Alexandria Engineering Journal (2018) 57, 2763–2774

HOSTED BY

Alexandria University

Alexandria Engineering Journal

www.elsevier.com/locate/aej
www.sciencedirect.com

ORIGINAL ARTICLE

Software bug prediction using weighted majority voting techniques

Sammar Moustafa, Mustafa Y. ElNainay *, Nagwa El Makky, Mohamed S. Abougabal

Computer and Systems Engineering Department, Alexandria University, Egypt

Received 5 September 2017; revised 12 December 2017; accepted 15 January 2018
Available online 15 November 2018

KEYWORDS

Modeling and prediction;
Product metrics;
Process metrics;
Classifier design and evaluation

Abstract Mining software repositories is a growing research field where rich data available in the different development software repositories, are analyzed and cross-linked to uncover useful information. Bug prediction is one of the potential benefits that can be gained through mining software repositories. Predicting potential defects early as they are introduced to the version control system would definitely help in saving time and effort during testing or maintenance phases. In this paper, defect prediction models that uses ensemble classification techniques have been proposed. The proposed models have been applied using different sets of software metrics as attributes of the classification techniques and tested on datasets of different sizes. The results show that change metrics outperform static code metrics and the combined model of change and static code metrics. Ensembles tend to be more accurate than their base classifiers. Defect prediction models using change metrics and ensemble classifiers have revealed the best performance, especially when the datasets used have imbalanced class distribution.

© 2018 Faculty of Engineering, Alexandria University. Production and hosting by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

Mining software repositories can help several development processes and activities including code programming, bug detection, testing, and maintenance [1]. Techniques used to mine software engineering data, along with the unique challenges of mining software data, are presented in number of surveys in the literature [2–6]. Software activities such as testing and bug fixing require huge number of resources which are quite expensive. Testing and bug fixing efforts comprise around 50–60% of the software development effort today [7]. Since the software products are too large, it is not possible to test all classes completely after every code change commit as this is going to be time and resource consuming. Identifying potentially buggy classes or methods after code change(s) at early stages and fixing them within the development time will result in significant time, resources and budget savings. Efficient identification of the fault proneness of a class change depends on many factors, most importantly the software metrics of this class/change.

There are number of different software metrics introduced in the literature [8–11]. Moreover, there have been various empirical studies that have used subsets of the metrics to detect

* Corresponding author.
E-mail addresses: ymustafa@alexu.edu.eg, ymustafa@vt.edu (M.Y. ElNainay).

Peer review under responsibility of Faculty of Engineering, Alexandria University.

https://doi.org/10.1016/j.aej.2018.01.003

1110-0168 © 2018 Faculty of Engineering, Alexandria University. Production and hosting by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

potentially buggy classes using single classifiers. Various ensemble classifiers methods are presented in the literature that have shown suprior performance over single classifiers in applications such as spam detection [12] and intrusion detection [13] but not widely applied to the bug prediction problem. Traditional learning models assume well distributed data classes. In many real-world data domains, however, the data are class-imbalanced, where the main class of interest is represented by few tuples. This is known as the class imbalance problem. However, none of the previous research on bug prediction has paid enough attention to this real-life problem.

In contrast to previous work, we build different models using ensemble classifiers and different sets of software metrics and evaluate them using datasets of different sizes to conclude on the best model for defect prediction, especially for imbalanced class distribution. The used ensembles are based on weighted majority voting techniques. An ensemble combines a series of base classifiers with the aim of creating an improved composite classification model. The metrics used are static code metrics and change metrics. A combined model of both types of metrics has been applied too. The primary contribution of this research is building and evaluating different combination of ensemble classifiers based on weighted majority voting techniques with different sets of software metrics to conclude on the best bug prediction model to be used with real-life software development datasets with imbalanced class distribution.

The remaining of this paper is organized as follows. In Section 2, the related work on different software metrics and the related work in the field of ensemble classifiers have been surveyed. In Section 3, the proposed bug prediction models of this research are explained in details. Datasets used, the evaluation environment, and performance metrics are described in Section 4. The evaluation of the different models and experimental results are discussed in Section 5. Finally, conclusions and possible future extensions are presented in Section 6.

2. Related work

In this section, different software metrics used for defect prediction are surveyed. Furthermore, previous work in ensemble classifiers is surveyed.

2.1. Related work in metrics selection

Different sets of static code metrics and change metrics have been used for defect prediction in the literature. Based on the study presented in [14], there is no one best set of static code metrics for defect prediction.

In [8], a new set of change metrics was introduced that captures changes done by the developers across different releases of the project. The authors performed a comparative analysis between product-related and process-related software metrics where their new set of metrics go under process-related metrics and static code metrics represent product-related metrics. The first step of their experiment was to build three models, one using only their new set of change metrics, referred to as change model, one using only the static code metrics, referred to as code model, and one using both types of metrics referred to as combined model. Their results strongly endorse building defect predictors using change data of source code. However, they did not automate the extraction of information from the repositories.

In [9], the model introduced for bug prediction is based on method level metrics and tries to predict whether a certain method is bug prone or not. They did not use the well known static code metrics set since it contains metrics which are not directly applicable to methods. Instead method level static code metrics, whose performance was proved to be good in previous studies [8,15,16] have been used. Furthermore, coarse grained change metrics are used. However, to build prediction models on the method level, it is necessary to track changes at a finer granularity to track the semantic of individual code changes.

In [10], predicting bugs using anti-patterns metrics, a new set of software metrics that studies the effect of bad code designs on increasing the density of bugs of a certain file, is introduced. Bad code designs are designs that violate the well known design patterns, for example anti-singleton, which is a class that provides mutable class variables, that consequently could be used as global variables. The results have proved that in general the density of bugs in a file with anti-patterns is higher than the density of bugs in a file without anti-patterns. The investigation of using different sets of anti-patterns was left for future.

In [11], a new set of metrics depending on change genealogies has been presented. Change genealogies were first introduced by Brudaru and Zeller [17]. A change genealogy is a directed acyclic graph structure modeling the dependencies between individual change sets. Four different sets of models of every classification technique were compared to each other: one model is based on code complexity metrics, one model is based on code dependency network metrics, one model is based on genealogy network metrics, and one model is based on a combined set code dependency and change genealogy network metrics [15,18]. Four open source projects were selected to apply the proposed models on them. The selected case studies were httpclient, jackrabbit, lucene and rhino. They have done a thorough comparison between the proposed models using k-nearest neighbor, recursive partition, support vector machine, tree bagging and random forest. Results have shown that the combined metrics have better performance over other separate set of metrics.

2.2. Related work in ensemble techniques

Different ensemble classifier techniques have been introduced in the literature. In [19], weighted majority voting (WM) and randomized weighted majority voting (RWM) techniques were introduced. Weighted majority voting is a pool of classifiers and a master algorithm. A non negative weight is associated with each algorithm (function) of the pool. All weights are initially zero unless specified otherwise. The master algorithm forms its prediction by comparing the weights of the classifiers and goes with the larger total (arbitrary in case of a tie). When the master algorithm makes a mistake, then the weights of the classifiers that agreed with the master algorithm will be penalized with a penalty fixed factor β such that $0 \leqslant \beta < 1$.

Randomized weighted majority voting has been introduced as well and proved that it outperforms weighted majority voting since it has more bounded error. The only randomness is when the learner makes its prediction, whereas the classifiers predictions are not random.

