all the bug reports in the corpus and the top word list of each topic. This list also contains the proportion of each top word in each topic. Our topic modeling phase consists of four steps, presented in Figure 6.1

1. **Parsing:** Raw incoming bug reports are in XML format; therefore, some non-textual information such as tags, attributes and declarations are also included. For each bug report, three sections: title, description and comments are extracted and combined into a single text file.

2. **Tokenization:** In this step, in order to transform these bug reports into a more processive form, we a tokenize stream of text from the previous step. The parsed stream of text is broken into words and unnecessary punctuation is removed.

3. **Stop Words Removal:** Since many words in English hold little to no meaning when alone, they will not provide useful information when transformed into a topic vector space that disregards their position. Therefore, these words are removed. The Stop list from mallet 2.0.7 is used in this step.

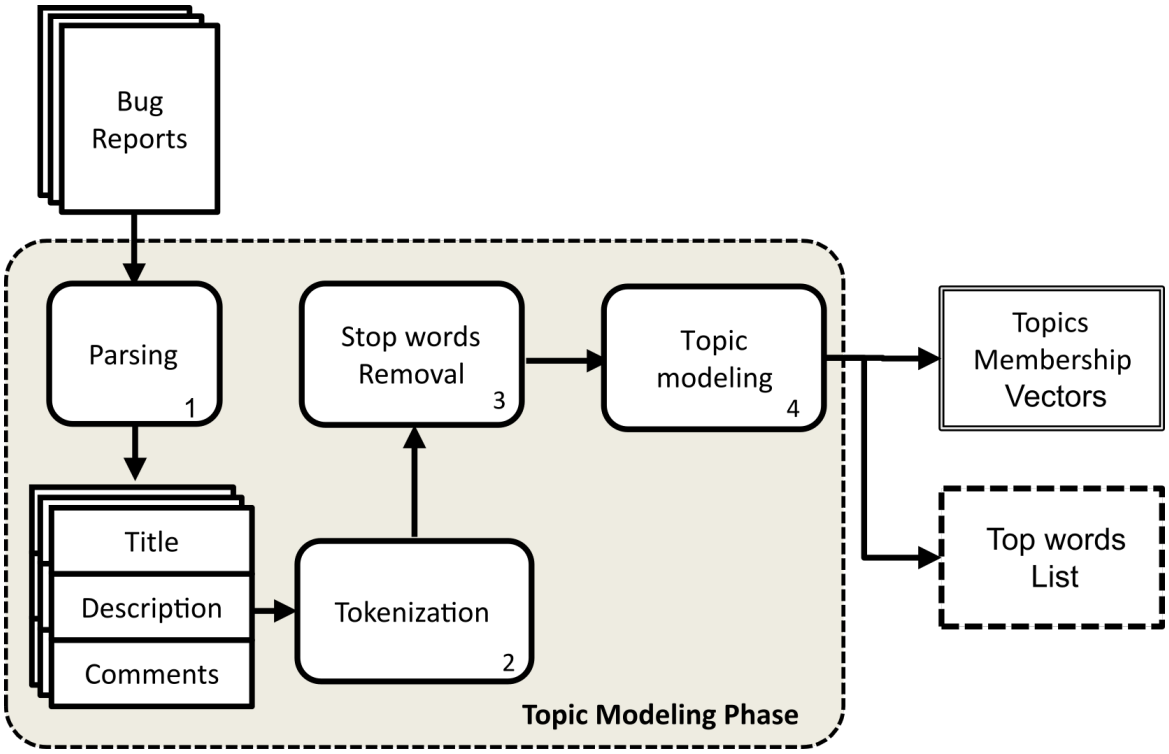

Figure 6.1. Unsupervised: Diagram of the Topic Modeling phase

4. **Topic Modeling:** Topic modeling is applied in this step in order to project bug reports onto the topic vector space. The Hierarchical Dirichlet Process (HDP) [19] is chosen as our topic modeler due to its ability to infer the number of topics automatically, hence reducing tuning effort and making our framework become much more automatic.

   We also provide results from Latent Dirichlet Allocation (LDA) topic modeling [18]. While this process is less automatic and some amount of tuning is required, it could provide a significant performance improvement when properly tuned. Note that even if we only use HDP and LDA in our experiments, our framework is modular. As such, other topic modelers can also be easily applied. This allows our framework to be adjusted according to data structure and the actual situation in which it will be employed.

The outputs of this process are bug reports in topic vector space and the top words list of each topic. Bug reports are represented by a set of topic membership vectors; each vector represents a bug report and consists of a set of topics with its proportion. Frequently co-occurring words are grouped into a topic and top words from each topic are output in the top words list.

6.2.2 Clustering Phase

To categorize bug reports without the need for any prior knowledge of the data, a clustering technique is used. In this phase, topic membership vectors representing bug reports from the previous phase are grouped and categorized according to their textual similarity. Any clustering algorithm can be used in this phase, depending on preferred categorization criteria and available information about dataset structure.

This available information can easily be utilized, which allows a boarder range of algorithm to be used and improves the categorizing performance. For example, when the number of categories existing in the dataset is known, a clustering algorithm that can specify the number of clusters should be employed. On the other hand, if there is no known information about the dataset structure, an algorithm that can automatically infer the number of clusters on its own should be used instead.

The clustering methods used in our experiments are Expectation Maximization (EM) and the X-means algorithm [70]; both are methods that can automatically estimate the number of clusters. All experiments using X-means in this chapter set a minimum number of clusters to 2 and maximum to 10. Both of the clustering algorithms are employed using Weka 3.6.3. [48] The output of this process is the cluster assignment, which indicates which bug report belongs to which cluster. Figure 6.2 summarizes our Clustering phase.

6.2.3 Cluster Labeling Phase

Understanding the result of clustering is difficult because normal clustering techniques do not provide adequate descriptions to understand the meaning behind their results. This means that the task of interpreting the meaning of each cluster

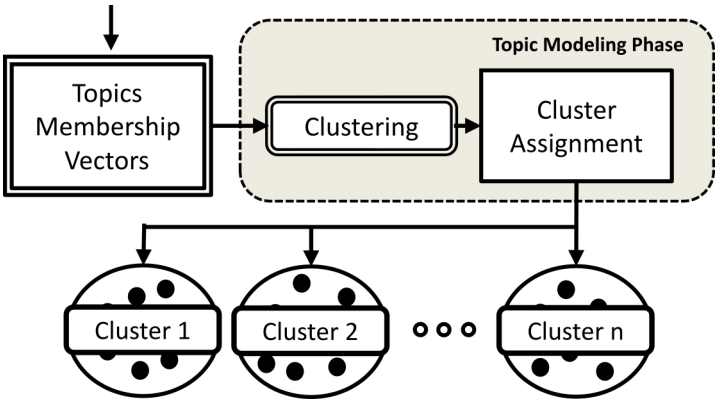

Figure 6.2. Unsupervised: Diagram of the Clustering phase

is often left to its user.

While there are many algorithms already proposed for cluster labeling [64–66] and topic naming [71, 72], they are not directly applicable for our work, which performs clustering in the topic space. Most cluster labeling algorithms are designed for word-level clustering, not for topic-level. Similarly, just finding a set of important topics and then using topic naming algorithms will either limit the labeling result to a set of terms or disregard the term co-occurrence between topics. With this in mind, we suggest three algorithms for labeling clusters in topic vector space.

Title of the closest instance: Cluster Labeling: One of the simplest methods for labeling a cluster of documents, this method sorts the instances (bug reports) in each cluster according to their distance from the cluster center, and then uses the title of the closest instance to label the cluster.

While this method is both straightforward and easy to implement, its result can be misleading; largely because the result depends on how informative the report's title is in that particular bug report.

Adjusted Jensen-Shannon Divergence: Cluster Labeling: Labeling according to the adjusted Jensen-Shannon Divergence, makes slight modifications to the important word extraction method proposed by Carmelet et al [65]. This method selects a set of top ranking topics for each cluster according to two criteria: First, the Jensen-Shannon Divergence, D_{JS}, between the

target cluster and the other clusters, has to be as high as possible. The Jensen-Shannon Divergence is measured for each dimension (topic), from the centroid of the target cluster to the centroid of other clusters. Second, in addition to having the highest possible Jensen-Shannon Divergence, the selected topic has to have an averaged topic distribution higher than the averaged topic distributions of that topic in other clusters. Figure 6.3 shows a diagram of this process. The $\mathrm{MaxD_{JS}}$ can be calculated by Eq.6.1.

