TLEL: A two-layer ensemble learning approach for just-in-time defect prediction

Xinli Yang\textsuperscript{a}, David Lo\textsuperscript{b}, Xin Xia\textsuperscript{a,*}, Jianling Sun\textsuperscript{a}

\textsuperscript{a}College of Computer Science and Technology, Zhejiang University, Hangzhou, China
\textsuperscript{b}School of Information Systems, Singapore Management University, Singapore

A R T I C L E   I N F O

Article history:
Received 2 January 2016
Revised 13 February 2017
Accepted 18 March 2017
Available online 22 March 2017

Keywords:
Ensemble learning
Just-in-time defect prediction
Cost effectiveness

A B S T R A C T

Context: Defect prediction is a very meaningful topic, particularly at change-level. Change-level defect prediction, which is also referred as just-in-time defect prediction, could not only ensure software quality in the development process, but also make the developers check and fix the defects in time [1].

Objective: Ensemble learning becomes a hot topic in recent years. There have been several studies about applying ensemble learning to defect prediction [2–5]. Traditional ensemble learning approaches only have one layer, i.e., they use ensemble learning once. There are few studies that leverage ensemble learning twice or more. To bridge this research gap, we try to hybridize various ensemble learning methods to see if it will improve the performance of just-in-time defect prediction. In particular, we focus on one way to do this by hybridizing bagging and stacking together and leave other possibly hybridization strategies for future work.

Method: In this paper, we propose a two-layer ensemble learning approach TLEL which leverages decision tree and ensemble learning to improve the performance of just-in-time defect prediction. In the inner layer, we combine decision tree and bagging to build a Random Forest model. In the outer layer, we use random under-sampling to train many different Random Forest models and use stacking to ensemble them once more.

Results: To evaluate the performance of TLEL, we use two metrics, i.e., cost effectiveness and F1-score. We perform experiments on the datasets from six large open source projects, i.e., Bugzilla, Columba, JDT, Platform, Mozilla, and PostgreSQL containing a total of 137,417 changes. Also, we compare our approach with three baselines, i.e., Deep-er, the approach proposed by us [6], DNC, the approach proposed by Wang et al. [2], and MKEL, the approach proposed by Wang et al. [3]. The experimental results show that on average across the six datasets, TLEL could discover over 70% of the bugs by reviewing only 20% of the lines of code, as compared with about 50% for the baselines. In addition, the F1-scores TLEL can achieve are substantially and statistically significantly higher than those of three baselines across the six datasets.

Conclusion: TLEL can achieve a substantial and statistically significant improvement over the state-of-the-art methods, i.e., Deep-er, DNC and MKEL. Moreover, TLEL could discover over 70% of the bugs by reviewing only 20% of the lines of code.

© 2017 Elsevier B.V. All rights reserved.

1. Introduction

To produce high-quality software, much effort needs to be invested to the process of testing and debugging. Unfortunately, developers often have limited resource and tight schedule, and are thus constrained to perform rigorous and comprehensive testing and debugging efforts on all parts of a code base. Defect prediction techniques are proposed to help prioritize software testing and debugging efforts; they can recommend software components that are likely to be defective to developers. Much research has been done on defect prediction; these techniques construct predictive classification models built on features such as lines of code, code complexity and number of modified files [7–9].

Many past defect prediction studies predict defects at coarse granularity level, such as file, package, or module [9–11]. In recent years, several research studies propose just-in-time defect prediction techniques that are able to predict defective changes (i.e., commits to a version control system) [1,12]. Just-in-time defect prediction is more practical because it can not only ensure software quality in the development process, but also make the developers check and...
fix the defects just at the time they are introduced. The advantage of just-in-time defect prediction includes: (1) it leads to smaller amount of code to be reviewed because only individual changes (rather than entire files or packages) need to be reviewed [13]; (2) it leads to an easier assignments of developers to fix bugs because we can easily identify the authors of the changes that introduce defects. In a recent work, Kamei et al. perform a large-scale empirical study on just-in-time defect prediction [1].

Ensemble learning becomes a hot topic in recent years. Many research studies have shown that ensemble learning can achieve much better classification performance than a single classifier [14,15]. There have been several studies about applying ensemble learning to defect prediction [2–5]. However, most ensemble learning approaches only use ensemble learning once. There are rare studies that leverages ensemble learning twice or more [16]. We notice that different ensemble learning methods are good for different datasets. Therefore, we assume that hybrid ensemble learning will improve the performance of just-in-time defect prediction much more. In particular, we focus on one way to do this by hybridizing bagging and stacking together. We leave other possibly hybridization strategies for future work.

We propose a novel approach TLEL. The approach can be seen as a two-layer ensemble learning technique. In the inner layer, we use bagging based on decision tree to build a Random Forest model. In the outer layer, we use stacking to ensemble many different Random Forest models.

To evaluate TLEL, we use two widely-used evaluation metrics: cost effectiveness [17–20], and F1-score [12,17,21,22]. Cost effectiveness evaluates prediction performance considering a given cost threshold, e.g., a certain percentage of code to inspect. For example, when a team has limited resources to inspect potentially buggy lines of code, it is crucial that by manually inspecting the top percentages of lines that are likely to be buggy, developers can discover as many bugs as possible. We measure cost effectiveness as the percentage of bugs that can be discovered by inspecting the top 20% LOC based on the confidence levels that a change classification technique outputs (PofB20) [1,8]. In addition, we also evaluate our method using the F1-score [12,17,21,22], which is a summary measure that combines both precision and recall. F1-score is a good evaluation metric when there is enough resource to inspect all predicted buggy changes. A higher F1-score usually means a better method for just-in-time defect prediction.

We perform experiments on six large-scale software projects from different communities, i.e., Bugzilla, Columba, JDT, Mozilla, Platform, and PostgreSQL containing a total of 137,417 changes. We compare our approach with three baselines, i.e., Deeper, the approach proposed by us [6], DNC, the approach proposed by Wang et al. [2], and MKEL, the approach proposed by Wang et al. [3]. The experimental results show that on average across the six projects, TLEL could discover over 70% of the bugs by reviewing only 20% of the lines of code, as compared with about 50% for the baselines. Also, TLEL can achieve F1-scores of 0.25-0.67, which are substantially and statistically significantly higher than those of the baselines.