In [20], a simple and effective method is introduced that aims to find the experts that have the lowest false positive and the lowest false negative rates besides the overall best expert, simultaneously. The introduced method is called cascading randomized weighted majority voting (CRWM), that presents a cascade version of the randomized weighted majority voting to find these best experts especially on data with large sizes. Results have shown that CRWM is dominant in most of the datasets. This is a promising technique for imbalanced datasets since the structure of the CRWM focuses on each class separately which provides a powerful facility in defining different cost functions on each class.

Other ensemble classification techniques have also been proposed in the literature [21]. The surveyed techniques have not been used in the defect prediction model. In this paper, we apply the weighted majority voting (WM), randomized weighted majority voting (RWM), cascading weighted majority voting (CWM), and cascading randomized weighted majority voting (CRWM) techniques using different classification methods and different sets of software metrics to conclude on the combination of the best ensemble method, set of classifiers, and set of software metrics.

3. The proposed bug prediction models

Each of the proposed software defect prediction models using weighted majority voting techniques is composed of two main operations. The first operation is the attributes (software metrics) retrieval. In this work, three sets of metrics were used as attributes/features: static code metrics, change metrics, and a combined model of them. Models using different sets of software metrics are compared together in the evaluation section.

The second operation is creating ensemble classifiers by using 4 types of weighted majority voting techniques: the weighted majority voting (WM), randomized weighted majority voting (RWM), cascading weighted majority voting (CWM), and cascading randomized weighted majority voting (CRWM) techniques. In every technique the ensemble is made of 4 different single classifiers and 1 ensemble classifier are combined namely, naive bayes (NB), decision trees (C4.5), support vector machine (SVM), logistic regression (LR) and random forest (RF) which is an ensemble of C4.5. In the evaluation section, the performance of the implemented ensemble classifiers is compared to the basic classifiers (NB, C4.5, SVM, LR and RF). The two aforementioned operations are described in more details in the following two subsections.

3.1. Metrics selection

In our proposed approaches, three types of metrics were selected. The selected metrics are the most widely used metrics in the related work.

1. Static code metrics.
2. Change metrics.
3. Combined model of both.

The static code metrics are used in literature in uncovering useful information about the code. The set of static code metrics used in this work is tabulated in Table 1. They are product-related metrics that are used in this work to be compared with change metrics in terms of prediction accuracy.

Table 1 Static code metrics used.

Static code metrics	Description
WMC	Weighted Methods per class
DIT	Depth of Inheritance Tree
NOC	Number Of Children
CBO	Coupling between Object classes
RFC	Response For a Class
LCOM	Lack of COhesion in Methods
LOC	Lines Of Code

The second type of metrics used in our models are change metrics. Change metrics are process-related metrics which require historical information on the system. It has been shown in literature that process metrics are better indicators of bugs with respect to performance, portability and the stability of the model using single classifiers [22]. Moreover, change

Table 2 List of class level change metrics.

Metric name	Description
Revisions	Number of revisions of a file
Refactoring[1]	Number of times a file has been refactored
Bug Fixes[2]	Number of times a file has been involved in bug fixing
Authors	Number of distinct authors that checked a file into the repository
LOC Added	Sum over all revisions of the lines of code added to a file
Max LOC Added	Maximum number of lines of code added for all revisions
AVE LOC Added	Average lines of code added per revision
LOC Deleted	Sum over all revisions of the lines of code deleted to a file
Max LOC Deleted	Maximum number of lines of code deleted for all revisions
AVE LOC Deleted	Average lines of code deleted per revision
CODECHURN	Sum of (added lines of code deleted lines of code) over all revisions
Max CODECHURN	Maximum CODECHURN for all revisions
AVE CODECHURN	Average CODECHURN per revision
Max CHANGESET	Maximum number of files committed together to the repository
AVE CHANGESET	Average number of files committed together to the repository
Age	Age of a file in weeks (counting backwards from a specific release)
Weighted Age	$\frac{\sum_{i=1}^{N} Age(i) \times LOCAdded(i)}{\sum_{i=1}^{N} LOCAdded(i)}$

[1] Computed by using the following query for determining if the origin of a revision was a refactoring: . . .revision comment LIKE '%refactor%'.

[2] Computed by using the following query for determining if the origin of a revision was a bug fix: . . .revision comment LIKE '%Fix %' AND comment NOT LIKE '% prefix %' AND comment NOT LIKE '% postfix %'.

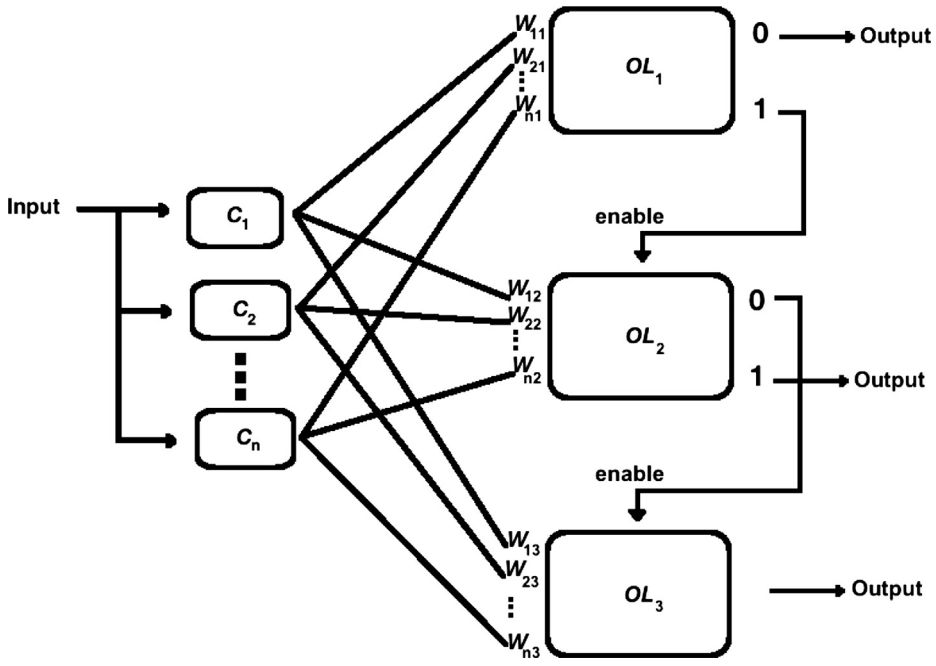


Fig. 1 Cascading randomized weighted majority voting technique.

Table 3 Datasets used.

Project	LOC	Files	Size	URL	Pos:Neg
Apache Log4j Trunk	30330	407 java files	17 M	http://svn.apache.org/repos/asf/logging/log4j/trunk	276:131
Apache Jmeter	111482	1359 java files	87 M	http://svn.apache.org/repos/asf/jmeter/trunk	1193:166
Apache Ant	139799	2052 Java files	68 M	http://svn.apache.org/repos/asf/ant/core/trunk	1917:135
Apache Tomcat	254953	2177 Java files	57 M	http://svn.apache.org/repos/asf/tomcat/tc7.0.x/trunk	1667:510

metrics have been proven to have better prediction accuracy than static code metrics [8]. The list of change metrics that are extracted and used in this work is described in Table 2.

The third type of metrics is the combined metrics that combine both product-related (static code metrics) and process-related metrics (change metrics). A thorough comparative analysis between the product, process, and combined metrics for defect prediction is carried out in the evaluation section.