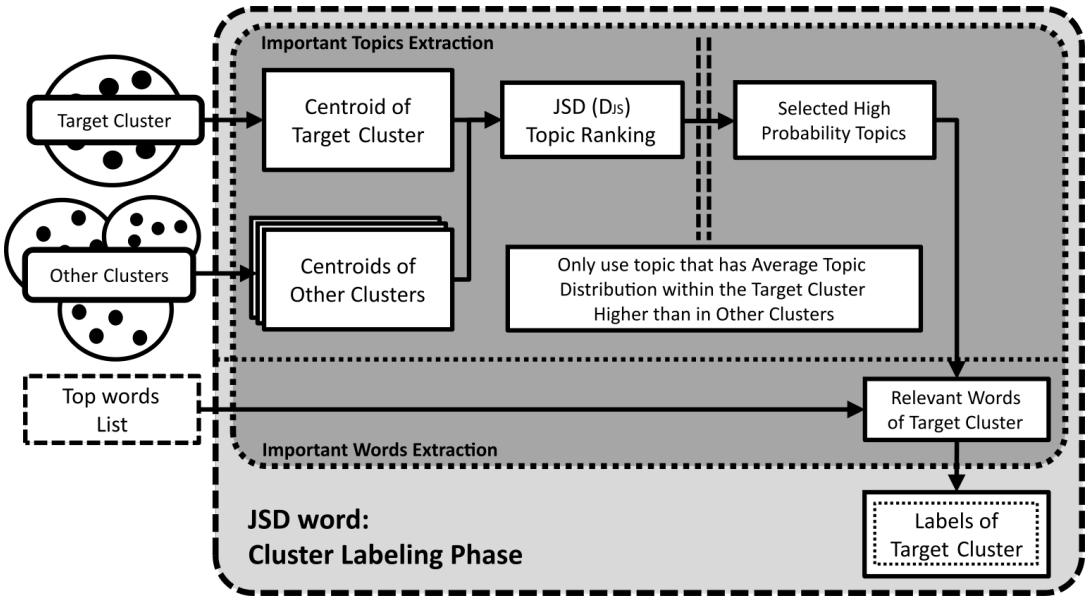

Figure 6.3. Unsupervised: Diagram of Adjusted Jensen-Shannon Divergence: Cluster Labeling

$$MaxD_{JS} = \max_{i=1}^{TopicNum} \left(\sum_{j=1}^{J} P(i) log \frac{P(i)}{M(i,j)} + \sum_{j=1}^{J} Q(i,j) log \frac{Q(i,j)}{M(i,j)} \right) \quad (6.1)$$

P is the centroid of the target cluster; P(i) is the location of P in $\mathrm{Topic_i}$ dimension. Q is centroid of other cluster; Q(i, j) is the location of other $\mathrm{cluster_j}$ in $\mathrm{Topic_i}$. J is the number of clusters. M(i,j) is computed by Eq.6.2.

$$M(i,j) = \frac{1}{2 \times (P(i) + Q(i,j))} \quad (6.2)$$

After obtaining a set of top ranking topics for each cluster, the top words of these clusters are then assigned as the clusters' labels.

NLP Chunk: Cluster Labeling: This novel method takes term co-occurrence between topics into account, labeling clusters with noun phrases extracted via a NLP chunker. This method consists of four main steps: Important Topics Extraction, Important Words Extraction, Noun Phrase Extraction, and NLP Chunk labeling. These steps are summarized in Figure 6.4.

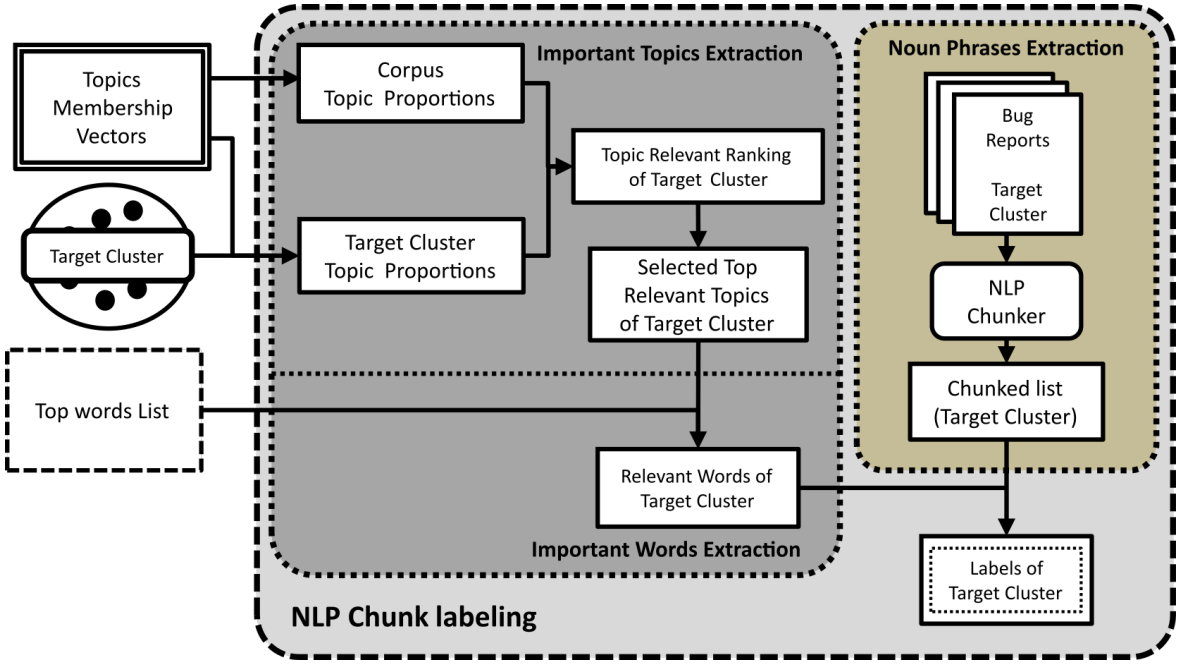

Figure 6.4. Unsupervised: Diagram of NLP Chunk: Cluster Labeling

1. **Important Topic Extraction:** The inputs to this step are topic membership vectors, the output of 6.2.1, and the assignment of the interested cluster from 6.2.2. A set of important topics for each cluster is acquired by ranking topics according to the ratio between each cluster's average probability distribution in that cluster and the average probability distribution of the entire dataset. The idea is that the topics with the highest probability distribution in the specific cluster when compared to the corpus average are

likely to be the best representatives for that cluster. This phrase, however, does not directly take each topic word distribution into account, but only selecting a top important topic for each cluster.

The average probability distribution of topic_i in cluster_j is equal to the sum of topic i proportion from all documents in cluster j divided by the number of documents in cluster j. The equation is show in Eq. 6.3.

$$AVGTopicDistribution(Topic_i, Cluster_j) = \frac{1}{n_j} \sum_{Report_x \in Cluster_j} TopicDistribution(Topic_i,\ Report_x) \tag{6.3}$$

The variable n_j is the number of instances (bug reports) that are assigned to cluster_j. For the entire dataset average distribution, simply replace cluster_j with the set of all instances in the bug report corpus.

The ratio of each Topic_i in cluster_j can then be calculated by dividing the average Topic_i proportion of cluster_j with the average Topic_i proportion of the entire document corpus. The equation is shown in Eq. 6.4.

$$Ratio(Topic_i, Cluster_j) = \frac{AVGTopicDistribution(Topic_i, Cluster_j)}{AVGTopicDistribution(Topic_i, EntireCorpus)} \tag{6.4}$$

All topics are ranked according to their ratio, and then output as a set of the top relevant topics. These topics can also be used to generate the label list of the cluster, by selecting the top word of each topic in the set. This relevant topics set is used as input for the following step.

2. **Important Word Extraction:** The previous step does not take the word distribution into account, this means it will neglects the second and later ranking words in the topic even though they might have probability distribution closed to the top one. Since same word can appear multiple times in the different topics, it is entirely possible that using only the topic distribution ratio will result in discarding potential representative words. To solve this problem, we propose using Eq. 6.5 to ranking words obtain from

the set of top relevant topics.

$$WordScore(word_k, Cluster_j) = \sum_{i=1}^{TopicNum} Ratio(Topic_i, Cluster_j) \times P(word_k|i) \tag{6.5}$$

$P(word_k|i)$ represents the probability distribution of $word_k$ in $topic_i$, and this combined with the summation of all relevant topics makes it possible for subsequent words to obtain a higher TermScore than a top word if it appears multiple times in several topics. Each word is then sorted according to their WordScore into a relevant word list, which can also be used as a label list. This relevant word list is used as input for the last step, NLP Chunks labeling.

