Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems | School of Computing and Information Systems

6-2016

Condensing class diagrams with minimal manual labeling cost

Xinli YANG

David LO
Singapore Management University, davidlo@smu.edu.sg

Xin XIA

Jianling SUN

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Software Engineering Commons

Citation

YANG, Xinli; David LO; XIA, Xin; and SUN, Jianling. Condensing class diagrams with minimal manual labeling cost. (2016). *COMPSAC 2016: Proceedings of the 40th IEEE Annual International Computers, Software and Applications Conference, Atlanta, Georgia, 10-14 June 2016*. 22-31.
Available at: https://ink.library.smu.edu.sg/sis_research/3566

This Conference Proceeding Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Published in COMPSAC 2016: Proceedings of the 40th IEEE Annual International Computers, Software and Applications Conference, Atlanta, Georgia, 10-14 June 2016, Pages 22-31.
http://doi.org/10.1109/COMPSAC.2016.83

Condensing Class Diagrams With Minimal Manual Labeling Cost

Xinli Yang*, David Lo†, Xin Xia*‡, and Jianling Sun*
*College of Computer Science and Technology, Zhejiang University, Hangzhou, China
†School of Information Systems, Singapore Management University, Singapore
zdyxl@zju.edu.cn, davidlo@smu.edu.sg, {xxia, sunjl}@zju.edu.cn

***Abstract*—Traditionally, to better understand the design of a project, developers can reconstruct a class diagram from source code using a reverse engineering technique. However, the raw diagram is often perplexing because there are too many classes in it. Condensing the reverse engineered class diagram into a compact class diagram which contains only the important classes would enhance the understandability of the corresponding project.**

A number of recent works have proposed several supervised machine learning solutions that can be used for condensing reverse engineered class diagrams given a set of classes that are manually labeled as important or not. However, a challenge impacts the practicality of the proposed solutions, which is the expensive cost for manual labeling of training samples. More training samples will lead to better performance, but means higher manual labeling cost. Too much manual labeling will make the problem pointless since the aim is to automatically identify important classes.

In this paper, to bridge this research gap, we propose a novel approach *MCCondenser* which only requires a small amount of training data but can still achieve a reasonably good performance. *MCCondenser* firstly selects a small proportion of all data, which are the most representative, as training data in an unsupervised way using k-means clustering. Next, it uses ensemble learning to handle the class imbalance problem so that a suitable classifier can be constructed based on the limited training data. To evaluate the performance of *MCCondenser*, we use datasets from nine open source projects, i.e., ArgoUML, JavaClient, JGAP, JPMC, Mars, Maze, Neuroph, Wro4J and xUML, containing a total of 2640 classes. We compare *MCCondenser* with two baseline approaches proposed by Thung et al., both of which are state-of-the-art approaches aimed to reduce the manual labeling cost. The experimental results show that *MCCondenser* can achieve an average AUC score of 0.73, which improves those of the two baselines by nearly 20% and 10% respectively.

Keywords*—*Class Diagram, Unsupervised Learning, Ensemble Learning, Manual Labeling, Cost Saving

I. Introduction

To better understand the design of a project, developers can reconstruct a class diagram from source code using a reverse engineering technique. However, the raw reverse engineered diagram is often confusing because there are often too many classes in it. Condensing the reverse engineered class diagrams becomes a necessity then. To condense a reverse engineered class diagram, one would need to find all the important classes. A diagram which only contains the important classes would be succinct, not cluttered with unnecessary details, and and help developers to understand a project better [1].

In recent years, there are several research studies that propose automated techniques to condense reverse engineered class diagrams [2], [3]. Osman et al. analyse many machine learning techniques and are the first to propose an automated approach to condense reverse engineered class diagrams [2]. Thung et al. extend their work by adding a set of network metrics as extra features for training and propose an optimistic classification technique to achieve better performance [3].

Although the above proposed techniques have achieved some good results, they are not practical enough because both of them use half the data (i.e., half of the classes in a diagram) as training samples, which means half the data are needed to be manually labeled first. The expensive cost for manually labeling training samples poses a challenge to the practicality of the proposed techniques. Therefore, in this work, our goal is to reduce manual labeling effort while still achieving a reasonable performance at the same time.

In this paper, to bridge this research gap, we propose an approach named *MCCondenser* (Minimal Cost Condenser) which addresses the reverse engineered class diagrams condensing problem requiring only *a small amount of training data*. *MCCondenser* mainly includes two phases: a sample selection phase and a model building phase. In the sample selection phase, we select a small proportion of all data, which are the most representative, as training data in an unsupervised way using k-means clustering. In the model building phase, we use ensemble learning to handle *class imbalance* problem and build an ensemble classifier based on the selected training samples. Class imbalance problem exists since most of the classes are unimportant and only a few are important ones.

To evaluate *MCCondenser*, we use AUC (Area Under the Receiver Operating Characteristic Curve) as the evaluation metric. AUC was also used to evaluate prior works by Osman et al. [2] and Thung et al. [3] that propose approaches to condense class diagrams. It is also widely used by many software engineering studies that leverage classification algorithms [4], [5], [6]. A higher AUC score means that a method achieves a better performance, with a perfect method achieving an AUC score of 1. An AUC score of more than 0.7 is often considered reasonably good in the literature [7], [8], [9]. We perform experiments on nine open source software projects from different communities, i.e., ArgoUML, JavaClient, JGAP, JPMC, Mars, Maze, Neuroph, Wro4J and xUML, containing a total of 2640 classes. We compare our approach with two

‡Corresponding author.

baseline approaches proposed by Thung et al. [3], [10], both of which are state-of-the-art approaches aimed to reduce the manual labeling cost. The experimental results show that *MCCondenser* can achieve an average AUC score of 0.73 and improves those of the two baselines by nearly 20% and 10% respectively.

The main contributions of this paper are:

1) We propose a novel approach *MCCondenser* for condensing the reverse engineered class diagrams, which only requires a small amount of training data, while still achieving a reasonable performance. The approach combines unsupervised learning and ensemble learning, and reduces the cost of manual labeling by a large amount.
2) We compare *MCCondenser* with two state-of-the-art baseline approaches developed for a similar purpose proposed by Thung et al. [3], [10] on nine software projects. The experiment results show that *MCCondenser* achieves substantial improvement over the two baselines.