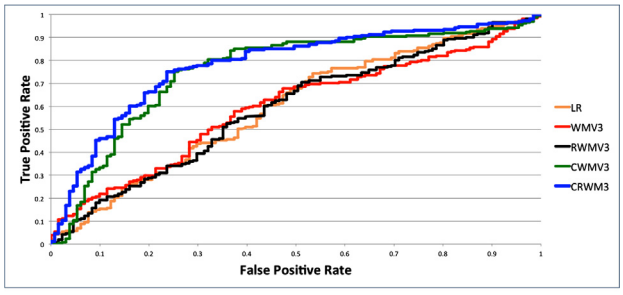
3.2. Ensemble weighted majority voting techniques

Ensemble methods are algorithms which take the advantage of an ensemble of classifiers to predict labels of data. The weighted majority voting algorithm was firstly introduced by Littlestone and Warmuth in 1994 [19]. Its idea depends on one learner that decides the final result by taking the weighted majority votes of the algorithms in the pool. They have shown that this algorithm is robust with respect to the errors in the data. For example if algorithm `A` makes at most `m` mistakes then, the Weighted majority voting algorithm will make at most `c(log|A|+m)` mistakes on that sequence where `c` is a fixed constant. A non negative weight is assigned to each algorithm of the pool. All weights are initially one (unless specified otherwise). The algorithm forms its prediction by comparing the total weights made for each class and predicts to the larger total (arbitrary in case of a tie). When the weighted majority algorithm makes a mistake, then the weights of those algorithms of the pool that agrees with it are each multiplied by a fixed β such that $0 \leqslant \beta < 1$. A randomized version of the weighted majority voting algorithm was also introduced in [19]. It has been proved that the randomized weighted majority is better than weighted majority with respect to the error bound. The only randomness is when the learner makes its prediction, where it predicts with probability $\frac{w_{i1}}{W_1}$ where w_{i1} is the weight of classifier i in learner 1 and $W_1 = \sum_i w_{i1}$. However, the classifiers predictions are not random.

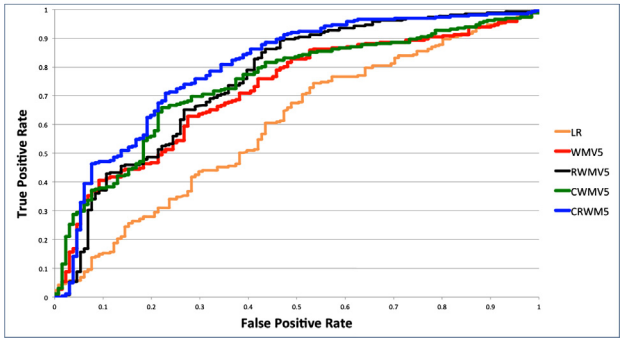
In the current paper, two types of cascading weighted majority voting techniques are implemented: cascading weighted majority voting and cascading randomized weighted majority voting. The cascading weighted majority voting technique has n base classifiers and N learners with weighted majority mechanism that exploits these classifiers, where `N = number of classes + 1` (thus `N = 3` in our problem as we have two classes: clean and buggy). For every new data instance, each of these n base classifiers predicts a label. Using the corresponding weight factors, these predictions are given to the learners in order for them to make their predictions. The first algorithm/classifier is responsible for predicting negative labels. If this classifier predicts the label as negative, the label is set to negative. Otherwise, the second classifier will be

Table 4 Log4j results using different metrics and different classifiers.

Log4j	SCM metrics			Change metrics			SCM + change metrics		
	FM	Ac	AUC	FM	Ac	AUC	FM	Ac	AUC
NB	21	38	0.49	81	73	0.69	70	63	0.361
LR	78.5	65.5	0.631	77	66	0.759	81	72	0.664
C4.5	74	62.5	0.535	**89**	**79**	**0.879**	82	75	0.452
SVM	78.2	66.1	0.56	84	77	0.723	82.7	72	0.677
RF	75.9	66.3	0.622	84	78	0.831	83	76	0.599
WM_3	76	63	0.58	82	73	0.781	82	72	0.672
CWM_3	81	72.9	0.75	84	76	0.778	84	77	0.756
RWM_3	79	67	0.6	83	75	0.775	82	73	0.75
$CRWM_3$	82.7	75	0.78	85	77.8	0.789	82.3	73.9	0.782
WM_5	78	66.8	0.716	83	76	0.799	82.9	75.4	0.8
CWM_5	81	69.2	0.744	85	78	0.839	84.5	77.1	0.825
RWM_5	83.5	74.4	0.755	84.4	77	0.822	84.7	76.9	0.815
$CRWM_5$	**85**	**76.7**	**0.798**	88	**79**	0.866	**85**	**77.8**	**0.827**

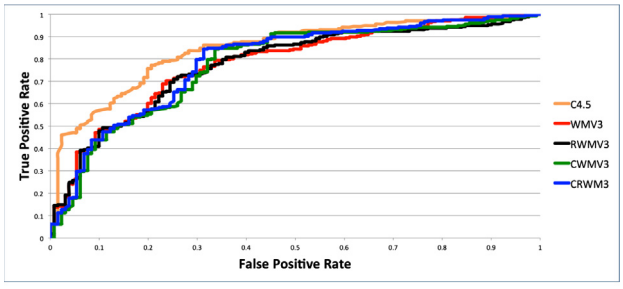


(2a) ROC for Log4j using static code metrics - 3 classifiers

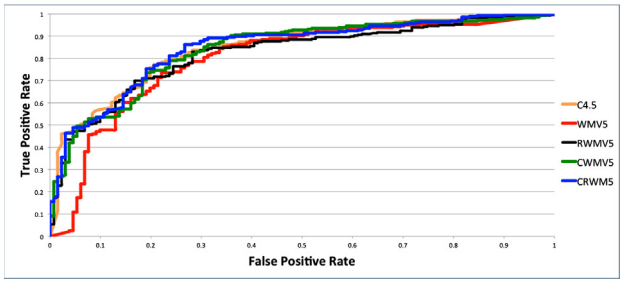


(2b) ROC for Log4j using static code metrics - 5 classifiers

Fig. 2 ROC for Log4j using static code metrics.



(2c) ROC for Log4j using change metrics - 3 classifiers



(2d) ROC for Log4j using change metrics - 5 classifiers

Fig. 3 ROC for Log4j using change metrics.

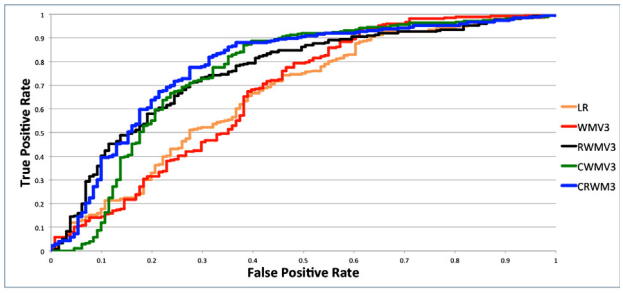
applied. If the second classifier predicts the label as positive, the output label will be positive. Otherwise, the third classifier will be applied and the output of this classifier would be the output label for the instance. Cascading randomized weighted majority voting depicted in Fig. 1 has the same concept of cascading weighted majority voting using N learners but with randomized weighted majority algorithm mechanism. It has been shown in contexts other than defect prediction that cascading randomized weighted majority voting outperforms randomized weighted majority voting especially in sufficiently large data sets. As the learners are using randomized weighted majority voting mechanism; in the specified learner, the weight factor of the classifiers that made the wrong prediction would be penalized by a constant factor β where $0.05 \leqslant \beta \leqslant 0.95$.

