Two Stage Model to Detect and Rank Software Defects on Imbalanced and Scarcity Data Sets

Teerawit Choeikiwong and Peerapon Vateekul

***Abstract*—In the software quality assurance process, it is crucial to prevent defective software to be delivered to customers since it can save the maintenance cost and increase software quality and reliability. Software defect prediction is recognized as an important process to automatically detect the possibility of having an error in the software. After defects are detected, it is then needed to identify their severity levels to avoid any effects that may obstruct the whole system. There were many trials attempts to capture errors by employing traditional supervised learning techniques. However, all of them are often faced with an imbalanced issue and scarcity of data, which causes decreased prediction performance. In this paper, we present a Two-Stage Model to detect and rank defects in software. The model focuses on two tasks. First, we will capture defects by applying an unbiased SVM called "R-SVM," which reduces a bias of the majority class by using the concept of threshold adjustment. Second, the detected modules will be ranked according to their severity levels by using our algorithm called "OS-YATSI," that combines semi-supervised learning and oversampling strategy to tackle the imbalanced issue. The experiment was conducted on 15 Java programs. The result showed that the proposed model outperformed all of the traditional approaches. In the defect prediction model, R-SVM significantly outperformed others on 6 programs in terms of F1. In the defect ranking model, OS-YATSI significantly outperformed all baseline classifiers on all programs at an average of 23.75% improvement in term of macro F1.**

***Index Terms*—software defect prediction, defect severity categorization, imbalanced issue, threshold adjustment, semi-supervised learning**

I. Introduction

SOFTWARE defect is an anomaly in the software. It is also referred to as a bug, fault, or error. It can be found in the source code. It may be a cause of failures to the software that cannot work properly, or does not meet the requirements specifications. As mentioned above, it is obvious that the creation of software products without any defects or bugs is difficult since human is a developer, which can cause the errors.

Software development organizations realize an importance of software production and quality assurance process to achieve the quality software that can respond to customer needs and actually works. However, to acquired quality software that is required a defect prediction, which is a key process in the field of software engineering. It is an attempt to automatically detect errors in the software, which can help developers to fix the bugs and prevent any serious damages to the whole system. Therefore, it is very important to detect all of the defects as early as possible before publishing the software.

Many researchers have been aware of the software defect issue and proposed several defect prediction frameworks by applying traditional supervised learning techniques [1-4], feature selection [5], and sampling strategies [6]. Unfortunately, all of these works showed relatively low prediction performance due to the class imbalanced issue, which is an important factor that tremendously drop prediction performance.

Class imbalanced issue is a major problem in the field of data mining since the technology application is diverse and still growing. Thus, the size of data also increases and it becomes difficult to classify. Imbalanced issue occurs where one of the two classes having more example (majority class) than other classes (minority class). The most of algorithm focuses on classification of majority class, while ignores minority class. Therefore, these classifiers always give better results with the majority class and poor results with minority class. For example, assume the percentage of defected modules is only 10%, while the remaining modules (90%) are non-defected. Although the detection system incorrectly classifies all modules as non-defected ones, the accuracy is still 90%!

Apart from detecting software defects, it is also important to rank them by their severity. Defect severity is a degree of impact that a defect has on the development or operation of a software system. Different defects have different impacts on the software. Some of them may only slow down the process, whiles others may be a cause of failures to the whole system. Therefore, it is important to categorize each defect by their severity levels, which can help developers to prioritize the defects and prevent any serious damages to the whole system.

There were many attempts to automatically classify defect severity. Almost of them required bug report from the user as an input. SEVERIS [7] is a software severity assessment system that utilize a textual description from reported issues. [8-10] applied traditional data mining techniques to predict a severity level from user feedbacks. [11, 12] employed a text mining algorithm along with a feature selection mechanism to select important keywords from bug reports. However, these works relied on the bug description, which means that

Manuscript received July 22, 2016.

Teerawit Choeikiwong is a graduate student with the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, 10330, Thailand (e-mail: Teerawit.Ch@student.chula.ac.th).

Peerapon Vateekul, Ph.D. is a lecturer with the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, 10330, Thailand (e-mail: Peerapon.V@chula.ac.th).

serious damages may already occur. Thus, it should be more efficient to early capture all defects along with their severity levels directly from a software metrics during the software production stage.

At the production stage, the number of defected modules is very small when comparing to the non-defected ones. Moreover, only a small number of defected modules are described along with their severity levels, while most of them do not have it. For example, the Eclipse PDE UI project [13] has 209 defective modules composing of 59 defects (28.23%) and 150 defects (71.77%) of known and unknown severity levels, respectively. Thus, it is not good idea to use a supervised learning algorithm that only relies on labeled data without considering imbalance and scarcity of the data. Furthermore, 46 defects (77.97%) of defects with severity levels are defined as "moderate effects (Level 2)" out of 3 levels. This situation is referred as "imbalanced problem" which is known to tremendously drop prediction performance.

In this paper, we aim to propose a model to detect bugs in software by applying an unbiased support vector machine called "R-SVM," our previous work [14, 15] which reduces a bias from majority class by using threshold adjustment. In addition, we also propose a model to rank the defect modules according to their severity levels by using our algorithm called "OS-YATSI," [16] that combines YATSI [17], a self-training semi-supervised learning algorithm, and SMOTE [18], an oversampling technique. It enhances a prediction performance by utilizing unlabeled data, while amending imbalanced issue all together. The experiment was conducted on 15 Java programs [13, 19] and, then, the result was compared to the original YATSI and several supervised learning techniques: Decision Tree (DT), Naïve Bayes (NB), k-NN and SVM.

The rest of paper is organized as follows. Section II presents an overview of the related work. Section III provides the background knowledge that use in this paper. The detail of the proposed method are presented in Section IV. Section V shows the data sets and the experimental results. Finally, this paper is concluded in Section VI.

II. Related Works

A. Related Works in Defect Prediction

In the field of software defect prediction, there were many trials that apply several machine learning techniques. [5] applied Naïve Bayes for constructing a model to predict software defects. There was an investigation on the feature selection strategy using information gain. The result showed that their system achieved 71 % of the mean probability detection (PD) and 25 % of the mean false alarm rate (PF). [1] introduced a novel algorithm called "GA-CSSVM," that built around SVM and used Genetic Algorithm (GA) to improve the cost sensitive in SVM. The experimental result showed that it reached a promising performance in terms of AUC. [2] proposed an algorithm called "Roughly Balanced Bagging (RBBag)" to predict fault in high assurance software. It employed the bagging concepts into two choices of classifiers: Naïve Bayes and C4.5. The result showed that RBBag model outperformed the classical models without the bagging concept. Moreover, RBBag is more effective when it was applied to Naïve Bayes than C4.5. However, all of these studies discard the imbalanced issue, so their prediction accuracy was limited.

[6] was aware of the imbalanced issue in the software defect prediction. There was an investigation on various approaches to handle the imbalanced issue including threshold moving, ensemble algorithms, and sampling techniques. The result showed that AdaBoost.NC is the winner, and it also outperformed other traditional approaches: NB and Random Forest (RF). Furthermore, a dynamic version of AdaBoost.NC was proposed and proved that it was better than the original one.

Recently, support vector machine (SVM) has been applied in the area of software defect prediction. It is one of the most popular classification techniques and demonstrates a good prediction performance. [3] employed SVM to detect bugs in the MDP data set. The analysis from the SVM results revealed that if a module has a large average of the decision values (SVM scores), there is high chance to found defects in it. [4] compared SVM to eight conventional classifiers, such as Neural Networks, Naïve Bayes, etc., on the MDP data set. The experiment demonstrated that SVM is the winner method. Thus, this is our motivation to apply a method built around SVM called "R-SVM" to detect the software errors.