3. **Noun Phrase Extraction:** In this step, a set of noun phrases are extracted from the interested cluster. The inputs for this step are raw bug reports from the interested cluster. A Natural Language Processing (NLP) chunker from Apache OpenNLP [68] Library version 1.5.3 is used to extract a set of noun phrases from these bug reports. A noun phrase is used instead of other phrase types, as it is more informative and more prevalent [73]. The output of this phrase, the chunked list, is used as input for the next step.

4. **NLP Chunks Labeling:** This step labels each cluster with its most representative noun phrases. The inputs for this step are a relevant word list and a chunked list. Each noun phrase in the chunked list is scored according to Eq. 6.6.

$$PhraseScore(Phrase_m, Cluster_j) = \sum_{k \in Phrase_m} WordScore(word_k, Cluster_j) \tag{6.6}$$

The $Phrase_m$ score in $Cluster_j$ is the summation of the WordScore of all words in $Phrase_m$. After the score calculations are complete, all phrases are then sorted using their score.

However, we found that using this scoring method alone will usually result in only a few of the most important words dominating the label list. To give the label list more variety, the score of words that already appeared several times in the phrase list are adjusted to have less and less impact on

the PhraseScore. This is done by reducing the score of the sorted phrase list, from Eq.6.6, using Eq.6.7.

$$ReductionScore(Phrase_m, Cluster_j) = \sum_{k \in Phrase_m} WordScore(word_k, Cluster_j) \times \frac{X_k^2}{(X_k^2 + 1)} \tag{6.7}$$

X is the number of times $word_k$ already appears in the higher ranking phrases. After this score reduction, the phrase list is then sorted again to represent its new ranking. The output of this process is a noun phrase list representing the interested cluster.

6.3. Experimental Design

We describe the measurements used, the experimented datasets, and the experimental-design in this section. The goal of these experiments is to demonstrate the performance of the proposed framework in various perspectives, and the specific goals for each experiment are given in the experimental-design subsection.

6.3.1 Measurements

In this section, we describe five measurements used in our experiments: Cluster's Purity, Accuracy, Precision, Recall and F-measure (F1). All clustering results are evaluated via external evaluation, by using known class labels.

Cluster's Purity: Cluster purity is a simple measure to evaluate how pure the cluster's result is according to the predetermine categories, classes. This measure can be calculated using Eq.6.8.

$$Purity(ClusterResult, Categories) = \frac{1}{n} \sum_j max_i |Cluster_j \cap Class_i| \tag{6.8}$$

The ClusterResult is a set of Clusters, {$Cluster_1$, $Cluster_2$,..., $Cluster_j$} while Categories represent the set of categories (classes) {$Categories_1$, $Categories_2$,...,

$\text{Categories}_i\}$. The n variable is the total number of instances in the entire data corpus.

To summarize, each cluster will be assigned to its predominant class. However, in the case where there is a class not assigned to any cluster, a cluster containing the highest ratio between the number of unassigned class and the number of instances in that cluster, will be reassigned.

Accuracy: Accuracy is similar to the cluster's purity, but it is used on the classification result instead of clustering. It grants a basic idea of how many instances are correctly classified in relation to the total number of instances. The arithmetic equation is shown in Eq. 6.9.

$$Accuracy = \frac{|Correctly\,Classified\,Instance|}{Total\,number\,of\,instance} \tag{6.9}$$

F-measure: F-measure, or F1, is used to evaluate the results of both clustering and classification. In the case of clustering, the same method as in cluster's purity is used to assign a class to each cluster. This measure is a harmonic means of precision and recall, which can be computed via Eq. 6.10 and Eq. 6.11, respectively. The calculation is applied to each class separately, then combined using a weight average.

A true positive indicates the number of instances in the interested class, a positive class, that are correctly classified while a false positive represents the number of instances from other classes mis-assigned into the positive class. A false negative is the number of positive classes that are incorrectly classified to any other classes.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{6.10}$$

This precision provides us with the insight as to how accurate the prediction is, in relation to the number of the predictions made.

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{6.11}$$

Recall indicates the completeness of the prediction in the interested class. In other words, recall shows how many instances in the interested class that

the classifier misses, in relation to the actual number of instances in the interested class.

When the two measures above are calculated, the f-measure can be computed using Eq. 6.12.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{6.12}$$

After obtaining the f-measure for each class, the final overall result is summarized using the weight average which takes the number of instances from each class into account.

6.3.2 Datasets

The experimented bug reports are from three datasets of our previous research [8], and which are manually classified by experts using a fixed set of rules. To represent our method performance by two possible situations, we use two sets of binary classes to evaluate its performance. First is to distinguish between the different projects, to represent this problem; namely, the bug reports from the Lucene and Jackrabbit (JCR) projects are combined. We then try to categorize the merged dataset back into its original categories: Lucene and JCR. This dataset is used in Experiment 2.

Second, we classify bug reports into their corresponding types. Originally, these bug reports were categorized into six categories. However, as the increasing number of categories make it harder to accurately classify bug reports [4, 5, 44], the number of categories is reduced to preserve the process performance. The used categories are BUG and Other Requests types; the Other Requests type is a composite type consisting of four non-bug report types: Improvement, Request for Enchantment (RFE), Task and Test. These datasets are used in Experiment 1,3,4 and 5. Each project class distribution is shown in Table 6.1.

6.3.3 Design For Each Experiment

There are a total of five experiments in this research, and each is designed to evaluate the different aspects of our method. The detail and idea behind each

Table 6.1. Unsupervised: Class distribution of the experimented datasets

Project	Total Number of Instances	Number of "Bug" Type	Number of "Other Requests" Type
Lucene	2382	1037	1345
Jackrabbit (JCR)	2328	1213	1115
HTTPClient	731	469	262

experiment are given below.

Experiment 1: Comparison between Bag-of-Words and Topic Modeling

The first experiment is designed to evaluate the performance of different ways to represent a document corpus. We experiment on a well known word-level approach, the bag-of-words model, using both term frequencies and term weighting. Then we compare their results with the topic modeling approaches.

First, the document corpus is transformed into term frequencies; each instance is a bug report and the value in each dimension represents the frequency of the associated term appearing in that particular bug report. After the transformation, the dataset is further processed to eliminate out the terms with an insignificant appearance, so that dimensions with a frequency below twenty are filtered out.

Tf-idf stands for term frequencyinverse document frequency, and it is a term weighting method regularly applied to the bag-of-words model. The tf-idf scores for each term in each document are calculated via the following equations.

$$tf(term_{k,}Doc_p) = freq(term_{k,}Doc_p) \tag{6.13}$$

Tf can be computed using Eq. 6.13. It represents how many times term_k appears in document_p.

$$idf(term_{k,}Doc_p) = log\frac{Number\,of\,Documents\,in\,the\,Corpus}{1 + Number\,of\,Documents\,Containing\,term_k} \tag{6.14}$$

Idf is calculated by Eq. 6.14. This measure indicates the significant of the term_k, according to its appearance in the entire data corpus. The less the term is found in other documents, the better its score.

$$tf - idf(term_{k,}Doc_p) = tf(term_{k,}Doc_p) \times idf(term_{k,}Doc_p) \tag{6.15}$$

After both tf and idf are obtained, the tf-idf can be calculated using Eq. 6.15. This tf-idf represents the importance of term_k in document_p.

Topic modeling is a natural language processing (NLP) process, and in this research, it is used to transform the bug report corpus into a vector matrix of documents and topics. Each topic modeling's topic is a group of words assumed to represent the same or similar meaning. There are two topic modelers used in this experiment, Latent Dirichlet Allocation (LDA) [18] and the Hierarchical Dirichlet Process (HDP) [19]. LDA is a method commonly used in research [5]; however, the method requires the user to specify the number of topics. This makes the method rather inconvenient when there is no prior knowledge about the dataset.

The second topic modeler, HDP, has the advantage that it can infer the number of topics automatically. The number of LDA's topics in this experiment are set to be similar to the HDP's topics. Since the output of each run from these topic modelers can be slightly different, due to their probabilistic property, the experiments are done on three separate runs for each topic modeler. These results are then averaged and shown in section 6.4.1.

These two topic modeling methods are experimented on to decide which method is more suitable for our categorization method.

Four document vector corpora are used as an input for X-means clustering [70]. The number of minimum clusters is two and the maximum is ten; each cluster is labeled according to the method described in subsection 6.3.1. As X-means results can change according to the cluster seeds used, we perform an experiment on one thousand different cluster seeds for each corpus, then average the results. All approaches are evaluated by five measurements described in section 6.3.1. This experiment is done on HttpClient and JCR dataset, by categorizing bug reports into BUG and Other Requests.

