Cross Project Defect Prediction Using Class Distribution Estimation and Oversampling

Nachai Limsettho[a], Kwabena Ebo Bennin[b,*], Jacky W. Keung[b], Hideaki Hata[a], Kenichi Matsumoto[a]

[a] *Graduate School of Information Science, Nara Institute of Science and Technology, Japan*
[b] *Department of Computer Science, City University of Hong Kong, Hong Kong*

Abstract

Context: Cross-project defect prediction (CPDP) which uses dataset from other projects to build predictors has been recently recommended as an effective approach for building prediction models that lack historical or sufficient local datasets. Class imbalance and distribution mismatch between the source and target datasets associated with real-world defect datasets are known to have a negative impact on prediction performance.

Objective: To alleviate the negative effects of class imbalance and distribution mismatch on performance of CPDP models by using Class Distribution Estimation and Synthetic Minority Oversampling Technique. A novel approach called Class Distribution Estimation with Synthetic Minority Oversampling Technique (CDE-SMOTE) is proposed to optimize and improve the CPDP performance and avoid excessive oversampling.

Method: The proposed CDE-SMOTE employs CDE to estimate the class distribution of the target project. SMOTE is then used to modify the class distribution of the training data until the distribution becomes the reverse of the approximated class distribution of the target project. Four comprehensive experiments are conducted on 14 open source software projects.

Results: The proposed approach improves the overall performance of CPDP

*Corresponding author
Email addresses: `nachai.limsettho.nz2@is.naist.jp` (Nachai Limsettho), `kebennin2-c@my.cityu.edu.hk` (Kwabena Ebo Bennin), `jacky.keung@cityu.edu.hk` (Jacky W. Keung), `hata@is.naist.jp` (Hideaki Hata), `matumoto@is.naist.jp` (Kenichi Matsumoto)

models when compared to the performance of other CPDP approaches. Significant improvements are observed in 63% of the test cases according to the Wilcoxon signed-rank tests with 16.421%, 29.687% and 20.259% improvements in terms of Balance, G-measure, and F-measure, respectively. Application of CDE-SMOTE on NN-filtered datasets significantly improved prediction performance.

Conclusions: CDE-SMOTE mitigates the class imbalance and distribution mismatch problems and also helps prevents excessive oversampling that results in performance degradation of prediction models. This approach is thus recommended for CPDP studies in software engineering.

Keywords: Cross-Project Defect Prediction, Software Fault Prediction, Oversampling, Class Imbalance Learning, Class Distribution Estimation

1. Introduction

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 [1, 2]. Defect prediction is known to work well when the prediction model is built using its own historical data [1, 2]. 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. [3] and Turhan et al [4].

Without sufficient historical or within-company data, cross-project defect prediction (CPDP) [5, 4, 3, 6], which selects and utilizes historical data from other similar projects for training predictors can be employed. The CPDP approach, although promising causes low prediction performance compared to the performance of within-company defect prediction models. This is mostly due to the distribution mismatch between the source and target projects [3, 6, 7, 5]. Additionally, class imbalance of defect datasets, a prevalent problem in within-company defect prediction [8, 9] is inherited by CPDP. Generally, defect prediction dataset contains more non-defective examples than the defective examples

[10]. 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 [10].

A common preprocessing technique adopted by researchers [11, 12, 13, 14, 15] for tackling the class imbalance and to enhance defect prediction performance is the application of sampling techniques such as over and under sampling [16, 17]. However, oversampling techniques have been shown to perform better than undersampling techniques in several empirical studies [11, 12, 13] 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 and non-defective modules in a target dataset cannot be easily assumed as in the within-project situation. The exact amount of the skewness vary depending on each project; while some might only have a very low number of defective modules (i.e. 8.1% in NetBSD), approximately half of the modules in another project might be defective (i.e. 49.1% in XFree86) as observed from the projects considered for our study.

To overcome the aforementioned problems above, TCSBoost [18, 7] and Boosting-SVM [19] are few of the recently proposed techniques. None of the existing methods in CPDP considers the distribution mismatch between source and target datasets and efficiently utilizes the estimated distribution of the target project when oversampling is applied. A transfer learning technique that estimates the class distribution of the target/unlabeled data and appropriately modifies the source data with this estimated value could be applied to improve the performance of the prediction model. As such, we investigate how the distribution of the target data could be estimated and how to use this estimated value for oversampling the source project in CPDP. We thus formulate and explore the following research questions:

RQ1: Is the prediction performance improved when the true class distribution of the target project is known beforehand?

RQ2: Can the actual distribution of the target project be estimated?

RQ3: Given that we can estimate the unlabeled data distribution, can we build a better cross-project defect prediction model based on this estimated value?

RQ4: Is the prediction performance improved when CDE-SMOTE is applied on two state-of-the-art filtering techniques - Nearest Neighbor(NN)/Burak filter and CLAMI?

The aim of our research is to improve the prediction performance of cross-project defect prediction models and alleviate the negative effects of class imbalance and distribution mismatch through modification of the training dataset's distribution by using SMOTE and a quantification technique. We propose 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. Although it is not practical to assume that the true distribution of the unlabeled data can be obtained, it can be estimated using a class distribution estimation (quantification) approach [20] mostly used in other domains of machine learning. By leveraging this estimated distribution, we can approximate the amount of oversampling required for each dataset and prevent excessive oversampling, which degrades prediction performance. The hypotheses and performance of CDE-SMOTE are validated and evaluated through four experiments together with Wilcoxon signed-rank statistical tests. We conduct extensive empirical studies on 14 open-source projects considering all of their possible cross-project pairs making a total of $14 \times 13 = 182$ cross-project pairs, and using 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 with statistical significance. It also improved the prediction performance compared to the CLAMI [21] and Burak [4] approaches.

The major contributions of this paper are:

- We empirically demonstrate the negative effects of building a cross-project defect prediction model without considering the distribution of the intended target projects and provide guidelines on how to improve the prediction performance using an estimated distribution.

- The first study to apply the quantification technique in the cross-project scenario, and empirically demonstrate the feasibility of using quantification to estimate the expected percentage of defective modules.

- We demonstrate that prediction performance of NN-Filter and CLAMI can be significantly improved when combined with our proposed approach.

The remainder of this paper is organized as follows. Section 2 summarizes related work in the area of cross-project defect prediction. In Section the 3, we describe the steps and procedures of CDE-SMOTE, as well as its base hypotheses. The experimental setup and results are explained in Section 4 and Section 5 respectively. Implications of the results and validity are discussed in Section 6, and conclusion in Section 7.

2. Background and Related Work

2.1. Defect Prediction Process

Defect prediction has the main objective of identifying fault-prone modules and aids in the effective allocation and prioritization of scarce testing resources [22]. Several defect prediction models have been proposed in the past decade employing various machine learning and statistical approaches. Conventional methods such as neural networks [23], support vector machine [24], bayesian classifier [25] and random forest [26] have been used. Menzies et al. [27] recommend the application of Naive Bayes with logNums preprocessing for better defect prediction. These proposed defect prediction models are however studied in the 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.

2.2. Defect Prediction with Limited Historical Project Data

Over the years, several studies [5, 4, 3, 6, 28, 21] have proposed different approaches 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.