B. Related Works in Defect Severity Categorization

There are many trials that applying text mining and machine learning techniques in the area of software defect severity prediction. In 2008, [7] proposed a method named SEVERIS (SEVERITY Issue assessment) based on a rule learning algorithm which also utilize the textual descriptions from issue reports. It was experimented on five nameless PITS projects consisting of 775 issue reports with about 79,000 words. By considering the top 100 terms, result showed that the method proposed is a good predictor for issue severity levels. The F-measure values is in the range of 65% - 98% for cases with more than 30 issue reports only.

In 2010, [8] applied Naïve Bayes algorithm to predict severity levels based on textual description of bug reports in binary classes. There was an investigation on the three open-source projects from Bugzilla. The result showed that it obtained a promising performance with precision and recall vary between 0.65-0.75 (Mozilla and Eclipse) and 0.70-0.85 (GNOME). Furthermore, this study has been extended to compare with four traditional classifier such as Naïve Bayes (NB), Naïve Bayes Multinomial (NBM), K-Nearest Neighbor (K-NN) and Support Vector Machines (SVM) [9]. The experiment revealed that Multinomial Naïve Bayes does not only show the highest accuracy, but it is also faster and requires a smaller training set than other classifiers.

In 2012, [11] was aware of the problem of how to find the potential indicators to improve the performance of severity prediction task. There was an investigation on three selection schemes namely Information Gain (IG), Chi-Square (CHI), and Correlation Coefficient (CC) based on the Naïve Bayes classifier. The experiment was conducted on four open-source components from Eclipse and Mozilla. The experimental results showed that the advantage of feature

selection can extract potential indicators and improve the performance of severity prediction. In 2014, [12] introduced an application of bi-grams and feature selection strategy for bug severity classification based on NB classifier. The result demonstrated that bi-grams and Chi-Square feature selection can help to enhance an accuracy of the severity categorization task.

As mentioned above, none of previous studies have ever applied semi-supervised learning approaches to improve a prediction performance by utilizing unlabeled data. Moreover, all of them ignored an imbalanced issue resulting in a prohibited accuracy.

III. Background Knowledge

A. Software Metrics

Measurement is considered as a key element in the software development process. It can helps to estimate the cost, effort, and timing of software development. In addition, It can help developers to know that software development is on target and schedule or not. For building a software, we use numerous software metrics to evaluate quality of software and also define the attribute of software. These software metrics reflect the benefits and one of the main benefits is to provide it provides information for software defect prediction.

Currently, there are many software metrics used for defect prediction in software. In this study, our intention is to point out that size and structure of software are reflect the defect prone in the software. We have studied and collected metrics from many researches [20-24] and use the software size and structure metrics by extracting from the source code with CKJM tool [25]. The details of software metrics used in the experiments as shown in Table 1.

TABLE I
Class Level Software Metrics.

	Metrics	Reference
WMC	Weight Method per Class	
NOC	Number of Children	
CBO	Coupling Between Object classes	C&K [20]
RFC	Response for a class	
LCOM	Lack of Cohesion in Methods	
Ca	Afferent couplings	Martin [21]
Ce	Efferent couplings	
NPM	Number of Public Methods	
DAM	Data Access Metric	
MOA	Measure of Aggregation	QMOOD [22]
MFA	Measure of Functional Abstraction	
CAM	Cohesion Among Methods of Class	
CBM	Coupling Between Methods	Tang [23]

B. Semi-Supervised Learning

Semi-supervised learning (SSL) is a class of machine learning that combines between supervised learning and unsupervised learning. Semi-supervised learning algorithm use both labeled data and unlabeled data for training. This algorithm can improve prediction accuracy by utilizing unlabeled data. In the literature survey [26], traditional semi-supervised learning algorithms are divided into four groups:

Self-training

Self-training is a method commonly used for semi-supervised learning. In this method, a classifier uses a small amount of labeled data for training and generate the prediction model. This model is used to label the unlabeled data. Typically the most confident unlabeled data from the new labeled one are added to the training set. The classifier is retrained and procedure repeated until convergence. This process is also called self-teaching or bootstrapping.

Co-Training

Co-training is a semi-supervised learning algorithm that needs two views of the data. Features are split into two sets and each classifier is trained with one of these sets. Each classifier predicts the labels of unlabeled data and teaches the other classifier with the most confident unlabeled data. After this step, classifiers are retrained and the procedure repeated.

Transductive Support Vector Machines (TSVMs)

Transductive Support Vector Machines (TSVMs) is an extension of traditional support vector machines with unlabeled data. In this method, the unlabeled data is also used. The aim is to label unlabeled data, so that maximum margin is reached on both labeled data and unlabeled data.

Graph-based methods

Graph-based methods define a graph where the nodes are the labeled and unlabeled data in the data set, and the edges represented as the similarity of examples. These methods are non-parametric, discriminative, and also transductive in nature.

As mentioned above, the success of semi-supervised learning depends on underlying assumptions in each model. In this paper, we use the self-training approach which is the most popular semi-supervised learning technique, since it is simple and can be easily applied to almost all existing classifiers.

C. Strategies to Handle Imbalanced Data Sets

To tackle imbalanced issue, a sampling technique has received the most attention and is reported to be the best strategy. These techniques are mainly dividing into two approaches as follows.

Undersampling (US)

Undersampling approach tries to balance between two classes by removing examples in the majority class until the desired class ratio has been achieved. Unfortunately, it is not suitable for small training data and it cannot guarantee to keep all important examples.

Oversampling (OS)

Oversampling approach is an opposite of the undersampling strategy. It helps to improve a balance between classes by replicating examples in the minority class; thus, it is suitable when there is a scarcity issue in the training data. However, a duplication of minority data can cause an overfitting issue, so it is common to generate new

minority examples instead. SMOTE (Synthetic Minority Over-sampling TEchnique) is chosen to use in this work and its details will be shown in Section IV.

D. Prediction Performance Metrics

In the domain of binary classification problem (defect and non-defect), it is necessary to construct a confusion matrix, which comprises of four based quantities: True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) as shown in Table II.

TABLE II
A CONFUSION METRIX.

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

These four values are used to compute Precision (Pr), Probability of Detection (PD), Probability of False Alarm (PF), True Negative Rate (TNR), F-measure [20], and G-mean [27] as shown in Table III.

TABLE III
PREDICTION PERFORMANCE METRICS.

Metrics	Definition	Formula
Precision	a proportion of examples predicted as defective against all of the predicted defective	$\frac{TP}{TP+FN}$
Probability of Detection (PD), Recall, TPR	a proportion of examples correctly predicted as defective against all of the actually defective	$\frac{TP}{TP+FN}$
Probability of False Alarm (PF), FPR	a proportion of examples correctly predicted as non-defective against all of the actually non-defective	$\frac{FP}{TN+FP}$
True Negative Rate (TNR)	a proportion of examples correctly predicted as non-defective against all of the actually non-defective	$\frac{TN}{TN+FP}$
G-mean	the square root of the product of TPR (PD) and TNR	$\sqrt{(TPR)\cdot(TNR)}$
F_β -measure	a weighted harmonic mean of precision and recall	$\frac{2(\mathrm{Pr})(\mathrm{Re})}{\mathrm{Pr}+\mathrm{Re}}$

As mentioned earlier, there are two ways to combine those common measures [28]: macro-averaging and micro-averaging as shown in Table IV. Macro-averaging gives an equal weight to each class, whereas micro-averaging gives an equal weights to each class based on a number of examples. In an imbalanced situation, it is appropriate to use macro-averaging over micro-averaging in order to avoid a dominance of majority classes.

TABLE IV
MACRO-AVERAGING AND MICRO-AVERAGING OF PRECISION, RECALL, AND F_B, I IS A CLASS INDEX.