4. Datasets and evaluation environment

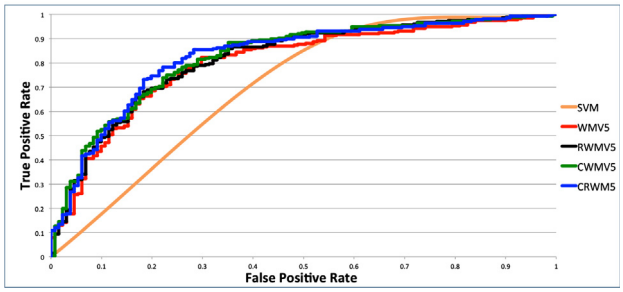
In this section, the datasets used in this work and their nature are presented. The SZZ algorithm used for labeling datasets is described. Finally, the evaluation environment and the used performance metrics are discussed.

4.1. Data collection

This subsection describes the datasets used in this proposal. They were all Apache open source java projects that can be found in Apache subversion website [23]. Table 3 lists the datasets used in this work. They are arranged in ascending order in terms of number of java files (number of instances). Close

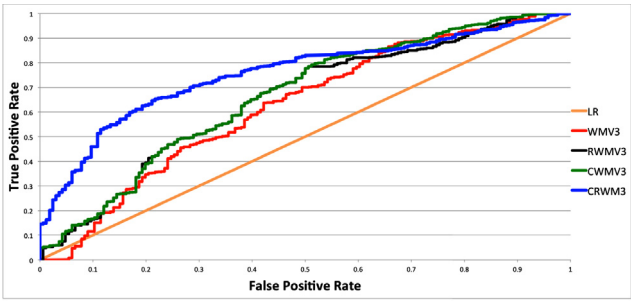


(2e) ROC for Log4j using combined metrics - 3 classifiers

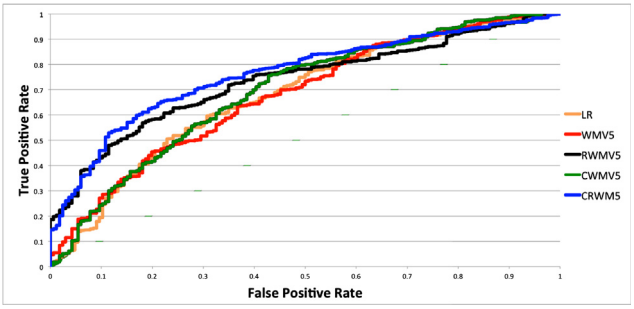


(2f) ROC for Log4j using combined metrics - 5 classifiers

Fig. 4 ROC for Log4j using combined metrics.



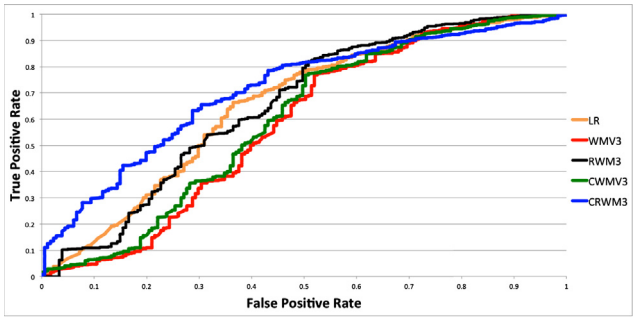
(3a) ROC for Jmeter using static code metrics - 3 classifiers



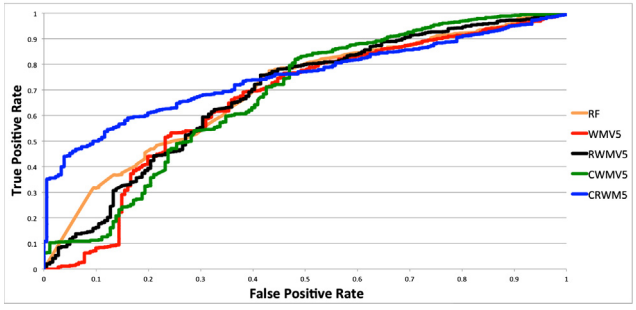
(3b) ROC for Jmeter using static code metrics - 5 classifiers

Fig. 5 ROC for Jmeter using static code metrics.

study to the last column in this table shows that the class distribution of the datasets is imbalanced. The ratio of the positive instances is too high with respect to the negative ones which means that the class distribution is skewed. Constant and relatively balanced class distribution is a rare case in the real world [24]. Thus, those datasets were selected with this nature in order to resemble a real world case.



(3c) ROC for Jmeter using change metrics - 3 classifiers



(3d) ROC for Jmeter using change metrics - 5 classifiers

Fig. 6 ROC for Jmeter using change metrics.

4.2. The SZZ algorithm for data labeling and validation

All classification techniques used in our models are supervised learning techniques; it should be provided with instances already classified as buggy or clean during training phase. Since software changes in the used datasets are not labeled, the labeling of the used datasets is performed using the SZZ algorithm [25]. The SZZ algorithm, called so after the first letters of the authors' last names (Jacek Sliwerski, Thomas Zimmermann and Andreas Zeller), is the algorithm implemented to label the instances by creating a link between Version Control System (VCS) commits and bugs. This is required for labeling data instances as buggy or clean to be used for classifiers training and validation. It identifies commits that fix bugs by matching bug numbers listed in commit messages with bugs in the bug database that have been marked as FIXED. The SZZ algorithm specifies regular expressions for identifying probable bug numbers in commit messages and for identifying keywords that are likely to indicate that a bug fix has occurred [25]. The basic idea of this algorithm can be summarized in the following bullets [26].

1. Start with a bug report in the bug database, indicating a fixed problem.
2. Extract the associated change from the version archive, thus giving us the location of the fix.
3. Determine the earlier change at this location that was applied before the bug was reported.

4.3. Evaluation environment and performance metrics

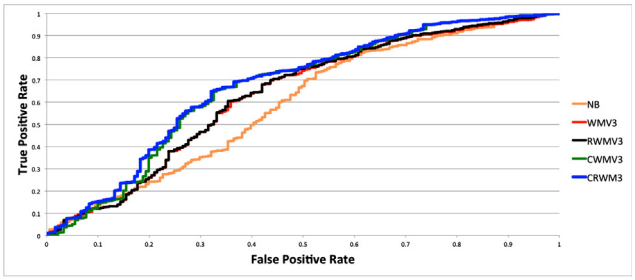
Evaluation was done based on the commonly used performance metrics in prediction models such as f-measure and

accuracy. The f-measure is the harmonic mean of precision and recall whereas accuracy is the common metric used by most of the related prediction models. However, the simple classification accuracy is a poor metric for measuring performance in class-imbalanced data. [27]. As the class distribution becomes more skewed, evaluation based on accuracy breaks down [28]. Since the datasets used in this paper are imbalanced (refer to Table 3), we used the Receiver Operating Characteristic (ROC) curve due to its usefulness in such datasets [27]. Thus, area under ROC curve (AUC) is studied in this paper. Mathematical definitions for the performance metrics used are listed in the following points.