Cross-Project Defect Prediction: With the availability of open source software projects, the feasibility of cross-project defect prediction where projects from different companies are used to train prediction models have been investigated in recent years but with an inconclusive result. The first to attempt cross-project defect prediction was Zimmermann et al. [5]. 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. Turhan et al. [4] 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. [29] proposed a new filter which outperformed the Burak filter proposed in the work by Turhan et al. [4]. The Peters filter selects training data considering the structure of the other projects and can select as few as one data instance for each test instance. By conducting large-scale cross-project defect prediction experiments on 34 data sets extracted from 10 open source projects, He et al. [3] 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. [5] and Turhan et al. [4]. Jureczko and Madeyski [30] 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. They argue projects in the same group tend to have similar characteristics and thus, no need for datasets to have historical datasets before defect prediction model could be constructed. Zhang et al. [31] studied 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 [32]. More recent work by Ryu et al. [18] uses transfer learning and boosting to deal with problems in cross-project defect prediction. The approach works by integrating a small amount of within project data to improve the prediction performance. The approach was improved with a different boosting technique a year later [7]. Poon et al. [33] proposed a credibility theory based Nave Bayes classifier that considers the data distribution mismatch between the source and target data. The proposed method provides a credibility factor that estimates the degree of reweighting the source data and target data to determine the extent of knowledge transfer from target data to source data whilst retaining the data distribution pattern of the source data.

Defect Prediction on Unlabeled Datasets: Compared to the cross-project defect prediction approaches, only few have studied the use of the unsupervised learning approach that creates the prediction model without the need for any labeled data. Catal et al. [28] proposed the use of a metric threshold. Similarly, Nam and Kim [21] proposed the method known as CLAMI, which computes threshold values for each metric value and cluster modules into two groups below or above the threshold (Defective and Clean). CLAMI requires no human expert judgment and does the labeling in an automated manner. CLAMI proved to outperform most traditional and state-of-the-art cross-project defect prediction approaches with statistical significance.

2.3. Handling of Skewed Dataset in Defect Prediction

Most studies conducted in the domain of cross-project defect prediction focuses on the techniques for selecting appropriate datasets for training the prediction model and inadvertently ignores the class distribution of the obtained datasets. Since defect prediction datasets are mostly highly skewed in favor of one class [10] with the non-defective modules most often dominating the defective modules, the acquired training data for a cross-project defect prediction will most likely also be highly skewed. As such, the appropriate preprocessing technique to be applied to the data becomes a research challenge. A common preprocessing technique adopted by researchers [12, 11, 13, 34] for enhanced defect prediction performance is the application of sampling techniques such as over and under sampling. Other research tackle and alleviate this problem by using boosting and instance weighting approaches such as by Ryu et al. [18, 7].

These techniques are applied to alleviate the negative effects of highly skewed or the imbalanced distribution of the defect prediction datasets [17]. However, oversampling techniques have been shown to perform better than undersampling techniques in several empirical studies [12, 11, 13, 34] whereby more minority or defective instances are added to the dataset. We thus adopt and focus on an oversampling approach for this study.

2.4. Synthetic Minority Oversampling Technique (SMOTE)

Among the oversampling techniques, SMOTE is one of the most prevalent techniques used for synthetic data generation [11] Proposed by Chawla et al. [35], 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 learning bias towards the minority class. Figure 1 shows how SMOTE works.

SMOTE generates synthetic examples by:

1. Choose an example from the minority class. We consider the middle instance that is circled in Figure 1.
2. Find the k-nearest minority instance neighbors of the chosen example. In Figure 1 these are the three surrounding circles (k=3).
3. Create synthetic examples, between the chosen example and its neighbors. The synthetic examples will have each of its features randomized between

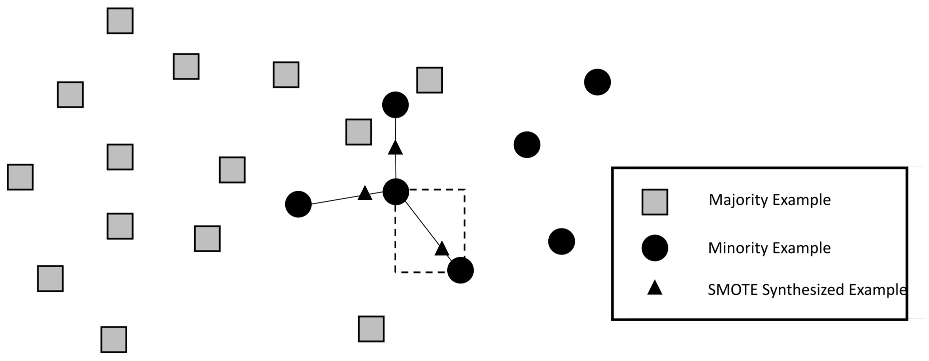

Figure 1: Diagram showing how SMOTE works (k=3)

the chosen and neighbor examples selected. In Figure 1, this is represented by the three triangles. The possible location for the synthetic example in the right corner is within the dashed box.

These steps are repeated until the target amount of oversampling is reached.

2.5. Class Distribution Estimation (CDE)

Class distribution estimation (CDE) or Quantification is a machine learning technique that estimates the proportion of each class in a given unlabeled dataset [20]. Unlike classification that is interested in the actual label of each instance, quantification is more interested in the distribution of each class. This approach has been successfully adopted in many fields such as in text mining [36], sentiment analysis [37] and epidemiology [38]. However, it is yet to be utilized in the field of software engineering. Our research uses this estimated class distribution approach to approximate the amount of oversampling needed for the target unlabeled project.

3. 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 on the performance of a cross-project defect prediction model by using CDE and SMOTE oversampling. The amount of oversampling is obtained by the estimated distribution of the unlabeled dataset in order to stop the needless oversampling which causes prediction performance degradation.

3.1. Theoretical basis of CDE-SMOTE

Based on the negative effects class imbalance and the class distribution mismatch between training and target projects do have on the performance of CPDP models, we argue that prediction performance will be significantly improved when the class distribution of the source project is similar to that of the target project. A known approach of improving prediction performance on the minority (defective modules) is by shifting the decision boundary of classifiers toward the region on the minority class [35, 17]. Thus, the underlining hypotheses behind CDE-SMOTE are formulated below:

First	The training dataset could be modified to better suit the class distribution of the target unlabeled dataset
Second	Without any prior knowledge of the label of the unlabeled dataset, the class distribution of the unlabeled dataset can be estimated.

The first hypothesis is based on the previous study of [39], with some modifications. Application of SMOTE can aid in the modification of the class distribution of the training data. The second hypothesis is however, based on the quantification technique [20] adopted from machine learning. This technique estimates the distribution of an unlabeled dataset by using the classification performance of the trained prediction model on the unlabeled dataset. More details of this technique is discussed below in 3.2.

3.2. Steps and Procedures

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

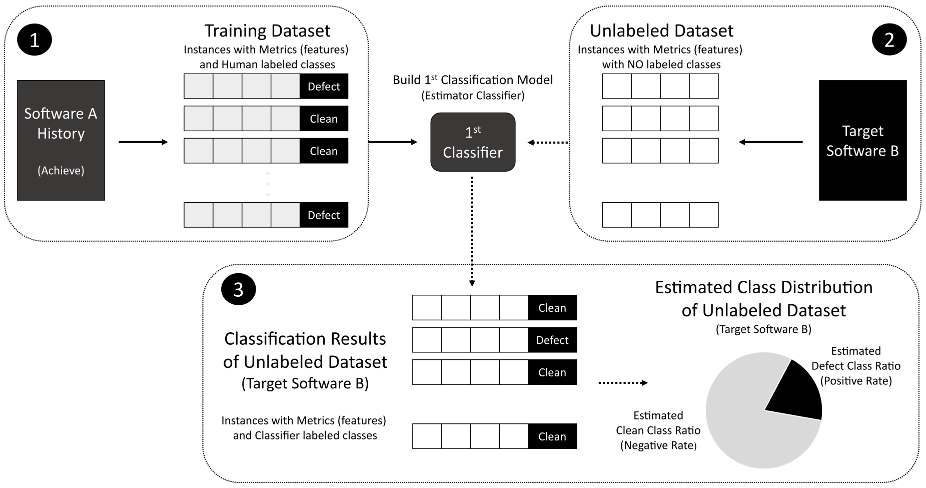
1
Training Dataset
Instances with Metrics (features) and Human labeled classes
Software A History
(Achieve)
Defect
Clean
Clean
Defect
Build 1st Classification Model
(Estimator Classifier)
1st Classifier
Unlabeled Dataset
Instances with Metrics (features) with NO labeled classes
2
Target Software B
3
Classification Results of Unlabeled Dataset
(Target Software B)
Instances with Metrics (features) and Classifier labeled classes
Clean
Defect
Clean
Clean
Estimated Class Distribution of Unlabeled Dataset
(Target Software B)
Estimated Defect Class Ratio (Positive Rate)
Estimated Clean Class Ratio (Negative Rate)