Experiment 2: Categorizing bug reports from two different projects

This experiment measures the ability of our method when trying to distinguish between two groups of bug reports that are not closely related. To achieve this, the bug report from the projects Lucene and JCR, are combined. The HDP is then employed to the topic model to the combined dataset, according to the results in Experiment 1. We then try to categorized bug reports from this dataset into their actual projects using four different approaches.

The first two are our unsupervised method employed using two different clustering algorithms, X-means [70] and Expectation Maximization (EM) Clustering [74], both capable of inferring the appropriate number of clusters each on their own.

For comparison, we compared our unsupervised method results with two classification algorithms; J48, an implementation of the C4.5 decision tree [75] and Logistic Regression [76]. These two supervised methods are used as an upper bound of the unsupervised method, as they are created from the prior knowledge (labeled training dataset) in which we assume that it may not exist or is very costly to obtain. All methods are evaluated in term of the cluster's purity/accuracy and f-measure. The results shown are the average values obtained from three separated HDP's runs. Both classification methods are trained and tested using 10-fold cross-validation.

Experiment 3: Categorizing bug reports into Bug and Other Requests

The goal of this experiment is to evaluate our method performance in a more complex situation, and to distinguish between the different types of bug reports in the same project. Different from the previous experiment, each project is separately experimented on; the experimented on projects are HTTPClient, JCR, and Lucene. The results of the X-means clustering are from the average of one thousand runs. Aside from this, the settings of this experiment are similar to 6.3.3.

Experiment 4: Cross-Project Classification Comparison

In this experiment, we look into more competitive methods of handling the lack of prior knowledge. Generally, when we want to categorize a project

in which no labeled training dataset is available, we usually employ cross-project classification [13]. Cross-project classification trains a classification model from the different but similar projects that already have training data readily available. While there are still some limitations, for many projects, this method is certainly more realistic than creating a whole new training dataset from scratch. Here, the cross-project classifiers are trained using training dataset from first one, and then two projects, and is then compared with our unsupervised approach.

Experiment 5: Cluster Labeling Results Comparison

This experiment shows the results of several cluster labeling algorithms. Most cluster labeling research is done on the word-document vectors [65], not the topic ones, so that some processes are required to make them capable of properly handling topic-level clusters. In this experiment, there is a total of three labeling methods, as described in section 6.2.3

6.4. Results

In this section, we present five experimental results as described in section 6.3.3. The experiments are performed on Lucene, Jackrabbit (JCR) and HTTPClient datasets.

6.4.1 Experiment 1: Comparison Between Bag-of-Words and Topic Modeling

While previous studies have shown the advantages of topic modeling in classification [5] and information retrieval (IR) [2,9], its efficiency in bug report clustering has hardly been explored. The goal of our experiment is to verify the benefits of using topic modeling and selecting the best topic modeler that is most compatible with our framework. Four different methods for representing the document corpus are investigated: Tf, Tf-idf, LDA, and HDP. The set of document vectors from each method is then used as input for X-means clustering. The experiment is done on HTTPClient and JCR datasets.

Table 6.2 presents the result of this experiment. The lower part of the table shows F-measure: per class and Weight Average summarizing the overall performance from both classes

Table 6.2. Unsupervised: Experiment 1 - Comparison between different Dataset Dimension

		HTTPClient				JCR			
		Bag-of-Words		Topic Model		Bag-of-Words		Topic Model	
		Tf	Tf-Idf	LDA	HDP	Count	Tf-Idf	LDA	HDP
Overall	Num Dim	1220	1220	52	52.67	1922	1922	52	55.67
	Accuracy	0.645	0.573	0.559	0.596	0.533	0.553	0.620	0.625
BUG	Precision	0.646	0.676	0.747	0.711	0.529	0.585	0.639	0.638
	Recall	0.991	0.725	0.490	0.641	0.938	0.653	0.661	0.649
	F-measure	0.782	0.629	0.539	0.656	0.676	0.576	0.595	0.625
Other	Precision	0.712	0.625	0.452	0.459	0.601	0.580	0.642	0.640
	Recall	0.025	0.302	0.684	0.516	0.091	0.443	0.575	0.600
	F-measure	0.048	0.224	0.510	0.467	0.155	0.436	0.564	0.605
WeightAVG	Precision	0.669	0.658	0.642	0.621	0.564	0.583	0.640	0.639
	Recall	0.645	0.573	0.559	0.596	0.533	0.553	0.620	0.625
	F-measure	0.519	0.484	0.529	0.588	0.426	0.509	0.580	0.616

From Table 6.2, we see that even with the removal of the terms with less frequency, the bag-of-words approaches still have a very high number of dimensions. This results in a very sparse high-dimensional space degrading the clustering efficiency. In both datasets, the topic model approaches perform better than the word-level approaches, and have higher weight average F-measures in all situations. While the Tf method has better accuracy in HTTPClient dataset, its performance in Other class is clearly suffered due to a heavy bias toward the majority class (BUG). This confirms that using topic modeling can efficiently reduce the sparseness of the bug report dataset and project it into a space more suitable for clustering.

When compared between the two topic modelers, the HDP method is shown to be more compatible with our method. The results of using HDP document

vectors are better than LDA in almost all aspects. Further more, HDP is also more automated than LDA,and is capable of assuming the number of topics by itself. With these two factors combined, we are confident that HDP topic modeling is more compatible with our method for representing a set of document vectors.

6.4.2 Experiment 2: Categorizing Bug Reports from Different Projects

This experiment evaluates the performance of our unsupervised framework in the simple task of capturing the structural differences of bug reports from two software engineering projects. In this experiment, our framework adopted two clustering methods, X-means and Expectation Maximization (EM), with both capable of assuming the number of cluster on their's own. The upper bounds are created using two, well-known classification algorithms: J48 and Logistic Regression. Both supervised and unsupervised learning are performed on the same HDP dimensions. The result is shown in Table 6.3.

Table 6.3. Unsupervised: Experiment 2 - Categorizing bug reports from different projects

		Train 2 dataset/ Lucene + JCR			
Overall	Num Instance	4710			
	Num Topic	57.333			
	Method	Xmeans	EM	J48	Logistic
	Accuracy	0.8671	0.7939	0.9155	0.9646
F1	JCR	0.8675	0.7891	0.9147	0.9640
	Lucene	0.8666	0.7914	0.9163	0.9653
	WeightAVG	0.8670	0.7876	0.9157	0.9647

As shown in Table 6.3, the performance of our method, while lower, is still comparable to the supervised learning one. For the X-means approach, its average F-measure is 5.62 and 11.26 percent lower than its upper-bounds, J48 and Logistic Regression, respectively. On the other hand, the EM clustering performance is

not as decent; its performance is 16.26 and 22.49 percent lower, despite having a higher number of clusters (12.667 while X-means only created 4).

6.4.3 Experiment 3: Categorizing Bug Reports in one Project into Bug and Other Requests (In-Project Classification)

This experiment categorizes bug reports from the one project into two different classes: BUG and Other Requests. The experiments are done on three different datasets: HTTPClient, JCR, and Lucene. Bug reports in each project are transformed into three sets of topic document vectors, using three separated HDP runs with the same setting. In each set, we perform 1,000 runs with the X-means approach for a total of 9,000 runs in this experiment.

The result is shown in Table 6.4. The value in each cell is averaged from three HDP runs, except for the X-means, which is from 3,000 runs.

Table 6.4. Unsupervised: Experiment 3 - Categorizing bug reports in one project into Bug and Other Requests

		HTTPClient			JCR			Lucene		
Overall	Instance	731			2328			2382		
	NumTopic	52.667			55.667			47.667		
	Methods	Xmeans	J48	Logistic	Xmeans	J48	Logistic	Xmeans	J48	Logistic
	Accuracy	0.596	0.626	0.710	0.625	0.636	0.745	0.601	0.633	0.710
F1	BUG	0.656	0.717	0.787	0.625	0.632	0.732	0.438	0.565	0.634
	OtherReq	0.467	0.448	0.550	0.605	0.639	0.756	0.688	0.682	0.760
	WeightAVG	0.588	0.621	0.702	0.616	0.636	0.745	0.579	0.631	0.705

Table 6.4 illustrates how this task of categorizing bug reports from the same project is a much harder task; all approaches perform worse here compared to the previous experiment.

As shown in Table 6.4, the F-measures of our X-means approach and J48 are still in close proximity, with a 5.90 percent difference in the average F-measure

from three projects. This shows that even in a complex environment the performance of our framework is comparable to J48, the well-known classifier.