Metrics	Macro-averaging	Micro-averaging
Precision	$MaPr = \frac{1}{\lvert L\rvert}\sum_{i=1}^{\lvert L\rvert} Pr_i$	$MiPr = \frac{\sum_{i=1}^{\lvert L\rvert} tp_i}{\sum_{i=1}^{\lvert L\rvert}(tp_i + fp_i)}$
Recall	$MaRe = \frac{1}{\lvert L\rvert}\sum_{i=1}^{\lvert L\rvert} Re_i$	$MiRe = \frac{\sum_{i=1}^{\lvert L\rvert} tp_i}{\sum_{i=1}^{\lvert L\rvert}(tp_i + fn_i)}$
F_β-measure	$MaF_\beta = \frac{1}{\lvert L\rvert}\sum_{i=1}^{\lvert L\rvert} F_{\beta,i}$	$MiF_\beta = \frac{(\beta^2+1)\times MiPr \times MiRe}{\beta^2 \times MiPr + MiRe}$

IV. A PROPOSED METHOD

In this section, we demonstrate the details of our proposed which is Two Stage Model. Fig. 1 shows an overview of our model consisting of two stages: (*i*) Defect Prediction and (*ii*) Defect Ranking

In the first stage, the R-SVM model is constructed based on the code features of the data. Then, the model predicts the class with a defect. In the second stage, we used the prediction result with only defective classes as an input to construct a ranking model. It builds around OS-YATSI to prioritize the defects according to their severity levels.

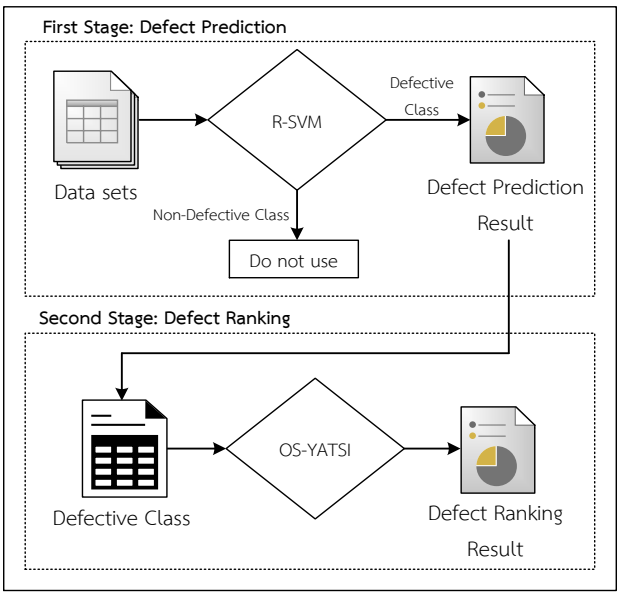


Fig. 1. Overview of our Two Stage Model.

A. Support Vector Machines

Support Vector Machine (SVM) [29] is a standout among the most well-known classification techniques which was introduced by Vapnik. It was shown to be more precise than other classification techniques, especially in the domain of text categorization. It builds a classification model by finding an optimal separating hyperplane as shown in Fig. 2 that maximizes the margin between the two classes. The training samples that lie at the margin of the class distributions in feature space called support vectors.

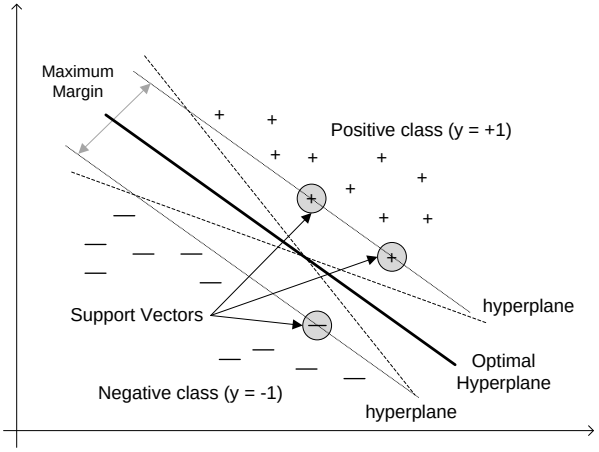


Fig. 2. Optimal separating hyperplane of SVM.

The purpose of SVM is to induce a hyperplane function (Equation 1), where $\vec{w}$ is a weight vector referring to "orientation" and *b* is a bias.

$$h(\vec{w},b) = \vec{w} \cdot x + b \tag{1}$$

Equation (2) shows the optimization function to construct SVM hyperplane, where C is a penalty parameter of error due to misclassifications.

$$\underset{w,b,\xi}{Minimize} \quad \frac{1}{2} w^T w + C\sum_{i=1}^{n} \xi_i$$

$$subject\ to \quad y_i\left(w^T \phi(x_i) + b\right) \geq 1 - \xi_i \quad , \xi_i \geq 0 \tag{2}$$

In a non-linear separable problem, SVM handles this by using a kernel function (non-linear) to map the data into a higher space, where a linear hyperplane cannot be used to do the separation. A kernel function is shown in (3).

$$K(x_i, x_j) \equiv \phi(x_i)\phi(x_j) \tag{3}$$

Unfortunately, although SVM has shown an impressive result, it still suffers from the imbalanced issue like other conventional classification techniques.

B. Threshold Adjustment (R-SVM)

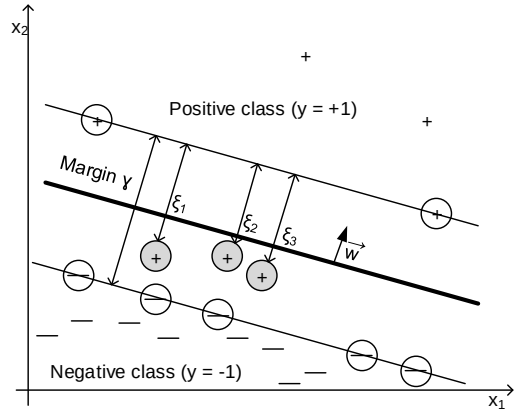
Although SVM has shown a good classification performance in many real-world data sets, it often gives low prediction accuracy in an imbalanced scenario. R-SVM [14] is an our earlier attempt that focuses to tackle this issue by applying the threshold adjustment strategy. To minimize a bias of the majority class, it translates a separation hyperplane in (1) without changing the orientation $\vec{w}$ by only adjusting *b*. After the SVM hyperplane has been induced from the set of training data mapped to SVM scores, *L*. The task is to find a new threshold, θ, that selected from the set of candidates thresholds, Θ, which gives the highest value of a user-defined criterion, *perf* (e.g., the F_1 metric):

$$\left\{\theta \in \Theta \middle| \theta = \max\left(perf\left(L, \Theta\right)\right)\right\} \tag{4}$$

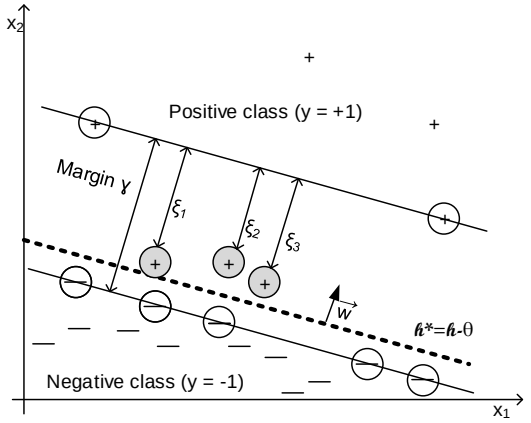
To avoid overfitting issue, the output θ is an average of the thresholds obtained from different training subsets. Finally, the SVM function is corrected as below:

$$h^*(x_i) = h(x_i) - \theta \tag{5}$$

Fig. 3 shows how "shifting" the hyperplane's bias downward in the bottom graph corrects the way SVM labels the three positive examples misclassified by the original hyperplane in the bottom graph (note that the hyperplane's orientation is unchanged).