Figure 2: CDE-SMOTE Diagram: Class Distribution Estimation

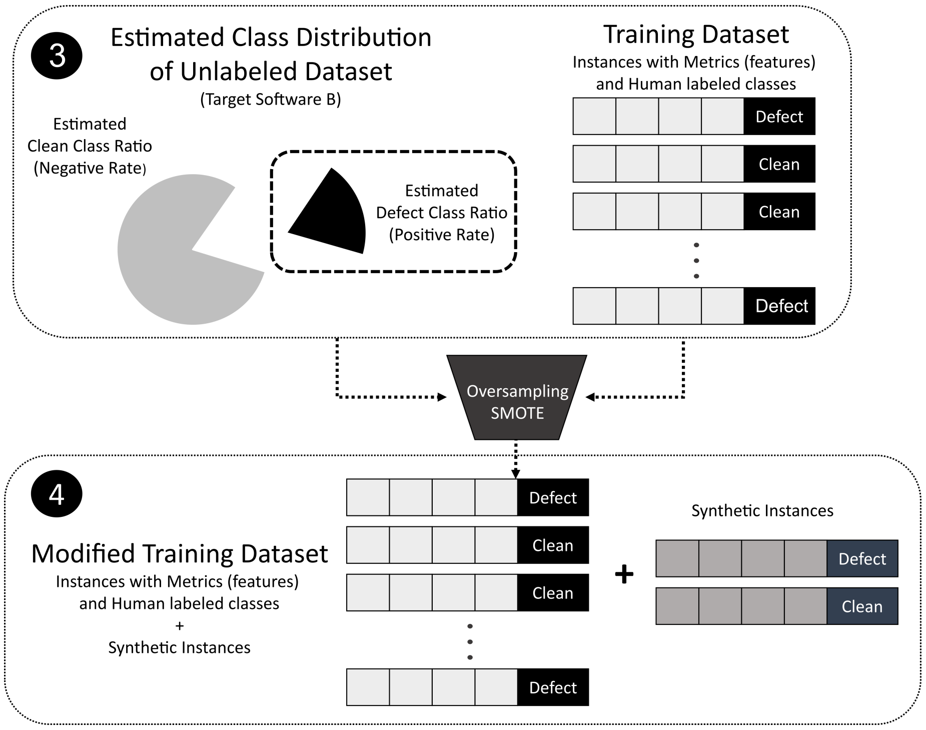
3
Estimated Class Distribution of Unlabeled Dataset
(Target Software B)
Estimated Clean Class Ratio (Negative Rate)
Estimated Defect Class Ratio (Positive Rate)
Training Dataset
Instances with Metrics (features) and Human labeled classes
Defect
Clean
Clean
Defect
Oversampling SMOTE
4
Modified Training Dataset
Instances with Metrics (features) and Human labeled classes + Synthetic Instances
Defect
Clean
Clean
Defect
+
Synthetic Instances
Defect
Clean

Figure 3: CDE-SMOTE Diagram: Class Distribution Modification

The first step, class distribution estimation, is shown in Figure 2. The approach starts by building the first classification model (estimator classifier) from the training dataset, for approximating the class distribution of the unlabeled dataset. The training dataset is a historical data from another software archive that is already labeled and modified using SMOTE to have an equal number of defective and clean classes. Next, the unlabeled dataset from the target software project is labeled (predicted) by the estimator classifier; this will yield a machine labeled result of the target software. The estimated distribution of target software is then obtained by the classification and count (CC) technique [20], through a simple count of the number of machine-labeled instances for each class. Practically, there might be some classification mistakes in the first labeled result, however, we assume that we will obtain an approximate class distribution ratio of the target project. This assumption is investigated in our results related to experiment 2.

The second step, class distribution modification, is shown in Figure 3. This part takes the estimated distribution and the training dataset as inputs to 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 and obtain the amount of oversampling required. The modification is done by oversampling the original unmodified training dataset by adding synthetic defective instances to the training dataset until the class distribution of the training data becomes the reverse of the estimated distribution of the unlabeled dataset. This is done in order to shift the decision boundary of the defect prediction model towards the region of minority class thus emphasizing the presence of the defective class samples. For example, if the distribution of the training is 6 : 4 and the estimated distribution of the unlabeled datasets is 8 : 2, synthetic examples will be added to the class with a ratio of 4 (40%) until the distribution of the training dataset becomes 2 : 8. These synthetic examples are generated by SMOTE [35], a well-known oversampling technique. Oversampling using the reverse of the target estimated distribution value acts as the limit for oversampling. Our aim is to improve the prediction performance

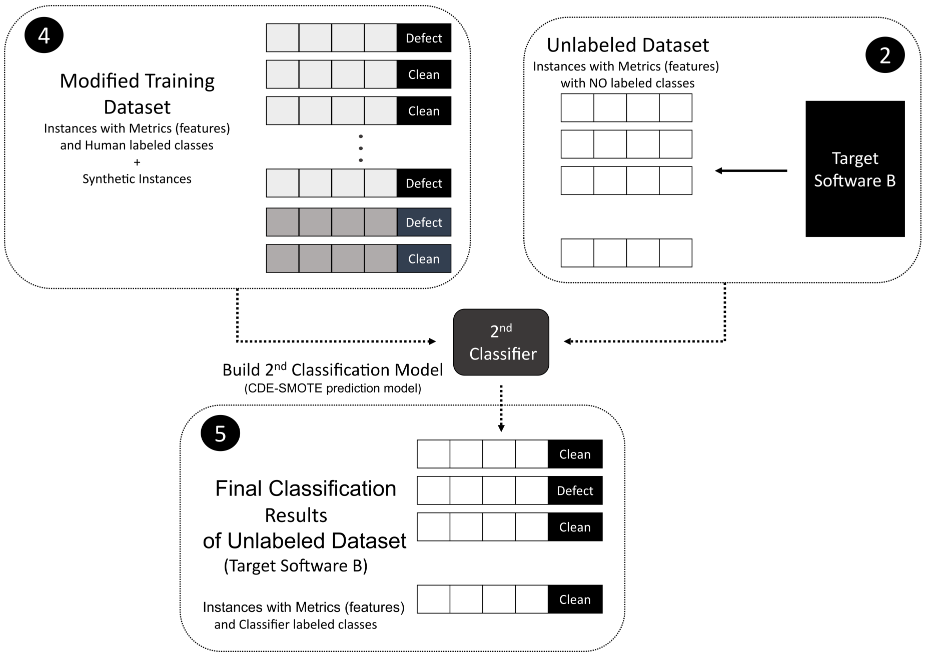
4
Modified Training Dataset
Instances with Metrics (features) and Human labeled classes
+
Synthetic Instances
Defect
Clean
Clean
Defect
Defect
Clean
Unlabeled Dataset
Instances with Metrics (features) with NO labeled classes
2
Target Software B
2nd Classifier
Build 2nd Classification Model
(CDE-SMOTE prediction model)
5
Final Classification Results of Unlabeled Dataset
(Target Software B)
Instances with Metrics (features) and Classifier labeled classes
Clean
Defect
Clean
Clean

Figure 4: CDE-SMOTE Diagram: Prediction Model Building

whilst the excessive oversampling of the minority class is avoided.