When compared to Logistic Regression, the difference here is significant; our method average F-measure is 0.123 (20.71 percent) lower. However, as mentioned previously, sometimes obtaining a training dataset in the first place can be quite challenging. In those situations, using our unsupervised framework can provide a good solution to the problem, as no training set is required.

6.4.4 Experiment 4: Comparison Between Our Method and Cross-Project Classification

While Experiments 6.4.2 and 6.4.3 are performed in a circumstance where the supervised approach has a very clear advantage, learning directly from the desired project, this experiment provides a more competitive situation.

Here, our unsupervised framework stays the same as in Experiment 6.4.3, while the classification models are trained using labeled datasets from other projects [13]. From the three datasets: HTTPClient, JCR, and Lucene, cross-project classification is performed using all possible combinations. Bug reports are transformed into three sets of topic vectors using three separated HDP runs, and their results are averaged.

Figure 6.5 presents this experimental result. The Y-axis is an average of the weighted F-measure while the X-axis shows the used methods, which are grouped according to the testing projects.

Figure 6.5 shows that, without any training data, the performance of our framework is better than the J48 by 5.16 percent. However, when compared to the Logistic Regression, its performance is 7.63 percent lower. While this indicates that if labeled datasets from similar projects are available, Logistic Regression can perform significantly better than our framework, but our framework can still provide a good alternative. As in many scenarios, even with cross-project classification, a labeled data matched with the desired categories can still be hard to acquire due to the numerous ways that bug reports can be categorized. For example: the types of package conflict [63], the types of bug report [8], the types of bugs, etc.

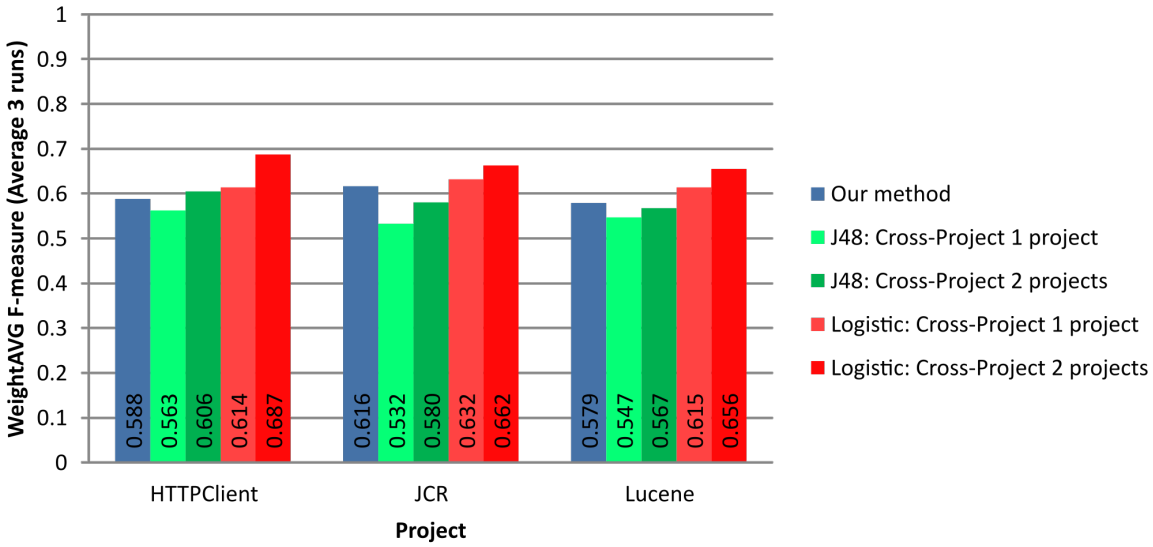

Figure 6.5. Unsupervised: F-measure comparison between Our Unsupervised Method and Cross-Project Classification

The four experiments above demonstrate that our framework is capable of classify bug reports automatically and reliably.

6.4.5 Experiment 5: Cluster Labeling Results

This experiment evaluates our cluster labeling process. Because there are only a few methods that can be directly applied to the cluster in topic space, we modify a cluster labeling method proposed by Carmelet et al [65]. This modified labeling method (JSD) and labeling using the title of the closest instance method (Section6.2.3) are employed as to provide comparison methods for cluster labeling.

Table 6.5 shows part of the labeling result, and this result is from one of the runs on the HTTPClient dataset from Experiment 2. The columns in this Table are the original cluster ID, while each row shows the cluster labels. The labeling methods are JSD, Labeling using the NLP Chunk, and the Title of the Closest Instance. These method are described in Section 6.2.3.

On the lower part of the table are rows that represent the number of instances

Table 6.5. Unsupervised: Experiment 5 - Cluster Labeling Results

	HttpClient		
	HDP 2 run		
	Cluster 1 BUG	Cluster 2 Others	Cluster 3 BUG
Jensen Shannon Divergence (JSD)	**connection connection connection connection** abstracthttpclient.java **timeout** getmethod rulebasedcollator logs **exception debug**	cache patch serializable equals encoding httpclient mockito socket encoding configuration	**cookie** uri http **authentication** http guide file **authentication debug** http
Labeling using NLP Chunk	**http request connection timeout**, **multi threaded connection manager**, **closed socket**, **s catch stale connection**	chunked encoding, alive debug headers transfer encoding, transfer encoding header, cache responses headers	**cookie domain attribute**, **cookie domain**, **domain public credentials**, **domain com cookies**
Title of Closet Instance	**Connection is not released back to the pool if a runtime**, **An IOException or RuntimeException leaves the underlying socket in an undetermined state**	RequestEntity EntityEnclosingMethod have inconsistent Javadocs use deprecated variables, Do not consume the remaining response content if the connection is to be closed	Invalid redirects are not corrected, **DigestScheme.authenticate returns invalid authorization string when algorithm is null**
Num Class BUG	89	163	217
Num Class Others	30	135	97

in each class: BUG and Other Requests. Each cluster is labeled by its majority class; the detail method is described in Section 6.3.1.

From the example result from Table 6.5, clusters 1 and 3 are labeled as BUG classes. Their results are reliable, having an internal cluster purity around 0.7. On the other hand, cluster 2 which is assigned to Other class has not quite as good a result. The bug reports within it are mixed, containing a high number of reports from both classes. The three labeling methods are present in Table 6.5 label cluster with different granularities: word, phrase and sentence level. We only discuss the quality of labels from cluster 1 and 3, as their purity are quite high so that it is easier to identify the main subjects in each cluster.

For the word level, with labeling using Jensen Shannon Divergence (JSD), we can see that while some labels are representative and make sense; e.g. "connection", "timeout", "exception" and "debug" in cluster 1, the majority of its reports are related to "connection problems". However some of its labels, such as "abstracthttpclient.java", "getmethod" or "rulebasedcollator", are clearly not that suitable to be a cluster label. Out of ten suggested labels, the numbers of usable labels are six and four, in cluster 1 and 3, respectively. This shows that using a word level approach for labeling a cluster of bug reports can introduce some problems since, unlike in a general document, many software engineering terms are too specific to be understood without their context.

For the sentence level, labeling is done using the title of the closest instance to the cluster center, and while it can provide a decent idea of what bug reports in that cluster are related to, there are a few problems. First, it largely depends on how informative the title is. As the report's title is written by just one submitter, whose knowledge related to that particular report is not guaranteed, it is very possible that its title will be misleading or not contain enough information. For example, the title "Invalid redirects are not corrected" is too short and can not be easily assigned to either of the two categories.

For the phrase level the generated labels are quite representative. In cluster 1, all of the top four labels are related to "connection" which is the main topic of cluster 1. Moreover, three out of these labels are associated with "connection problems", topics strongly correlated with the BUG category to which the majority of reports in cluster 1 were classified. The same trend can be found in

cluster 3, related to "authentication", which means that all the suggested phrases are representative as they all contain words like "cookie" or "credentials". This indicates that this labeling method can provide a high quality set of labels that are both compact and representative.

6.5. Threats to Validity

This research is subject to threats to validity induced by the limitations of our approach. The most important threats are listed below.

6.5.1 Measurements used

The evaluation measurements used in this research are: cluster purity, accuracy, precision, recall and F-measure. While these measurements are well-known and commonly used in many past studies, we still can not guarantee that the result would be the same if other measurements are used instead.

6.5.2 The categories of bug reports

As mentioned previously, bug reports can be categorized in many ways. Our experiments are performed only on two sets of categories, and as such, the performance of our framework in other sets of categories still cannot be guaranteed.