The main contributions of this paper are:

1. We propose a novel approach TLEL, which can be seen as a two-layer ensemble learning technique, to achieve a better performance for just-in-time defect prediction problem.
2. We compare TLEL with three baselines, i.e., Deeper, DNC and MKEL, on six large software projects. The experiment results show that our approach can achieve a better performance than all of them.

The rest of our paper is organized as follows. Section 2 introduces the background of our work. Section 3 presents the overall framework of our approach and elaborates the techniques that we use in our approach. Section 4 describes our experiments and the results. Section 5 presents some discussions about our work. Section 6 discusses the related work. Conclusion and future work are presented in the last section.

2. Preliminaries and motivation

In this section, we first introduce the general method of just-in-time defect prediction in Section 2.1. Next, we introduce ensemble learning in Section 2.2. Technical motivation will be presented at last.

2.1. Just-in-time defect prediction

Just-in-time defect prediction aims to predict if a particular file involved in a commit (i.e., a change) is buggy or not. Traditional just-in-time defect prediction techniques typically follow the following steps:

1. **Training Data Extraction.** For each change, label it as buggy or clean by mining a project’s revision history and issue tracking system. Buggy change means the change contains bugs (one or more), while clean change means the change has no bug.
2. **Feature Extraction.** Extract the values of various features from each change. Many different features have been used in past change classification studies. The features include change diffusion (which represents the number of files a change involves), change size (which represents the number of lines of code churned in a change), change purpose (which represents whether a change is a defect fix) and so on [1].
3. **Model Learning.** Build a model by using a classification algorithm based on the labeled changes and their corresponding features.
4. **Model Application.** For a new change, extract the values of various features. Input these values to the learned model to predict whether the change is buggy or clean.

2.2. Ensemble learning

Ensemble learning becomes more and more popular in recent years. Generally, different classifiers have many different characteristics, such as the intrinsic principle and the sensitivity to different training data. It is likely that different classifiers make different predictions for the same data. Ensemble learning can improve the classification performance by combining the predictions of multiple different classifiers into a single robust prediction [14,15]. The two key parts of ensemble learning are base learners and ensemble methods. There are many classification techniques that can be used as base learners such as support vector machine, decision tree [23]. Also, there are mainly three ensemble methods, i.e., bagging, boosting and stacking [24].

In this paper, for the ensemble methods, we use both bagging and stacking to create a two-layer ensemble learning approach. Bagging, also referred to as bootstrapped aggregation, can reduce the variance of the prediction [24]. In bagging, data are sampled uniformly from the original training data set with replacement, so that different sets of sampled data lead to different models even if the algorithms of the models are the same. Eventually, the class of the majority vote from the different models becomes the final prediction label. Stacking is a very general ensemble learning approach, in which two levels of classification are used [24]. In the first level, several different classifiers are trained based on the training dataset. In the second level, a final classifier is trained based on the output of the first-level classifiers.

For the base learner, almost all the classification techniques can be used. However, different techniques have different theoretical
basis so that suitable to different problems. Here we introduce five of the popular classification techniques mentioned above. We select one with best performance as base classifier from them.

1. **Naive Bayes.** Naive Bayes is a probabilistic model based on Bayes theorem for conditional probabilities [23]. Naive Bayes assumes the feature variables are independent of one another. The simplification can quantify the relationship between the feature variables and the target labels as a conditional probability much easier.

2. **Support Vector Machine.** Support Vector Machine (SVM) is developed from traditional linear models [23]. As with all traditional linear models, it uses a separating hyperplane as the decision boundary to differentiate two classes. However, traditional linear models only consider empirical error, while SVM considers structural error which includes both empirical error and confidence error. Therefore, the separating hyperplane achieved by SVM has maximum margin between two classes, which makes SVM one of the best classifiers.

3. **Decision Tree.** Decision Tree is modeled with the use of a set of hierarchical decisions on the feature variables, arranged in a tree-like structure [23]. In the tree-constructing process, Decision Tree can rapidly find the feature variables that differentiate different classes the most. In addition, it can generate explicit rules for different classes, while many other classifiers can not.

4. **Linear Discriminant Analysis.** Linear Discriminant Analysis (LDA) is similar to Principle Component Analysis (PCA) in that they both look for a linear combination of feature variables that best explains the data [23]. However, PCA doesn’t consider differences between classes, while LDA attempts to model the difference and uses a perpendicular hyperplane to the most discriminating direction as a binary class separator.

5. **Nearest Neighbor Classifier.** Nearest Neighbor Classifier is an instance-based classifier [23]. The principle of it is very simple: Similar instances have similar class labels. For an unlabeled instance, we can check k most similar neighbors of it, and determine its label by the label the majority of the k neighbors belong to. There are many criteria to measure the similarity, such as Euclidean distance and Manhattan distance. In our paper, we use Euclidean distance.

### 2.3. Technical motivation

The effectiveness of our approach relies on two observations:

**Observation 1.** Decision tree is a good classifier for just-in-time defect prediction.

**Observation 2.** Ensemble classifier can achieve better performance than that of a single classifier.

To demonstrate the first observation, we make a rough investigation to look for the technique that performs the best for just-in-time defect prediction. We perform experiments on six datasets, i.e., Bugzilla, Columba, Eclipse JDT, Eclipse Platform, Mozilla and PostgreSQL using ten-fold cross validation [1]. We choose five popular classification techniques which are introduced briefly in the above Section, i.e., Naive Bayes (NB), Support Vector Machine (SVM), Decision Tree (DT), Linear Discriminant Analysis (LDA) and Nearest Neighbours (NN), as the candidates. We train the different classifiers using the same features [2] and we use PofB20 and F1-Score [3] to make comparison of their performances.