(a) SVM hyperplanes before threshold adjustment.



(b) SVM hyperplanes after threshold adjustment.

Fig. 3. SVM hyperplanes before (a) and after (b) threshold adjustment. The classification of three examples is thus corrected [14].

C. OS-YATSI

In this section, we demonstrate the details of our proposed defect severity classification called "OS-YATSI". The process of our method consisting of three main modules: (*i*) Oversampling, (*ii*) Semi-supervised Learning, and (*iii*) Unlabeled Selection Criteria as shown in Fig. 4.

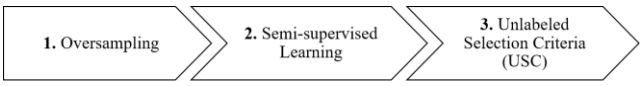


Fig. 4. A process diagram of the proposed method.

1) Oversampling (OS)

This module aims to alleviate a bias from the majority severity level. SMOTE, an oversampling strategy, is chosen since the scarcity of defects. It generates synthetic examples from the minority class following the equation below:

$$x_{new} = x_i + (\hat{x}_i - x)(r) \tag{6}$$

First, *i*-th minority example (x_i) is randomly selected along with its nearest neighbor in the minority class ($\hat{x}_i$).

Second, a new synthetic example (x_{new}) is calculated from the equation (6), where *r* is a random number between 0-1.

Finally, this process repeats until all minority examples are processed and generated their synthetic examples.

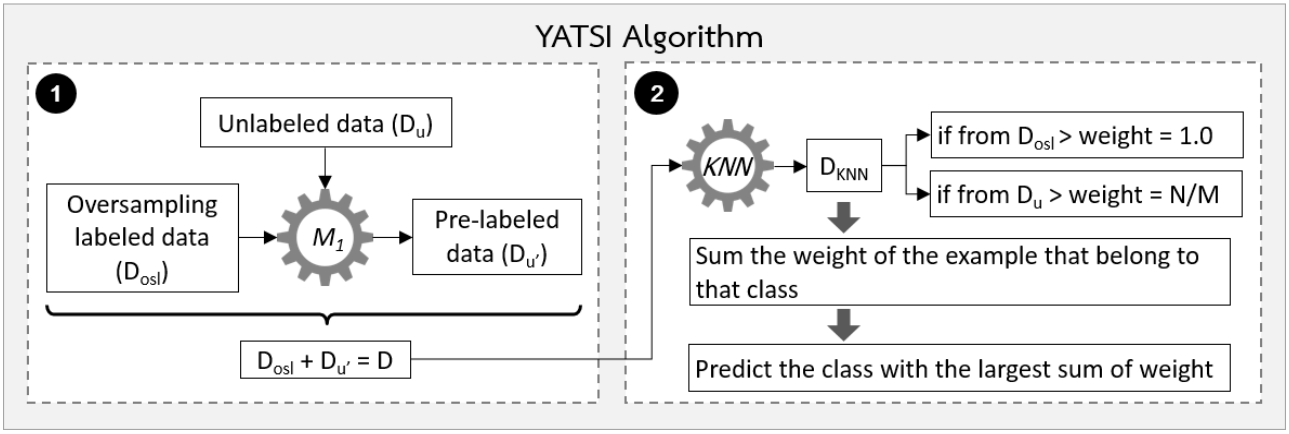


Fig. 5. A procedures of YATSI algorithm.

2) Semi-Supervised Learning

In the bug repositories, some defects have severity levels reported (labeled data), while most of them do not have it (unlabeled data). This process focuses on utilizing unlabeled defects by employing a semi-supervised learning classifier called "Yet Another Two Stage Idea (YATSI)," which consists of two stages as shown in Fig. 5.

In the first stage, an initial classifier is constructed only from the oversampling labeled data from Module 1 (Oversampling). Then, it is used to predict a severity level for each unlabeled data. The output unlabeled data with predicted severity are called "pre-labeled data."

In the second stage, the nearest neighbor algorithm is applied on a merged data set between the labeled and pre-labeled data to determine a predicted severity level of the unlabeled data. A weighting strategy is referred to as the amount of trust. It is applied to a distance during the process of finding a neighbor. As a default value, the weight of the labeled data is set to 1, while the weight of the unlabeled data is equal to $F \times (N/M)$, where N and M denote the number of labeled and unlabeled data, consecutively, and F denotes a user-defined parameter between 0 and 1 showing a trust on the unlabeled data.

For the last stage, all unlabeled data are assigned to their *actual* severity level. The *k*-nearest neighbor is employed. It predicts the level that gives the largest total weighting score.

3) Unlabeled Selection Criteria (USC)

After Module 2 (SSL), all unlabeled defects are already annotated and have their severity level, so an enhanced training data can be created by combining between the labeled and unlabeled data.

For the labeled data, we choose the oversampling data from Module 1 (Oversampling) to avoid the imbalanced issue. For the unlabeled data, the traditional semi-supervised classifier usually uses all of them without concerning the imbalanced issue. However, the preliminary experiment showed that there is still an imbalanced issue in the unlabeled data.

Therefore, this module called "USC" is proposed as a criteria to select examples in the unlabeled data set to include in the training data set while maintaining the balance of data for each severity level as summarized below:

1. Find the class with the smallest amount of example (also called minority class) and add all examples in that class to the training data.
2. Select examples for each severity level equally to those in the minority class by their prediction score from module 2 (Semi-supervised Learning)

From Fig. 6, we illustrate examples of using the USC. First, we founded that the low severity level with 5 examples are minority class. Then, we add all examples of these classes to the training data. Finally, we select examples for each remaining classes (medium and high) equally to those minority class by their confidence value and add all examples of them to the training data.

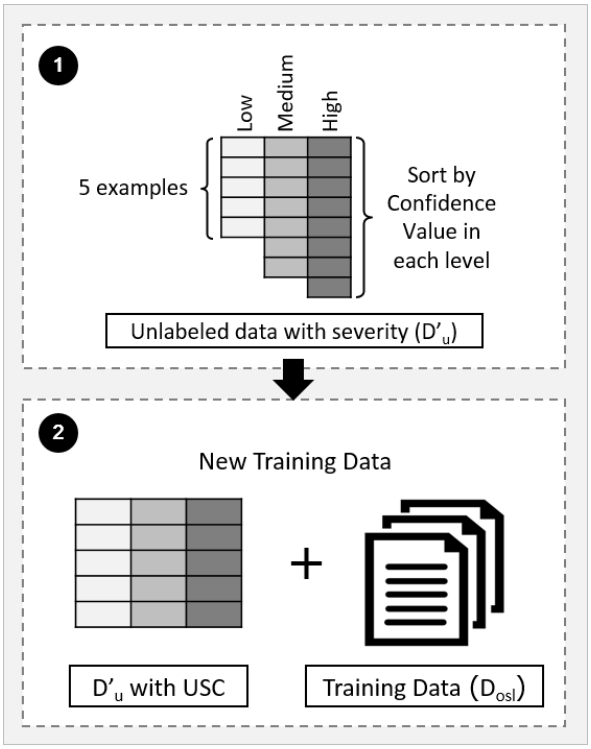


Fig. 6. A process of the Unlabeled Selection Criteria.

The pseudo code for OS-YATSI is shown below.