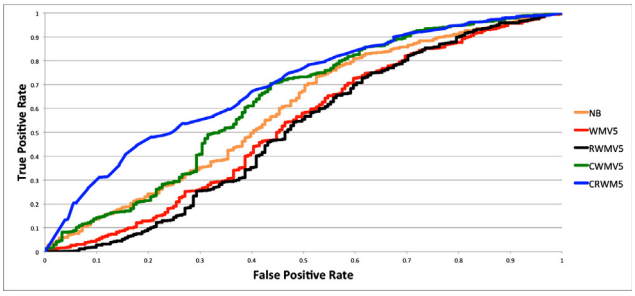
- $\texttt{Precision (PC)} = \texttt{Number of positive instances that were correctly classified as such/Total number of instances classified as positive}$
- $\texttt{TruePositiveRate(TPR)} = \frac{\texttt{Positivescorrectlyclassified}}{\texttt{Totalpositives}}$
- $\texttt{FalsePositiveRate(FPR)} = \frac{\texttt{Negativesincorrectlyclassified}}{\texttt{Totalnegatives}}$
- $\texttt{F-measure(FM)} = \frac{\texttt{2 xPrecisionxRecall}}{\texttt{Precision+Recall}}$ where $\texttt{Recall} = \texttt{TPR}$
- $\texttt{Accuracy(Ac)} = \frac{\texttt{TruePositives+TrueNegatives}}{\texttt{n}}$ where $\texttt{n} = \texttt{total number of instances}$
- An ROC curve for a given model shows the trade-off between the true positive rate (TPR) and the false positive rate (FPR) [29] where AUC is the area under the ROC curve

Base classifiers selected were naive bayes (NB), decision trees (C4.5), logistic regression (LR) support vector machine (SVM), and random forest (RF) as mentioned in the previous section. The first three classifiers were selected following the selection taken in [8] since the work introduced there was implemented in this paper. Moreover, ensemble weighted majority voting techniques of these three specific classifiers were executed. SVM and RF were selected since they are well known of their strong performance in terms of the prediction accuracy and their different nature from the other three selected classifiers.

Weka [30], the open source machine learning tool, was used in executing all the classifiers mentioned above, as well as, the ensemble classifiers that were implemented in our work. For each dataset, 27 experiments were carried out. 5 classifiers and 4 ensemble weighted majority voting techniques were executed across 3 different metrics types. 10-fold cross-validation was used in all the experiments. For each run, performance metrics such as FM, Ac and AUC were calculated.



(3e) ROC for Jmeter using combined metrics - 3 classifiers



(3f) ROC for Jmeter using combined metrics - 5 classifiers

Fig. 7 ROC for Jmeter using combined metrics.

5. Performance evaluation

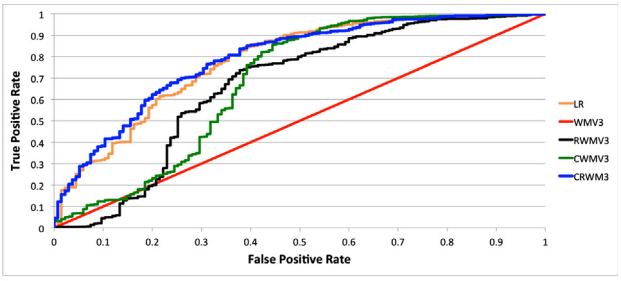
Experiments were carried out using datasets listed in Table 3. For each dataset, experiments were performed using the 3 different sets of metrics and the 5 different base classifiers as well as the 4 ensemble weighted majority voting techniques of these classifiers mentioned in the previous section. The weighted majority voting techniques are implemented once with 3 classifiers (NB, LR, and C4.5) and once with the 5 classifiers.

Table 5 Jmeter results using different metrics and different classifiers.

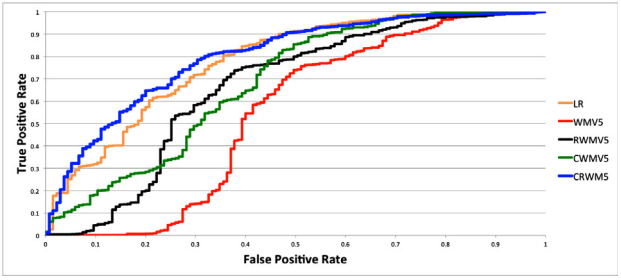
Jmeter	SCM metrics			Change metrics			SCM + change metrics		
	FM	Ac	AUC	FM	Ac	AUC	FM	Ac	AUC
NB	87	81.4	0.649	90	82.7	0.61	88.8	80	0.59
LR	88	83.2	0.679	92	85.7	0.65	92	85	0.45
C4.5	88	83.3	0.521	92	85.5	0.57	92	80	0.58
SVM	92	86	0.56	92.3	85.9	0.565	92	86.5	0.56
RF	91	84	0.646	90	83.4	0.69	92	86	0.43
WM_3	92.8	86.6	0.631	92.3	85.7	0.58	92	85.9	0.63
CWM_3	93	87.1	0.665	92	86.9	0.59	92.7	86.3	0.66
RWM_3	93.2	87	0.662	92.9	86.7	0.65	92.7	86.5	0.63
$CRWM_3$	93	87.6	0.7	92	87	0.7	**92.8**	86.6	0.678
WM_5	92.9	86.7	0.678	92.7	86.3	0.66	92	86.3	0.65
CWM_5	93.5	87.5	0.693	93	86.9	0.67	92.8	86.9	0.65
RWM_5	93.3	87.4	0.735	93	87	0.684	92.8	86.5	0.51
$CRWM_5$	**93.5**	**88**	**0.765**	**93.4**	**88**	**0.749**	**92.8**	**86.9**	**0.695**

Table 6 Ant results using different metrics and different classifiers.

Ant	SCM metrics			Change metrics			SCM + change metrics		
	FM	Ac	AUC	FM	Ac	AUC	FM	Ac	AUC
NB	94	90	0.75	94	90.2	0.56	92	91	0.5
LR	96	91	0.78	96	93	0.73	94	91.3	0.44
C4.5	96	90	0.62	96	93	0.56	94	91	0.5
SVM	96.2	93	0.585	96.6	93.4	0.585	96.6	93.4	0.57
RF	95.9	92.1	0.76	96.5	93.2	0.76	96.6	93.4	0.61
WM_3	96	92	0.68	96	93	0.7	96.5	93	0.66
CWM_3	96.5	93.3	0.68	96	93.9	0.7	96	93.6	0.71
RWM_3	96	93	0.6	96.8	93.7	0.69	96.6	93.2	0.71
$CRWM_3$	96.6	93.4	0.79	97	94.2	0.75	96.6	93.7	0.72
WM_5	96.2	92.7	0.53	96	93.2	0.6	96.5	93.1	0.72
CWM_5	96.4	93	0.68	96.8	93.9	0.71	96.6	93.4	0.759
RWM_5	96.4	93	0.66	96.9	94	0.66	96.6	93.5	0.73
$CRWM_5$	**96.7**	**94**	**0.801**	**97.1**	**94.7**	**0.828**	**97**	**94**	**0.779**

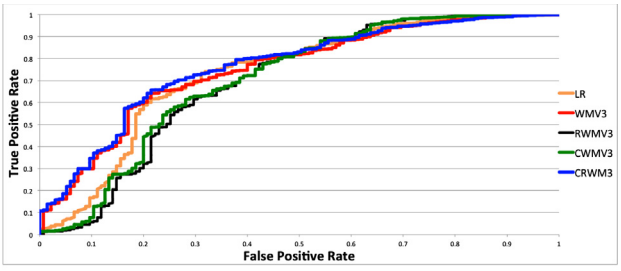


(4a) ROC for Ant using static code metrics - 3 classifiers

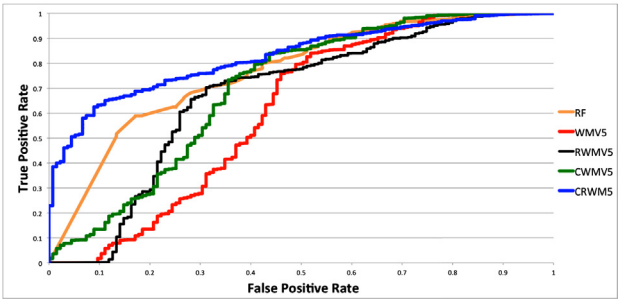


(4b) ROC for Ant using static code metrics - 5 classifiers

Fig. 8 ROC for Ant using static code metrics.