**Tables 1** and **2** present PofB20 and F1-score of five classification techniques for just-in-time defect prediction. Note that SVM has F1-score of NaN in three datasets (i.e., JDT, Mozilla and Platform), which results from the severe imbalance problem of the three datasets (in which the minority class occupies only less than 15% of the whole dataset). The severely imbalanced training data leads to the construction of a poor SVM model and thus the SVM model predicts all the testing data as the majority class. Also because of the same reason, the values of PofB20 generated by SVM in the three datasets are very small (less than 20%). From the table, we can see that Decision Tree, whose average PofB20 is 53% and F1-score is 40%, performs the best for just-in-time defect prediction. Specifically, in terms of PofB20, decision tree can beat the other four classification techniques for all datasets except for one situation, where SVM has a little higher PofB20 than decision tree for the dataset Columba. In terms of F1-score, Decision Tree can beat the other four classification techniques for all datasets except for two situations, where Naive Bayes has a little higher F1-Score than Decision Tree for the datasets JDT and Platform. Therefore, we can conclude that Decision Tree is the best classifier among the five for just-in-time defect prediction and we will use it as base classifier in our proposed approach.

The second observation has been demonstrated by many past studies [14,15]. There are two main components in the error of a classifier, i.e., bias and variance. Bias is the difference between the decision boundary of a classifier and the true decision boundary. Variance is caused by different training data. Ensemble classifier can often be used to reduce bias or/and variance [24]. Therefore, ensemble classifier can achieve better performance than that of a single classifier.

The above two observations motivate us to build an ensemble classifier based on decision tree. There have been several studies that leverage bagging [5,25,26] and stacking [25,26] in defect prediction. Unfortunately, whether their combination can improve the performance of defect prediction has not been studied yet. Interestingly, we find that bagging and stacking performs better for different datasets; thus, motivating our choice to combine them to allow the strengths of one to cover for the weaknesses of the other. **Tables 3** and **4** present PofB20 and F1-score of bagging and stacking techniques (based on decision trees) for just-in-time defect prediction using the dataset that we have described earlier in this section. From the table, we can see that stacking is better than bagging on some datasets (i.e., JDT, Mozilla, Platform C in terms of PofB20), while bagging is the better than stacking on

---

1. Detail information of the experiment setup is presented in Section 4.1.
2. Detail information of the features we use are presented in Section 3.1.
3. Detail information of the two evaluation metrics are presented in Section 4.2.
some other datasets (e.g., Bugzilla, Columba and PostgreSQL C in terms of PoIB20). Therefore, we propose to combine bagging and stacking to build a two-layer ensemble learning model.

3. Our proposed approach

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

3.1. Overall framework

Fig. 1 presents the overall framework of our proposed approach TLEL. The framework contains two phases: the model building phase and the prediction phase. In the model building phase, our goal is to build an ensemble classifier, by leveraging ensemble learning and decision tree, from historical changes with known labels (i.e., buggy or clean). In the prediction phase, this ensemble classifier would be used to predict if an unknown change would be buggy or clean.

Our framework first extracts a number of features from a set of training changes (i.e., changes with known labels) (Step 1). Features are various quantifiable characteristics of changes that could potentially distinguish changes that are buggy from those that are clean. In this paper, we use the 14 basic features proposed by Kamei et al. [1] as shown in Table 5. In addition, all the features are normalized using z-score method so that the values of all features are in the same order of magnitude.

Next, we construct the base learners based on decision tree (Step 2–4). For each base learner, we firstly perform random under-sampling \(^4\) to handle the class imbalance problem \(^3\) (Step 2). Then, the sampled data is used to train a classifier using Random Forest \(^6\), which is an advanced version of bagging of decision trees (Step 3). After the classifier is trained, we can assign it with a weight (Step 4). The trained classifier together with its weight can be seen as a single unit of the ensemble learner (i.e. base learner). Note that random under-sampling will generate different sampled

---

\(^4\) Detail information of this technique is presented in Section 3.2.

\(^3\) Detail information of this technique is presented in Section 3.3.

\(^6\) Detail information of this technique is presented in Section 3.4.
data every time so that we can learn many different Random Forest classifiers and corresponding weights when we repeat these steps.

After we have several trained Random Forest classifiers, we construct an ensemble classifier based on them and their corresponding weights using stacking. Note that our training process uses bagging and stacking in turn based on decision tree. Therefore, our approach can be seen as a two-layer ensemble learning technique.

In the prediction phase, the ensemble classifier is then used to predict whether a change with an unknown label is buggy or clean. For each of such changes, our framework first extracts the same set of features and normalize the values of the features using the same method as the model building phrase (Step 5). Next, these features are input into all the trained Random Forest classifiers (Step 6). With these classifiers, different prediction results would be generated. In the end, with the weights of these classifiers, we ensemble the different prediction results to produce a final prediction result, which is one of the following labels: buggy or clean (Step 7).

3.2. Z-score method

Considering that the values of the 14 basic change features are not in the same order of magnitude, we perform data normalization on these features. In this paper, we use the z-score method to do the normalization [23]. It transforms all values to make their average value be 0 and their variance be 1. Given a feature $f$, we denote the mean and variance of the initial values in $f$ as $mean(f)$ and $std(f)$ respectively. For each value $f_i$ of the feature $f$, the normalized value $z_i$ is computed as:

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

3.3. Random under-sampling

Random under-sampling [36] is one of the effective approaches to deal with class imbalance problem. It randomly deletes data belonging to the majority class until the amount of data in the majority class is approximately equal to the minority. Random under-sampling can help the learned classifier not to be biased to the majority class, thus in most case it can improve the performance of the classifier [35,37]. In just-in-time defect prediction, the number of clean changes is much more than the buggy changes, which will lead to a bad classifier or even training failure. Therefore, random under-sampling is essential and important for just-in-time defect prediction to make the number of buggy (majority class) and clean (majority class) changes equal.