Algorithm	Pseudo code for OS-YATSI algorithm
Input:	A set of label $L = \{l_1, l_2, l_3\}$, classifier C, labeled data D_l, unlabeled data D_u, oversampling ratio R_{os}, oversampling labeled data D_{osl}, number of nearest neighbors K, $N = \|D_{osl}\|$, $M = \|D_u\|$, unlabeled data example d_u

Step1:	Find the majority class l_M with $\|D_M\|$ examples in the labeled data D_l
	Create a set of minority classes L_m that excludes the majority class l_M
	While(L_m is not empty)
	Find the class l_M in L_m with the least number of examples, $\|D_m\|$
	Compute the number of examples $\|D'_m\|$ if oversampling using SMOTE with R_{os}
	If (Diff($\|D'_m\|$, $\|D_M\|$) < Diff($\|D_m\|$, $\|D_M\|$)) Then
	Oversampling the class using SMOTE with R_{os}
	Add the new oversampled example into D_{osl}
	Else
	Remove class l_M from a set of classes L_m

Step2:	Use the classifier C to construct the initial model M_1 by using D_{osl}
	Use the M_1 to "pre-label" all the examples of D_u
	For(i=1 to N)
	Weight = 1.0
	For(j=1 to M)
	Weight = $(N/M) * WeightFactor\ F$
	Combine D_{osl} and D_u to generate D
	For every example in D_u
	Find the K-nearest neighbors to the example from D to produce set D_{kNN}
	For i=1 to K
	If(class of D_{kNN} = 1) sum weight1 of D_{kNN}
	If(class of D_{kNN} = 2) sum weight2 of D_{kNN}
	If(class of D_{kNN} = 3) sum weight 3of D_{kNN}
	Predict the actual class with the largest total weighting score

Step3:	For unlabeled data D_u Find the class with smallest amount of example and produce set C_{small} For another class Select examples equally to C_{small} with their prediction score and produce set $C_{balance}$ Merge C_{small} and $C_{balance}$ to produce balance unlabeled data D'_u

V. Experiments and Results

A. Data Sets

We use the public benchmark presented in [13, 19], which contains a metrics that describe software artifacts from 15 open-source java projects. The data set statistics is shown in Table V. From the statistics, it has shown that the public benchmark suffers from the imbalanced issue and scarcity of the data. An average percentage of the severity class is 28.51%, and the lowest percentage is only 10.19% in Synapse.

In addition, we select the three java projects to use in the ranking model. There are enough training examples to predict the defect severity, which comprises of three severity levels: Low, Medium, and High. The severity statistics is shown in Table VI.

TABLE V
Defect Statistics for Each Data Set.

Data	#classes	Defect Class	Non-Defect Class	%Defect
Ant	125	20	105	16.00%
Camel	608	216	392	35.53%
Ivy	352	40	312	11.36%
Jedit	272	90	182	33.09%
Log4j	135	34	101	25.19%
Lucene	195	91	104	46.67%
Pbeans	26	20	6	76.92%
Poi	237	141	96	59.49%
Synapse	157	16	141	10.19%
Velocity	229	78	151	34.06%
Xalan	723	110	613	15.21%
Xerces	440	71	369	16.14%
Eclipse JDT Core	997	206	791	20.66%
Eclipse PDE UI	1,497	209	1,288	13.96%
Mylyn	1,862	245	1,617	13.16%
Average	7,855	1,587	6,268	28.51%

TABLE VI
Severity Statistics for Each Data Set.

Data	#classes	Severity (Sev.) levels				%Sev.
		lv. 1	lv. 2	lv. 3	N/A	
Eclipse JDT Core	206	12	19	10	165	19.90%
Eclipse PDE UI	209	7	46	6	150	28.23%
Mylyn	245	127	15	3	100	59.18%
Average	220	48.67	26.67	6.33	138.33	35.77%

B. Experimental Setup

In this section, show how to conduct the experiments in this paper. We divided the experiments into two parts: (*i*) software defect prediction and (*ii*) software defect severity ranking. It starts from the data preprocessing including numeric-to-nominal conversion and scaling [30] all values into a range of [0, 1]. Then, we compare the prediction performance among different approaches, which can explained the details of each experiment as the following steps. Note that all experiments are based on 10-fold cross validation.

1) Software Defect Prediction

- Step1: find the baseline method which is the winner of the traditional classifiers: Decision Tree (DT), Naïve Bayes (NB), k-NN and SVM
- Step2: compare R-SVM to the baseline method (Step1) along with a significance test using unpaired t-test at a confidence level of 95%

2) Software Defect Severity Ranking

- Step1: find the best setting of the winner method from the first experiment (software defect prediction) whether or not the oversampling is necessary to construct an initial model.
- Step2: find the best setting for OS-YATSI whether or not USC is necessary. Then, compare OS-YATSI to the baselines (Step1) and Original YATSI along with a significance test using unpaired t-test at a confidence level of 95%

C. Results and Discussion

In this section, we first compare the performance of the software defect prediction model and the two existing software defect severity ranking models. The result will demonstrate that our method can give the best performance.

In addition, we summarized the data characteristics for each methods in the experiments to make it more understandable as shown in Table VII.

TABLE VII
DATA CHARACTERISTICS OF EACH METHODS IN THE EXPERIMENTS.

Method Name	Data Characteristics			
	Training Data		Unlabeled Data	
	Original	Over sampling	Without USC	With USC
Baseline	✓			
Original (OG)	✓			
Oversampling (OS)		✓		
YATSI	✓		✓	
OS-YATSI w/o USC		✓	✓	
OS-YATSI w/ USC		✓		✓

1) Results of Software Defect Prediction

The comparison of the baseline methods. In order to get the baseline for each data set, four classifiers: Decision Tree, Naïve Bayes, k-NN, and SVM, were tested and compared in terms of PD, PF, F1, and G-Mean (Tables VIII – XI). For each row in the tables, the boldface method is a winner on that data set. From the result, DT and k-NN showed the best performance in almost all data sets, while others gave the moderate and low performance on some data set, especially for the Velocity, NB gave the worst performance for 35.30% For Table X – XI, it is interesting that F1 and G-mean unanimously showed the same winners. Since F1 and G-mean are suitable metrics for imbalanced data, we selected the winner as a baseline using F1 and G-mean as summarized in Table XII.

TABLE VIII
PREDICTION PERFORMANCE: PD

Data	Prediction model			
	DT	*k-NN*	*NB*	*SVM*
Ant	**0.857**	0.747	0.774	0.765
Camel	0.684	0.646	**0.749**	**0.749**
Ivy	**0.891**	0.878	0.492	0.621
Jedit	0.803	0.853	**0.863**	0.814
Log4j	0.751	0.773	**0.922**	0.843
Lucene	0.601	0.786	**0.875**	0.731
Pbeans	**0.900**	**0.900**	0.700	0.850
Poi	0.702	0.680	0.283	**0.794**
Synapse	**0.809**	0.780	0.687	0.808
Velocity	0.735	**0.740**	0.252	0.714
Xalan	0.825	0.829	**0.876**	0.744
Xerces	**0.824**	0.718	0.743	0.582
Eclipse JDT Core	0.836	0.855	**0.946**	0.881
Eclipse PDE UI	0.866	0.768	**0.925**	0.823
Mylyn	0.866	0.799	**0.911**	0.832
Avg.	0.797	0.783	0.733	0.770
SD	0.085	0.072	0.224	0.084

TABLE IX
PREDICTION PERFORMANCE: PF

Data	Prediction model			
	DT	*k-NN*	*NB*	*SVM*
Ant	0.161	**0.066**	0.181	0.134
Camel	**0.321**	0.365	0.755	0.646
Ivy	0.154	0.195	0.150	**0.138**
Jedit	**0.210**	0.243	0.448	0.232
Log4j	0.228	**0.138**	0.406	0.286
Lucene	**0.301**	0.344	0.541	0.313
Pbeans	0.050	0.300	**0.000**	0.200
Poi	0.270	0.164	**0.086**	0.363
Synapse	0.114	**0.100**	0.200	0.157
Velocity	0.238	0.272	**0.112**	0.318
Xalan	**0.152**	0.180	0.618	0.277
Xerces	**0.100**	0.144	0.575	0.348
Eclipse JDT Core	0.169	**0.154**	0.643	0.402
Eclipse PDE UI	0.143	**0.120**	0.725	0.492
Mylyn	0.135	**0.129**	0.676	0.449
Avg.	0.183	0.194	0.408	0.317
SD	0.077	0.091	0.262	0.142