The final step, prediction model building, as shown in Figure 4, adopts the modified training dataset from the second part to create the second classification model which we refer to as the CDE-SMOTE prediction model. The unlabeled dataset from the target software project is then labeled (predicted) by this classifier producing the final classification results of the target software project which is evaluated in our results related to experiments 3 and 4.

4. Experimental Setting

To evaluate the proposed CDE-SMOTE approach, this section describes the datasets, prediction models, evaluation measures, and validation procedures used for this study.

4.1. Defect Prediction algorithms

For a more comprehensive study, both single based classifiers and ensemble classifiers were used for the experiments. The cross-project defect prediction experiments are conducted with the following seven classification algorithms: • J48 Decision Tree [40]

- Random Forest (10 trees) [41]
- Naive Bayes (NB) [42]
- Logistic Regression [43]
- kNN (k = 3) [44]
- Vote ensemble: Average of Probability (J48+NB) [45]
- Vote ensemble: Average of Probability (J48+NB+kNN(3)) [45]

The first five models are well known and commonly used in several defect prediction studies, while the remaining two are ensemble classifiers created from the combinations of two or three classification algorithms. The classification algorithms used in this papers are all implemented in WEKA Machine Learning Toolkit, version 3.6.3. [46].

4.2. 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 1. We experiment on a wide variety of class distribution to examine the impact different class distribution between the training and target projects have on prediction performance. The metrics of each software repository are collected from their respective commit logs using Git/CVS version control tools to extract seven common process metrics as recommended by Moser et al. [47]. Modules labeled as error in the commit logs and having error density values greater than zero, are thus labeled as "Defective" in our datasets. Table 2 presents nine metrics commonly used in defect prediction [26] extracted from each software project.

Table 1: The 14 Open-source Datasets extracted with their respective defect class distribution

Datasets	# Instances	%Defective Module	Datasets	# Instances	%Defective Module
Clam Antivirus	1597	5.8	GANYMEDE	184	15.8
NetBSD	6781	8.1	OpenBSD	1706	16.1
Scilab Website	2636	8.8	Squid	250	23.6
OpenNMS	1203	10.2	WineHQ	1627	35.9
Samba Development	1623	11.3	XFree86	713	49.1
Helma.org	312	12.2	Hylafax.org	137	51.1
Spring	926	13.6	Ipnetfilter	151	61.6

Table 2: 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.

4.3. Evaluation Criteria

Two sets of performance measurements are used for this study. One for measuring the performance of the defect prediction models and second for measuring the mismatch during and after 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 and F-measure. These measures are widely used in the defect prediction field, which emphasizes the importance of the defective class.

- *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 paper, only the F-measure of the Defective-class is evaluated:

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

- *Precision*: the number of predicted defect modules which are actually defective modules, this is used to calculate the F-measure:

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

The predicted results by the prediction models built on the modified training data are compared with the classification model built from the unmodified training data (the original classifier). Wilcoxon signed rank tests are further performed to find the statistical significance of the results. The Wilcoxon Win-Tie-Loss across all the five measures as well as the percentage improvements are presented in Section 5.

The second set of measures aims to measure the performance of the class distribution estimation model, 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*: the distribution of the defective module in terms of the ratio between the number of defectives and the total number of modules:

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

  Three *Positive Rate* measures 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.

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

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

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

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

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

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

4.4. Validation Procedure

We conduct four different experiments to validate the hypotheses formulated and the proposed CDE-SMOTE approach. We explain how each experiment was conducted. These experiments provide answers to the four research questions.

Experiment 1: Oracle and Original Classifiers Comparison

For the first experiment, we aim to validate the first hypothesis of CDE-SMOTE and investigate whether the performance of the prediction models can be improved if the true distribution of unlabeled dataset, $PositiveRate_{Unlabeled}$, is known beforehand. By confirming our first hypothesis, we demonstrate the possibility of improving the cross-project prediction model through the modification of the class distribution of the training dataset. We also demonstrate the danger of building prediction models on the 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 is not practical in most cross-project defect prediction scenario. This knowledge is used to adjust the number of training instances of each class in the training dataset to make the training dataset more similar and 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 class distribution of the 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" because it obtains information that is not possible to obtain in a normal and practical scenario. We then examine the prediction results of this model against the true labels of the unlabeled dataset and evaluate its performance. The evaluation 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. The above 7 classification algorithms are run on each cross-project pair making a total of $182 \times 7 = 1,274$ runs. The Wilcoxon test is performed on each training dataset selected across all of its cross-project pairs for each performance measure and reported in terms of Wins, Ties or Losses depending on its significance level 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 to validate the second hypothesis of CDE-SMOTE and investigate the impact of applying the quantification technique on the performance of cross-project defect prediction models. Practically, the actual class distribution of the testing or unlabeled data is unknown or unobtainable. The question we ask ourselves is, can we estimate the actual class distribution of the unlabeled dataset? We thus focus on using the training (labeled) data, the same data used as the input for the already built classification model and examine the quantification performance of the classification and count (CC) technique applied on the cross-project models.

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

Experiment 3: CDE-SMOTE and Original Classifiers Comparison

Given that we can estimate the class distribution of the unlabeled data, can we build a better cross-project defect prediction model based on this estimated value? This experiment demonstrates and provides the practical contribution of this study. That is, to estimate the class distribution of the unlabeled data and accordingly use the estimated value to modify the class distribution of the training 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 procedure and percentage improvement in the same manner as in Experiment 1. The results are presented in Section 5.3. In contrast to experiment 1, the actual class distribution of the unlabeled dataset is never used in this experiment so as to avoid contaminating the trained classifier. The estimated class distribution of the unlabeled dataset is rather used in this experiment.

Experiment 4: CDE-SMOTE and Related Works Comparison

The final experiment is conducted to compare the predictive performance of CDE-SMOTE with two state-of-the-art filtering techniques discussed in our related works: Burak filter [4] and CLAMI [21].

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 how similar they are 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. The Burak filter approach assumes that there is a large amount of training dataset composed of historical data from several software engineering projects. Therefore, per each run of experiment, 1 dataset is selected as the testing/unlabeled dataset whilst the remaining 13 datasets are combined to create a composite 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 a 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 metric 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 defect prediction models.

While 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, CDE-SMOTE 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, we compare the performances of using only CDE-SMOTE and using only CLAMI. Similar to the setups in experiment 3, each run is performed 14 times making a total of $14 \times 13 = 182$ cross-project pairs. However, the original classifier is substituted with CLAMI and compared to CDE-SMOTE. We use the Logistic Regression Classifier for CLAMI as the authors recommend that model for better performance [21]. The results of Both Burak and CLAMI experiments are presented in Section 5.4.

5. Results and Analysis

In this section, we discuss and present the results of the four conducted experiments. The answers to the research questions are also provided.

5.1. RQ1: Is the prediction performance improved when the true class distribution of the target project is known beforehand?

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 shows the summarized result, the Y-axis denotes the number of Win, Tie or Loss while the X-axis denotes the classification algorithm used.