The rest of our paper is organized as follows. Section II introduces the background of our work and technical rationale of our proposed approach. Section III presents the overall framework of our approach and elaborates the techniques that we use in our approach. Section IV describes our experiments and the results. Section V discusses the related work. Conclusion and future work are presented in the last section.

II. Preliminaries

In this section, we first introduce the basic concepts of condensing reverse engineered class diagram in Section II-A. We then present the challenge of condensing reverse engineered class diagrams and state our problem definition in Section II-B. Next, we briefly introduce the two main techniques we leverage in our approach, namely unsupervised learning and ensemble learning, in Section II-C and II-D respectively. Finally, we present the technical rationales of our proposed approach in Section II-E.

A. Class Diagram Condensing

Approaches that condense reverse engineered class diagrams aim to identify important classes from a project so that the diagram is smaller and not cluttered with unnecessary details [2], [3], with an end goal of improving the understanding of a project [1]. Typically, an approach that condenses a reverse engineered class diagram follows these steps:

1) **Reverse Engineering.** Use a system modeling tool such as MagicDraw[1] to generate a class diagram that describes a piece of code.
2) **Feature Extraction.** Extract a set of software metrics that can characterize classes in the reverse engineered class diagrams. We can use a software metric tool such as SDMetrics[2] to extract class diagram metrics. SDMetrics can calculate a total of 32 metrics which can be divided into 5 categories namely size, coupling, inheritance, complexity and diagram. Also, we can construct a network in which the nodes are classes and the edges are various relationships between pairs of classes, and extract network metrics as features (c.f., [3]).
3) **Model Building.** Build a discriminative model by using a classification algorithm trained from a set of labeled training data samples (i.e., classes that are manually marked as important or not) characterized by the features extracted. This trained discriminative model is then able to predict the label of an unknown instance (i.e., an unknown class) whether it is "important" or "unimportant".
4) **Model Application.** For an unlabeled class, first extract the values of various features that characterize it, and then input these values to the learned model which will predict whether the class is "important" or "unimportant".

[1]www.nomagic.com

[2]www.SDMetrics.com

B. Problem Definition

Given a number of unlabelled classes, we want to select as few as possible training samples (i.e., classes in a diagram) to manually label, and automatically build a statistical model based on these training samples. This trained statistical model is then used to label the remaining classes automatically. Our approach should meet two requirements to be practical. First, it should minimize the manual labeling effort. Second, it should still achieve a reasonable performance .

C. Unsupervised Learning

Unsupervised learning is a category of machine learning methods [11]. As its name implies, unsupervised learning does not require any labels. Since it does not require labels, which often needs to be manually assigned, it is valuable for many research problems where manual labeling is an expensive process. K-means clustering is one of the classic unsupervised learning algorithms [12]. It groups a set of data into several clusters by minimizing the mean squared distance from each data point to its cluster centroid.

D. Ensemble Learning

Ensemble learning is a technique to improve classification accuracy by combining many different models. Generally, different classifiers have different characteristics, such as their intrinsic working principles and their sensitivity to different training data. For some data samples, different classifiers may make different predictions. Ensemble learning can improve the classification performance by combining the predictions of multiple different models (aka. classifiers) into a single robust prediction model [13], [14]. Different models can be constructed by using different classification algorithms, or different subsets of a training data, which are then combined together. In this paper, we use stacking as a method to combine the prediction models. Stacking is a very general ensemble learning approach, in which two levels of classification are used [15]. In the first level, several different classifiers are trained based on the training dataset. In the second level, a final classifier is trained based on the output of the first-level classifiers.

E. Technical Rationales

Our proposed approach *MCCondenser* combines unsupervised learning and ensemble learning. The effectiveness of *MCCondenser* relies on the following two rationales.

Rationale 1: K-means clustering can select better training data than random selection.

Rationale 2: An ensemble of classifiers can achieve better performance than a single classifier.

The first rationale seems intuitive. To demonstrate it, we perform experiments on six datasets, i.e., ArgoUML, JavaClient, JPMC, Maze, Neuroph, xUML. We use the same features (cf. Section III-A), the same amount of training data (i.e., one-tenth of all data), the same classifier (i.e., random forest) and the evaluation metric AUC score (cf. Section IV-B). The only difference is the method to select training data. Let N be the amount of training data. In the first approach, we randomly select N training data from the datasets, while in the second approach, we first group all the data into N clusters and then select training data by picking one instance from each cluster. Table I shows the experiment results which validate our rationale. In all of the datasets, the performance of the approach using K-means clustering is better than that of the approach using random selection.

TABLE I. AUC SCORES OF TWO APPROACHES WITH DIFFERENT METHODS TO SELECT TRAINING DATA

Project	Random	K-means Clustering
ArgoUML	0.5976	0.6514
JavaClient	0.8315	0.8586
JPMC	0.5342	0.6407
Maze	0.4789	0.5815
Neuroph	0.8026	0.8296
xUML	0.8719	0.9239

The second rationale has been proven by many researches [13], [14]. There are two main components in the error of a classifier, i.e., bias and variance. Bias is the difference between the decision boundary of a classifier and the true decision boundary. Variance means that different choices of training data lead to different models. By building an ensemble of classifiers, we can reduce both bias and variance [15]. Therefore, an ensemble of classifiers can often achieve better performance than a single classifier.

III. Our Proposed Approach

In this section, we present the details of our proposed approach *MCCondenser*. We first present the overall framework of *MCCondenser*, and then we describe in detail the individual steps in the overall framework.

A. Overall Framework

Figure 1 presents the overall framework of our proposed approach *MCCondenser*. The framework contains three phases: the sample selection phase, the model building phase and the prediction phase. In the sample selection phase, we select the most representative samples as training data in an unsupervised way and manually label them as important or unimportant classes. In the model building phase, we build a classifier (i.e., a statistical model) from the limited training data selected in the sample selection phase by leveraging an ensemble learning technique. In the prediction phase, this classifier would be used to predict if an unknown class would be important or not.