TABLE X
PREDICTION PERFORMANCE: F1

Data	Prediction model			
	DT	*k-NN*	*NB*	*SVM*
Ant	0.790	**0.816**	0.784	0.799
Camel	0.608	**0.642**	0.629	0.619
Ivy	**0.870**	0.846	0.596	0.703
Jedit	0.798	**0.814**	0.748	0.795
Log4j	0.756	**0.807**	0.793	0.793
Lucene	0.623	**0.742**	0.725	0.719
Pbeans	0.583	**0.837**	0.767	0.827
Poi	0.706	0.704	0.405	**0.710**
Synapse	0.816	**0.826**	0.725	0.820
Velocity	0.682	**0.720**	0.355	0.699
Xalan	**0.734**	0.726	0.702	**0.734**
Xerces	**0.856**	0.771	0.639	0.602
Eclipse JDT Core	0.833	**0.851**	0.731	0.772
Eclipse PDE UI	**0.862**	0.813	0.698	0.711
Mylyn	**0.742**	0.729	0.704	0.730
Avg.	0.751	0.776	0.667	0.736
SD	0.094	0.062	0.129	0.067

TABLE XI
PREDICTION PERFORMANCE: G-MEAN

Data	Prediction model			
	DT	*k-NN*	*NB*	*SVM*
Ant	0.814	**0.905**	0.881	0.893
Camel	0.776	**0.783**	0.616	0.714
Ivy	**0.931**	0.914	0.788	0.847
Jedit	0.806	**0.832**	0.817	0.827
Log4j	0.855	**0.857**	0.850	0.827
Lucene	0.580	**0.783**	0.705	0.718
Pbeans	0.785	**0.951**	0.858	0.867
Poi	0.819	0.851	0.749	**0.827**
Synapse	0.897	**0.900**	0.819	0.898
Velocity	0.856	**0.873**	0.867	0.816
Xalan	**0.913**	0.907	0.743	0.851
Xerces	**0.927**	0.882	0.797	0.762
Eclipse JDT Core	0.812	**0.892**	0.733	0.848
Eclipse PDE UI	**0.828**	0.806	0.702	0.801
Mylyn	**0.730**	0.713	0.725	0.720
Avg.	0.822	0.857	0.777	0.814
SD	0.089	0.063	0.075	0.060

TABLE XII
THE WINNER OF THE BASELINE METHOD FOR EACH DATA SET IN TERMS OF F1 AND G-MEAN.

Project	Winner Method	F1	G-mean
Ant	*k-NN*	0.816	0.905
Camel	*k-NN*	0.642	0.783
Ivy	*DT*	0.870	0.931
Jedit	*k-NN*	0.814	0.832
Log4j	*k-NN*	0.807	0.857
Lucene	*k-NN*	0.742	0.783
Pbeans	*k-NN*	0.837	0.951
Poi	*SVM*	0.710	0.827
Synapse	*k-NN*	0.826	0.900
Velocity	*k-NN*	0.720	0.873
Xalan	*DT*	0.734	0.913
Xerces	*DT*	0.856	0.927
Eclipse JDT Core	*k-NN*	0.851	0.892
Eclipse PDE UI	*DT*	0.862	0.828
Mylyn	*DT*	0.742	0.730
Avg.	-	0.789	0.862
SD	-	0.068	0.064

The comparison of R-SVM and the baseline methods. In this section, we compare R-SVM to the baseline methods, which are obtain from the previous experiment as shown in Tables VIII-XI. In Table XIII shows a comparison in terms of PD, PF, F1, and G-mean. All of the performance metrics give the same conclusion that R-SVM outperforms the baseline methods in almost all of the data sets. From 15 data sets, R-SVM *significantly* won 5, 6, and 5 on PD, F1, and G-mean, respectively.

For more details about the overall performance, Fig. 7 shows the number of data sets that each method is the winner. R-SVM outperformed the baseline methods on 10 data sets. Furthermore, R-SVM on both F1 and G-mean outperformed others on 8 data sets and achieved an average F1 at 0.750. Thus, this demonstrates that it is effective to apply R-SVM as a core module for early detect imperfect software system.

TABLE XIII
COMPARISON PREDICTION PERFORMANCE MEASURES BETWEEN R-SVM AND THE BASELINE METHOD FROM TABLE XII. THE BOLDFACE METHOD IS A WINNER ON THAT DATA SET.

Project	PD		PF		F1		G-mean	
	Baseline	R-SVM	Baseline	R-SVM	Baseline	R-SVM	Baseline	R-SVM
Ant	0.857	**0.902**	**0.066****	0.275	0.816	**0.832****	**0.905**	0.886
Camel	0.749	**0.905***	**0.321**	0.471	0.642	**0.650**	0.783	**0.804****
Ivy	**0.891**	0.831	**0.138****	0.313	**0.870****	0.796	**0.931**	0.864
Jedit	**0.863**	0.840	**0.210**	0.224	**0.814**	0.785	0.832	**0.878****
Log4j	0.922	**0.983***	**0.138***	0.297	**0.807**	0.775	0.857	**0.860**
Lucene	**0.875**	0.941	**0.301**	0.334	**0.742**	0.692	0.783	**0.806***
Pbeans	0.900	**1.000**	**0.000**	0.003	0.837	**0.893**	0.951	**0.952**
Poi	0.794	**0.811**	**0.086***	0.279	0.710	**0.714***	0.827	**0.943****
Synapse	0.809	**0.875**	**0.100****	0.211	0.826	**0.836****	0.900	**0.906**
Velocity	0.740	**0.784**	0.112	**0.041***	0.720	**0.731****	**0.873**	0.837
Xalan	**0.876**	0.843	**0.152****	0.385	0.734	**0.757****	**0.913****	0.845
Xerces	0.824	**0.858***	**0.100**	0.121	**0.856****	0.703	**0.927***	0.874
Eclipse JDT Core	0.946	**0.990****	**0.154****	0.372	**0.851****	0.744	**0.892***	0.840
Eclipse PDE UI	**0.925**	0.914	**0.120****	0.627	**0.862****	0.716	**0.828***	0.744
Mylyn	0.911	**0.954****	**0.129****	0.535	0.742	**0.767****	0.730	**0.805****
Avg.	0.859	0.874	0.174	0.318	0.733	0.750	0.852	0.861
SD	0.063	0.066	0.125	0.161	0.061	0.072	0.069	0.054

* and ** represent a significant difference at a confidence level of 95% and 99%, respectively.