6.5.3 Experimented Datasets

Our experiments are performed on the dataset of previous study. Even if these datasets are manually inspected and cross-validated, some errors might still remain, which could slightly change the result of our experiments. The research subjects of our experiment are also limited; all experiments are performed on bug reports of projects written in Java using the JIRA bug tracker. This data might not be representative of other programming languages or bug tracker systems.

6.5.4 Heuristics parameters

Some parameters used in our approach and evaluation are heuristics. These parameters are:

- The number of top words for topic modeling is set at 50 for both the HDP and LDA topic modelers
- The minimum and maximum number of clusters for the X-means algorithm (Set to two and ten receptively)
- Number of suggested labels for each label algorithm in Experiment 6.4.4

6.5.5 Coverage of other possible parameters and approaches

Due to the modular property of our framework, there are simply too many ways that it can be adjusted. While this property is intended and beneficial, there are just too many possible setting combinations for us to experiment on. For instance, instead of LDA or HDP, the topic modeling algorithm can easily be changed, the same could be said for the clustering algorithm as well.

6.6. Conclusion

In this chapter we propose a framework for categorizing bug reports automatically by utilizing topic modeling and clustering algorithms. Compared to the traditional supervised learning method, our framework provides the definitive benefit of not requiring any training dataset, while still having comparable, albeit lower, performance to the supervised approach. Our framework could be deployed to automatically classify bug reports for a newly deployed project, to categorize bug reports in to several groups based on its text or to help as a label suggestion system for manual data inspection.

In addition, we also presented a new cluster labeling method that can be used in topic space. Different from previous approaches, it takes into account both topic and word distribution. Its labels are also created from a noun phrase, making them both compact and meaningful.

Future work is to further improve the categorization performance while still retaining its nonparametric and non prior knowledge properties. To further prove the generality of our framework, we also aim to do more experiments on a wider range of categories, projects, and bug tracker systems.

Chapter 7

Conclusion

This dissertation focuses on how to reduce the amount of human effort required in software development. The motivation came from when we were collecting, processing, and preparing our software engineering dataset for further analysis. What we have found is that a good number of man hours are needed to complete these given tasks, weeks were spent in order to sort through and labeled data according to the given categories.

All the works we did point us to the major problem in the software development, how difficult it is to actually deploy and used automated software engineering techniques. While the outcome of these techniques can be very rewarding, the process of deploying them is often difficult and labor intensive hindering their used in the real life scenario. To improve the ease of deployment for software engineering techniques, this dissertation investigates various perspectives and means of how could we further reinforce these automated techniques to minimize the human resources required. Two of the most important tasks in software development-bug report categorization and defect prediction are the focuses of this dissertation.

Bug report categorization, the task of categorizing unlabeled bug reports into the target categories, can serve several purposes in software development. As most of bug report's information is in natural language form, the majority of categorization approach employ NLP techniques to transform the textual information into the formats that are easier for further processing, namely to the feature vectors format. To this end, one of the most recent techniques used are

topic modeling which combines several occurring wording into a topic then used them as features. NLP technique holds many advantages over the previous techniques used in the bug report categorization field. However, the main problems of it are its requirement for the number of topics and how difficult it is to efficiently estimate the appropriate number of topics, as its largely depended on the target bug report archive. This means for the most part when a project wants to use this topic modeling technique a certain amount of human resource is needed to spend on parameter tuning. Our dissertation experiments on using nonparametric topic modeling and proposes a method for automatically classify bug reports base on its textual information without the need to do a parameter tuning which further reduces time and effort need to process these bug report. The result from our experiment demonstrates that this nonparametric method performance is comparable, though lowers, to the parametric one.

Another obstacle for the deployment of software engineering tasks is the requirement for labeled historical data. As the labeling process for a new training dataset is very costly, a project with unavailable labeled data will struggle to deploy the supervised automated approach. To this end, there are two main solutions: cross-project and unsupervised approaches. The first solution, cross-project approach, can achieve better performance but has more limitation; as it still required the knowledge from another similar project. If the said similar project is not obtainable, then the unsupervised approach is needed instead. Our dissertation proposed solutions for both situations, an enchant in performance when the cross-project approach is possible, and purely unsupervised solution for when it can not be applied. For the cross-project solution, The proposed approach, CDE-SMOTE, alleviates the detrimental effect of class distribution different and highly skew dataset. It can be used by practitioners to predict the defect-proneness of their software-engineering module and could be easily applied to any software engineering project. While for the unsupervised, the framework for categorizing bug reports automatically by utilizing topic modeling and clustering algorithms is proposed, compared to the traditional supervised learning method, our framework provides the definitive benefit of not requiring any training dataset, while still having comparable, albeit lower, performance to the supervised approach. Our framework could be deployed to automatically

classify bug reports for a newly deployed project, to categorize bug reports into several groups based on its text or to help as a label suggestion system for manual data inspection. In addition, we also presented a new cluster labeling method that can be used in topic space. Different from previous approaches, it takes into account both topic and word distribution. Its labels are also created from a noun phrase, making them both compact and meaningful.

We believe that this research could contribute to the following:

- We proposed the nonparametric approach to topic model the natural language in the bug reports and show that the new features still retain the pattern which can easily be categorized by classifier algorithm
- We demonstrate the detrimental effects of building a cross-project model without considering the distribution of the intended target projects and how to improve the prediction performance using an estimated distribution.
- We confirm that the class distribution of the unlabeled project could be estimated even in the cross-projects situation, and provides the guideline of how to estimate this distribution.
- We proposed an unsupervised categorization framework for bug reports that could perform even when there is no historical dataset and has a comparable performance to the supervised approach.
- We proposed a new cluster labeling algorithm for topic features utilizing both topic and word distribution.

7.1. Future Work

For the future work, we will investigate more on the factors related to our three main topics:

- For automatic topic modeling, we plan to tackle the lack of data and imbalanced dataset, the problems found in multiclass bug report corpus. We also want to improve the nonparametric classification performance. Last, we aim generalized our result by experiment on other project written in other programming language and different bug tracking systems.

- For cross-project, we plan to take more measurement metrics and include other techniques for recording all faulty models. We also intend to further optimize the CDE-SMOTE by testing other quantification and class distribution modification techniques.
- For unsupervised categorization, the future work is to further improve the categorization performance while still retaining its nonparametric and non-prior knowledge properties. To further prove the generality of our framework, we also aim to do more experiments on a wider range of categories, projects, and bug tracker systems.

Bibliography

[1] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, (Washington, DC, USA), pp. 1–, IEEE Computer Society, 2007.

[2] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proceedings of the 27th international conference of Automated Software Engineering*, pp. 70–79, IEEE, 2012.

[3] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pp. 385–390, March 2012.

[4] N. K. Nagwani and S. Verma, "A comparative study of bug classification algorithms," *International Journal of Software Engineering and Knowledge Engineering*, vol. 24, no. 01, pp. 111–138, 2014.

[5] N. Pingclasai, H. Hata, and K. Matsumoto, "Classifying bug reports to bugs and other requests using topic modeling," in *Proceedings of the 20th International Conference on Software Engineering, Asia-Pacific*, vol. 2, pp. 13–18, Dec 2013.

[6] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, (New York, NY, USA), pp. 34–43, ACM, 2007.

[7] A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features," in *2010 Asia Pacific Software Engineering Conference*, pp. 366–374, Nov 2010.

[8] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 392–401, IEEE Press, 2013.

[9] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 522–531, IEEE Press, 2013.

[10] K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent dirichlet allocation," in *Proceedings of the 5th India Software Engineering Conference*, ISEC '12, (New York, NY, USA), pp. 125–130, ACM, 2012.

[11] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.

[12] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 485–496, 2008.

[13] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Software Engineering*, vol. 19, no. 2, pp. 167–199, 2012.

[14] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[15] M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara, and K. Matsumoto, "A dataset of high impact bugs: Manually-classified issue reports," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 518–521, May 2015.

[16] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, (New York, NY, USA), pp. 91–100, ACM, ACM, 2009.

[17] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pp. 409–418, IEEE, 2013.

[18] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *International Journal of Machine Learning Research*, vol. 3, pp. 993–1022, Mar. 2003.

[19] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei, "Hierarchical Dirichlet processes," *Journal of the American statistical association*, vol. 101, no. 476, 2006.

[20] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: An empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, (New York, NY, USA), pp. 25–33, ACM, 2006.

[21] A. Tosun, A. B. Bener, and R. Kale, "AI-based software defect predictors: Applications and benefits in a case study.," in *22th Innovative Applications of Artificial Intelligence Conference*, pp. 1748–1755, 2010.