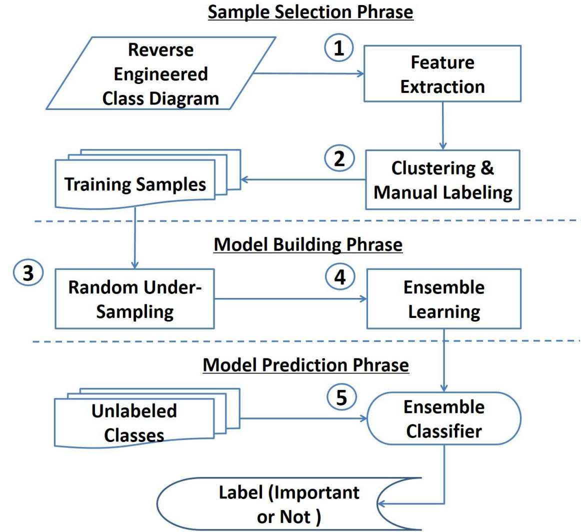

Fig. 1. The Overall Framework of *MCCondenser*.

Our framework first extracts a number of features from the classes of a project (Step 1). Features are various quantifiable characteristics of the classes that could potentially distinguish the important classes from the unimportant ones. In this paper, we use the 18 features proposed by Thung et al. [3] as shown in Table II. The features are from three kinds of metrics, i.e., size, coupling and network metrics. The network metrics are computed by first constructing a network given a class diagram where the nodes in the network are classes and the edges are various relationships between pairs of classes. In addition, all the features are normalized using z-score method (cf. Section III-B) so that the values of all features are in the same order of magnitude. With these normalized features, we form a feature vector for each class of the project, which is also called a data sample.

Next, we select the most representative samples as training data using k-means clustering (cf. Section III-C) and label them manually (Step 2). Note that the more the amount of training data is, the more the manual labeling cost is. Thus, the goal of this step is to minimize the number of data samples to label.

To deal with the class imbalance problem (i.e., there are more unimportant than important classes), we use random under-sampling (cf. Section III-D) to build different base classifiers, and use ensemble learning (cf. Section III-E) to combine them (Steps 3-4). The base classifier can deal with the data imbalance problem and the ensemble learning can overcome the information deficiency problem of a single base classifier due to random under-sampling. The ensemble of different base classifiers considers all the information the training samples provide so that the performance of the ensemble classifier can be much better than that of a single base classifier.

In the prediction phase, we use the ensemble classifier to

TABLE II. EIGHTEEN FEATURES USED IN OUR APPROACH

Name	Category	Description
NumAttr	Size	The number of attributes in a class [2]
NumOps	Size	The number of methods in a class [2]
NumPubOps	Size	The number of public methods in a class [2]
Setters	Size	The number of methods whose names start with 'set' [2]
Getters	Size	The number of methods whose names start with 'get', 'is', or 'has' [2]
DepOut	Coupling	The number of dependencies where a class uses other classes [2]
DepIn	Coupling	The number of dependencies where a class is used by other classes [2]
ECAttr	Coupling	The number of times a class is externally used as an attribute type [2]
ICAttr	Coupling	The number of attributes in a class having another class or interface as their types [2]
ECPar	Coupling	The number of times a class is externally used as a parameter type [2]
ICPar	Coupling	The number of parameters in a method of a class having another class or interface as their types [2]
Barycenter	Network	The barycenter centrality score of a class in the network [3]
Betweenness	Network	The betweenness centrality score of a class in the network [3]
Closeness	Network	The closeness centrality score of a class in the network [3]
Eigenvector	Network	The eigenvector centrality score of a class in the network [3]
Hub	Network	The hub score of a class in the network [3]
Authority	Network	The authority score of a class in the network [3]
PageRank	Network	The page rank score of a class in the network [3]

predict whether a class with an unknown label is important or not. For each of such classes, our framework first extracts its feature vector and inputs it into all of the trained base classifiers. Each of these classifiers generates a prediction result which may be different from one another. In the end, we ensemble the different prediction results to produce a final prediction result (cf. Section III-F), which is one of the following labels: important or unimportant (Step 5).

B. Data Normalization: Z-Score Method

Considering that the values of the 18 features are not in the same order of magnitude, we perform data normalization on these feature values. In this paper, we use the z-score method to do the normalization [12]. It transforms all values of a feature to make them subject to the Gaussian distribution with a zero mean and a variance of 1. Given a feature f, we denote the mean and variance of its values as $mean(f)$ and $std(f)$ respectively. For each value f_i of the feature f, its normalized value z_i is computed as:

$$z_i = \frac{f_i - mean(f)}{std(f)}$$

C. Sample Selection: K-means Clustering

K-means clustering is a classic unsupervised learning technique [12]. The aim of k-means clustering is to group a set of data points into several clusters so that the mean squared distance from each data point to its cluster centroid is minimized. Initially, k-means clustering randomly picks k data points as k initial centroids, one for a cluster respectively. Next, each data point is assigned to its nearest centroid. In the process, k clusters are formed. Next, the k centroids are updated based on data points in the same cluster. The centroids are calculated by averaging all the data points in the same cluster. Note that these centroids do not necessarily correspond to any data points. The process is repeated for a number of iterations until the k centroids do not change any further.

In *MCCondenser*, after we have k clusters, for each cluster, we pick the data point (i.e., feature vector) that is the nearest to its centroid as a training sample. Thus, at the end we have k training samples. The selection strategy has two advantages. First, the k training samples are the most representative because they are the closest to the center of their corresponding cluster. At the same time, the k training samples have the biggest diversity because they are the farthest to one another. Therefore, the k training samples can be a suitable training set for model building.

K-means clustering has two major tunable parameters. One is the number of clusters k and the other is the metric to compute the distance between two data points. We set k to be equal to the labeling budget (i.e., number of classes to label). We use Euclidean distance to measure the distance between two data points. Euclidean distance between two data points is the length of the line segment connecting them.

D. Model Building: Random Under-Sampling

Random under-sampling is one of the effective approaches to deal with class imbalance problem [12], [16]. Class imbalance problem is always a big problem in machine learning. It can lead to a classifier that performs poorly. Random under-sampling can help the trained classifier so that it is not biased to the majority class (in our case: unimportant class), thus in most cases it can improve the performance of the classifier [17], [18]. Just as the name implies, random under-sampling randomly deletes data belonging to the majority class until the amount of data in the majority class is approximately equal to that of the minority class.

In our problem, we use random under-sampling to make the number of important (minority) and unimportant (majority) classes in the training data equal.

