Software Defect Prediction Using Semi-supervised Learning with Change Burst Information

[Anonymized for Double-Blind Review]

***Abstract*—Software defect prediction is an important software quality assurance technique. It utilizes historical project data and previously discovered defects to predict potential defects. However, most of existing methods assume that large amounts of labeled historical data are available for prediction, while in the early stage of the life cycle, projects may lack the data needed for building such predictors. In addition, most of existing techniques use static code metrics as predictors, while they omit change information that may introduce risks into software development. In this paper, we take these two issues into consideration, and propose a semi-supervised based defect prediction approach - *extRF*. extRF extends the classical supervised *Random Forest* algorithm by self-training paradigm. It also employs change burst information for improving accuracy of software defect prediction. We also conduct an experiment to evaluate extRF against three other supervised machine learners (i.e., Logistic Regression, Naïve Bayes, Random Forest) and comparing the effectiveness of code metrics, change burst metrics, and a combination of them. Experimental results show that extRF trained with a small size of labeled dataset achieves comparable performance to some supervised learning approaches trained with a larger size of labeled dataset. When only 2% of Eclipse 2.0 data are used for training, extRF can achieve F-measure about 0.562, approximate to that of LR (a supervised learning approach) at labeled sampling rate of 50%. Besides, change burst metrics outperform code metrics in that F-measure rises to a peak value of 0.75 for Eclipse 3.0 and JDT.Core.**

***Keywords*—Defect Prediction; Software Quality Assurance; Semi-supervised Learning; Change Metrics.**

I. Introduction

Software Defect Prediction is one of the most important software quality assurance techniques. Ensuring software quality is a complex and time-consuming activity. Defect prediction models can be used to direct test effort to defect-prone code. Latent defects can be detected in code before the system is delivered to users. Each year defects of code cost industry billions of dollars to find and fix. Models which predict defects efficiently have the potential to save companies a large amount of money. Since the costs are so huge, even a small improvement in ability to find and fix defects can make a significant difference to cost reduction.

Many defect prediction methods have been proposed in recent years. For example, Akiyama [1] proposes that the number of software defects in the early software development phase has a relation with the lines of code. The equation $D = 4.86 + 0.018L$ holds, showing that there are approximately 22.86 defects per thousand lines of code. Menzies et al. [2] build defect prediction models based on 21 static code attributes. Kim et al. [3] predict defects by caching locations that are adjusted by software change history data. Nagappan et al. [4] propose to extract principle components from metrics and use these principle components for defect prediction. Zimmermann and Nagappan [5] propose models based on the structural measurement of dependency graphs.

In the literature, metrics used in defect prediction can be divided into four categories: code complexity metrics, code churn information, change history, and structure of software development organizations. Different kinds of software metrics and machine learning methods (or statistical methods) elaborate to construct prediction model. The model is then used to predict defect-proneness for modules of which the fault content is unknown.

A. Motivation

Most of existing defect prediction methods are supervised, which assumes defective information of historical data is sufficient and available. As shown in figure 1, supervised-learning models are trained on labeled dataset, for which the defective or defect-free information are known. Intuitively, larger the size of the training set, better the accuracy of prediction. However, projects often lack such data during the early phase of software development. When the size of training dataset is small, prediction performance could be dramatically reduced. Thus, here exists a constraint on supervised learning - the size of labeled training dataset should be as large as possible.

Another limitation is that, most of existing techniques use static code metrics as predictors, but static code metrics can not show change information in software development. It has been believed that change information may bring serious risks to software artifacts [6]. Without change information, effective defect prediction is difficult to achieve.

Considering problems mentioned above, we propose a semi-supervised based defect prediction approach - ***extRF***. In particular, we try to answer the following questions:

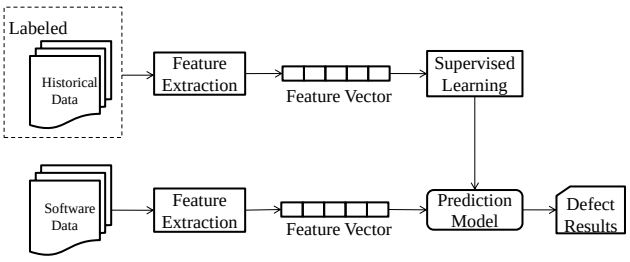


Fig. 1: Defect Prediction with Supervised Learning

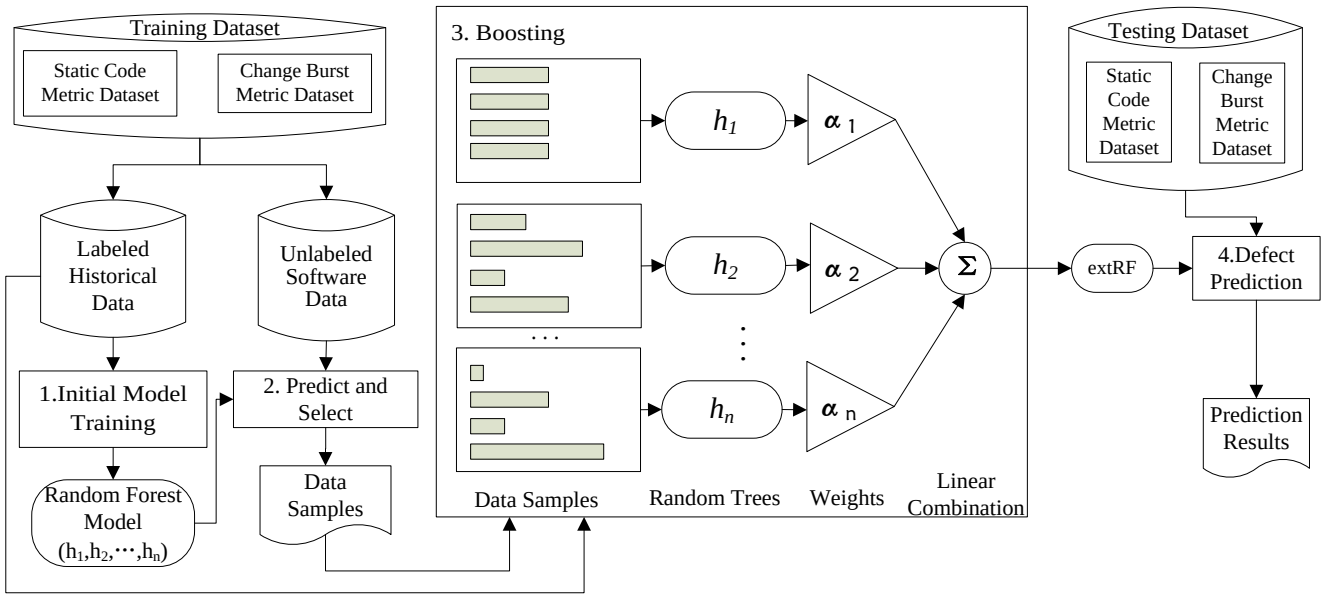


Fig. 2: Overview of Semi-supervised Approach - extRF

RQ_1: How to design an effective approach when few modules with defect content are known?

RQ_2: Can the semi-supervised learning approach with few labeled data samples achieve comparable results to the supervised learning approach with a larger set of labeled data samples?

RQ_3: Are change burst metrics more efficient than static code metrics for defect prediction?

B. Contribution

Unlike the defect prediction with supervised learning (see Figure 1), extRF is a semi-supervised learning approach to software defect prediction. It extends the classical supervised algorithm [7] named *Random Forest* [8] by incorporating the self-training paradigm [9]; it also employs change burst information for improving the accuracy of software defect prediction. As figure 2 shows, the process of extRF includes four steps. First, we sample a small percentage of modules from data set, and assumes their defective information are known (i.e. labeled dataset). These sampled data set are used to train a prediction model (Step 1). Then we use the model to predict the defectiveness of remaining unsampled data set, which are assumed as unlabeled data set (Step 2). The most confident samples are selected out, and used to refine the initial model (Step 3). Finally, the refined model is used to predict the defectiveness of new software modules (Step 4).

The contributions of our work can be summarized as the following three points:

- For determining which unlabeled samples are confident enough to label, extRF adopts the majority voting strategy from Random Forest.
- We introduce a boosting [10] process into extRF to decide the weight of each base learner. The final prediction result is produced by a weighted combination of component classifiers (i.e. random tree), instead of unweighted majority voting. Research from David Bowes et al. [11] has shown that ensembles not based on majority voting are likely to perform best.
- We also conduct a series of experiments for evaluating extRF and compare the effectiveness of code metrics and change burst metrics. Experimental results show that extRF trained with a small size of labeled dataset achieves comparable performance to some supervised learning approaches trained with a much larger size of labeled dataset. Especially, when only 2% of Eclipse **2.0** data are used for training, extRF achieves an F-measure value of 0.562, close to that of LR (a supervised learning approach) at labeled sample rate of 50%. Besides, change burst metrics outperform code metrics in that the corresponding F-measure achieves a peak value of 0.75 when applied to extRF.

Note that our extRF approach requires neither the data which should be described by sufficient defectiveness information, nor the special learning algorithms which frequently employ time-consuming cross validation in learning. Thus, extRF could be easily applied in Software Defect Prediction.

The rest of this paper is organized as follows: Section II presents methodology for our optimized Random Forest approach. In Section III, we describe the experiments and then analyse the results. In Section IV, we provide a general discussion of commonly used defect prediction techniques and metrics. Finally, conclusions and future work are offered in Section V.

II. METHODOLOGY

Our study aims to provide an effective and practical approach for identifying fault-prone files, when historical data with defective or defect-free information is not sufficient. In this section, we discuss the design and application of our approach.

In software quality prediction, Random Forest appears to offer consistently good performance across different data sets [12]. One of the reasons for good performance is its robustness to noise. But it only fits for supervised learning, in which large amounts of labeled data samples exist in training dataset. In practice, defect information of code files are often lacked in the early phase of software life cycle. To address this problem, we extend Random Forest into a semi-supervised setting.

We first clarify the notation. Let *X* be a $(n+m) \times p$ matrix that denotes software features. *n* is the size of labeled dataset *L* and *m* is the size of unlabeled dataset *U*. Specifically, $X = (X_l, X_u)^T$, where $X_l = (x_1, x_2, \ldots, x_n)^T$ and $X_u = (x_n + 1, x_n + 2, \ldots, x_{n+m})^T$. In addition, let the $(n+m) \times 1$ vector $Y = (Y_l, Y_u)^T$ be the labels where $Y_l = (y_1, y_2, \ldots, y_n)^T$ are known and $Y_u = (y_{n+1}, y_{n+2}, \ldots, y_{n+m})^T$ are unknown. The observed labels are binary variables, $y_i \in \{0, 1\}$, where 0 denotes the absence of faults and 1 denotes faulty modules.

Our approach is composed of four steps. First, we train a Random Forest model on *L*, and it is used to predict on unlabeled data set *U*. Different random trees have different prediction results. Second, we count the degree of agreement (denoted as *confidence*) for each data sample. Third, we select out *M* most confident data samples. These data samples are added into a new data set *L'* along with their *voting labels*. At last, a boosting procedure is conducted and a specific weight is set to each random tree.