3.4. Inner layer ensemble: bagging

In the inner layer, we combine decision tree and bagging to build a Random Forest model. Random Forest is an advanced bagging technique based on decision tree [24]. Bagging works best when the base learners are independent and identically distributed. However, traditional decision trees constructed using bagging can’t meet this condition. Random Forest solve the problem by introducing randomness into the model building process of each decision tree. In the construction of traditional decision trees, the split of each node are performed by considering the whole set of features, while in random forest, the split in each tree are performed by considering only a random subset of all features. The randomized decision trees have less correlation so that bagging them performs better.

3.5. Outer layer ensemble: stacking

Due to random under-sampling, we can learn different Random Forest classifiers trained by different subsets of training data. Therefore, we use stacking to ensemble them once more in the outer layer. We simply assign all the classifiers equal weights since all the training data should be treated equally.

For an unlabeled change $x$, we first input its normalized features into all the trained Random Forest classifiers to obtain different prediction results $p$. Each $p$ generated by a Random Forest classifier is either 1 (buggy) or −1 (clean). Specifically, assume in a Random Forest classifier, there are $nb$ decision trees that classify $x$ as buggy while $nc$ decision trees that classify $x$ as clean. If $nb > nc$, then the final prediction result $p$ generated by the specific Random Forest classifier will be 1, and otherwise −1. Since we have many Random Forest classifiers, each of which can generate a prediction result, we simply add all the prediction results due to equal weights of all the classifiers to generate $Ensemble(x)$, as follows:

$$Ensemble(x) = \sum p_{x,i}$$

According to the above formula, $Ensemble(x)$ can be positive or negative or even 0, depending on the number of prediction results $p$ that are 1 or -1. From the ensemble score, we compute the output score $Out(x)$ as:

$$Out(x) = \frac{Ensemble(x)}{LOC(x)}$$

In the above equation, $LOC(x)$ refer to the number of total lines of code in $x$. If $Out(x) \geq 0$, we predict the change as buggy; else we predict it as clean.

The output score of a change considers both the likelihood of the change to be buggy and the effort to review the change. Therefore, the output score is a better indicator for sorting changes to be reviewed than the ensemble score. There are prior studies that also take into consideration review cost by dividing with $LOC$ [38–40]. Mende et al. describe a model that takes the module size measured in lines of code into account [38,39]. Kamei et al. revisit bug prediction by making use of effort-aware models [40]. These studies conclude that models perform better when taking review cost into account.

Note that our two-layer ensemble learning approach is not mathematically equivalent to a single-layer random forest with the same number of trees. Below we elaborate it with an example where there are totally 100 decision trees. For a single-layer random forest, its prediction result $p (1 or -1)$ depends on $nb$ and $nc$ (here $nb + nc$ equals to 100). On the contrary, for our two-layer ensemble learning approach, it contains 10 random forests, each of which has 10 decision trees and generates a prediction result $p (1 or -1)$ depends on $nb$ and $nc$ (here $nb + nc$ equals to 10). The output of the 10 random forests are then added together to generate a final result. Due to random under-sampling, the result of a single random forest is not reliable enough while our approach considers the results of 10 random forests. Therefore, our two-layer ensemble learning approach can have more robust and better performance than that of a single-layer random forest.

4. Experiments and results

In this section, we evaluate the effectiveness of TIEL. The experimental environment is an Intel(R) Core(TM) T6570 2.00 GHz CPU, 8 GB RAM desktop running Windows 7. We first present our experiment setup and evaluation metrics in Sections 4.1 and 4.2 respectively. We then present six research questions and our experiment results that answer these research questions in Section 4.3.
similarly, stance confusion percentage from specific approaches 4.1.

We evaluate TLEL on six datasets from six well-known open source projects, which are Bugzilla, Columba, Eclipse JDT, Eclipse Platform, Mozilla and PostgreSQL. These datasets are also used by Kamei et al. [1]. Table 6 summarizes the statistics of each dataset, containing the period of each dataset, the total number of instances (i.e., changes), and the proportions of the defective changes. Note that all the datasets are imbalanced. The most imbalanced dataset, Mozilla, contains only 5% defects, while the most balanced dataset, Bugzilla, contains 36% defects.

We use ten-fold cross validation [23] to evaluate the performance of TLEL. In 10-fold cross validation, we randomly divide each of the datasets into 10 folds, in which 9 folds are used as training dataset, and the remaining one fold is used as testing dataset. Also, cross validation means each fold is used as testing dataset once. Furthermore, we ensure that each fold has the same class proportion as the original dataset. To make the experiment results more convincing, we run ten-fold cross validation 100 times and record the average performance. Cross validation is a standard evaluation setting, which is widely used in software engineering studies [41,42].

4.2. Evaluation metrics

We use two evaluation metrics to evaluate the performance of our approach TLEL. One is cost effectiveness and the other is F1-score.

4.2.1. Cost effectiveness

Cost effectiveness is often used to evaluate defect prediction approaches [18–20,43,44]. Cost effectiveness is measured by computing the percentage of buggy changes found when reviewing a specific percentage of the lines of code. To compute cost-effectiveness, given a number of changes, we firstly sort them according to their output scores. We then simulate to review the changes one-by-one from the highest ranked change to the lowest ranked change and record buggy changes found. Using this process we can obtain the percentage of buggy changes found when reviewing different percentages of lines of code (1%–100%).

4.2.2. F1-score

The F1-score is a commonly-used measure to evaluate classification performance [21,23]. It combines Precision and Recall and can be derived from a confusion matrix, as shown in Table 7. The confusion matrix lists all four possible prediction results. If an instance is correctly classified as “buggy”, it is a true positive (TP); if an instance is misclassified as “buggy”, it is a false positive (FP). Similarly, there are false negatives (FN) and true negatives (TN). Based on the four numbers, Precision, Recall and F1-score are calculated. Precision is the ratio of correctly predicted “buggy” instances to all instances predicted as “buggy” (\( \text{Precision} = \frac{TP}{TP + FN} \)). Recall is the ratio of the number of correctly predicted “buggy” instances to the actual number of “buggy” instances (\( \text{Recall} = \frac{TP}{TP + FN} \)). Finally, F1-score is a harmonic mean of Precision and Recall: \( \text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \). F1-score is often used as a summary measure to evaluate if an increase in precision outweighs a reduction in recall (and vice versa).