E. Model Building: Ensemble Learning

Although random under-sampling can avoid the class imbalance problem, it may lose some important information because it deletes some training samples. To make up for this deficiency, we introduce ensemble learning. Ensemble learning can improve the classification performance by combining

the predictions of multiple different classifiers into a single robust prediction [13], [14]. Specifically, we first train different base classifiers with different subsets of the training samples acquired by multiple random under-sampling. For each base classifier, we assign it a weight by evaluating its performance. The weight is computed by two steps. First, we compute the AUC score s of each classifier i on the whole training data set. Second, we transform s to the weight w using the following formula:

$$w_i = Base^{s_i}$$

In the formula, $Base$ can be any positive number. The bigger the $Base$ is, the bigger weight a classifier with a higher AUC has. By default, we set $Base$ as 2.

MCCondenser generates $NLearner$ base classifiers. Next, we select classifiers whose weights are different from one another to ensemble. If there are many classifiers with the same weight, we randomly select one. This strategy can reduce the time and space cost (since we store and run less classifiers), while still highly likely to retain all the information contained in the training samples. By default, $NLearner$ is set as 100 to guarantee the diversity of base classifiers.

F. Model Prediction

For an unlabeled class x, we first input its normalized feature vector into all of the trained base classifiers to obtain a set of prediction results. We denote the prediction result of the ith classifier given an unlabeled class x as $p_{x,i}$. Next, we ensemble the different prediction results p to achieve a final ensemble score $Final(x)$ using the following formula:

$$Final(x) = \sum_i w_i \times p_{x,i}$$

$Final(x)$ indicates the likelihood of the class x to be important. If the score is above 0, we assign it to the important class, otherwise we assign it to the unimportant class.

IV. Experiments and Results

In this section, we evaluate the performance of *MCCondenser*. The experimental environment is an Intel(R) Core(TM) T6570 2.10 GHz CPU, 4GB RAM desktop running Windows 7 (64-bit). We first present our experiment setup and evaluation metrics in Sections IV-A to IV-C. We then present five research questions and our experiment results that answer these questions in Section IV-D.

A. Datasets

We evaluate *MCCondenser* on nine datasets from nine well-known open source projects, which are ArgoUML, JavaClient, JGAP, JPMC, Mars, Maze, Neuroph, Wro4J and xUML[3]. These datasets were also used by Osman et al. [2] and Thung et al. [3]. Table III summarizes the statistics of each dataset. We can see that in seven out of nine projects, the datasets are severely imbalanced (i.e., there are many more unimportant classes than important ones).

[3]Datasets are publicly available at: http://sites.google.com/site/classdiag/dataset.zip.

TABLE III. Statistics Of The Datasets Used In Our Study. Ratio = ratio of important to total classes (in percentages).

Project	# Total Classes	# Important Classes	Ratio
ArgoUML	903	44	4.87%
JavaClient	214	57	26.64%
JGAP	171	18	10.52%
JPMC	121	24	19.83%
Mars	840	29	3.45%
Maze	59	28	47.45%
Neuroph	161	24	14.90%
Wro4J	87	11	12.64%
xUML	84	37	44.05%

B. Evaluation Metric

We use Area Under the Receiver Operating Characteristic Curve (AUC) to evaluate the effectiveness of our approach *MCCondenser*. AUC, which was also used in the studies by Osman et al. [2] and Thung et al. [3], is a commonly-used measure to evaluate classification performance. Many other software engineering studies also use AUC as an evaluation metric [4], [5], [6]. The AUC score ranges from 0 to 1, with 1 representing perfect prediction performance. Generally, an AUC score above 0.7 is considered reasonable [7], [8], [9].

To compute AUC, we first plot the Receiver Operating Characteristic Curve (ROC). ROC is a plot of the true positive rate (TPR) versus false positive rate (FPR). TPR and FPR can be derived from a confusion matrix, as shown in Table IV. The confusion matrix lists all four possible prediction results. If a class is correctly classified as "important", it is a true positive (TP); if a class is misclassified as "important", it is a false positive (FP). Similarly, there are false negatives (FN) and true negatives (TN). Based on the four numbers, TPR and FPR are calculated. TPR is the ratio of the number of correctly predicted as "important" classes to the actual number of "important" classes (*TPR* = $\frac{TP}{TP+FN}$). FPR is the ratio of wrongly predicted as "important" classes to the actual number of "unimportant" classes (*FPR* = $\frac{FP}{FP+TN}$). With the ROC, AUC can be calculated by measuring the area under the curve. AUC measures the ability of a classification algorithm to correctly rank classes as important or unimportant. The larger the AUC is, the better is the performance of a classification algorithm.

TABLE IV. Confusion Matrix

	Predicted (to be) Important	Predicted (to be) Unimportant
Truly Important	TP	FN
Truly Unimportant	FP	TN

We use the AUC score as an evaluation metric because it has been shown to be suitable for imbalanced data [19]. As mentioned above, in seven out of the nine datasets, the datasets are severely imbalanced.

C. Experimental Settings

MCCondenser uses few training samples so that the cost of manual labeling can be reduced by much. Therefore, in our experiments, we only select one-tenth of all data as training samples. This means that we set the parameter k of the k-means clustering algorithm as one-tenth of the number of

samples in a dataset (i.e., one-tenth of the number of classes in a reversed engineer class diagram).

For the ensemble learning process, the base classifier we use is random forest. Random forest is an advanced machine learning technique based on decision tree. Osman et al. have shown that random forest outperforms many other algorithms for condensing class diagrams [2]. In a random forest, there are $NTree$ decision trees. $NTree$ is a tunable parameter and we set it as 10 by default.

There are totally three tunable parameters in *MCCondenser*, i.e, $Base$, $NLearner$ and $NTree$. In our experiments, we use the default values of them except for the last research question, in which we change the parameter settings since we want to investigate the influence of different values of these parameters on the performance of *MCCondenser*.

We test the performance of *MCCondenser* in the remaining nine-tenth of the data. To reduce the bias due to training set selection, we repeat all experiments 10 times and report the average performance.

D. Research Questions

Our experiments are designed to answer the following research questions:

RQ1 How effective is MCCondenser?

Motivation. In the first research question, we want to investigate the effectiveness of our approach *MCCondenser*. We need to compare it with some baselines. The baseline approaches are selected based on the following two criteria. First, the approach should be aimed to reduce manual labeling, that is, it should work with small amount of training data. Second, the approach should work for the problem of condensing class diagram.