A. Semi-supervised learning - extRF

Several researches have been proposed by extending Random Forest model to fit semi-supervised learning setting. M. Li et al. [13] propose a novel active semi-supervised learning method ACoForest which is able to sample the modules that are helpful for learning a good prediction model. They present an iteration process. For each random tree h_t , a concomitant ensemble H_{-t} is constructed to label the unlabeled dataset. Then the unlabeled data samples whose labeling confidences are above threshold θ will be used to refresh h_t. H. Lu et al. [14] propose a self-training approach, which initializes a Random Forest model on labeled data samples. They refine each base learner (i.e. random tree) through an iteration process, by expanding the labeled dataset from unlabeled dataset. Both of them have the following disadvantages:

- When the dataset is too large, the process of predicting the unlabeled data set in each iteration is very time-consuming.
- The ensemble result is just based on majority voting of each random tree. But in practice, different random tree may have different prediction power on unlabeled data set.

TABLE I: DESCRIPTION OF EXTRF ALGORITHM

Algorithm: extRF
Input: labeled data set L, unlabeled dataset U threshold θ, number k the number of random trees N
Output: weight α_t for each random trees
Procedure:
1: train a Random Forest model on L with N random trees $H = \{h_1, h_2, \ldots, h_T\}$
2: use current Random Forest model to predict U
3: find those data samples of which the confidence exceeds the threshold θ, and pick out top k of them
4: add the selected data samples along with their voting labels into L, to form a new data set L'
5: **Repeat until** each h_t in H has been set a weight α_t
6: initialize the weight vector of data samples in L'
7: **For** each random tree h_t in H,
8: update h_t with L'
9: calculate the error rate ε_t of h_t
10: calculate the weight α_t of h_t
11: update the weight vector of data samples in L'
12: **end For**
13: **End of Repeat**
14: **Return** weight α_t

Different from their approaches, we predict unlabeled data set only once by using the initial Random Forest model. Then we select the most confident data samples out. Table I describes details of our extRF approach. When deciding the confidence of a corresponding data sample, we use the majority voting schema. That is, if the number of voting for a specific label (0 or 1) exceeding a given threshold θ, then it is added into a new set along with the voting results.

After picking out these confident unlabeled data samples, we introduce a boosting process into Random Forest algorithm. A specific weight is then assigned for each random tree.

We combine the labeled dataset and the selected data samples along with their voting labels into a new data set $L' = \{X_{l'}, Y_{l'}\}$. Assume there are k data samples selected out, then there are total $n+k$ data samples in L'. We initialize the weight of each data sample in L' as

$$D_i^0 = \frac{1}{n+k}, \quad where\ i = \{1, 2, \ldots n+k\} \tag{1}$$

Then in t-th iteration, the random tree h_t is updated by the expanding data set L'. And the error rate of h_t is calculated by

$$\varepsilon_t = \sum_{i=1}^{n+k} D_i^t I(h_t(x_i) \neq y_i) \tag{2}$$

where $I(.)$ is an indicator function showing whether the predicting result is consistent with $Y_{l'}$.

The weight of random tree h_t is calculated based on the error rate.

$$\alpha_t = \frac{1}{2} ln(\frac{1-\varepsilon_t}{\varepsilon_t}) \tag{3}$$

Next we update weight of each data sample in L'.

$$D_i^{t+1} = \begin{cases} \frac{D_i^t e^{\alpha}}{\sum_{i=1}^{n+k} D_i^t} & if\ x_i\ is\ misclassified \\ \\ \frac{D_i^t e^{-\alpha}}{\sum_{i=1}^{n+k} D_i^t} & otherwise \end{cases}$$

We aim to increase the weights of misclassified data samples, and decrease the weights of data samples classified correctly. Each random tree in Random Forest is refined with these weighted data samples. By enhancing the influences of misclassified data samples, the accuracy of prediction model is presumably improved.

If each random tree has been set to a specific weight, the iteration procedure ends. The final strong classifier is a weighted linear combination of each weak classifiers:

$$H(x) = \sum_{t=1}^{T} \alpha_t h_t(x) \tag{4}$$

B. Ensemble learning

In our extRF approach, we introduce a boosting process into the well-known Random Forest algorithm, in order to assign a specific weight for each random tree.

Boosting and Random Forest all belong to the machine learning setting called *Ensemble Learning* [7]. Ensemble learning paradigms train multiple weak classifiers and then combine their predictions. Ensemble techniques can significantly improve the generalization ability of single classifiers.

An ensemble is usually built in two steps. The first step is to generate multiple component classifiers, and the second step is to combine their predictions. According to the way to generate component classifiers, ensemble learning algorithms fall into two categories – algorithms that generate component classifiers in parallel, and algorithms that generate component classifiers in sequence. Bagging [15] is one representative of the first category. It generates each classifier on an example set bootstrap [16] sampled from the original training set in parallel, and then combines their predictions using majority voting. Other well-known algorithms in the first category include Random Subspace [17] and Random Forest [8]. In the second category, the representative algorithm is AdaBoost [18] , which sequentially generates a series of classifier focus on the training examples that are misclassified by the former classifiers. Other well-known algorithms in the second category include Arc-x4 [19] and LogitBoost [20].

Our extRF approach first trains a Random Forest model on labeled data samples. And then we use the model to predict unlabeled data samples. The most confident data samples are selected out to refine each random tree in Random Forest. Each weak classifier (ie. random tree) in Random Forest is assigned a same weight. But corresponding to classification accuracy, they vary from each other. A simple linear combination with equal weights of them may not improve prediction performance of the ensemble effectively. In order to enhance the influence of those component classifiers with high predictive power, we introduce a boosting process into initial model. Each component classifier is assigned a specific weight. Thus, the random tree with high accuracy has stronger impact on final prediction model.