(4c) ROC for Ant using change metrics - 3 classifiers



(4d) ROC for Ant using change metrics - 5 classifiers

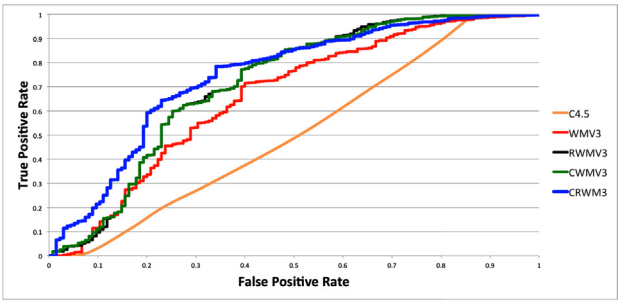
Fig. 9 ROC for Ant using change metrics.

The subscripts 3 and 5 in the tables of results identify the two different implementations.

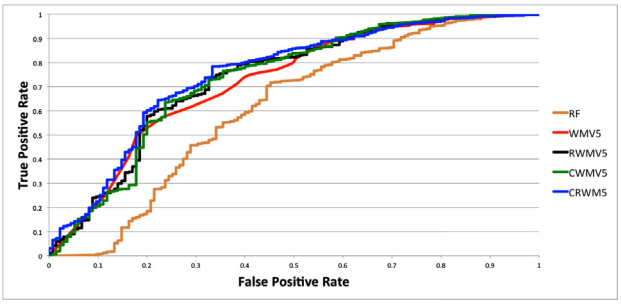
5.1. The effect of metrics and ensemble classifiers

Results of the experiments are discussed and tabulated in the following points. The best Ac, FM or AUC are in bold and the second best Ac, FM or AUC are underlined within the same set of metrics. ROC modeling is also demonstrated in this subsection. ROC graphs are two-dimensional graphs in which TPR is plotted on the Y-axis and FPR is plotted on the X-axis. An ROC graph depicts relative trade-offs between benefits (true positives) and costs (false positives). The use of ROC graphs in the machine learning community has increased rapidly especially due to the realization that simple classification accuracy is often a poor metric for measuring performance [24,31]. Moreover, ROC graphs have properties that make them especially useful for domains with skewed class distribution. ROC graphs are plotted for all the 4 datasets, also AUC is tabulated for each curve. It will be noticed that tuned SVM did not produce significantly high AUC since SVM can be limited in their performance on highly imbalanced datasets [32]. It is worthwhile noting that SVM results listed below are the best results reached using tuned SVM parameters (`kernel = radial basis function (RBF), gamma = 0.01 and c = 10`). The initial results of SVM using its default parameters (`kernel = ploy kernel and c = 1`) were not as good as expected.

Results of log4j project are tabulated in Table 4. They show that $CRWM_5$ performs better, in terms of predictive accuracy across all sets of metrics (which verifies the conclusion reached in [20]). Figs. 2a and b represent ROC curves for Log4j when using static code metrics with 3 and 5 ensemble classifiers, respectively. It can be noticed that cascading randomized



(4e) ROC for Ant using combined metrics - 3 classifiers



(4f) ROC for Ant using combined metrics - 5 classifiers

Fig. 10 ROC for Ant using combined metrics.

majority voting technique with 5 classifiers ($CRWM_5$) has produced the maximum AUC in this case. Figs. 3a and d represent ROC curves for Log4j when using change metrics with 3 and 5 ensemble classifiers, respectively. It can be observed that C4.5 has produced the maximum AUC in this case followed by $CRWM_5$. Finally, Figs. 4a and b represent ROC curves for Log4j when using combined metrics with 3 and 5 ensemble classifiers respectively. The $CRWM_5$ has produced the maximum AUC. In Tables 4–7, bold is used to highlight the best result and underline is used for the second best result.

Results of the Jmeter dataset are tabulated in Table 5. They show that $CRWM_5$ performs better, in terms of predictive accuracy, f-measure and AUC across all sets of metrics. Figs. 5a and 5b represent ROC curves for Jmeter when using static code metrics with 3 and 5 ensemble classifiers, respectively. Figs. 6a and 6b represent ROC curves for Jmeter when using change metrics with 3 and 5 ensemble classifiers, respectively. Whereas, Figs. 7a and 7b represent ROC curves for Jmeter when using combined metrics with 3 and 5 ensemble classifiers, respectively.

Results of the Ant dataset are tabulated in Table 6. They show that $CRWM_5$ performs better, in terms of predictive accuracy, f-measure and area under ROC curve across all sets of metrics. Figs. 8a and 8b show ROC plots for Ant dataset using static code metrics with 3 and 5 ensemble classifiers, respectively. Figs. 9a and 9b represent ROC plots for the Ant dataset using change metrics for the 3 and 5 ensemble classifiers, respectively. Figs. 10a and 10b demonstrate ROC plots for the Ant dataset using combined metrics. The first figure represents the 3 classifiers and their ensemble while the second represents the 5 classifiers and their ensemble.

Results of the Tomcat dataset are tabulated in Table 7. They show that $CRWM_5$ performs better, in terms of predictive accuracy and AUC across all sets of metrics used. Figs. 11a and 11b show ROC plots for Tomcat dataset using static code metrics with 3 and 5 ensemble classifiers, respectively. Fig. 12a and Fig. 12b represent ROC plots for Tomcat dataset using change metrics with 3 and 5 ensemble classifiers, respectively. Figs. 13a and 13b demonstrated ROC plots for Tomcat using combined metrics. The first figure represents the 3 classifiers and their ensemble while the second represents the 5 classifiers and their ensemble.

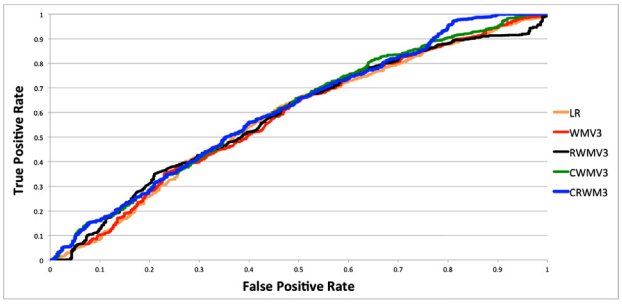
5.2. Sensitivity of classifiers' accuracy to parameter beta

The beta parameter is the penalty parameter that is multiplied by weights of those classifiers which agreed with the wrong final decision taken by the learner. This parameter has a constant value that lies between 0 and 1. It is used in all the weighted majority techniques discussed before. This subsection will discuss the sensitivity of cascading randomized weighted majority voting accuracy with respect to this parameter. It has been shown that $\beta = 0.5$ gives maximum accuracy with cascading randomized weighted majority voting technique in [20].