Approach. We compare *MCCondenser* against two state-of-the-art baseline approaches, both of which are proposed by Thung et al. [3], [10]. The first baseline is an active semi-supervised approach initially designed for defect categorization [10]. It is aimed to reduce manual labeling of defect samples and it has been proven to work well when only one-tenth of all data samples are used as training samples. It is referred to as *Baseline-1* in the following text. The second baseline is an optimistic classification technique for condensing class diagram [3]. It also deals with label scarcity problem by optimistically assigning labels to some of the unlabeled data. It is referred to as *Baseline-2* in the following text.

For comparability sake, we use one-tenth of all data as training samples for all the approaches. We use the evaluation metric AUC mentioned above to make comparisons. To increase the confidence of the results, we repeat all experiments 10 times and report the average results. In addition, we also calculate p-value (using Wilcoxon signed-rank statistical test with Bonferroni correction) and Cliff's delta to better investigate whether or not our approach improves the baselines significantly and substantially.

Results. Tables V and VI present the AUC values of *MCCondenser* as compared with those of the two baselines, respectively. From the tables, we can see that *MCCondenser* achieves an average AUC score of 0.73, while Baseline-1 and Baseline-2 achieve AUC scores of 0.62 and 0.66 respectively. Compared to the two baselines, *MCCondenser* improves the AUC scores by 18% and 10% respectively, which is a substantial improvement.

TABLE V. AUC OF MCCONDENSER COMPARED WITH BASELINE-1 [10]. IMPROVE. (%) = IMPROVEMENT OF MCCONDENSER OVER BASELINE-1 (IN PERCENTAGES).

Project	**Baseline-1**	**MCCondenser**	**Improve. (%)**
ArgoUML	0.5057	0.6582	30.16%
JavaClient	0.7348	0.8355	13.70%
JGAP	0.5000	0.5533	10.66%
JPMC	0.5966	0.7743	29.79%
Mars	0.5252	0.7764	47.83%
Maze	0.5968	0.5841	-2.13%
Neuroph	0.7392	0.8940	20.94%
Wro4J	0.6163	0.6801	10.35%
xUML	0.7988	0.8149	2.02%
Average	0.6237	0.7301	18.15%

TABLE VI. AUC OF MCCONDENSER COMPARED WITH BASELINE-2 [3]. IMPROVE. (%) = IMPROVEMENT OF MCCONDENSER OVER BASELINE-2 (IN PERCENTAGES).

Project	**Baseline-2**	**MCCondenser**	**Improve. (%)**
ArgoUML	0.5966	0.6582	10.33%
JavaClient	0.7800	0.8355	7.12%
JGAP	0.6878	0.5533	-19.56%
JPMC	0.5725	0.7743	35.25%
Mars	0.6987	0.7764	11.12%
Maze	0.5218	0.5841	11.94%
Neuroph	0.7726	0.8940	15.71%
Wro4J	0.5322	0.6801	27.79%
xUML	0.8201	0.8149	-0.63%
Average	0.6647	0.7301	9.88%

TABLE VII. MAPPINGS OF CLIFF'S DELTA VALUES TO THEIR INTERPRETATIONS [20]

Cliff's Delta (δ)	Interpretation
-1 $<=\delta<$ 0.147	Negligible
0.146 $<=\delta<$ 0.33	Small
0.33 $<=\delta<$ 0.474	Medium
0.474 $<=\delta<=$ 1	Large

To better demonstrate the superiority of our approach, we perform the Wilcoxon signed-rank statistical test with Bonferroni correction to compute the p-value, and also compute the Cliff's delta. Wilcoxon signed-rank statistical test is often used to check if the difference in two data groups is statistically significant (which corresponds to a p-value of less than 0.05) or not. We include the Bonferroni correction to counteract the impact of multiple hypothesis tests. Cliff's delta is often used to check if the difference in two data groups are substantial. The range of Cliff's delta is in [-1, 1], where -1 or 1 means all values in one group are smaller or larger than those of the other group, and 0 means the data in the two groups is similar. The mappings between Cliff's delta scores and effectiveness levels are shown in Table VII. By computing the p-value and Cliff's delta, the extent of which our approach improves over the two baselines can be more rigorously assessed.

Results. Tables VIII and IX present the adjusted p-values and Cliff's deltas of *MCCondenser* compared with the two

TABLE VIII. Adjusted P-Values (after Bonferroni Correction) and Cliff's Delta Comparing the AUC scores of MCCondenser with Those of Baseline-1

Project	Adjusted P-Value	Cliff's Delta
ArgoUML	0.01758	1 (large)
JavaClient	0.1230	0.82 (large)
JGAP	0.5801	0.8 (large)
JPMC	0.01758	1 (large)
Mars	0.01758	1 (large)
Maze	1	-0.16 (small)
Neuroph	0.0879	0.7 (large)
Wro4J	1	0.24 (small)
xUML	1	0 (negligible)

TABLE IX. Adjusted P-Values (after Bonferroni Correction) and Cliff's Delta Comparing the AUC scores of MCCondenser with Those of Baseline-2

Project	Adjusted P-Value	Cliff's Delta
ArgoUML	0.4395	0.62 (large)
JavaClient	0.5801	0.48 (large)
JGAP	0.3340	-0.52 (negligible)
JPMC	0.0527	0.94 (large)
Mars	0.1758	0.52 (large)
Maze	1	0.36 (medium)
Neuroph	0.2461	0.62 (large)
Wro4J	1	0.42 (medium)
xUML	1	-0.14 (negligible)

baselines in terms of their AUC scores respectively. From the tables, we can see the effectiveness of our approach more clearly. Compared with Baseline-1, *MCCondenser* statistically significantly (i.e., adjusted p-value < 0.05) achieves a better performance in three out of the nine datasets, and substantially (i.e., Cliff's delta is not negligible) achieves a better performance in seven out of the nine datasets. Compared with Baseline-2, *MCCondenser* statistically substantially achieves a better performance in seven out of the nine datasets.

MCCondenser is more effective than the baselines. On average, it achieves an AUC of 0.73, which improves the AUC of the two baselines by nearly 20% and 10% respectively.

RQ2 What is the difference in MCCondenser performance when limited and much training data is available?

Motivation. Intuitively, with more training data the performance of our approach would improve. In this research question, we would like to investigate how big is the gap in performance when we use little and much training data. The gap in performance highlights how much performance do we sacrifice for a big reduction in labelling cost.