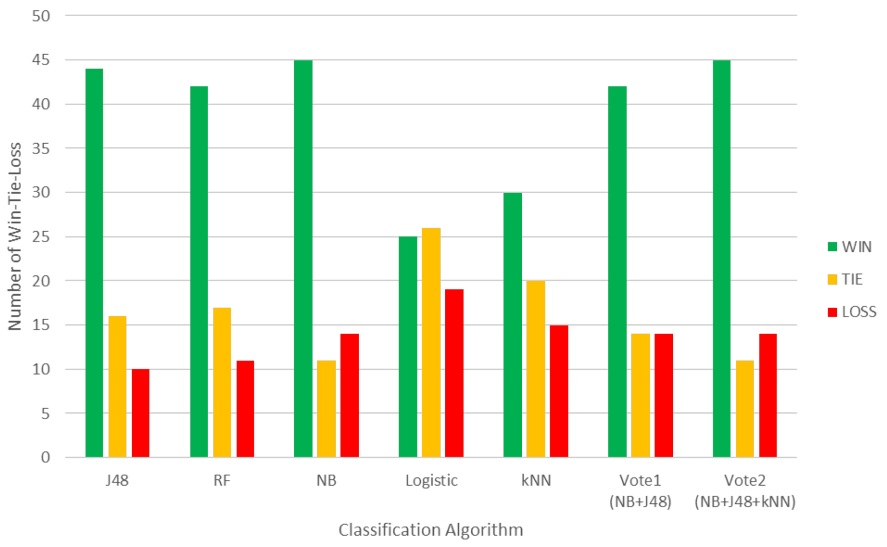

Figure 5: Oracle comparison to the Original Classifier: Wilcoxon Win-Tie-Loss

We observe from Figure 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 Wilcoxon statistical test.

The average increase in performance of each classifier is shown in Table 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.

Table 3: Oracle performance increase (%) 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

Across these seven classification algorithms, we can see the increase in performance 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 G-measure decrease in Logistic Regression, none of the algorithms experienced a performance degradation in the results. We have enough evidence to conclude that the use of the oracle classifier improves prediction performance across all models and performance measures with statistical significant results.

5.2. RQ2: Can the actual distribution of the target project be estimated?

As it is not practical to assume that the distribution of the unlabeled data is known beforehand, this experiment investigates the practicality of estimating the distribution of the unlabeled dataset. First, the average mismatch class

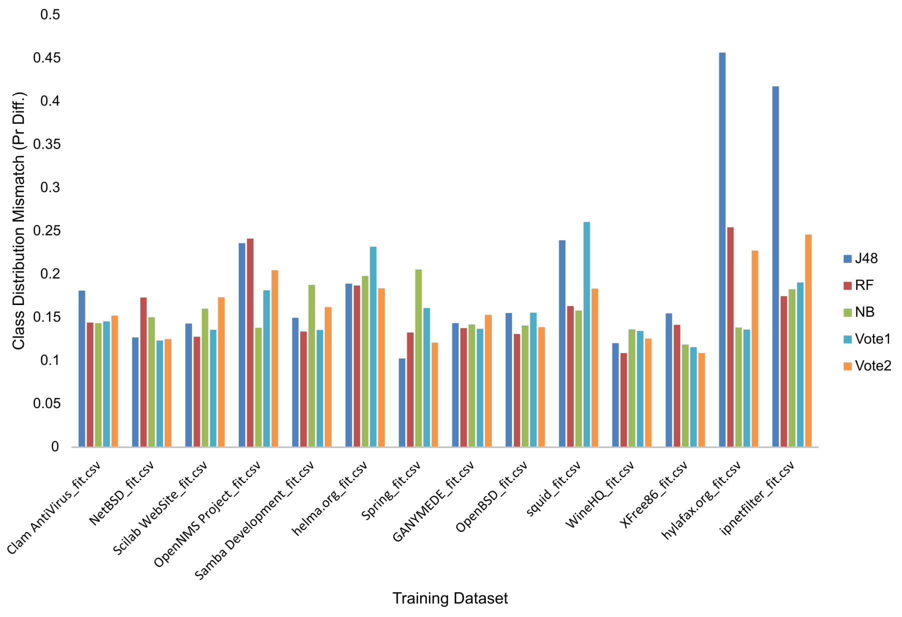

Figure 6: Actual class distribution mismatch ($PredictedMismatch_{Value}$) between train and unlabeled datasets in the actual value

distributions in term of positive rate (PR) between the estimation and the true value ($PredictedMismatch_{Value}$), averaged across all cross-project pairs for each training dataset, are shown in Figure 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 6 as well as our further experiments, because these two algorithms could not accurately estimate the distributions of the unlabeled datasets. They produced worst mismatched estimations, generating 160% and 190% more error at maximum compared to the use of the original unmodified training dataset.

From Figure 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.

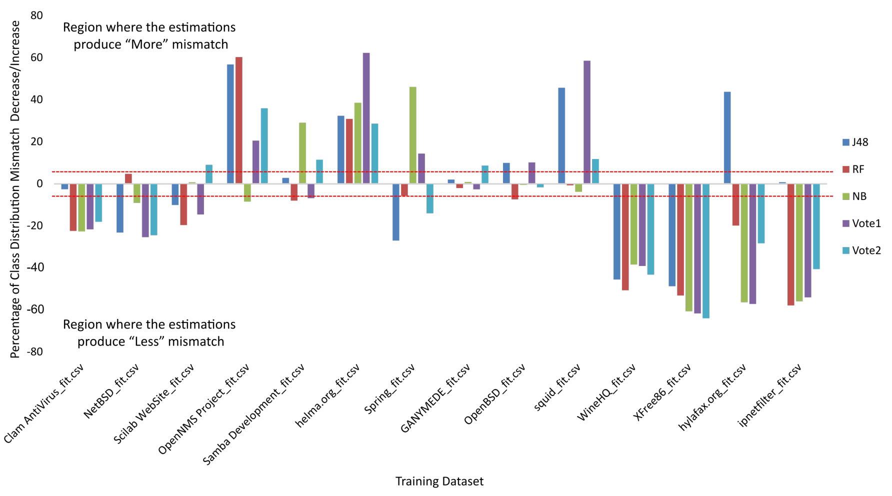

Figure 7: Class Distribution Mismatch Compared to the original Training Data: Percentage differrent betwen actual test data error and CDE estimation ($PredictedMismatch_{\%Diff}$)

Comparing the estimation result with the mismatch of the original training dataset, we present the increase/decrease percentage mismatch ($PredictedMismatch_{\%Diff}$) in Figure 7. The X-axis displays the training dataset used where each result is averaged across all the unlabeled datasets. The Y-axis shows the increase and decrease in percentage error of the estimated distribution, $PredictedMismatch_{Value}$, 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 when we assume that the distribution of the unlabeled dataset is the same as the training dataset. In constrast, the increase in error implies that the estimated class distribution 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 7, we observe that on several instances, the classification and count (CC) 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 which shows that the class distribution of the unlabeled dataset could be estimated.

5.3. RQ3: Given that we can estimate the unlabeled data distribution, can we build a better cross-project defect prediction model based on this estimated value?

This experiment investigates the practicality of using the estimated class distribution of the target project. 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 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.

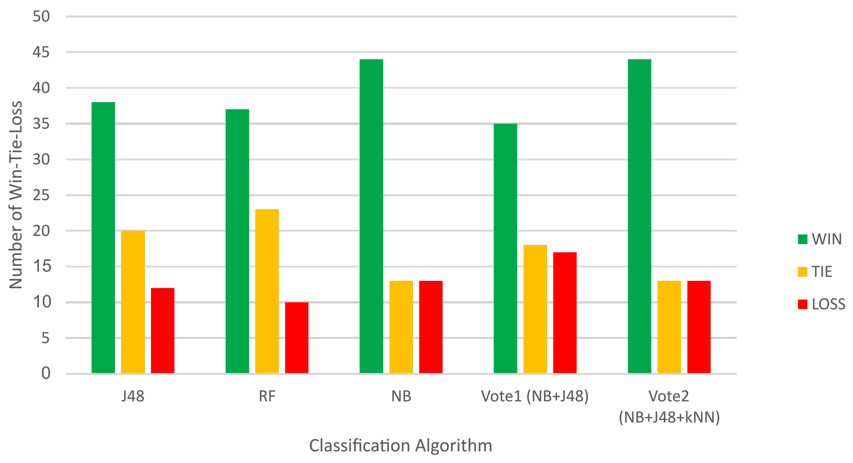

Figure 8: CDE-SMOTE comparison to the Original Classifier: Wilcoxon Win-Tie-Loss

The results in Figure 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 4 summarized with respect to their Balance, G-measure and F-measure values which are averaged across 182 cross-project pairs. From Table 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. The first column presents the training dataset, the remaining columns are the Balance, G-measure, and F-measure values respectively. The performances shown for each training dataset are the averaged across all 13 cross-project pairs. We conclude that application of CDE-SMOTE can significantly improve performance of CPDP.