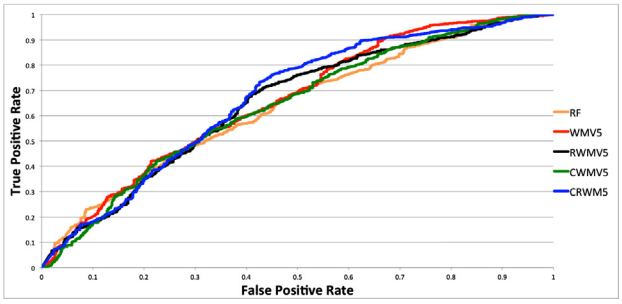
Fig. 14a shows the sensitivity of accuracy of $CRWM_5$ to beta parameter when using static code metrics. We can notice

Table 7 Tomcat results using different metrics and different classifiers.

Tomcat	SCM metrics			Change metrics			SCM + change metrics		
	FM	Ac	AUC	FM	Ac	AUC	FM	Ac	AUC
NB	77	66.4	0.548	84	74.7	0.68	83.9	73.7	0.67
LR	83	71.4	0.58	86	76.8	0.68	86	76.5	0.68
C4.5	83.6	72.7	0.58	87	79	0.63	84.6	75	0.66
SVM	82	70.9	0.57	86.7	77.6	0.6	86.7	76.8	0.585
RF	80	69	0.637	85.5	76.6	0.7	85	76.6	0.68
WM_3	81.7	70	0.581	82	72	0.7	83	75	0.69
CWM_3	98.3	74.6	0.6	85.9	76.4	0.7	85.8	76.4	0.7
RWM_3	78	67	0.6	85	75	0.68	85.3	76.2	0.7
$CRWM_3$	**98.6**	74.9	0.64	88.9	79	0.72	85.8	77.9	0.71
WM_5	82.7	72	0.601	82	72	0.7	84	75	0.71
CWM_5	84.5	74.6	0.65	88.5	79.5	0.7	87	77	0.72
RWM_5	83	73	0.597	87	78.6	0.7	86.4	77.8	0.71
$CRWM_5$	85.3	**75.5**	**0.68**	**89.5**	**81**	**0.747**	**87**	**78**	**0.736**

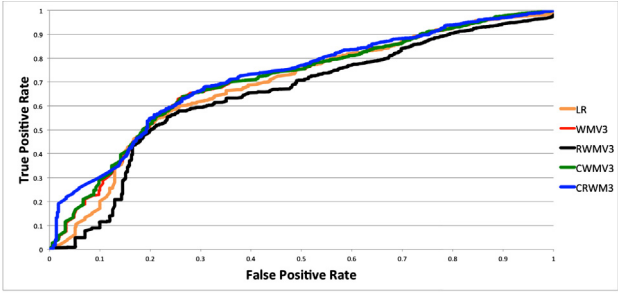


(5a) ROC for Tomcat using static code metrics - 3 classifiers

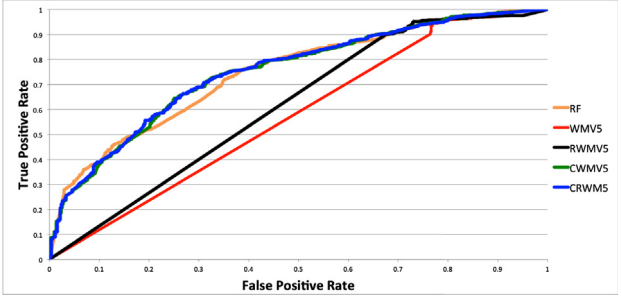


(5b) ROC for Tomcat using static code metrics - 5 classifiers

Fig. 11 ROC for Tomcat using static code metrics.

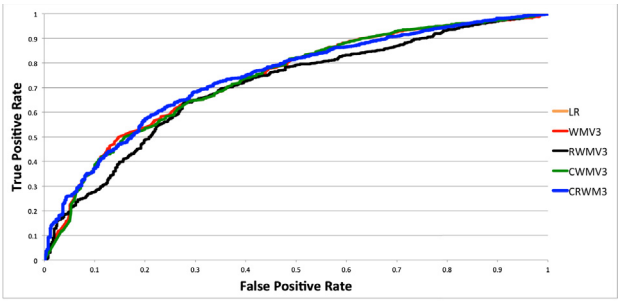


(5c) ROC for Tomcat using change metrics - 3 classifiers

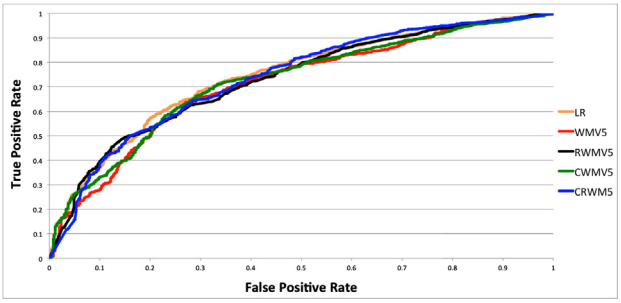


(5d) ROC for Tomcat using change metrics - 5 classifiers

Fig. 12 ROC for Tomcat using change metrics.



(5e) ROC for Tomcat using combined metrics - 3 classifiers



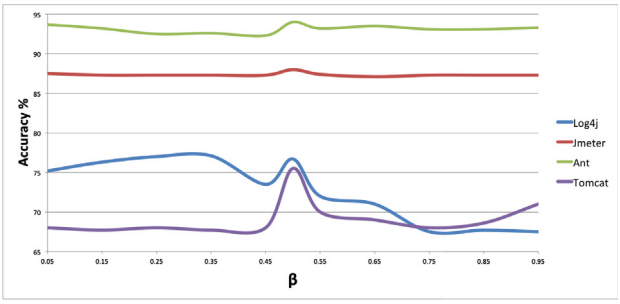
(5f) ROC for Tomcat using combined metrics - 5 classifiers

Fig. 13 ROC for Tomcat using combined metrics.

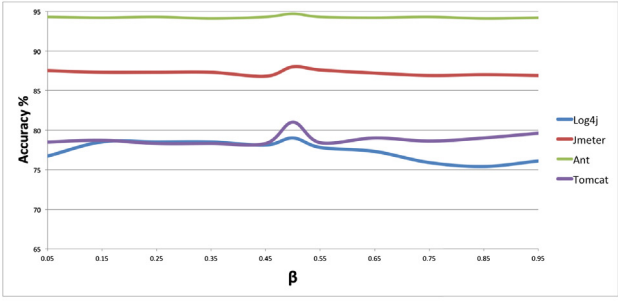
that Log4j does not follow the expectations as it gives maximum accuracy when $\beta = 0.34$. This may be due its small size (407 instances, 17 MB). Nevertheless, all the other datasets (Jmeter, Ant and Tomcat), that have relatively larger sizes, followed and verified the conclusion in [20] having the maximum accuracy when $\beta = 0.5$.