C. Base learner of the ensemble

One important factor in ensemble learning is how to choose the base learner. The performance improvement of each component classifier does not necessarily lead to the performance improvement of the ensemble. According to Krogh and Vedelsby [21], an ensemble exhibits its generalization power when the average error rate of component classifiers is low, and the diversity between component classifiers is high. To obtain good performance, the diversity between component classifiers should be maintained when the ensemble exploits the unlabeled data.

To maintain the diversity in the semi-supervised learning process, a well-known ensemble method named *Random Forest* [8] is used to construct base learners in extRF. Since Random Forest injects certain randomness in the tree learning process, any two trees in the Random Forest could still be diverse even if their training data are similar.

III. Evaluation and Results Analysis

We have performed a series of comparative experiments on Eclipse data set. Results show that extRF is superior to traditional supervised approach on three data sets (except SWT), when little labeled historical data is available. ExtRF with few labeled data samples performs comparable to that of supervised model trained with a larger labeled data set. Besides, change burst model outperforms code model among all the four defect prediction approaches. Especially when we combine extRF with change burst metrics, accuracy increases extremely high.

A. Experiment Setting

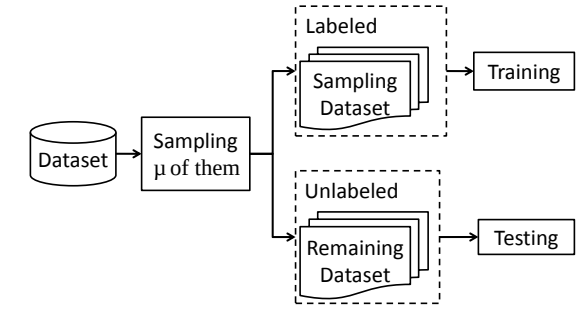
We adopt three widely used machine learners: Logistic Regression, Naïve Bayes, and Random Forest. To evaluate good performance of change bursts metrics and our approach extRF, we conduct a series of comparative experiments.

Table II shows the combination of different prediction models and different metrics.

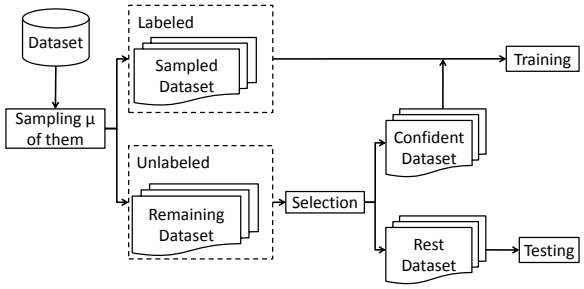
Next we illustrate sample-based evaluation process of supervised and semi-supervised defect prediction, as Figure 3 shows.

TABLE II: Combination of Different Prediction Models and Different Metrics

	Code Metrics	**Change Burst Metrics**	**Composed Metrics**
LR	CM×LR	CBM×LR	CompM×LR
NB	CM×NB	CBM×NB	CompM×NB
RF	CM×RF	CBM×RF	CompM×RF
extRF	CM×extRF	CBM×extRF	CompM×extRF



(a) Prediction with Supervised Learners



(b) Prediction with Semi-Supervised Learners

Fig. 3: Sample-based Evaluation Process of Supervised and Semi-Supervised Prediction

1) Prediction with supervised learners: We first perform experiments to evaluate the effectiveness of the defect prediciton models constructed using randomly selected samples and supervised machine learners. For each data set, we randomly sample a small portion of modules as the labeled training set according to a sampling rate μ, while the remaining data samples are used as unlabeled and testing set. Here we use five different sampling rates $\{2\%, 5\%, 10\%, 25\%, 50\%\}$, in order to explore influence of different sampling intensity. The labeled data set is used to initiate machine learners. The remaining unlabeled data set is taken as testing set. We repeat the sampling process for 100 times, and the average performance of the compared method is recorded.

2) Prediction with semi-supervised learners: Then we perform experiments to evaluate the effectiveness of the defect prediction models constructed using random sampling and semi-supervised learning. Following the same experimental setting described above, for each data set, we use five different sampling rate $\{2\%, 5\%, 10\%, 25\%, 50\%\}$. The labeled data set is used to initiate Random Forest. Part of the unlabeled data will be selected out, and then the remaining unlabeled data set is taken as testing data set. For each of the 20 experimental settings (4 data sets $\times$ 5 sampling rates), we repeat the sampling process for 100 times and the average results are recorded.

B. Dataset

We perform our experiments on the Eclipse platform. Eclipse is a popular open source system that has been extensively studied before. In the experiment, we focused on the metrics on two versions (Eclipse 2.0 and 3.0) and two components of version 3.0 (*org.eclipse.jdt.core* and *org.eclipse.swt*). We download the repository from CVS[1] and the bug reports from Bugzilla[2]. Table III shows the defects information in these datasets.

TABLE III: DEFECTIVE INFORMATION OF DATASET

Data set	Instances	Defective Modules
Eclipse 2.0	6729	2611 (38.80%)
Eclipse 3.0	9740	7948 (81.6%)
JDT.Core	939	502 (53.46%)
SWT	843	208 (24.67%)

Metrics used in this experiment are shown in below:

1) Code Metrics: Source code downloaded from CVS are preprocessed, and only files with defect history in Bugzilla are left and tracked. Understand[3] tool is used to extract code metrics. Table IV(a) shows some common file-level metrics used in our experiment.

2) Change Burst Metrics: Nachiappan Nagappan et al. [22] introduce the concept of change bursts and extract them from a series of changes. A change burst is a sequence of consecutive changes. It is defined by two parameters – gap size and burst size. Figure 4 illustrates how to determine a change burst. Gap size is the minimum distance between two changes. Burst Size is the minimum number of changes in a burst. Increasing gap size yields longer bursts, and increasing burst size eliminates shorter bursts. They conducted an empirical study on Windows Vista, and found that the features of such change bursts have the highest predictive power for defect-prone components. They also provided an open data set[4] for four versions of Eclipse, but have not analysed these data so far. The data set contains all the necessary change burst metrics. In this paper, we use the data set to evaluate predictive power of change burst metrics on Eclipse. Table IV(b) list some representative change burst metrics used in our experiment.