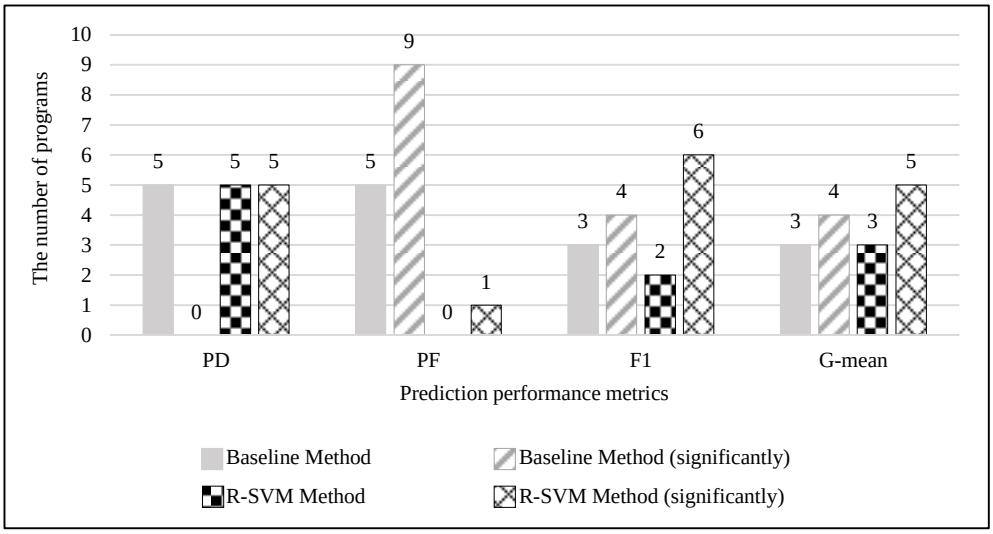


Fig. 7. The number of wom data sets on R-SVM and the baseline methods in terms of PD, PF, F1 and G-mean.

TABLE XIV
COMPARISON PREDICTION PERFORMANCE MEASURES OF THE CLASSICAL CLASSIFIERS.
THE BOLDFACE METHOD IS A WINNER ON THAT DATA SET.

Precision

Data	DT		k-NN		NB		SVM		R-SVM	
	Macro	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Micro*
JDT Core	0.330	0.445	0.302	0.317	**0.419**	**0.419**	0.186	0.186	**0.377**	**0.488**
PDE UI	**0.263**	**0.630**	0.261	0.679	0.255	0.255	0.258	0.258	**0.323**	**0.761**
Mylyn	0.291	0.855	**0.361**	**0.758**	0.318	0.318	0.292	0.292	0.338	0.759
Avg.	0.295	0.643	0.308	0.585	0.331	0.331	0.245	0.245	0.346	0.669
SD	0.034	0.205	0.050	0.235	0.083	0.083	0.054	0.054	0.028	0.157

Recall

Data	DT		k-NN		NB		SVM		R-SVM	
	Macro	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Micro*
JDT Core	**0.393**	**0.445**	0.272	0.317	0.356	0.356	0.329	0.329	**0.355**	**0.488**
PDE UI	0.318	0.630	0.290	0.679	0.233	0.233	**0.326**	**0.326**	0.299	0.761
Mylyn	0.325	0.855	**0.347**	**0.758**	0.323	0.323	0.333	0.333	**0.348**	**0.759**
Avg.	0.345	0.643	0.303	0.585	0.304	0.304	0.329	0.329	0.334	0.669
SD	0.041	0.205	0.039	0.235	0.064	0.064	0.004	0.004	0.031	0.157

F1-measure

Data	DT		k-NN		NB		SVM		R-SVM	
	Macro	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Micro*
JDT Core	0.349	0.445	0.280	0.317	**0.350**	**0.350**	0.230	0.230	**0.336**	**0.488**
PDE UI	0.275	0.630	0.275	0.679	0.241	0.241	**0.288**	**0.288**	**0.296**	**0.761**
Mylyn	0.307	0.855	**0.351**	**0.758**	0.315	0.315	0.311	0.311	0.341	0.759
Avg.	0.310	0.643	0.302	0.585	0.302	0.302	0.276	0.276	0.324	0.669
SD	0.037	0.205	0.043	0.235	0.056	0.056	0.042	0.042	0.025	0.157

2) Results of Software Defect Severity Ranking

The comparison of the baseline. In order to get the baseline methods for each data set, four classifiers: DT, NB, k-NN, and SVM were tested and compared in terms of Pr, Re, and F1 (Table XIV). For each row in the table, the boldface method is a winner on that data set. From the result, R-SVM showed the best performance in terms of macro and micro-average on JDT Core and PDE UI, while the Mylyn data has been effective from k-NN method, which are not significantly better than R-SVM. Hence, we selected the R-SVM on both macro and micro-average as a baseline methods in terms of F1 as summarized in Table XV to construct an initial model.

The comparison of an initial model with and without oversampling. In this section, we aim to give the best setting of an initial model by testing whether or not the oversampling training data can deal with the imbalanced issue and improve the prediction performance. We use R-SVM as a predictor to build an initial model. For each row in the tables, the boldface method is a winner on that data set. The results in Table XVI demonstrate that the R-SVM with OS (oversampling) performs better than without OS (original) all data sets both macro and micro-average. The results imply that the oversampling strategy is suitable when there is a scarcity in the training data since it can helps to improve the prediction performance.

TABLE XV
THE WINNER OF THE BASELINE METHOD FOR EACH DATA SET IN TERMS OF F1-MEASURE.

Data	Winner	F1	
		Macro	*Micro*
JDT Core	R-SVM	0.336	0.488
PDE UI	R-SVM	0.296	0.761
Mylyn	R-SVM, k-NN	0.341	0.759
Avg.	-	0.324	0.669
SD	-	0.025	0.157

TABLE XVI
A COMPARISON OF AN INITIAL MODEL BETWEEN WITH AND WITHOUT OVERSAMPLING IN TERMS OF F1-MEASURE.

Data	OG (original)		OS (oversampling)	
	Macro	*Micro*	*Macro*	*Micro*
JDT Core	0.459	0.491	**0.484**	**0.513**
PDE UI	0.290	0.629	**0.430**	**0.746**
Mylyn	0.349	0.800	**0.361**	**0.807**
Avg.	0.366	0.640	0.425	0.689
SD	0.086	0.155	0.062	0.155

The comparison of OS-YATSI, YATSI and baseline methods. In this section, we compare OS-YATSI to the R-SVM with the oversampling training data (OS) which are obtain from the Table XVI. Furthermore, we also compare to the original YATSI as well. In addition, we aim to give the best setting of OS-YATSI by testing whether or not the Unlabeled Selection Criteria can handle the imbalanced issue and improve the prediction performance.

In Table XVII shows a comparison in terms of Pr, Re, and F1 for both macro and micro-average. All of the measures give the same conclusion that OS-YATSI both with and without USC outperforms the other method in almost all of the data sets. Moreover, OS-YATSI with USC is the best setting since it performs better than without USC all data sets, which indicate that all the unlabeled data is not always enhance the performance and it may be reduced as well.

In *macro-average*, OS-YATSI with USC *significantly* won 3, 2, and 3 on Pr, Re, and F1, respectively. On average, *macro F1* of OS-YATSI with USC outperforms that of the OS for 23.75%, especially for the JDT Core data set showing 50.57% improvement. Furthermore, it also outperforms the original YATSI for 55.60% on JDT Core data set. Consequently, this demonstrates that it is effective to apply OS-YATSI with USC as a main mechanism of software defect severity categorization.

TABLE XVII
COMPARISON PREDICTION PERFORMANCE MEASURES OF OS, OS-YATSI WITH AND WITHOUT USC, AND YATSI. THE BOLDFACE METHOD IS A WINNER ON THAT DATASET.