Table 4: CDE-SMOTE performance increase (Percentage) when compared to original classifier

	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: 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

5.4. RQ4: Is the prediction performance improved when CDE-SMOTE is applied on two state-of-the-art filtering techniques - Nearest Neighbor(NN)/Burak filter and CLAMI?

To demonstrate the practicality and effectiveness of CDE-SMOTE, we compare our proposed method to other related works. Two well-known defect prediction approaches are implemented: Burak filter and CLAMI.

The Burak filter experiments were performed across 14 extracted datasets and the average increase in performance for each classifier after CDE-SMOTE is applied to the training dataset selected by Burak filter is presented in Table 6.

Table 6: Performance increase (%) 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

The results in Table 6 indicate that by taking into account the distribution difference, the prediction performance could be significantly improved. Across all prediction models, we observe an average of 20% increase regarding the Balance and F-measure values and 40% regarding the G-measure values. The win-tie-loss (Wilcoxon signed-rank tests at $p < 0.05$) results of CDE-SMOTE+BURAK Filter against the BURAK Filter alone is displayed in Figure 9. Compared to just using Burak's filter alone, the 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 losing in terms of Probability of False Alarm (PF) across all the prediction models. This demonstrates that CDE-SMOTE can be used in conjunction with Burak filter and it does provide a significant improvement in prediction performance.

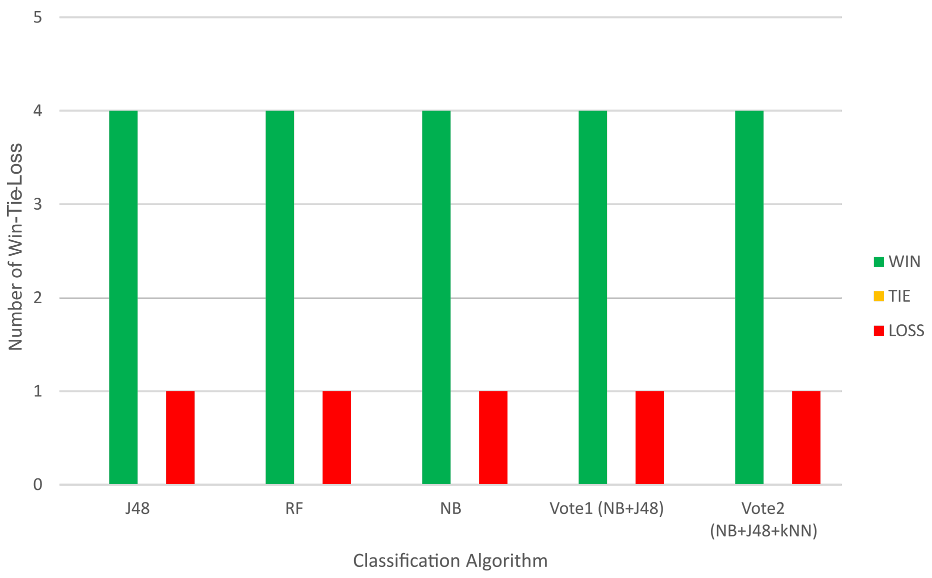

Figure 9: Wilcoxon Win-Tie-Loss comparison of CDE-SMOTE with Burak Filter vrs Burak Filter:

In contrast to the Burak experiments, the CLAMI algorithm was directly trained on a single dataset, which was selected as the unlabeled dataset because CLAMI requires no training dataset. The results from the 14 trained CLAMI datasets was then compared to the CDE-SMOTE results from Experiment 3. Table 7 displays the averaged prediction performances of CLAMI across 14 unlabeled datasets and across 182 cross-project pairs for CDE-SMOTE.

As shown in Table 7, 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 prediction approach, it should, however, be noted that, the results computed

Table 7: Performance increase (%) comparison between CLAMI and CDE-SMOTE (trained and tested on Cross-Project pairs randomly chosen)

	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

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 10 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.

Considering the Balance and G-measure values (Figure 10), there is almost no difference between CLAMI and CDE-SMOTE. Their medians are exactly the same (0.640) with slightly larger ranges for the CDE-SMOTE values. Regarding the F-measure values, CDE-SMOTE shows an improvement over CLAMI with 13.16% increase 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 compute another Win-Tie-Loss comparison for each selected cross-project pair. Win is defined as the case where CDE-

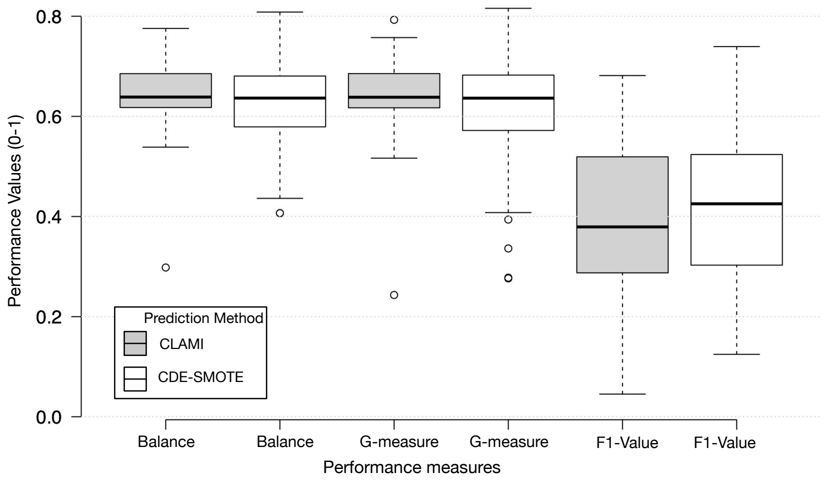

Figure 10: Box plots Performance (Actual value) comparison between CLAMI and CDE-SMOTE (VOTE 2) for Cross-Project training datasets chosen at random

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 11 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).

We observe that the overall percentage of the "Success" case 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 produced a "Success" compared to CLAMI, offering better prediction performances. This shows that CDE-SMOTE could achieve better performance results than the CLAMI algorithm when the training dataset is carefully selected.

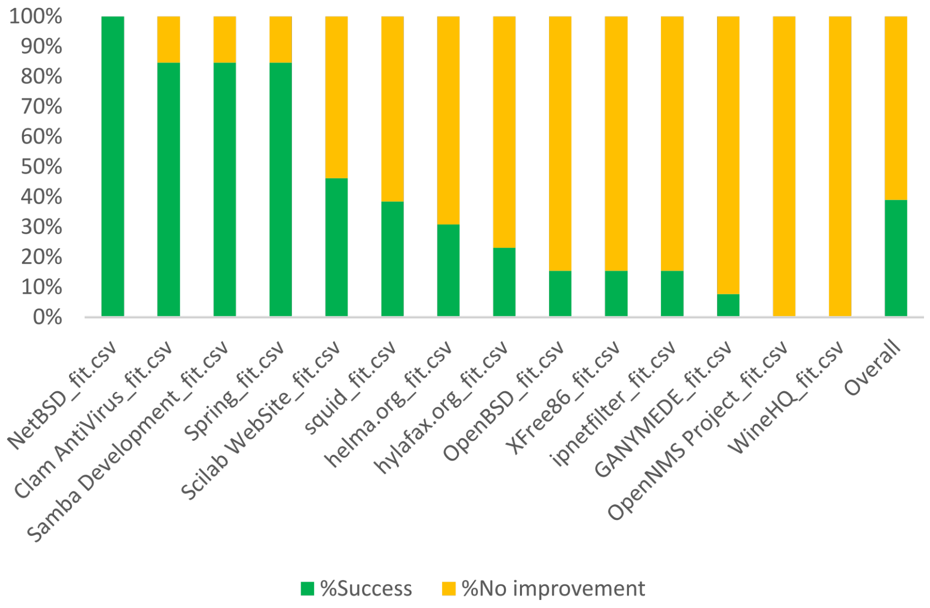

Figure 11: Percentage of cases that CDE-SMOTE shows Significant improvement over CLAMI

In Table 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 11, the results in Table 8 also indicates that when the training datasets are carefully selected, CDE-SMOTE approach does perform significantly better than CLAMI. We observe how CLAMI performs worse for the NetBSD dataset. When we 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 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.