4.3. Research questions

To evaluate the performance of TLEL, we compare it against three baselines. The first baseline is a deep learning approach proposed by us earlier [6]. The approach firstly uses deep belief network to generate a more expressive feature set, and then uses Random Under-Sampling and Logistic Regression. It is referred to as Deeper in the following text. The other two baselines are state-of-the-art approaches for defect prediction. One is a dynamic AdaBoost.NC approach proposed by Wang et al. [2]. The approach is based on AdaBoost and decision tree, but it can adjust the parameter in the training process dynamically. It is referred to as DNC in the following text. The other is a multi-kernel ensemble learning approach proposed by Wang et al. [3]. The approach is a boosting of multiple SVMs each with different kernel functions. It is referred to as MKEL in the following text. We examine our approach in terms of its effectiveness, stability and efficiency in the first four research questions.

TLEL has two tunable parameters, i.e., NLR and NTree. NLR is used to specify the number of base ensemble learners (Random Forest Classifiers) constructed. NTree is used to specify the number of decision trees in each Random Forest. In our experiments, we assign NLR as 10 and NTree as 10 by default. For the fifth research question, we investigate the influence of different values of these parameters.

**RQ1** How effective is TLEL?

**Motivation.** To validate the effectiveness of TLEL, we compare it with the three baselines mentioned above.

**Approach.** We use the two evaluation metrics mentioned above, i.e., cost effectiveness and F1-score, to make comparisons. They are commonly-used measures to evaluate the performance of a defect prediction approach. To make our results more convincing, we perform 10-fold cross validation 100 times and report the average results. In addition, to make fair comparisons, all the ensemble approaches have the same number of base learners. Specifically, there are 100 decision trees in TLEL and DNC, and there are 100 SVMs in MKEL.

For cost effectiveness, we record the percentage of buggy instances found when adding every one percentage of lines of code reviewed. So we will have 100 average values corresponding to the percentage of buggy instances found when reviewing 1%–100% lines of code. We specifically focus on the percentage of buggy instances found when reviewing 20% lines of code, which is referred to as PopB20 [8]. For F1-score, we calculate the average of the 100 F1-score values that we obtain after performing 100 times 10-fold cross validation. We use this average value to compare with the baselines.

In addition, we also calculate p-value and cliff delta to better investigate whether or not TLEL improve the baselines significantly and substantially.

**Results.** Tables 8–11 present the PopB20, Precision, Recall and F1-score values of TLEL as compared with those of the three baselines. From these tables, we can conclude several points.
Table 8
PoB20 values of TLEL and the three baselines.

<table>
<thead>
<tr>
<th>Project</th>
<th>Deeper (%)</th>
<th>DNC(%)</th>
<th>TLEL(%)</th>
<th>MKEL(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>43.52</td>
<td>43.47</td>
<td>61.67</td>
<td>33.02</td>
</tr>
<tr>
<td>Columbia</td>
<td>41.33</td>
<td>42.39</td>
<td>58.85</td>
<td>30.05</td>
</tr>
<tr>
<td>JDT</td>
<td>48.81</td>
<td>54.20</td>
<td>72.55</td>
<td>26.00</td>
</tr>
<tr>
<td>Mozilla</td>
<td>68.30</td>
<td>72.52</td>
<td>82.40</td>
<td>30.98</td>
</tr>
<tr>
<td>Platform</td>
<td>57.25</td>
<td>62.82</td>
<td>77.08</td>
<td>48.94</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>54.11</td>
<td>56.63</td>
<td>70.64</td>
<td>33.33</td>
</tr>
<tr>
<td>Average</td>
<td><strong>52.22</strong></td>
<td><strong>55.34</strong></td>
<td><strong>70.53</strong></td>
<td><strong>37.05</strong></td>
</tr>
</tbody>
</table>

Table 9
Precision of TLEL and the three baselines.

<table>
<thead>
<tr>
<th>Project</th>
<th>Deeper (%)</th>
<th>DNC(%)</th>
<th>TLEL(%)</th>
<th>MKEL(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>57.28</td>
<td>57.15</td>
<td>62.39</td>
<td>36.71</td>
</tr>
<tr>
<td>Columbia</td>
<td>48.01</td>
<td>45.75</td>
<td>51.22</td>
<td>30.55</td>
</tr>
<tr>
<td>JDT</td>
<td>26.02</td>
<td>27.37</td>
<td>29.34</td>
<td>14.20</td>
</tr>
<tr>
<td>Mozilla</td>
<td>13.23</td>
<td>20.11</td>
<td>15.79</td>
<td>5.19</td>
</tr>
<tr>
<td>Platform</td>
<td>26.31</td>
<td>28.66</td>
<td>31.42</td>
<td>14.66</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>46.93</td>
<td>43.58</td>
<td>49.86</td>
<td>25.00</td>
</tr>
<tr>
<td>Average</td>
<td><strong>36.30</strong></td>
<td><strong>37.10</strong></td>
<td><strong>40.00</strong></td>
<td><strong>21.05</strong></td>
</tr>
</tbody>
</table>

Table 10
Recall of TLEL and the three baselines.

<table>
<thead>
<tr>
<th>Project</th>
<th>Deeper (%)</th>
<th>DNC(%)</th>
<th>TLEL(%)</th>
<th>MKEL(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>69.83</td>
<td>74.90</td>
<td>75.92</td>
<td>1</td>
</tr>
<tr>
<td>Columbia</td>
<td>67.37</td>
<td>76.52</td>
<td>74.33</td>
<td>1</td>
</tr>
<tr>
<td>JDT</td>
<td>69.06</td>
<td>72.32</td>
<td>73.48</td>
<td>1</td>
</tr>
<tr>
<td>Mozilla</td>
<td>68.00</td>
<td>61.01</td>
<td>77.75</td>
<td>1</td>
</tr>
<tr>
<td>Platform</td>
<td>69.84</td>
<td>76.49</td>
<td>77.40</td>
<td>1</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>66.71</td>
<td>81.41</td>
<td>76.97</td>
<td>1</td>
</tr>
<tr>
<td>Average</td>
<td><strong>68.47</strong></td>
<td><strong>73.77</strong></td>
<td><strong>75.99</strong></td>
<td><strong>1</strong></td>
</tr>
</tbody>
</table>

Table 11
F1-score of TLEL and the three baselines.