Approach. We compare the performance of MCCondenser trained using 10% (the default setting) and 90% of the data. We also use AUC as the yardstick for comparison. For each setting, we repeat the experiments 10 times and report the average AUC.

Results. Table X presents the performance of *MCCondenser* when it is trained with 10% of the data compared with its performance when it is trained with 90% of the data. We notice a performance drop of 0.13 when the size of the training data is reduced. In spite of this, the first setting only needs one-tenth of all labels, which can reduce the cost of manual labeling by a large amount. A large reduction in labeling cost only causes a relatively small reduction in AUC.

TABLE X. The Difference of AUC Scores Between MCCondenser Using 10% and 90% Training Data

Project	MCCondenser (10%)	MCCondenser (90%)
ArgoUML	0.6586	0.8172
JavaClient	0.8397	0.8926
JGAP	0.6342	0.8965
JPMC	0.7242	0.8028
Mars	0.7509	0.8766
Maze	0.5461	0.6972
Neuroph	0.9061	0.9449
Wro4J	0.6952	0.8821
xUML	0.8109	0.9244
Average	0.7295	0.8594

With 90% of the data used to train it, MCCondenser *achieves an AUC of 0.83, which is only 0.13 more than the AUC of* MCCondenser *trained with 10% of the data. Thus, the reduction in labeling effort is much less than the reduction in AUC, demonstrating the effectiveness of* MCCondenser.

RQ3 Do k-means clustering and ensemble learning both contribute to the performance of our approach?

Motivation. We have validated the effectiveness of *MCCondenser* through the above two research questions. *MCCondenser* clearly outperforms the two state-of-the-art baselines. In this RQ, we want to go further by investigating the individual contribution of the two key steps of *MCCondenser*, i.e., k-means clustering and ensemble learning.

Approach. To measure the individual contribution of k-means clustering and ensemble learning to the overall performance of *MCCondenser*, we create two incomplete versions of *MCCondenser* – referred to as *Cluster* and *Ensemble* respectively. For *Cluster*, we use k-means clustering to select training samples in the same way as *MCCondenser* but only build a single model without using ensemble learning. For *Ensemble*, we do not use k-means clustering. Instead, we randomly select one-tenth of all data as training samples but build an ensemble model using ensemble learning. We can then observe the individual contribution of k-means clustering by comparing the AUC scores of *Ensemble* and *MCCondenser*, and the individual contribution of ensemble learning by comparing the AUC scores of *Cluster* and *MCCondenser*. Following RQ1, we randomly pick 10% of the data for training, and the rest for evaluation.

TABLE XI. Individual Contribution of K-means Clustering and Ensemble Learning

Project	Cluster	Ensemble	MCCondenser
ArgoUML	0.6391	0.6466	0.6582
JavaClient	0.8520	0.7919	0.8355
JGAP	0.5704	0.6354	0.5533
JPMC	0.6920	0.5990	0.7743
Mars	0.6194	0.7822	0.7764
Maze	0.5841	0.5143	0.5841
Neuroph	0.7756	0.7895	0.8940
Wro4J	0.7462	0.5854	0.6801
xUML	0.9190	0.8765	0.8149
Average	0.7109	0.6912	0.7301

Results. Table XI shows the performance of *Cluster* and *Ensemble* which sheds light to the individual contributions of k-means clustering and ensemble learning. We can note that *MCCondenser* outperforms *Cluster* and *Ensemble* by 3% and 6% respectively. Moreover, *MCCondenser* outperforms both *Cluster* and *Ensemble* in four out of the nine datasets, i.e., ArgoUML, JPMC, Maze and Neuroph. These indicate that both k-means clustering and ensemble learning contribute to the overall performance of *MCCondenser*, and removing any one of them degrades the overall performance.

Both K-means clustering and ensemble learning contribute to the good performance of MCCondenser.

RQ4 How much time does it take for MCCondenser to run?

Motivation. Now that we have examined the effectiveness of our approach *MCCondenser* and the contributions of its key steps, we shall test its efficiency. The efficiency of an approach is also an important factor that contributes towards the practicality of a proposed approach.

Approach. In order to answer the question, we measure the training and testing time of *MCCondenser*. The training time includes the time taken for the sample selection phase and model building phase. The testing time is the time taken for predicting all the test samples. Following RQ1, we randomly pick 10% of the data for training, and the rest for testing.

TABLE XII. TRAINING TIME OF MCCONDENSER AND THE TWO BASELINES (IN SECONDS)

Project	**Baseline-1**	**Baseline-2**	**MCCondenser**
ArgoUML	5.0646	0.1190	2.1257
JavaClient	0.9093	0.0410	0.9574
JGAP	0.7688	0.0352	0.9442
JPMC	0.5031	0.0306	0.8881
Mars	4.6919	0.1083	1.4519
Maze	0.2614	0.0222	0.0330
Neuroph	0.6965	0.0336	0.9426
Wro4J	0.3859	0.0253	0.8730
xUML	0.3570	0.0234	0.8668
Average	1.5154	0.0487	1.0092

TABLE XIII. TESTING TIME OF MCCONDENSER AND THE TWO BASELINES (IN SECONDS)

Project	**Baseline-1**	**Baseline-2**	**MCCondenser**
ArgoUML	0.0054	0.0043	0.1635
JavaClient	0.0048	0.0038	0.0340
JGAP	0.0047	0.0034	0.0143
JPMC	0.0046	0.0036	0.0207
Mars	0.0049	0.0039	0.1179
Maze	0.0045	0.0034	0.0045
Neuroph	0.0047	0.0035	0.0298
Wro4J	0.0044	0.0042	0.0175
xUML	0.0045	0.0035	0.0177
Average	0.0047	0.0037	0.0467

Results. Tables XII and XIII present the training and testing time of *MCCondenser* and the two baselines on the nine datasets. For *MCCondenser*, it takes about 1 second to finish training a statistical model and takes a negligible amount of time to predict whether a class is important or unimportant. We believe the efficiency of *MCCondenser* is acceptable.

On average, MCCondenser needs about 1 second to build a statistical model and less than 0.1 seconds to predict if a class is important or not, which we believe to be good enough in practice.

RQ5 What is the effect of varying the three tunable parameters of MCCondenser on its performance?

Motivation. As mentioned in Section IV-C, *MCCondenser* has three tunable parameters, i.e. $Base$, $NLearner$ and $NTree$. In this RQ, we want to investigate the effect of varying their default values on the performance of *MCCondenser*.