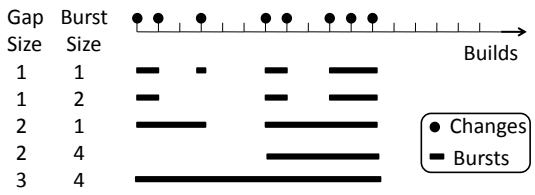


Fig. 4: How Gap Size and Burst Size Determine Change Burst from A Sequence of Changes

[1]http://archive.eclipse.org/arch/
[2]https://bugs.eclipse.org/bugs/
[3]https://scitools.com/
[4]https://github.com/kimherzig/change_burst_data.git

TABLE IV: SOME REPRESENTATIVE METRICS USED IN EXPERIMENTS

(a) FILE-LEVEL CODE METRICS

Understand Metrics	Code Metrics
SumCyclomatic	WMC
MaxInheritanceTree	DIT
PercentLackOfCohesion	LCOM
CountOutput	FanOut
CountLineCode	LOC
CountInput	FanIn
CounDeclMethodAll	RFC
CountClassDerived	NOC
CountClassCoupled	CBO
CountDeclClassMethod	NOM
CountDeclClassVariable	NOA

(b) CHANGE BURST METRICS

Change Burst Metrics	Brief Description
NumberOfChanges	number of builds in which the file has changed
NumberOfConsecutiveChanges	number of consecutive for a given file
NumberOfChangeBursts	number of change bursts for a given file
TotalBurstSize	number of changed builds in all change bursts
MaximumChangeBurst	maximum number of builds in all change bursts
NumberOfChangesEarly	compute number of changes for early periods
NumberOfChangesLate	compute number of changes for early periods
TimeFirstBurst	when the first burst occurred
TimeLastBurst	when the last burst occurred
TimeMaxBurst	when the greatest burst occurred
PeopleTotal	number of people who ever committed a change
TotalPeopleInBurst	total number of people involved across all bursts
MaxPeopleInBurst	maximum number of people involved in a burst
ChurnTotal	total churn over the lifetime of a file
TotalChurnInBurst	total churn in all change bursts
MaxChurnInBurst	maximum churn across all bursts

C. Evaluation Measures

In the experiment, we employ three commonly used evaluation measures to evaluate the performance of defect prediction models, including precision, recall and F-measure. These measures can be defined by using A, B, C and D in Table V. Here, A, B, C and D are the number of defective modules that are predicted as defective, the number of defective modules that are predicted as defect-free, the number of defect-free modules that are predicted as defective, and the number of defect-free modules that are predicted as defect-free, respectively.

The recall rate is defined as A/(A + B). It denotes the ratio of the number of defective modules that are correctly classified as defective to the total number of defective modules. The precision evaluates the correct degree of prediction model and is defined as A/(A + C). Obviously, a good prediction model desires to achieve high value of recall rate and precision. However, there exists a tradeoff between them. Therefore, a comprehensive measure of recall rate and precision is necessary. F-measure is the harmonic mean of mean of recall rate and precision, which is defined as:

$$F\text{-}measure = \frac{2 \times recall \times precision}{recall + precision} \tag{5}$$

All the above evaluation measures range from 0 to 1. Obviously, an ideal defect prediction model should hold high values of recall rate and F-measure. In the experiment, we evaluate the performances of all defect prediction models in terms of recall, precision and F-measure values. It is noted that we do not specially report results with respect to the precision measure since it has been included in the comprehensive F-measure.

TABLE V: CONFUSION MATRIX OF DEFECT PREDICTION RESULTS

	Predict as Defective	Predict as Defect-free
Defective Modules	A	B
Defective-free Modules	C	D

D. Result and Analysis

Table VI illustrates the performance of Naïve Bayes, Logistic Regression, Random Forest, and extRF on change burst metrics over all dataset. When different values of labeled data sampling rate μ are set, F-measures of the same prediction approach on a dataset are different. The best performance in terms of F-measure on each dataset is boldfaced. It can be easily observed that, extRF performs significantly better than Naïve Bayes, Logistic Regression, and Random Forest on Eclipse 2.0, Eclipse 3.0, and JDT.Core in terms of all the five sampling rates. Besides, it is noted that Random Forest performs best among all the three supervised learning methods. With respect to SWT dataset, it takes on a new characteristic. Naïve Bayes achieves the best performance among all the labeled data sampling rate. It may be caused by that the defective modules in SWT are too few. The imbalance in data distribution may have bad influence on accuracy of prediction model.

Figure 5 further depicts the change trend of prediction model performance. It can be easily observed that increasing labeled data sampling rate μ may only result in marginal performance improvement for extRF. Taking Eclipse 3.0 as an example, the value of F-measure has no significant change when we increase sampling rate μ from 2% to 10%. If we further increase sampling rate to 50%, the value of F-measure