[22] T. M. Khoshgoftaar, A. S. Pandya, and D. L. Lanning, "Application of neural networks for predicting program faults," *Annals of Software Engineering*, vol. 1, no. 1, pp. 141–154, 1995.

[23] F. Xing, P. Guo, and M. R. Lyu, "A novel method for early software quality prediction based on support vector machine," in *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pp. 213–222, Nov 2005.

[24] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *Software Engineering, IEEE Transactions on*, vol. 33, no. 10, pp. 675–686, 2007.

[25] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, (Piscataway, NJ, USA), pp. 200–210, IEEE Press, 2012.

[26] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.

[27] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 91–100, ACM, 2009.

[28] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Information and Software Technology*, vol. 54, no. 3, pp. 248–256, 2012.

[29] C. Catal, U. Sevim, and B. Diri, "Clustering and metrics thresholds based software fault prediction of unlabeled program modules," in *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, pp. 199–204, April 2009.

[30] J. Nam and S. Kim, "Clami: Defect prediction on unlabeled datasets (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pp. 452–463, Nov 2015.

[31] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, p. 9, ACM, 2010.

[32] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: An empirical study of commercial software projects," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 1042–1051, IEEE Press, 2013.

[33] M. Porter, "Readings in information retrieval," 1997.

[34] K. W. Church, "A stochastic parts program and noun phrase parser for unrestricted text," in *Proceedings of the Second Conference on Applied Natural Language Processing*, ANLC '88, (Stroudsburg, PA, USA), pp. 136–143, Association for Computational Linguistics, 1988.

[35] A. K. Ingason, S. Helgadóttir, H. Loftsson, and E. Rögnvaldsson, *A Mixed Method Lemmatization Algorithm Using a Hierarchy of Linguistic Identities (HOLI)*, pp. 205–216. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[36] M. A. Kachites, "Mallet: A machine learning for language toolkit," *URL: http://mallet. cs. umass. edu*, 2002.

[37] D. Čubranić, "Automatic bug triage using text categorization," in *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, Citeseer, 2004.

[38] S. Nessa, M. Abedin, W. E. Wong, L. Khan, and Y. Qi, *Software Fault Localization Using N-gram Analysis*, pp. 548–559. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[39] I. G. Caron, A. W. Carter, D. M. Canady, and T. Corbett, "Cross-project namespace compiler and method," July 4 2000. US Patent 6,083,282.

[40] G. Forman, "Quantifying counts and costs via classification," *Data Mining and Knowledge Discovery*, vol. 17, no. 2, pp. 164–206, 2008.

[41] A. Esuli and F. Sebastiani, "Optimizing text quantifiers for multivariate loss functions," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 9, no. 4, p. 27, 2015.

[42] W. Gao and F. Sebastiani, "Tweet sentiment: From classification to quantification," in *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*, pp. 97–104, ACM, 2015.

[43] G. King, Y. Lu, *et al.*, "Verbal autopsy methods with multiple causes of death," *Statistical Science*, vol. 23, no. 1, pp. 78–91, 2008.

[44] N. Limsettho, H. Hata, and K. Matsumoto, "Comparing hierarchical dirichlet process with latent dirichlet allocation in bug report multiclass classification," in *Proceedings of the 15th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pp. 1–6, June 2014.

[45] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?," *IEEE Transactions on Software Engineering*, vol. 36, pp. 618–643, Sept 2010.

[46] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, p. 23, ACM, 2008.

[47] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pp. 271–280, Nov 2012.

[48] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update; sigkdd explorations, 2009," *Software available at http://www. cs. waikato. ac. nz/ml/weka*, vol. 11, no. 1, pp. 10–18, 2009.

[49] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass support vector machines," *IEEE Transactions on Neural Networks*, vol. 13, pp. 415–425, Mar 2002.

[50] J. Riquelme, R. Ruiz, D. Rodríguez, and J. Moreno, "Finding defective modules from highly unbalanced datasets," *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos*, vol. 2, no. 1, pp. 67–74, 2008.

[51] V. García, J. S. Sánchez, and R. A. Mollineda, "On the effectiveness of preprocessing methods when dealing with different levels of class imbalance," *Knowledge-Based Systems*, vol. 25, no. 1, pp. 13–21, 2012.

[52] N. Japkowicz and S. Stephen, "The class imbalance problem: A systematic study," *Intelligent data analysis*, vol. 6, no. 5, pp. 429–449, 2002.

[53] A. A. Shanab, T. M. Khoshgoftaar, R. Wald, and A. Napolitano, "Impact of noise and data sampling on stability of feature ranking techniques for biological datasets," in *Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on*, pp. 415–422, IEEE, 2012.

[54] S. L. Phung, A. Bouzerdoum, and G. H. Nguyen, *Learning pattern classification tasks with imbalanced data sets*. 2009.

[55] D. Ryu and J. Baik, "Effective multi-objective naïve bayes learning for cross-project defect prediction," *Applied Soft Computing*, 2016.

[56] D. Ryu, O. Choi, and J. Baik, "Value-cognitive boosting with a support vector machine for cross-project defect prediction," *Empirical Software Engineering*, vol. 21, no. 1, pp. 43–71, 2016.

[57] B. X. Wang and N. Japkowicz, "Boosting support vector machines for imbalanced data sets," *Knowledge and Information Systems*, vol. 25, no. 1, pp. 1–20, 2010.

[58] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 37th International Conference on*

Software Engineering - Volume 2, ICSE '15, (Piscataway, NJ, USA), pp. 99–108, IEEE Press, 2015.

[59] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, pp. 321–357, 2002.

[60] J. C. Xue and G. M. Weiss, "Quantification and semi-supervised classification methods for handling changes in class distribution," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, (New York, NY, USA), pp. 897–906, ACM, 2009.

[61] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pp. 181–190, May 2008.

[62] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, (New York, NY, USA), pp. 495–504, ACM, 2010.

[63] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Why do software packages conflict?," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, (Piscataway, NJ, USA), pp. 141–150, IEEE Press, 2012.

[64] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "Ausum: approach for unsupervised bug report summarization," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, p. 11, ACM, 2012.

[65] D. Carmel, H. Roitman, and N. Zwerdling, "Enhancing cluster labeling using wikipedia," in *Proceedings of the 32Nd International ACM SIGIR*

Conference on Research and Development in Information Retrieval, SIGIR '09, (New York, NY, USA), pp. 139–146, ACM, 2009.

[66] D. Carmel, E. Yom-Tov, A. Darlow, and D. Pelleg, "What makes a query difficult?," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 390–397, ACM, 2006.

[67] N. Limsettho, H. Hata, A. Monden, and K. Matsumoto, "Automatic unsupervised bug report categorization," in *Proceedings of the 6th International Workshop on Empirical Software Engineering in Practice*, pp. 7–12, IEEE, 2014.

[68] J. Baldridge, "The opennlp project," *URL: http://opennlp. apache. org/index. html,(accessed 2 February 2012)*, 2005.

[69] H. M. Wallach, "Topic modeling: Beyond bag-of-words," in *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, (New York, NY, USA), pp. 977–984, ACM, 2006.

[70] D. Pelleg and A. W. Moore, "X-means: Extending k-means with efficient estimation of the number of clusters," in *Proceedings of the Seventeenth International Conference on Machine Learning*, pp. 727–734, 2000.

[71] J. H. Lau, K. Grieser, D. Newman, and T. Baldwin, "Automatic labelling of topic models," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pp. 1536–1545, Association for Computational Linguistics, 2011.

[72] A. Hindle, N. A. Ernst, M. W. Godfrey, and J. Mylopoulos, "Automated topic naming," *Empirical Software Engineering*, vol. 18, no. 6, pp. 1125–1155, 2013.

[73] E. F. Tjong Kim Sang and S. Buchholz, "Introduction to the conll-2000 shared task: Chunking," in *Proceedings of the 2Nd Workshop on Learning Language in Logic and the 4th Conference on Computational Natural Language Learning - Volume 7*, ConLL '00, (Stroudsburg, PA, USA), pp. 127–132, Association for Computational Linguistics, 2000.

[74] X. Jin and J. Han, "Expectation maximization clustering," in *Encyclopedia of Machine Learning* (C. Sammut and G. Webb, eds.), pp. 382–383, Springer US, 2010.

[75] J. R. Quinlan, *C4.5: programs for machine learning*. Elsevier, 2014.

[76] A. Genkin, D. D. Lewis, and D. Madigan, "Large-scale bayesian logistic regression for text categorization," *Technometrics*, vol. 49, no. 3, pp. 291–304, 2007.