Approach. In order to answer this question, we perform three sets of experiments. In each set, we only change one single parameter, and keep the other parameters at their default values, to clearly measure the influence of changing the value of an individual parameter to *MCCondenser*'s performance. For example, in the first set of experiments, we fix $NTree$ as 10 and $NLearner$ as 100, and only change the value of $Base$. Following RQ1, we randomly pick 10% of the data for training, and the rest for evaluation. We also use AUC as a yardstick to measure performance.

TABLE XIV. THE EFFECT OF VARYING THE VALUE OF PARAMETER $Base$ TO 2, 5, AND 10, WHEN $NTree$=10 AND $NLearner$=100

Project	**2**	**5**	**10**
ArgoUML	0.6582	0.6618	0.6648
JavaClient	0.8355	0.8582	0.8274
JGAP	0.5533	0.5702	0.5980
JPMC	0.7743	0.6889	0.6426
Mars	0.7764	0.7092	0.7456
Maze	0.5841	0.6066	0.5766
Neuroph	0.8940	0.8448	0.7962
Wro4J	0.6801	0.7700	0.7505
xUML	0.8149	0.8179	0.8319
Average	0.7301	0.7253	0.7148

TABLE XV. THE EFFECT OF VARYING THE VALUE OF PARAMETER $NLearner$ TO 10, 50, AND 100, WHEN $Base$=2 AND $NTree$=10

Project	**10**	**50**	**100**
ArgoUML	0.6275	0.6090	0.6582
JavaClient	0.8248	0.8358	0.8355
JGAP	0.5807	0.5609	0.5533
JPMC	0.6786	0.6969	0.7743
Mars	0.7445	0.7261	0.7764
Maze	0.5966	0.5590	0.5841
Neuroph	0.8007	0.8348	0.8940
Wro4J	0.7687	0.7761	0.6801
xUML	0.8107	0.8177	0.8149
Average	0.7148	0.7129	0.7301

Results. Tables XIV to XV present the effect of modifying each of the three parameters. From these tables, we can conclude several points:

First, for some of the datasets, we can find that setting a different value for one of the three parameters indeed have some influence on the performance. For example, for dataset Neuroph, when $Base$ is fixed as 2 and $NLearner$ is fixed as 100, setting $NTree$ as 10 leads to an AUC of 0.89, while setting it as 50 leads to an AUC of 0.80.

Second, different datasets often have their own sets of optimal parameters. For instance, when $NTree$ is fixed as 10

TABLE XVI. THE EFFECT OF VARYING THE VALUE OF PARAMETER $NTree$ TO 10, 20, AND 50, WHEN $Base$=2 AND $NLearner$=100

Project	**10**	**20**	**50**
ArgoUML	0.6582	0.6436	0.6734
JavaClient	0.8355	0.8434	0.8292
JGAP	0.5533	0.6105	0.5609
JPMC	0.7743	0.5819	0.7077
Mars	0.7764	0.7390	0.7284
Maze	0.5841	0.6541	0.6234
Neuroph	0.8940	0.8993	0.7952
Wro4J	0.6801	0.8170	0.7294
xUML	0.8149	0.7857	0.8090
Average	0.7301	0.7305	0.7174

and $NLearner$ is fixed as 100, among the three settings of $Base$ (i.e., 2, 5, and 10), setting it to 5 leads to the lowest AUC score of 0.71 for dataset Mars, while it leads to the highest AUC score of 0.86 for dataset JavaClient.

All the three parameters have some influence on the performance of MCCondenser. In addition, different datasets have their own sets of optimal parameters.

E. Threats to Validity

Threats to construct validity relate to the suitability of our evaluation metrics. We use AUC which was used by prior approaches that condense class diagrams [2], [3]. AUC is also frequently used in many past software engineering studies as an evaluation metric (e.g., [4], [5], [6]). Thus, we believe there is little threat to construct validity.

Threats to internal validity relate to errors in our experiments. We have double checked our implementations and repeat all the experiments 10 times. Hence, we believe there are minimal threats to internal validity.

Threats to external validity relate to the generalizability of our results. We have evaluated our approach on 2640 classes from nine open source projects. In the future, we plan to reduce this threat further by analyzing more datasets from more open source projects and also commercial software projects.

V. RELATED WORK

In this section, we first highlight some prior works which identify the important classes in subsection V-A. Then, since our work aims to reduce the manual labeling cost, we present several prior works with the same purpose as ours in subsection V-B. At last, we introduce some software engineering studies that also use ensemble learning.

A. Studies Identifying Important Classes

The most related works to ours are the recent studies by Osman et al. [2] and Thung et al. [3]. Osman et al. analyse nine classification algorithms from the machine learning community for identifying important classes in a reverse engineered class diagram [2]. By including only the important classes, they condense the original class diagram. They conclude that the class diagram metrics from the coupling and size categories are good predictors for identifying important classes. They also show that k-nearest neighbour and random forest are the two best performing classification algorithms for the problem. Thung et al. extend Osman et al.'s work by adding a new set of network metrics as extra features for training and propose an approach called optimistic classification to achieve better performance [3]. The optimistic classification technique can deal with data scarcity problem by optimistically assigning labels to some of the unlabeled data and use them for training a better model.

Aside from the two recent works highlighted above, there are also other older studies that also identify important classes [21], [22], [23]. Zaidman et al. propose an approach for identifying key classes based on dynamic coupling and web mining [21]. Steidl et al. use several network metrics, some of them are obtained by running PageRank and HITS algorithms, to identify important classes [22]. The network metrics are calculated from a class dependency graph. Hammad et al. present a method that can assign importance scores to classes based on records in a version control system [23].

In this paper, we have compared our work against the state-of-the-art work that identifies important classes, i.e., [3]. Our experiments demonstrate that our approach outperforms that work by a substantial margin.

B. Studies Reducing Manual Labeling Cost

There are many software engineering studies that aim to minimize the manual labeling cost [10], [24], [25]. We highlight some of them below. Due to the page limit, the survey here is by no means complete.