TABLE VI: F-MEASURES OF DIFFERENT CLASSIFIERS USING CODE METRICS

Data Set	μ	Classifier			
		NB	LR	RF	extRF
Eclipse 2.0	2%	0.432	0.535	0.508	**0.562**
	5%	0.43	0.538	0.517	**0.556**
	10%	0.426	0.542	0.523	**0.573**
	25%	0.413	0.558	0.55	**0.578**
	50%	0.409	0.563	0.557	**0.595**
Eclipse 3.0	2%	0.690	0.675	0.694	**0.748**
	5%	0.702	0.680	0.702	**0.762**
	10%	0.698	0.677	0.719	**0.757**
	25%	0.680	0.690	0.726	**0.784**
	50%	0.673	0.714	0.739	**0.797**
JDT.Core	2%	0.667	0.611	0.684	**0.719**
	5%	0.685	0.681	0.614	**0.716**
	10%	0.688	0.693	0.626	**0.725**
	25%	0.659	0.695	0.644	**0.731**
	50%	0.652	0.706	0.68	**0.75**
SWT	2%	**0.605**	0.53	0.426	0.551
	5%	**0.599**	0.536	0.421	0.558
	10%	**0.602**	0.557	0.436	0.572
	25%	**0.597**	0.56	0.468	0.589
	50%	**0.603**	0.577	0.553	0.592

only increases slightly about 0.029. As for JDT.Core, F-measure has no dramatically improvement when we increase sampling rate μ from 2% to 25%. If we further increase μ to 50%, F-measure rises a little. Another interesting phenomenon is that, when we increase labeled data samples rate on Naïve Bayes, the F-measure lowers down on Eclipse 2.0, Eclipse 3.0, and SWT. For JDT.Core, performance of Naïve Bayes has a little improvement when labeled data sampling rate μ increases from 2% to 10%. But when we continue to increase μ to 25%, F-measure has a sharp decrease. What's more, F-measure of Logistic Regression keeps increasing when sampling rate μ rises from 2% to 50%. Especially for SWT data set, F-measure grows like an upward parabolic curve.

Figure 6 is drawn by extracting out F-measures of extRF at $\mu = 2\%$ and F-measures of supervised approaches at $\mu = 50\%$. It is noted that even when labeled data sampling rate is very low, extRF still obtains comparable prediction performance to traditional supervised learning methods. Especially for JDT.Core and Eclipse 3.0, the result of extRF is even better in contrast with other three supervised learning methods.

To further analyze predictive power of static code metrics and change burst metrics, we design comparative experiments on Eclipse 3.0 data set. Results are shown in figure 7. All of the four defect prediction approaches are analysed. Obviously change burst model have higher prediction accuracy than that of static code model. Especially for extRF approach, change burst model improve the prediction accuracy to a large extent

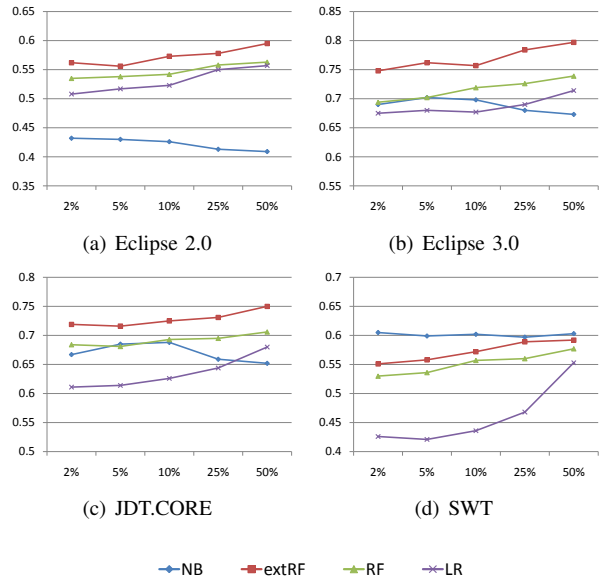


Fig. 5: F-measures of extRF and Compared Models at Different Sampling Rate

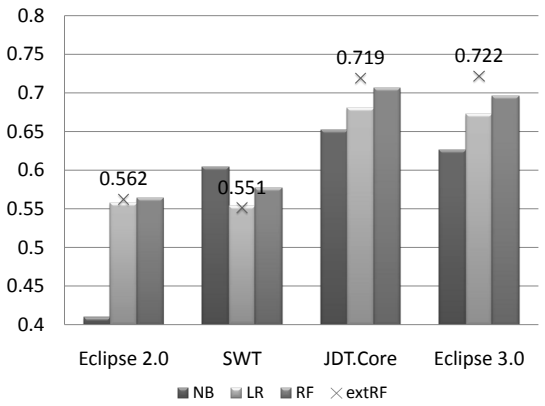


Fig. 6: F-measures of extRF at $\mu = 2\%$ and Compared Models at $\mu = 50\%$

in comparison with static code model. Besides, if we combine code metrics and change burst metrics, there is no significant improvement. Thus, there is no need using such a large composed metrics for defect prediction.

Figure 8 shows the box-plots for F-measures obtained from 100 trials with sampling rate $\mu = 2\%$ using extRF. In descriptive statistics, a box-plot is a convenient way of graphically depicting groups of numerical data through their quartiles. It also has lines extending vertically from boxes that indicates variability outside upper and lower quartiles. For all the four data sets, the inter-quartiles are all narrow (less than 0.090). Especially for Eclipse 3.0 and JDT.Core, the inter-quartile is 0.035 and 0.030 respectively. The ranges between upper tails and lower tails of box-plots are all less than 0.195. The box-plots demonstrate that results of all the 100 trials on such experimental setting distribute centrally around the

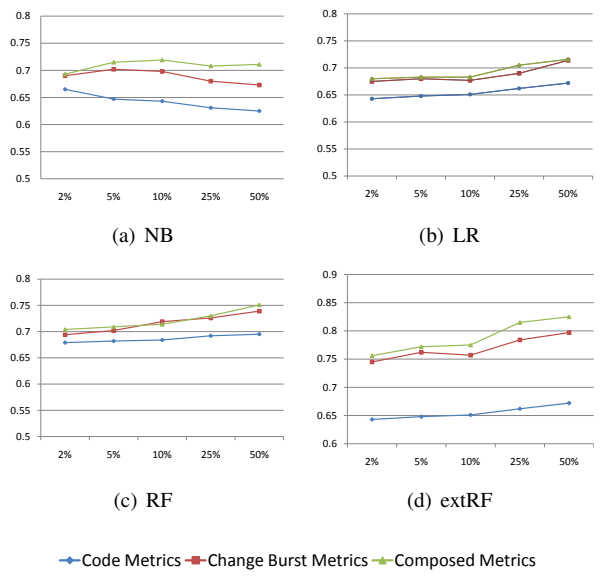


Fig. 7: F-measures of Code Metrics and Change Burst Measures at Different Sampling Rate on Eclipse 3.0