Precision

Data	OS		YATSI		OS-YATSI w/o USC		OS-YATSI w/ USC	
	Macro	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*
JDT Core	0.530	0.526	0.510	0.509	0.759	0.777	**0.794****	**0.791**
PDE UI	0.763	0.761	0.759	0.725	0.737	0.813	**0.853***	**0.853***
Mylyn	0.769	0.764	0.693	0.693	0.776	0.820	**0.880****	**0.873****
Avg.	0.687	0.684	0.654	0.642	0.757	0.803	0.842	0.839
SD	0.136	0.137	0.129	0.117	0.020	0.023	0.044	0.043

Recall

Data	OS		YATSI		OS-YATSI w/o USC		OS-YATSI w/ USC	
	Macro	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*
JDT Core	0.526	0.526	0.509	0.509	0.763	0.777	**0.791**	**0.791**
PDE UI	0.761	0.761	0.725	0.725	0.716	0.813	**0.853***	**0.853***
Mylyn	0.764	0.764	0.693	0.693	0.772	0.820	**0.873****	**0.873****
Avg.	0.684	0.684	0.642	0.642	0.750	0.803	0.839	0.839
SD	0.137	0.137	0.117	0.117	0.030	0.023	0.043	0.043

F1-measure

Data	OS		YATSI		OS-YATSI w/o USC		OS-YATSI w/ USC	
	Macro	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*	*Macro*
JDT Core	0.526	0.526	0.509	0.509	0.761	0.777	**0.792****	**0.791**
PDE UI	0.758	0.761	0.718	0.725	0.724	0.813	**0.853***	**0.853***
Mylyn	0.749	0.764	0.681	0.693	0.774	0.820	**0.871****	**0.873****
Avg.	0.678	0.684	0.636	0.642	0.753	0.803	0.839	0.839
SD	0.131	0.137	0.112	0.117	0.026	0.023	0.041	0.043

* and ** represent a significant difference at a confidence level of 95% and 99%, respectively.

VI. CONCLUSION

Software defect prediction is a vital part of software development. It is crucial to identify the severity levels after bugs are detected. Unfortunately, the overall performance of the existing techniques are still not satisfied due to two major problems. First, the scarcity of defects that have a small number of labeled data, while the remaining are left unlabeled. Second, some severity levels have defects much larger than others causing an imbalanced issue.

In this paper, we presented the two stage models by incorporating R-SVM, semi-supervised learning, and oversampling strategy. There are two modules in the system: (*i*) defect prediction and (*ii*) defect severity ranking. First, the R-SVM classifier, our version of SVM tailored to domains with imbalanced classes, is created to predict the defective class in the software system. It reduces a bias of the majority class by using threshold adjustment concept to adjust the separation hyperplane. Second, the defected classes are identified severity levels by using our algorithm called "OS-YATSI." It employs semi-supervised learning to fully utilize both labeled and unlabeled data and oversampling defects in the minority class to alleviate the imbalanced issue.

In the experiment, we divided the experiments into two parts: (*i*) software defect prediction and (*ii*) software defect severity ranking. The experiment was conducted on 15 java projects. In the software defect prediction, R-SVM was compared to four conventional classifiers: Decision Tree, Naïve Bayes, k-NN, and SVM. The result showed that R-SVM enhanced the correct classification of the minority class and overcame the imbalanced issue. In the software defect severity ranking, we compared OS-YATSI to the same four conventional classifiers in the first experiment and also compared to original YATSI as well. Experimental results revealed that OS-YATSI with USC significantly surpassed all baselines on all data sets in terms of *macro F1*.

In the future, we plan to study other software defect repositories. In addition, it is necessary to investigate further algorithms to deal with the imbalanced and scarcity data.

REFERENCE

[1] S. Bo, L. Haifeng, L. Mengjun, Z. Quan, and T. Chaojing, "Software defect prediction using dynamic support vector machine," presented at the In Proceeding of the 9th International Conference on Computational Intelligence and Security, 2013.

[2] N. Seliya, T. M. Khoshgoftaar, and J. Van Hulse, "Predicting Faults in High Assurance Software," in *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, 2010, pp. 26-34.

[3] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Software defect prediction using static code metrics underestimates defect-proneness," in *Neural Networks (IJCNN), The 2010 International Joint Conference on*, 2010, pp. 1-7.

[4] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, vol. 81, pp. 649-660, 2008.

[5] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," presented at the IEEE Transactions on Software Engineering, 2007.

[6] W. Shuo and Y. Xin, "Using Class Imbalance Learning for Software Defect Prediction," *Reliability, IEEE Transactions on*, vol. 62, pp. 434-443, 2013.

[7] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, 2008, pp. 346-355.

[8] 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*, 2010, pp. 1-10.

[9] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing Mining Algorithms for Predicting the Severity of a Reported Bug," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, 2011, pp. 249-258.
[10] K. K. Chaturvedi and V. B. Singh, "Determining Bug severity using machine learning techniques," in *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on*, 2012, pp. 1-6.
[11] Y. Cheng-Zen, H. Chun-Chi, K. Wei-Chen, and C. Ing-Xiang, "An Empirical Study on Improving Severity Prediction of Defect Reports Using Feature Selection," in *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, 2012, pp. 240-249.
[12] N. K. Singha Roy and B. Rossi, "Towards an Improvement of Bug Severity Classification," in *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, 2014, pp. 269-276.
[13] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, 2010, pp. 31-41.
[14] P. Vateekul, S. Dendamrongvit, and M. Kubat, "Improving SVM Performance in Multi-Label Domains: Threshold Adjustment," *International Journal on Artificial Intelligence Tools*, 2013.
[15] T. Choeikiwong and P. Vateekul, "Software Defect Prediction in Imbalanced Data Sets Using Unbiased Support Vector Machine," in *Information Science and Applications*, J. K. Kim, Ed., ed Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 923-931.
[16] T. Choeikiwong and P. Vateekul, "Improve Accuracy of Defect Severity Categorization Using Semi-Supervised Approach on Imbalanced Data Sets," *Lecture Notes in Engineering and Computer Science: Proceedings of The International MultiConference of Engineers and Computer Scientists 2016*, IMECS 2016, 16-18 March, 2016, Hong Kong, pp. 434-438.
[17] K. Driessens, P. Reutemann, B. Pfahringer, and C. Leschi, "Using Weighted Nearest Neighbor to Benefit from Unlabeled Data," in *Advances in Knowledge Discovery and Data Mining*. vol. 3918, W.-K. Ng, M. Kitsuregawa, J. Li, and K. Chang, Eds., ed: Springer Berlin Heidelberg, 2006, pp. 60-69.
[18] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *J. Artif. Int. Res.*, vol. 16, pp. 321-357, 2002.
[19] T. Menzies, Krishna, R., Pryor, D., "The Promise Repository of Empirical Software Engineering Data," ed, 2015.
[20] S. R. Chidamber and C. F. Kemerer, "A metrics suit for object oriented design," presented at the IEEE Transactions on Software Engineering, 1994.
[21] R. Martin, "OO Design Quality Metrics - An Analysis of Dependencies," presented at the Proc. of Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94, 1994.
[22] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, pp. 4-17, 2002.
[23] T. Mei-Huei, K. Ming-Hung, and C. Mei-Hwa, "An empirical study on object-oriented metrics," in *Software Metrics Symposium, 1999. Proceedings. Sixth International*, 1999, pp. 242-249.
[24] T. J. McCabe, "A complexity measure," presented at the IEEE Transactions on Software Engineering, 1976.
[25] M. Jureczko and D. Spinellis, "Using Object-Oriented Design Metrics to Predict Software Defects," in *Models and Methodology of System Dependability. Proceedings of 2010: Fifth International Conference on Dependability of Computer Systems*, Poland, 2010, pp. 69-81.
[26] X. Zhu, "SemiSupervised classification learning survey," Computer Sciences TR 1530 , University of Wisconsin-Madison 1530, Dec. 2005 2005.
[27] M. Kubat and S. Matwin, "Addressing the curse of imbalanced training sets: one-sided selection," presented at the Proc. 14th International Conference on Machine Learning, 1997.
[28] Y. Yang, "An Evaluation of Statistical Approaches to Text Categorization," *Inf. Retr.*, vol. 1, pp. 69-90, 1999.
[29] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, 2 ed. s.l.: Morgan Kaufmann, 2006.
[30] H. Chih-Wei, C. Chih-Chung, and L. Chih-Jen, *A Practical Guide to Support Vector Classification*. National Taiwan University: Department of Computer Science, 2003.