One of the most recent work is by Thung et al. who propose an active semi-supervised approach for defect prediction with minimal labeled data [10]. The approach actively selects a small subset of diverse and representative defect samples to manually label and build a prediction model based on both labeled and unlabeled defect samples in a semi-supervised way. The biggest advantage of the approach is that it can minimize the manual labeling cost and still achieve a reasonable performance in the meantime. Zhong et al. advocate the use of unsupervised learning techniques to estimate software quality [24]. They first use clustering to group many software modules. And then with the help of human expert, label each cluster as either fault-prone or not based on its representative point. Finally, the unlabeled module is assigned the same label as that of its corresponding cluster.

In this paper, we have compared our work against the state-of-the-art work that reduces manual labeling cost, i.e., [10]. Our experiments demonstrate that our approach outperforms that work by a substantial margin.

C. Studies Leveraging Ensemble Learning

There are a large number of software engineering studies which leverage ensemble learning [26], [27], [28], [29]. Xia et al. propose an approach called ELBlocker to predict blocking bugs (i.e., the bugs that block other bugs from being fixed) by leveraging ensemble learning [26]. Zhang et al. conduct an empirical study of classifier combination for cross-project defect prediction [27]. They investigate seven ensemble learning algorithms and find that several of them perform better than the state-of-the-art approach for cross-project defect prediction namely CODEP. Peng et al. compare experimentally

the performance of several popular ensemble learning methods based on an analytic hierarchy process for software defect prediction [28]. The results show that ensemble methods can in general improve the classification performance for software defect prediction. Zheng proposes a software reliability prediction system based on neural network ensembles [29]. The performance of the system is significantly better than that of a single neural network.

VI. Conclusion and Future Work

In this paper, we propose a novel approach *MCCondenser* which aims to minimize the manual labeling cost for condensing reverse engineered class diagrams. The approach first uses k-means clustering to select the most representative one-tenth of all data as training samples in an unsupervised way, and then uses ensemble learning to handle class imbalance problem and build an ensemble classifier based on the selected training samples. We evaluate our approach on datasets taken from nine open source projects and use a commonly-used evaluation metric AUC. We compare our approach with two state-of-the-art approaches designed for a similar purpose proposed by Thung et al. [3], [10]. The results show that *MCCondenser* can achieve an average AUC score of 0.73, which improves those of the two baselines by nearly 20% and 10% respectively. We have also demonstrated that the two key steps of *MCCondenser*, i.e., k-means clustering and ensemble learning, both contribute to its overall performance. Moreover, our experiments highlight the efficiency of *MCCondenser* that is able to learn a statistical model in around 1 second, and predict if a class is important or not in a fraction of a second.

In the future, we plan to perform experiments on more datasets to reduce the threats to external validity. We also plan to further reduce the manual labeling cost by developing a more advanced approach requiring even less training samples, or even following a completely unsupervised way.

References

[1] A. M. Fernández-Sáez, M. Genero, M. R. V. Chaudron, D. Caivano, and I. Ramos, "Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance?: A family of experiments," *Information & Software Technology*, vol. 57, pp. 644–663, 2015.

[2] M. H. Osman, M. R. Chaudron, and P. Van Der Putten, "An analysis of machine learning algorithms for condensing reverse engineered class diagrams," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 140–149.

[3] F. Thung, D. Lo, M. H. Osman, and M. R. Chaudron, "Condensing class diagrams by analyzing design and network metrics using optimistic classification," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 110–121.

[4] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 1–10.

[5] 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.

[6] D. Romano and M. Pinzger, "Using source code metrics to predict change-prone java interfaces," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 303–312.

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

[8] D. Romano and M. Pinzger, "Using source code metrics to predict change-prone java interfaces," in *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, 2011, pp. 303–312.

[9] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, 2011, pp. 83–92.

[10] F. Thung, X.-B. D. Le, and D. Lo, "Active semi-supervised defect categorization," in *Proceedings of the 23nd International Conference on Program Comprehension*. ACM, 2015, pp. 60–70.

[11] T. Hastie, R. Tibshirani, and J. Friedman, *Unsupervised learning*. Springer, 2009.

[12] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan kaufmann, 2006.

[13] G. Brown, J. L. Wyatt, and P. Tiňo, "Managing diversity in regression ensembles," *The Journal of Machine Learning Research*, vol. 6, pp. 1621–1650, 2005.

[14] E. K. Tang, P. N. Suganthan, and X. Yao, "An analysis of diversity measures," *Machine Learning*, vol. 65, no. 1, pp. 247–271, 2006.

[15] C. Aggarwal, *Data Mining: The Textbook*. Springer International Publishing, 2015.

[16] H. He and E. A. Garcia, "Learning from imbalanced data," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 21, no. 9, pp. 1263–1284, 2009.

[17] 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*. IEEE, 2007, pp. 196–204.

[18] T. M. Khoshgoftaar, X. Yuan, and E. B. Allen, "Balancing misclassification rates in classification-tree models of software quality," *Empirical Software Engineering*, vol. 5, no. 4, pp. 313–330, 2000.

[19] F. J. Provost, T. Fawcett, and R. Kohavi, "The case against accuracy estimation for comparing induction algorithms." in *ICML*, vol. 98, 1998, pp. 445–453.

[20] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[21] A. Zaidman and S. Demeyer, "Automatic identification of key classes in a software system using webmining techniques," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 6, pp. 387–417, 2008.

[22] D. Steidl, B. Hummel, and E. Juergens, "Using network analysis for recommendation of central software classes," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 93–102.

[23] M. Hammad, M. L. Collard, J. Maletic *et al.*, "Measuring class importance in the context of design evolution," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*. IEEE, 2010, pp. 148–151.

[24] S. Zhong, T. M. Khoshgoftaar, and N. Seliya, "Unsupervised learning for expert-based software quality estimation." in *HASE*. Citeseer, 2004, pp. 149–155.

[25] N. Seliya and T. M. Khoshgoftaar, "Software quality analysis of unlabeled program modules with semisupervised clustering," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 37, no. 2, pp. 201–211, 2007.

[26] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "Elblocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, vol. 61, pp. 93–106, 2015.

[27] 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. IEEE, 2015, pp. 264–269.

[28] Y. Peng, G. Kou, G. Wang, W. Wu, and Y. Shi, "Ensemble of software defect predictors: an ahp-based evaluation method," *International Journal of Information Technology & Decision Making*, vol. 10, no. 01, pp. 187–206, 2011.

[29] J. Zheng, "Predicting software reliability with neural network ensembles," *Expert systems with applications*, vol. 36, no. 2, pp. 2116–2122, 2009.