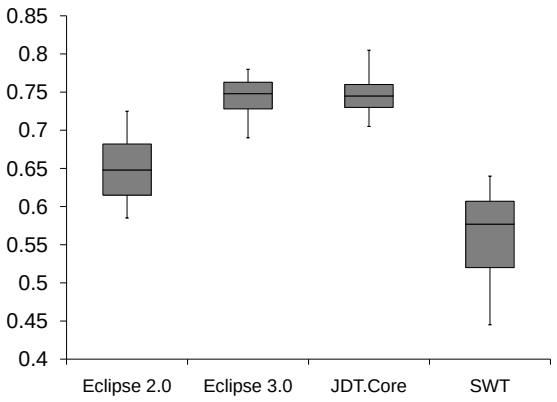


Fig. 8: Box-plots of F-measures obtained from 100 Trials on Change Burst Metrics with Sampling Rate $\mu = 2\%$ using extRF

median value. That is to say, our result is confident.

In summary, our results confirm that the proposed approach extRF can achieve better defect prediction performance in situations when modules with defect content are too few.

E. Answers to Research Questions

RQ_2: Can the semi-supervised learning approach with few labeled data samples achieve comparable results to the supervised learning approach with a larger set of labeled data samples?

From tables and figures above, we can conclude that our extRF approach is appropriate to be applied to the situation where historical data is not sufficient in software defect prediction. For training data set with partially labeled, extRF can still achieve a high prediction accuracy, while traditional supervised learning approaches fail to predict defects effectively.

RQ_3: Are change burst metrics more efficient than static code metrics for defect prediction?

From figure 7, it shows that change burst metrics have much more predictive power than code metrics. Specifically, a combination of our extRF approach with change burst metrics could improve prediction accuracy to a large extent. Figure 8 further demonstrates that our results are convincible and confident.

F. Threats to Validity

Followings are several potential threats to the validity with respect to the experiments:

1) Bias of Evaluation Measures: In our research we only employ the widely used F-measure to report the performance of defect prediction. Other measures, such as area under curve (AUC) and g-measure (harmonic mean of pd and 1-pf) are not used. They are also comprehensive measures.

2) Comparison Accuracy: We only conduct comparison experiments with traditional supervised approach, but have not implemented effective comparison experiments with other existing semi-supervised approach. Some other researchers have proposed several semi-supervised approaches to defect prediction. For example, H. Lu et al. [14] propose a self-training approach on NASA data set, and M. Li et al. [13] propose an active semi-supervised learning method on Lucene, XALAN, and Eclipse data set. Since the authors of these semi-supervised methods do not provide the program codes, we can not conduct effective comparison experiments with them.

IV. Related Work

A. Semi-supervised Learning

In traditional supervised learning, all training data should be labeled before learning, and classifiers are then trained on these labeled data. When a portion of training data samples are unlabeled, and effective way to combining labeled and unlabeled data in learning is known as semi-supervised learning [23]–[25] , where an initial hypothesis is firstly learned from the labeled data and then refined through the unlabeled ones labeled by certain automatic labeling strategy.

There have been many semi-supervised approach adopted to defect prediction. Khoshgoftaar et al. [26] utilize EM-based semi-supervised learning algorithm [23] in this domain. It estimates the parameters of a generative model and the probability of unlabeled examples being in each class. Their work shows that EM-based semi-supervised software quality modeling provides better performance compared to a standard supervised learning approach using C4.5. N. Seliya and T.M. Khoshgoftaar [27] propose a semi-supervised clustering approach. It extends traditional unsupervised learning (clustering) into semi-supervised context so that better partitions (or grouping) is achieved with the use of unlabeled data. It improves the performance compared to the corresponding unsupervised learning, but does not perform as well as supervised learning. Besides, it is not an entirely automated approach and

requires software engineering experts in the loop. Hence its not likely that semi-supervised clustering is a good candidate for practical applications.

Both semi-supervised approaches above follow an *inductive* learning strategy [23], of which the model is built from training data and then is used to predict on test data (remaining components, new versions of similar projects). Different from above methods, Huihua Li et al. [14] implement a *tansductive* [23] semi-supervised learning approach, which predicts the labels of unlabeled data, and then uses the data to supplement the model. Results show that the approach perform significantly better than one of the best performing supervised learning algorithm - Random Forest - in situations when few modules with known fault content are available. Ming Li et al. [13] present a defect prediction approach which extends Random Forest into semi-supervised learning setting. In their method, random forest is trained using initially labeled modules. Each random tree is then iteratively refined with the original labels and the labels which are assigned to previously unlabeled modules. When the stop criteria is reached, the majority voting from the ensemble forms the prediction.

In this paper, we present a new transductive semi-supervised approach to defect prediction. But since there exists difference in the evaluation strategy, and the authors do not provide the program codes, it is impractical to compare the results of our approach with semi-supervised approaches mentioned above. However, we can compare our results with the literature on supervised learning, as in both cases only the predicted labels for unlabeled modules form the basis of evaluation.

B. Change Information

Despite the significant effort spent in research and practice, the relationship between software defects and the various software artifacts are still unknown. Regarding predictor variables, we can identify three different approaches: product-centric, process-centric, and a combination of both.