<table>
<thead>
<tr>
<th>Project</th>
<th>Deeper</th>
<th>DNC</th>
<th>TLEL</th>
<th>MKEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>0.6292</td>
<td>0.6472</td>
<td>0.6850</td>
<td>0.5371</td>
</tr>
<tr>
<td>Columbia</td>
<td>0.5606</td>
<td>0.5721</td>
<td>0.6065</td>
<td>0.4680</td>
</tr>
<tr>
<td>JDT</td>
<td>0.3779</td>
<td>0.3971</td>
<td>0.4194</td>
<td>0.2488</td>
</tr>
<tr>
<td>Mozilla</td>
<td>0.2215</td>
<td>0.3023</td>
<td>0.2625</td>
<td>0.0987</td>
</tr>
<tr>
<td>Platform</td>
<td>0.3822</td>
<td>0.4169</td>
<td>0.4471</td>
<td>0.2558</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>0.5509</td>
<td>0.5675</td>
<td>0.6052</td>
<td>0.4000</td>
</tr>
<tr>
<td>Average</td>
<td><strong>0.4537</strong></td>
<td><strong>0.4839</strong></td>
<td><strong>0.5043</strong></td>
<td><strong>0.3352</strong></td>
</tr>
</tbody>
</table>

First, from Table 8, we can see that the PoB20 values of TLEL range from 59% to 82%, which exceed those of the baselines substantially for all the datasets. On average, over 70% of the buggy instances can be found by reviewing only 20% of the lines of code, which is a substantial improvement as compared to the results achieved by the baselines. In addition, the result is also competitive with results reported by many recent studies about defect prediction [38,45]. For example, Ostrand et al. found on average 83% of the defects in 20% of the files [45]. However, note that their 20% of the files actually contains over 50% of the lines of code. And Mendes et al. evaluate a model named LoC-MOM on two datasets KC1 and PCS, and find that when considering 20% of the files, LoC-MOM is able to identify around 55% of the defects in KC1 and over 90% in PCS [38].

Second, from Tables 9 to 11, we can find that in terms of precision, TLEL is the best performer by achieving an average precision of 60%. And in terms of recall, TLEL is better than Deeper and DNC. Although MKEL has higher Recall than TLEL, the Precision of MKEL is rather low. Also, in terms of F1-score, which is the summary of the above two indicators, TLEL is the best predictor by achieving an average F1-score of 49%.

Third, in the experiment, we find that the approach MKEL is the worst in terms of both PoB20 and F1-score in all of the datasets. The direct reason could be that the base learner of MKEL is SVM, which has been demonstrated (in Section 2.3) to have worse performance than decision tree for just-in-time defect prediction. In addition, there are actually two big weaknesses of MKEL. First, it needs two very huge three-dimension kernel matrices, one for training data and the other for testing data, which leads to a huge space complexity. Second, in MKEL the weight update strategy suffers from an algorithmic issue. In the strategy, the weights of defective samples will always increase and not decrease at all, while the weights of non-defective samples will always decrease and not increase at all. Although it seems that the strategy can be a solution to the class imbalance problem, it will lead to infinite loop when sampling using the weights in a later round, say, the 90th round of Boosting. This is the case because each time the sampled data must contain two classes, but the tiny weights of non-defective samples in the later round won’t allow it. Due to the reason, we set the number of boosting rounds as 50, which we have empirically tried and found to be a suitable number that will avoid the infinite loop issue.

In summary, TLEL is more effective than those baselines. To better demonstrate the superiority of our approach, we perform the Wilcoxon signed-rank statistical test with Bonferroni correction to compute the p-value. We also compute the Cliff’s delta. Wilcoxon statistical test is often used to check if the difference in two means is statistically significant (which corresponds to a p-value of less than 0.05). We include the Bonferroni correction to counteract the impact of multiple hypothesis tests. Cliff’s delta is often used to check if the difference in two means are substantial. The range of Cliff’s delta is in [−1, 1], where −1 or 1 means all values in one group are smaller or larger than those of the other group, and 0 means the data in the two groups is similar. The mappings between Cliff’s delta scores and effectiveness levels are shown in Table 12. Note that since MKEL suffers from performance and algorithmic issues, we won’t compare our approach with it in the remainder of this section. By computing the p-value and Cliff’s delta, the extent of which our approach improves over the two baselines can be more rigorously assessed.

Tables 13 and 14 present p-values and Cliff’s deltas of TLEL compared with the two baselines for each of the six datasets. From the two tables, we can see the effectiveness of our approach more clearly. In terms of cost effectiveness, TLEL improves the performance of the baselines statistically significantly and substantially.
in all datasets. In terms of F1-score, TLEL improves the performance of the baselines statistically significantly and substantially in five out of the six datasets.

**TL EL is more effective than the two baselines for just-in-time defect prediction.** On average, by reviewing only 20% lines of code, over 70% of the buggy changes can be found with it.

**RQ2 How effective is TLEL when different percentages of LOC are inspected?**

*Motivation.* We have validated the effectiveness of TLEL in terms of cost effectiveness and F1-score in the first research question. We have demonstrated that TLEL outperforms the baselines in terms of cost effectiveness statistically significantly and substantially. We want to go further by showing the percentage of buggy instances found when reviewing different amount of lines of code using TLEL. Given the same amount of lines of code reviewed, the more buggy instances found, the more useful an approach is.

**Approach.** We record the percentage of buggy instances found when adding every one percentage of lines of code reviewed. So we will have 100 average values corresponding to the percentage of buggy instances found when reviewing 1%-100% lines of code. We can generate a figure whose x-axis represents the percentage of code reviewed and y-axis represents the percentage of defects found for each dataset. In each chart there are three lines, representing TLEL, Deeper and DNC correspondingly.