Fig. 14b demonstrates the relation between $CRWM_5$ accuracy and beta parameter when using change metrics. We can observe that accuracy curves of all the 4 datasets have their peak when $\beta = 0.5$. Fig. 14c presents this relation when using combined metrics. Log4j gets out of the track one more time where it gives maximum accuracy when $\beta = 0.05$. The remaining data sets give maximum accuracy when $\beta = 0.5$ as concluded in [20].

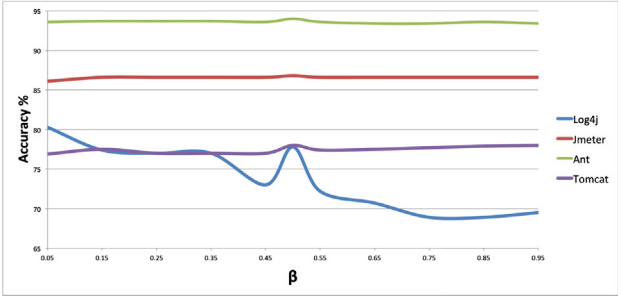
A thorough time complexity analysis of the proposed models is out of scope of this paper. However, ensemble classifiers



(6a) Sensitivity of Accuracy to parameter beta using static code metrics and CRWM5



(6b) Sensitivity of Accuracy to parameter beta using change metrics and CRWM5



(6c) Sensitivity of Accuracy to parameter beta using combined metrics and CRWM5

Fig. 14 Effect of parameter beta on accuracy of results.

shall need more processing time compared to their base classifiers and as the number of classifiers increases in the model, more processing time shall be needed.

6. Conclusions and future work

In this paper, software defect prediction models based on weighted randomized majority voting techniques have been proposed. The models have been applied using different sets of software metrics on real datasets of different sizes. The used datasets reflect real-life software projects data with imbalanced class distribution. The results presented in this paper strongly endorse building defect predictors using change metrics and cascading ensemble classifiers. Change metrics have shown better performance than static code metrics and combined static code and change metrics. Results also indicated that ensemble classifiers have solved the problem of class-imbalanced datasets by improving their classification accuracy.

The proposed models can be extended along different directions. Working on method level instead of the class level is one of those extensions. Using other sets of metrics introduced in the literature such as Anti-design patterns metrics and Change genealogies metrics and comparing them against one another is another direction. Testing the proposed models with more datasets, especially large ones, is also needed. Towards better addressing of the class imbalance problem, it is highly recommended to apply other strategies such as oversampling or undersampling together with the ensemble classifiers techniques.

References

[1] T. Xie, J. Pei, A.E. Hassan, Mining software engineering data, in: 29th International Conference on Software Engineering-Companion, 2007. ICSE 2007 Companion, IEEE, 2007, pp. 172–173.

[2] H. Kagdi, M.L. Collard, J.I. Maletic, A survey and taxonomy of approaches for mining software repositories in the context of software evolution, J. Software Mainten. Evol.: Res. Practice 19 (2) (2007) 77–131.

[3] P. Anbalagan, M. Vouk, On mining data across software repositories, in: 2009 6th IEEE International Working

Conference on Mining Software Repositories, IEEE, pp. 171–174.
[4] T. Xie, S. Thummalapenta, D. Lo, C. Liu, Data mining for software engineering, Computer 42 (8) (2009) 55–62.
[5] M. Halkidi, D. Spinellis, G. Tsatsaronis, M. Vazirgiannis, Data mining in software engineering, Intell. Data Anal. 15 (3) (2011) 413–441.
[6] M. D'Ambros, H. Gall, M. Lanza, M. Pinzger, Analysing software repositories to understand software evolution, in: Software Evolution, Springer, 2008, pp. 37–67.
[7] D. Mishra, A. Mishra, Software process improvement in smes: a comparative view, Comput. Sci. Inf. Syst. 6 (1) (2009) 111–140.
[8] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: 2008 ACM/IEEE 30th International Conference on Software Engineering, IEEE, 2008, pp. 181–190.
[9] E. Giger, M. D'Ambros, M. Pinzger, H.C. Gall, Method-level bug prediction, in: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, 2012, pp. 171–180.
[10] S.E.S. Taba, F. Khomh, Y. Zou, A.E. Hassan, M. Nagappan, Predicting bugs using antipatterns, in: ICSM, vol. 13, 2013, pp. 270–279.
[11] K. Herzig, S. Just, A. Rau, A. Zeller, Predicting defects using change genealogies, in: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2013, pp. 118–127.
[12] R. Neumayer, Clustering Based Ensemble Classification for Spam Filtering, Citeseer, 2006.
[13] W. Fan, S.J. Stolfo, Ensemble-based adaptive intrusion detection, in: SDM, SIAM, 2002, pp. 41–58.
[14] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Trans. Software Eng. 33 (1) (2007) 2–13.
[15] T. Zimmermann, N. Nagappan, Predicting defects using network analysis on dependency graphs, in: Proceedings of the 30th International Conference on Software Engineering, ACM, 2008, pp. 531–540.
[16] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, A.E. Hassan, Revisiting common bug prediction findings using effort-aware models, in: 2010 IEEE International Conference on Software Maintenance (ICSM), IEEE, 2010, pp. 1–10.
[17] I.I. Brudaru, A. Zeller, What is the long-term impact of changes?, in: Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering, ACM, 2008, pp 30–32.
[18] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: International Workshop on Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007., IEEE, 2007, p. 9.
[19] N. Littlestone, M.K. Warmuth, The weighted majority algorithm, in: 30th Annual Symposium on Foundations of Computer Science, 1989, IEEE, 1989, pp. 256–261.
[20] M. Zamani, H. Beigy, A. Shaban, Cascading randomized weighted majority: a new online ensemble learning algorithm, Intelligent Data Analysis 20 (4) (2016) 877–889.
[21] L. Rokach, Ensemble methods for classifiers, in: Data Mining and Knowledge Discovery Handbook, Springer, 2005, pp. 957–980.
[22] F. Rahman, P. Devanbu, How, and why, process metrics are better, in: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013, pp. 432–441.
[23] Apache Subversion (Accessed: 2015). <http://svn.apache.org/repos/asf/>.
[24] F.J. Provost, T. Fawcett, et al., Analysis and visualization of classifier performance: comparison under imprecise class and cost distributions, in: KDD, vol. 97, 1997, pp. 43–48.
[25] C. Williams, J. Spacco, Szz revisited: verifying when changes induce fixes, in: Proceedings of the 2008 Workshop on Defects in Large Software Systems, ACM, 2008, pp. 32–36.
[26] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes?, ACM Sigsoft Software Engineering Notes, vol 30, ACM, 2005, pp. 1–5.
[27] T. Fawcett, An introduction to roc analysis, Pattern Recogn. Lett. 27 (8) (2006) 861–874.
[28] J. Egan, Signal Detection Theory and Roc Analysis. Series in Cognition and Perception, 1975.
[29] J. Han, J. Pei, M. Kamber, Data Mining: Concepts and Techniques, Elsevier, 2011.
[30] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten, The weka data mining software: an update, ACM SIGKDD Explor. Newsl. 11 (1) (2009) 10–18.
[31] F. Provost, T. Fawcett, Robust classification systems for imprecise environments, in: AAAI/IAAI, 1998, pp. 706–713.
[32] K. Drosou, S. Georgiou, C. Koukouvinos, S. Stylianou, Support vector machines classification on class imbalanced data: a case study with real medical data, J. Data Sci. 12 (4) (2014) 143–155.