The most studied approach is to relate software defects to the product itself: measures of static or dynamic structure of source code. Schroter et al. [28] investigate the usage relationship between software components and software defects, and find it effective for predicting the most defect-prone components for the Eclipse project. Zimmermann et al. [29] extract defects from bug database of the Eclipse project, and additionally annotate such data with a vast amount of size and complexity metrics extracted from source code. They find a significant correlation between complexity metrics and pre- and post-release defects. Moreover, they use logistic regression models for predicting successfully defects at a package level. Menzies et al. [2] propose the Naïve Bayes learner and a feature selection method based on information theory and obtain very accurate results for defect prediction on the MDP repository for NASA projects. They conclude that there exists no best set of code metrics for defect prediction but such set rather depends on the characteristics of single data sets, feature selection methods, and machine learners. Nagappan et al. [4] arrive at similar conclusion, and they suggest a general methodology for selecting relevant complexity metrics for a given data set and creating a defect prediction model using such metrics.

The second branch in defect prediction research aims at relating various process artifacts such as change history of source files, changes in the team structure, testing effort, or technology and other human factors to software defects. Graves et al. [30] predict fault incidences using software change history based on a weighted time stamp model using the sum of contributions from all changes to a module, where large and/or recent changes contribute the most to fault potential. Moser et al. [6] use *change metrics* such as the number of revisions or refactoring to predict defects in Eclipse classes. They report that cost-sensitive classification yields more than 75% of correctly classified files and a recall of more than 80%. Hassan [31] measures the complexity of the change process by assessing how much modifications are scattered across space and time. The resulting entropy metrics are evaluated to be better predictors than prior faults.

V. Conclusion and Future Work

In this paper, we propose a semi-supervised approach extRF to defect prediction. It extends Random Forest into a semi-supervised ensemble learning, refining each random tree based on self-training paradigm. A boosting process is introduced, and the final prediction result is produced by a weighted combination of random trees. The extRF approach makes defect prediction available even when defective information of historical data are not sufficient. Our experiments are conducted on Eclipse data set. We focus the metrics on two versions (Eclipse 2.0 and 3.0), and two components of version 3.0 (JDT.Core and SWT). Experiment results show that extRF trained with a small size of labeled dataset achieves comparable accuracy to that of supervised approach trained with a larger size of labeled dataset. When employing extRF on change burst metrics, defect prediction achieves a best performance with F-measure of 0.75.

In the future, we would like to employ more data set to validate the generalization ability of our approach. We will also evaluate effectiveness of our solution for semi-supervised setting with other approaches.

References

[1] F. Akiyama, "An example of software system debugging.," in *IFIP Congress (1)*, pp. 353–359, 1971.

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

[3] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*, pp. 489–498, IEEE Computer Society, 2007.

[4] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*, pp. 452–461, ACM, 2006.

[5] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*, pp. 531–540, ACM, 2008.

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

[7] T. G. Dietterich, "Ensemble learning," *The handbook of brain theory and neural networks*, vol. 2, pp. 110–125, 2002.

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

[9] C. Rosenberg, M. Hebert, and H. Schneiderman, "Semi-supervised self-training of object detection models," 2005.

[10] Y. Freund, R. Schapire, and N. Abe, "A short introduction to boosting," *Journal-Japanese Society For Artificial Intelligence*, vol. 14, no. 771-780, p. 1612, 1999.

[11] D. Bowes, T. Hall, and J. Petrić, "Different classifiers find different defects although with different level of consistency," in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, p. 3, ACM, 2015.

[12] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pp. 417–428, IEEE, 2004.

[13] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou, "Sample-based software defect prediction with active and semi-supervised learning," *Automated Software Engineering*, vol. 19, no. 2, pp. 201–230, 2012.

[14] H. Lu, B. Cukic, and M. Culp, "A semi-supervised approach to software defect prediction," in *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pp. 416–425, July 2014.

[15] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[16] B. Efron and R. J. Tibshirani, *An introduction to the bootstrap*. CRC press, 1994.

[17] T. K. Ho, "The random subspace method for constructing decision forests," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 20, pp. 832–844, Aug 1998.

[18] Y. Freund and R. E. Schapire, "A desicion-theoretic generalization of on-line learning and an application to boosting," in *Computational learning theory*, pp. 23–37, Springer, 1995.

[19] L. Breiman, "Bias, variance, and arcing classifiers," 1996.

[20] J. Friedman, T. Hastie, R. Tibshirani, *et al.*, "Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors)," *The annals of statistics*, vol. 28, no. 2, pp. 337–407, 2000.

[21] A. Krogh, J. Vedelsby, *et al.*, "Neural network ensembles, cross validation, and active learning," *Advances in neural information processing systems*, vol. 7, pp. 231–238, 1995.

[22] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pp. 309–318, Nov 2010.

[23] O. Chapelle, B. Schölkopf, A. Zien, *et al.*, "Semi-supervised learning," 2006.

[24] M. Seeger, "Learning with labeled and unlabeled data," tech. rep., 2000.

[25] X. Zhu, "Semi-supervised learning literature survey," Tech. Rep. 1530, Computer Sciences, University of Wisconsin-Madison, 2005.

[26] N. Seliya and T. M. Khoshgoftaar, "Software quality estimation with limited fault data: a semi-supervised learning perspective," *Software Quality Journal*, vol. 15, no. 3, pp. 327–344, 2007.

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

[28] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pp. 18–27, ACM, 2006.

[29] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pp. 9–9, IEEE, 2007.

[30] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *Software Engineering, IEEE Transactions on*, vol. 26, no. 7, pp. 653–661, 2000.

[31] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 78–88, IEEE Computer Society, 2009.