**Results.** Fig. 2 shows six charts comparing the cost effectiveness of our approach TLEL with two baselines, Deeper and DNC, for different percentages of LOC inspected. The black solid curve corresponds to TLEL, the blue dashed curve corresponds to Deeper and the red dashed curve corresponds to DNC. From the charts, we can see that the red solid curves are always more convex than the blue dashed curves, which means that our approach can always detect more buggy changes than the two baselines in the whole range of percentages of LOC inspected. Therefore, the performance of our approach TLEL is much better than DNC and Deeper in terms of the cost effectiveness.

**TL EL can identify more buggy changes than Deeper and DNC for a wide range of lines of code inspected.**

**RQ3 What is the benefit of using two ensemble layers in TLEL?**

*Motivation.* We have validated the effectiveness of TLEL through the above two research questions. TLEL clearly outperforms the two state-of-the-art baselines. In this RQ, we want to go further by investigating the individual contribution of the two ensemble layers of TLEL.

**Approach.** To measure the individual contribution of the two ensemble layers to the overall performance of TLEL, we create two incomplete versions of TLEL – referred to as Sub-1 and Sub-2 respectively. For Sub-1, we use the inner layer ensemble bagging method to create a single random forest. The detail is the same as Section 3.4 but we do not use stacking. For Sub-2, we only use the outer layer ensemble stacking method to create another kind of ensemble of decision trees. The detail is the same as Section 3.5 except that we replace random forest with decision tree. Note that both TLEL and Sub-2 use undersampling, so we also apply undersampling to Sub-1. That is, in Sub-1 we first use undersampling to balance the training data and then build a single random forest. In addition, all the approaches take review cost into consideration by dividing with LOC. We can then observe the individual contribution of the two ensemble layers by comparing the performance of Sub-1, Sub-2 and TLEL. Also note that the total number of base learners used by all the approaches are the same (i.e., 100) for a fair comparison.

**Results.** Tables 15 and 16 show the performance of Sub-1, Sub-2 and TLEL. From the tables, we can note that TLEL outperforms Sub-1 and Sub-2 in all the datasets in terms of both PoF20 and F1-score. To better demonstrate the superiority of our approach to Sub-1 and Sub-2, we compute p-values (with Bonferroni correction) and Cliff’s delta as we do in RQ1.
Tables 17 and 18 present p-values and Cliff’s deltas of TLEL compared with Sub-1 and Sub-2 for each of the six datasets. From the two tables, we can see the effectiveness of our approach more clearly. Through Wilcoxon signed-rank statistical test and Cliff’s delta we find that the improvement achieved by our approach is statistically significant and substantial to both Sub-1 and Sub-2 in terms of both cost effectiveness and F1-score. It indicates that both of the two ensemble layers contribute to the overall performance of TLEL, and removing any one of them degrades the overall performance.

Both of the two ensemble layers contribute to the good performance of TLEL.

**RQ4** What is the effect of varying the amount of training data on the effectiveness of TLEL?

Motivation. For some projects, the amount of training data (i.e., changes known to be buggy or non-buggy) can be limited. Thus, in this research question, we want to investigate the stability of TLEL by varying the amount of training data.

Approach. In the above research questions, we perform 10-fold cross validations which means that 90% of the data are used for training and 10% of data are used for testing. In this RQ, we perform 2-fold to 10-fold cross validations on the datasets. To make the results more convincing, we also perform each kind of cross validation 10 times. For each dataset, we plot two curves on one
Results. Fig. 3 presents the PofB20 values (red solid line) and F1-scores (blue dashed line) for different cross validations. In the figure, the curves are very stable. In terms of PofB20, the biggest fluctuation is less than 2%. In terms of F1-score, the biggest fluctuation is less than 3%. Therefore, we can conclude that TLEL has good stability and can work with different amount of training data very well.

TLEL is stable and able to work well for reduced amount of training data.

RQ5 How much time does it take for TLEL to run?

Motivation. Now that we have examined the effectiveness and the stability (with reduced training data) of our approach TLEL, we shall test the efficiency of TLEL. The efficiency of an approach is also an important indicator to evaluate whether or not the approach is good enough.

Approach. In order to answer the question, we measure the training and testing time of TLEL. The training time includes the time

![Fig. 3. Two-to-ten fold validation results on six datasets.](chart showing the PofB20 values and F1-scores for 2-folds to 10-folds cross validations.)
Results. Tables 19 and 20 present the training time and testing time of TLEL and the baselines on the six datasets. From Table 19, it takes only less than 4 s on average for TLEL to finish training a statistical model, while Deeper needs more than 10 s and DNC needs more than 200 s. From Table 20, the testing time of all approaches are very small, less than 1 s, which is quite acceptable.

On average, TLEL needs less than 4 s to build a statistical model and about 0.1 s to do the prediction, which we believe to be reasonably good.

RQ6 What is the effect of varying the two parameters settings? Motivation. We have shown the superiority of our approach TLEL in terms of its effectiveness, stability and efficiency. Note that in TLEL, there are two parameters (i.e., NTREE and NLEARNER) that can be tuned. Therefore, we want to examine the effect of varying these parameters.

Approach. In order to answer the question, we perform two sets of experiments. In each set, we only change one parameter and fix the other parameter to see its individual influence to TLEL. For example, to examine the effect of parameter NTREE, we only vary the value of NTREE and fix NLEARNER to its default value (i.e., 10). We vary NTREE from 1 to 20 when fixing NLEARNER, and vary NLEARNER from 1 to 20 when fixing NTREE. Therefore, the total number of decision trees varies from 10 to 200. For each dataset, we plot two curves on one chart showing the PofB20 values and F1-scores for varying the two parameters settings.

Results. Figs. 4 and 5 present the effect of varying the values of the two parameters NTREE and NLEARNER on the performance of TLEL on six datasets. From these figures, we can conclude several points.

First, for NTREE, we can see that varying their values has little influence on the performance in terms of both PofB20 and F1-score. In all the datasets, the values of the two metrics change little when the values of NTREE change. The biggest fluctuation is less than 0.05.