Table 8: Performance increase (%) comparison between CLAMI and CDE-SMOT when the Cross-Project training dataset is carefully selected

	Vote 2 (J48+RF+kNN)				Vote 2 (J48+RF+kNN) (Con.)		
Training Dataset	Increase performance (Percentage)			Training Dataset	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	

6. Discussion

6.1. Summary of Results

Results from experiment 1 show the danger of applying training data from one project to predict another project without considering their class distributions. The results also indicate that the low performance caused by the class distribution could be mitigated by appropriately modifying the class distribution of training dataset. Using the modified dataset, significant improvements (an increase of at least 5% in performance) are observed in 64% of the test cases according to Wilcoxon signed rank test and thus validates our first hypothesis.

Experiment 2 demonstrates that in the practical scenario of a cross-project prediction, the class distribution of an unlabeled dataset can be estimated. An average positive rate (PR) error of 0.1689 was observed in comparison to just using the unmodified training data. These estimations significantly reduced 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. The results validate RQ3 by confirming that the estimated distribution can be used as a substitute for the actual distribution and can significantly improve (increase by at least 5%) the cross-project defect prediction performance in 63% of the test cases according to Wilcoxon signed rank tests.

The final 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 the prediction performance of models by 27%. In comparison to CLAMI, when the training dataset is randomly selected, a slight improvement in F-measure is observed, while significant performance improvements are observed when similar cross-project pairs were selected.

6.2. General Discussion and Implications of Results

From the empirical results and statistical tests computed, the success of CDE-SMOTE confirms the association between defect prediction performance and class distribution of defect datasets. We have provided sufficient support that oversampling can improve the performance of cross-project prediction models when appropriately applied. Based on the hypothesis of CDE-SMOTE, not only have we shown that class distribution of the training dataset is important but the amount of oversampling applied to the training data is important as well. The results for CDE-SMOTE is dependent on the strategy to use the reverse of the class distribution of the testing data. We use the reverse of the estimated class distribution value to dictate the amount of oversampling since the stability conclusion on the amount of oversampling to be applied for defect prediction studies is yet to be established [9].

The experiments also demonstrate that even after preprocessing the data using the Burak filter, CDE-SMOTE can significantly help improve the overall performance of prediction models trained on these datasets. The results of CDE-SMOTE and CLAMI were also very competitive when CDE-SMOTE was applied on randomly paired projects. Although CLAMI is useful and effective for unsupervised classification, we note that not all projects are suitable for the CLAMI threshold approach. An example is observed in experiment 4 where CLAMI could not capture the underlying structure of the NetBSD project and achieved a very low performance (Balance: 0.298, G-measure: 0.243, and F-measure: 0.045). Carefully pairing the cross-projects considering their similarities and applying CDE-SMOTE will improve the prediction performance even if the datasets are unlabeled as demonstrated by the CLAMI experiments. This also confirms the conclusion made by He et al. [3] that carefully selecting the training data from different projects is very essential for improved prediction perform. Our approach is practically feasible and easy to implement. The general 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 can be applied to improve performance of cross-project prediction models.

6.3. Threats to Validity

As an empirical study, there are several potential limitations. The construct, internal and extenal threats to validity in this study are discussed in this section.

6.3.1. Construct Validity

We considered only process metrics which are different from static code and other object-oriented metrics. These metrics however, have been used for most prediction studies and have been shown in literature to perform better at predicting defects [48, 49, 50, 47, 51]. The performance measures used for the experiment were carefully selected to ensure the reliability of the results. We adopted several performance measures which are used in similar cross project prediction studies for evaluation. Other performance measures will be considered in a future study.

6.3.2. Internal Validity

The method of labeling (faulty or clean) the modules for each dataset using comments from the commit logs poses a possible threat to the results obtained. The datasets used for the experiments were extracted from commit logs using open source tools. 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 modules aside those in the commit logs. The proposed approach is compared with two state-of-the-art techniques. For a fair comparison, we implemented the algorithms of these techniques in strict adherence to the authors instructions.

6.3.3. Statistical Conclusion Validity

In this study, we used the Wilcoxon sign rank test for the win-tie-loss analysis. We however acknowledge the existence of more robust non-parametric statistical test such as the Brunner's test and cliff's effect size recommended by Kitchenham et al. [52]. These tests will be considered in future studies.

6.3.4. 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. We acknowledge the existence of several methods for tackling the class distribution challenge such as undersampling, oversampling, resampling and cost-sensitive classifier. However, we only considered the SMOTE approach to aid handle the class imbalance issue. This approach is widely used in defect prediction studies and regarded as an efficient method for handling the class imbalance issue of defect datasets. Impact of these other methods for modifying the training dataset distribution in the cross-project scenario is left for future studies. In estimating the class distribution of the unlabeled dataset, while there are several ways to quantify the class distribution of the unlabeled dataset, we adopted the classification and count (CC) technique. 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, seven classification algorithms were considered in this study. These algorithms are widely used for several defect prediction studies. We also extended our study to include ensemble techniques. However, many classification algorithms were not considered. Consideration of more prediction models is left for future studies.

7. Conclusions

Class imbalance and distribution mismatch of datasets are associated with real world defect prediction datasets and significantly affects the performance of the prediction models trained on cross-project datasets. This study proposed an approach for improving the prediction performance of cross-project defect prediction models utilizing class distribution estimation and SMOTE referred to as CDE-SMOTE. CDE-SMOTE alleviates the negative effect of class distribution difference between the source and target projects and class imbalance on prediction performance. We validate our approach by conducting four empirical experiments over 14 open source projects and 7 prediction models. The 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 can be estimated and mitigated by using oversampling to shift decision boundary toward that minority class, thus improving the overall defect prediction performance. The results of this paper emphasizes the importance of class distribution and its effects on the performance of defect prediction models. Specifically, the major contributions of this paper are:

- The study demonstrate the detrimental effects of building a cross-project defect prediction model without considering the distribution of the intended target projects and how to improve the prediction performance using an estimated distribution.
- The study confirms that the class distribution of the unlabeled project could be estimated even in the cross-projects situation, and provide the steps on how to estimate this distribution appropriately.

The results obtained by our approach indicates that CDE-SMOTE could be used by practitioners to predict the defect-proneness of software project modules and could be easily applied to any software engineering project. For future studies, we plan to consider more measurement software data metrics and include other techniques for recording all faulty models. We also intend to further optimize the CDE-SMOTE approach by considering other quantification and class distribution modification techniques. Lastly, we aim to compare our proposed approach with some genetic algorithm frameworks [53].

8. Acknowledgements

This research was supported by JSPS KAKENHI Grant number 26330086, was conducted as a part of the Program for Advancing Strategic International Networks to Accelerate the Circulation of Talented Researchers: Interdisciplinary Global Networks for Accelerating Theory and Practice in Software Ecosystem, Japanese Goverment Scholarship (Monbukagakusho) and was supported in part by the General Research Fund of the Research Grants Council of Hong Kong (No. 125113, 11200015 and 11214116), and the research funds of City University of Hong Kong (No. 7004683 and 7004474).