[77] H. Zeng and D. Rine, "Estimation of software defects fix effort using neural networks," in *Proceedings of the 28th Annual International Computer Software and Applications Conference*, vol. 2, pp. 20–21 vol.2, Sept 2004.

[78] A. Panichella, R. Oliveto, and A. D. Lucia, "Cross-project defect prediction models: L'union fait la force," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pp. 164–173, Feb 2014.

[79] Y. Zhang, D. Lo, X. Xia, and J. Sun, "An empirical study of classifier combination for cross-project defect prediction," in *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, vol. 2, pp. 264–269, July 2015.

[80] P. Abate and R. Di Cosmo, "Predicting upgrade failures using dependency analysis," in *27th International Conference on Data Engineering*, pp. 145–150, IEEE, 2011.

[81] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-based learning algorithms," *Machine learning*, vol. 6, no. 1, pp. 37–66, 1991.

[82] J. J. Amor, G. Robles, and J. M. Gonzalez-Barahona, "Effort estimation by characterizing developer activity," in *Proceedings of the 2006 International Workshop on Economics Driven Software Engineering Research*, EDSER '06, (New York, NY, USA), pp. 3–6, ACM, 2006.

[83] J. Anvik, "Automating bug report assignment," in *Proceedings of the 28th international conference on Software engineering*, pp. 937–940, ACM, 2006.

[84] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in *Proceedings of the 28th international conference on Software engineering*, pp. 361–370, ACM, 2006.

[85] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 3, p. 10, 2011.

[86] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, 2010.

[87] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis.," in *OSDI*, pp. 237–250, 2010.

[88] M. Bates, "Models of natural language understanding," *Proc. National Academy of Sciences*, vol. 92, no. 22, pp. 9977–9982, 1995.

[89] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[90] A. W. Brown and K. C. Wallnau, "The current state of CBSE," *IEEE Software*, vol. 15, pp. 37–46, 1998.

[91] T. Long, I. Yoon, A. Memon, A. Porter, and A. Sussman, "Enabling collaborative testing across shared software components," in *Proc. 17th International ACM Sigsoft Symposium on Component-Based Software Engineering*, (Marcq-en-Baroeul, France), pp. 55–64, ACM, 2014.

[92] N. V. Chawla, "Data mining for imbalanced datasets: An overview," in *Data Mining and Knowledge Discovery Handbook*, pp. 875–886, Springer, 2010.

[93] C. Szyperski, *Component Software. Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[94] P. Abate, R. Di Cosmo, L. Gesbert, F. Fessant, R. Treinen, and S. Zacchiroli, "Mining component repositories for installability issues," in *Proc. 9th IEEE Working Conf. on Mining Software Repositories*, 2015.

[95] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the complexity of large free and open source package-based software distributions," in *ASE 2006*, pp. 199–208, IEEE, 2006.

[96] C. Elkan, "The foundations of cost-sensitive learning," in *International joint conference on artificial intelligence*, vol. 17, pp. 973–978, Citeseer, 2001.

[97] N. Feamster and H. Balakrishnan, "Detecting bgp configuration faults with static analysis," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 43–56, USENIX Association, 2005.

[98] W. R. Gilks, *Markov chain monte carlo*. Wiley Online Library, 2005.

[99] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 projects: Lightweight cross-project anomaly detection," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, (New York, NY, USA), pp. 119–130, ACM, 2010.

[100] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Applied statistics*, pp. 100–108, 1979.

[101] T. Hastie, R. Tibshirani, *et al.*, "Classification by pairwise coupling," *The annals of statistics*, vol. 26, no. 2, pp. 451–471, 1998.

[102] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pp. 45–54, IEEE, 2013.

[103] T. Hofmann, "Probabilistic latent semantic indexing," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 50–57, ACM, 1999.

[104] R. Di Cosmo, P. Trezentos, and S. Zacchiroli, "Package upgrades in FOSS distributions: Details and challenges," in *International Workshop on Hot Topics in Software Upgrades*, HotSWUp '08, (New York, NY, USA), pp. 7:1–7:5, ACM, 2008.

[105] G. H. John and P. Langley, "Estimating continuous distributions in Bayesian classifiers," in *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pp. 338–345, Morgan Kaufmann Publishers Inc., 1995.

[106] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto, "The effects of over and under sampling on fault-prone module detection," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pp. 196–204, IEEE, 2007.

[107] A. Kapoor, "Web-to-host: Reducing total cost of ownership," tech. rep., Technical Report 200503, The Tolly Group, 2000.

[108] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to Platt's SMO algorithm for SVM classifier design," *Neural Computation*, vol. 13, no. 3, pp. 637–649, 2001.

[109] J. Kittler, M. Hatef, R. P. Duin, and J. Matas, "On combining classifiers," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 20, no. 3, pp. 226–239, 1998.

[110] S. Kotsiantis, D. Kanellopoulos, P. Pintelas, *et al.*, "Handling imbalanced datasets: A review," *GESTS International Transactions on Computer Science and Engineering*, vol. 30, no. 1, pp. 25–36, 2006.

[111] P. S. A. Krogh, "Learning with ensembles: How over-fitting can be useful," in *Proceedings of the 1995 Conference*, vol. 8, p. 190, 1996.

[112] L. I. Kuncheva, *Combining pattern classifiers: methods and algorithms*. John Wiley & Sons, 2004.

[113] S. Le Cessie and J. C. Van Houwelingen, "Ridge estimators in logistic regression," *Applied statistics*, pp. 191–201, 1992.

[114] N. Limsetto and K. Waiyamai, "Handling concept drift via ensemble and class distribution estimation technique," in *Advanced Data Mining and Applications*, pp. 13–26, Springer, 2011.

[115] T. Long, I. Yoon, A. Porter, A. Sussman, and A. Memon, "Overlap and synergy in testing software components across loosely coupled communities," in *Proc. 23rd Int. Symposium on Software Reliability Engineering (ISSRE)*, pp. 171–180, IEEE, 2012.

[116] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*, vol. 1. Cambridge university press Cambridge, 2008.

[117] A. K. McCallum, "Mallet: A machine learning for language toolkit." http://mallet.cs.umass.edu, 2002.

[118] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *Communications Surveys & Tutorials, IEEE*, vol. 10, no. 4, pp. 56–76, 2008.

[119] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, *et al.*, "Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies," tech. rep., Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, 2002.

[120] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," 1998. Technical Report MSR-TR-98-14.

[121] J. Platt *et al.*, "Fast training of support vector machines using sequential minimal optimization," *Advances in kernel methods?support vector learning*, vol. 3, 1999.

[122] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Software Engineering, 2003. Proceedings. 25th International Conference on*, pp. 465–475, IEEE, 2003.

[123] I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, and M. Welling, "Fast collapsed Gibbs sampling for latent Dirichlet allocation," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, (New York, NY, USA), pp. 569–577, ACM, 2008.

[124] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, p. 61, ACM, 2012.

[125] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Learning from the future of component repositories," *Science of Computer Programming*, vol. 90, no. B, pp. 93–115, 2014.

[126] M. Stokely, *The FreeBSD Handbook*. FreeBSD Mall, 3 ed., 2004.

[127] U. Stu, W. Melssen, and L. Buydens, "Facilitating the application of support vector regression by using a universal Pearson VII function based kernel," *Chemometrics and Intelligent Laboratory Systems*, vol. 81, pp. 29–40, 2006.

[128] Y. Sun, M. S. Kamel, A. K. Wong, and Y. Wang, "Cost-sensitive boosting for classification of imbalanced data," *Pattern Recognition*, vol. 40, no. 12, pp. 3358–3378, 2007.

[129] Z. Sun, Q. Song, and X. Zhu, "Using coding-based ensemble learning to improve software defect prediction," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 42, no. 6, pp. 1806–1817, 2012.

[130] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *Proceedings of the 19th International Working Conference on Reverse Engineering*, pp. 205–214, Oct 2012.

[131] L. Nussbaum and S. Zacchiroli, "The Ultimate Debian Database: Consolidating bazaar metadata for quality assurance and data mining," in *7th*

IEEE Working Conference on Mining Software Repositories (MSR 2010), (Cape Town, South Africa), 2010.

[132] B. X. Wang and N. Japkowicz, *Foundations of Intelligent Systems: 17th International Symposium, ISMIS 2008 Toronto, Canada, May 20-23, 2008 Proceedings*, ch. Boosting Support Vector Machines for Imbalanced Data Sets, pp. 38–47. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[133] Y. Zhang, R. Jin, and Z.-H. Zhou, "Understanding bag-of-words model: a statistical framework," *International Journal of Machine Learning and Cybernetics*, vol. 1, no. 1-4, pp. 43–52, 2010.