References

[1] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: a benchmark and an extensive comparison, Empirical Software Engineering 17 (4-5) (2012) 531–577.

[2] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, Software Engineering, IEEE Transactions on 34 (4) (2008) 485–496.

[3] Z. He, F. Shu, Y. Yang, M. Li, Q. Wang, An investigation on the feasibility of cross-project defect prediction, Automated Software Engineering 19 (2) (2012) 167–199.

[4] B. Turhan, T. Menzies, A. B. Bener, J. Di Stefano, On the relative value of cross-company and within-company data for defect prediction, Empirical Software Engineering 14 (5) (2009) 540–578.

[5] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, 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, ACM, 2009, pp. 91–100.

[6] Y. Ma, G. Luo, X. Zeng, A. Chen, Transfer learning for cross-company software defect prediction, Information and Software Technology 54 (3) (2012) 248–256.

[7] D. Ryu, O. Choi, J. Baik, Value-cognitive boosting with a support vector machine for cross-project defect prediction, Empirical Software Engineering 21 (1) (2016) 43–71.

[8] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, 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, IEEE, 2007, pp. 196–204.

[9] K. E. Bennin, J. Keung, A. Monden, Y. Kamei, N. Ubayashi, Investigating the effects of balanced training and testing datasets on effort-aware fault prediction models, in: Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual, Vol. 1, IEEE, 2016, pp. 154–163.

[10] J. Riquelme, R. Ruiz, D. Rodríguez, J. Moreno, Finding defective modules from highly unbalanced datasets, Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos 2 (1) (2008) 67–74.

[11] V. García, J. S. Sánchez, R. A. Mollineda, On the effectiveness of preprocessing methods when dealing with different levels of class imbalance, Knowledge-Based Systems 25 (1) (2012) 13–21.

[12] N. Japkowicz, S. Stephen, The class imbalance problem: A systematic study, Intelligent data analysis 6 (5) (2002) 429–449.

[13] A. A. Shanab, T. M. Khoshgoftaar, R. Wald, 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, IEEE, 2012, pp. 415–422.

[14] S. Wang, X. Yao, Using class imbalance learning for software defect prediction, IEEE Transactions on Reliability 62 (2) (2013) 434–443.

[15] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, S. Mensah, Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction, IEEE Transactions on Software Engineering.

[16] K. Bennin, J. Keung, A. Monden, P. Phannachitta, S. Mensah, The significant effects of data sampling approaches on software defect prioritization and classification, in: 11th International Symposium On Empirical Software Engineering and Measurement, ESEM 2017, 2017.

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

[18] D. Ryu, J.-I. Jang, J. Baik, A transfer cost-sensitive boosting approach for cross-project defect prediction, Software Quality Journal (2015) 1–38`doi:10.1007/s11219-015-9287-1`.
URL `http://dx.doi.org/10.1007/s11219-015-9287-1`

[19] B. X. Wang, N. Japkowicz, Boosting support vector machines for imbalanced data sets, Knowledge and Information Systems 25 (1) (2010) 1–20. `doi:10.1007/s10115-009-0198-y`.
URL `http://dx.doi.org/10.1007/s10115-009-0198-y`

[20] G. Forman, Quantifying counts and costs via classification, Data Mining and Knowledge Discovery 17 (2) (2008) 164–206.

[21] J. Nam, S. Kim, Clami: Defect prediction on unlabeled datasets (t), in: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, 2015, pp. 452–463. doi:10.1109/ASE.2015.56.

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

[23] T. M. Khoshgoftaar, A. S. Pandya, D. L. Lanning, Application of neural networks for predicting program faults, Annals of Software Engineering 1 (1) (1995) 141–154.

[24] F. Xing, P. Guo, 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), 2005, pp. 213–222. doi:10.1109/ISSRE.2005.6.

[25] G. J. Pai, J. B. Dugan, Empirical analysis of software fault content and fault proneness using bayesian methods, Software Engineering, IEEE Transactions on 33 (10) (2007) 675–686.

[26] H. Hata, O. Mizuno, T. Kikuno, Bug prediction based on fine-grained module histories, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 200–210.
URL http://dl.acm.org/citation.cfm?id=2337223.2337247

[27] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, Software Engineering, IEEE Transactions on 33 (1) (2007) 2–13.

[28] C. Catal, U. Sevim, 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, 2009, pp. 199–204. doi:10.1109/ITNG.2009.12.

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

[30] M. Jureczko, 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, ACM, 2010, p. 9.

[31] Y. Zhang, D. Lo, X. Xia, 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, 2015, pp. 264–269. doi:10.1109/COMPSAC.2015.58.

[32] A. Panichella, R. Oliveto, 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, 2014, pp. 164–173. doi:10.1109/CSMR-WCRE.2014.6747166.

[33] W. N. Poon, K. E. Bennin, J. Huang, P. Phannachitta, J. W. Keung, Cross-project defect prediction using a credibility theory based naive bayes classifier, in: Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on, IEEE, 2017, pp. 434–441.

[34] M. Tan, L. Tan, S. Dara, C. Mayeux, Online defect prediction for imbalanced data, in: Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 99–108.
URL http://dl.acm.org/citation.cfm?id=2819009.2819026

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

[36] A. Esuli, F. Sebastiani, Optimizing text quantifiers for multivariate loss functions, ACM Transactions on Knowledge Discovery from Data (TKDD) 9 (4) (2015) 27.

[37] W. Gao, 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, ACM, 2015, pp. 97–104.

[38] G. King, Y. Lu, et al., Verbal autopsy methods with multiple causes of death, Statistical Science 23 (1) (2008) 78–91.

[39] J. C. Xue, 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, ACM, New York, NY, USA, 2009, pp. 897–906. doi:10.1145/1557019.1557117.
URL http://doi.acm.org/10.1145/1557019.1557117

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

[41] L. Breiman, Random forests, Machine learning 45 (1) (2001) 5–32.

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

[43] S. Le Cessie, J. C. Van Houwelingen, Ridge estimators in logistic regression, Applied statistics (1992) 191–201.

[44] D. W. Aha, D. Kibler, M. K. Albert, Instance-based learning algorithms, Machine learning 6 (1) (1991) 37–66.

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

[46] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten, The WEKA data mining software: An update, SIGKDD Explor. Newsl. 11 (1) (2009) 10–18.

[47] R. Moser, W. Pedrycz, 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, 2008, pp. 181–190. `doi:10.1145/1368088.1368114`.

[48] M. D'Ambros, M. Lanza, R. Robbes, An extensive comparison of bug prediction approaches, in: Proceedings of 2010 7th IEEE Working Conference on Mining Software Repositories (MSR), IEEE, 2010, pp. 31–41.

[49] T. Mende, R. Koschke, Revisiting the evaluation of defect prediction models, in: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, ACM, 2009, p. 7.

[50] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, A. E. Hassan, Revisiting common bug prediction findings using effort-aware models, in Proceedings of 2010 IEEE International Conference on Software Maintenance (ICSM) (2010) 1–10.

[51] K. E. Bennin, K. Toda, Y. Kamei, J. Keung, A. Monden, N. Ubayashi, Empirical evaluation of cross-release effort-aware defect prediction models, in: Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on, IEEE, 2016, pp. 214–221.

[52] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, A. Pohthong, Robust statistical methods for empirical software engineering, Empirical Software Engineering (2016) 1–52`doi:10.1007/s10664-016-9437-5`.
URL `http://dx.doi.org/10.1007/s10664-016-9437-5`

[53] J. Murillo-Morera, C. Castro-Herrera, J. Arroyo, R. Fuentes-Fernández, An automated defect prediction framework using genetic algorithms: A

validation of empirical studies, Inteligencia Artificial 19 (57) (2016) 114–137.