Second, for NLEARNER, we can find that varying its value has some influence on the performance in terms of both PofB20 and F1-score. Specifically, for PofB20 there is some improvement when NLEARNER varies from 1 to 10. However, the performances are similar when NLEARNER varies from 10 to 20. This indicates that the performance of TLEL improves with the increase of the number of base learners in the beginning, but remain stable when the number of base learners increases to a proper number.

Third, when considering NTREE and NLEARNER together, we can find that the same number of base learners (decision trees) does not necessarily lead to the same performance. For example, the performance generated when NTREE is 5 and NLEARNER is 10 is better than the performance generated when NTREE is 10 and NLEARNER is 5. Tables 21 and 22 present a clear comparison of the two parameter settings in terms of PofB20 and F1-score. We can clearly see that the performance of the first parameter setting is better than that of the second parameter setting in terms of both PofB20 and F1-score. It indicates implicitly that the two ensemble layers have different contributions to TLEL.

TLEL performance remains more or less the same when NTREE is increased. On the other hand, its performance improves when NLEARNER is increased but the performance gain tapers off after NLEARNER reaches a certain number (i.e., 10).

5. Discussion

We have investigated six research questions about TLEL and shown the superiority of our approach TLEL. However, there still exists one point that can be discussed.

TLEL is compared with three baselines. Among them, Deeper is a state-of-the-art deep learning approach for just-in-time defect prediction, while DNC and MKEL are two state-of-the-art ensemble learning approaches. Actually, when comparing different ensemble approaches, it would be good to also see the differences in performance when using the same base learner. However, the base learners of TLEL and DNC are the same (i.e., decision tree), while the base learner of MKEL is SVM. To better demonstrate the superiority of our ensemble learning approach, we create a variant of TLEL (referred to as TLEL_SVM) by replacing the base learner of TLEL with SVM, and we compare MKEL, TLEL_SVM and TLEL. Note that the experiment setting is the same as the one described in Section 4.

Tables 23 and 24 present the performances of MKEL, TLEL_SVM and TLEL in terms of PofB20 and F1-score. From these tables, we can see that when using the same base learner, TLEL_SVM is better
Fig. 4. The effect of varying parameter $NTree$ when $NLearn=10$ on the performance of our approach on six datasets.

Table 23
The performance of TLEL compared with TLEL_SVM in terms of Pof$B_20$.

<table>
<thead>
<tr>
<th>Project</th>
<th>MKEL (%)</th>
<th>TLEL_SVM (%)</th>
<th>TLEL (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>33.02</td>
<td>55.37</td>
<td>61.67</td>
</tr>
<tr>
<td>Columbia</td>
<td>30.05</td>
<td>54.45</td>
<td>58.85</td>
</tr>
<tr>
<td>JDT</td>
<td>26.00</td>
<td>61.17</td>
<td>72.55</td>
</tr>
<tr>
<td>Mozilla</td>
<td>50.98</td>
<td>61.22</td>
<td>82.40</td>
</tr>
<tr>
<td>Platform</td>
<td>48.94</td>
<td>61.63</td>
<td>77.08</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>33.33</td>
<td>58.08</td>
<td>70.64</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>37.05</strong></td>
<td><strong>58.69</strong></td>
<td><strong>70.53</strong></td>
</tr>
</tbody>
</table>

Table 24
The performance of TLEL compared with TLEL_SVM in terms of F1-score.

<table>
<thead>
<tr>
<th>Project</th>
<th>MKEL</th>
<th>TLEL_SVM</th>
<th>TLEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>0.5371</td>
<td>0.6061</td>
<td>0.6850</td>
</tr>
<tr>
<td>Columbia</td>
<td>0.4608</td>
<td>0.5482</td>
<td>0.6065</td>
</tr>
<tr>
<td>JDT</td>
<td>0.2488</td>
<td>0.3538</td>
<td>0.4194</td>
</tr>
<tr>
<td>Mozilla</td>
<td>0.0987</td>
<td>0.1969</td>
<td>0.2625</td>
</tr>
<tr>
<td>Platform</td>
<td>0.2558</td>
<td>0.3435</td>
<td>0.4471</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>0.4000</td>
<td>0.5357</td>
<td>0.6052</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.3352</strong></td>
<td><strong>0.4307</strong></td>
<td><strong>0.5043</strong></td>
</tr>
</tbody>
</table>
than MKEL for all datasets in terms of PofB20 and F1-score, which further demonstrates the superiority of our ensemble learning approach TLEL. In addition, the performance of TLEL_SVM is worse than that of TLEL. It indicates that decision tree is indeed better than SVM as the base learner, which corresponds to the first observation described in Section 2.3.

5.1. Threats to validity

Threats to internal validity relate to errors in our experiments. We have double checked our experiments and implementations. Still, there could be errors that we did not notice. Threats to external validity relate to the generalizability of our results. We have evaluated our approach on 137,417 changes from six open source projects. In the future, we plan to reduce this threat further by analyzing even more datasets from more open source and commercial software projects. Threats to construct validity refer to the suitability of our evaluation metrics. We use cost effectiveness and F1-score which are also used by past software engineering studies to evaluate the effectiveness of various prediction tech-

Fig. 5. The Effect of Varying Parameter N Learner When N Tree = 10 on the Performance of Our Approach on Six Datasets.
niques [12,17–19,21,22,47]. Thus, we believe there is little threat to construct validity.

6. Related work

We classify related work into two parts. The first part is about studies on defect prediction. The second part is about studies on ensemble learning.

6.1. Defect prediction

There are some prior studies on just-in-time defect prediction. Mockus et al. predict defects at change-level in a telecommunication system [27]. They propose a number of measures that characterize a change including change diffusion, change size, change purpose and so on, and use logistic regression to do prediction. All the change measures satisfy three basic conditions: The measure can be computed automatically from changes, the measure can be obtained immediately after changes, and the measure can reflect a property of changes. Kim et al. predict defects at change-level in 12 open source projects [12]. They use Support Vector Machine to predict whether or not a change will lead to a bug. Kamei et al. perform a large-scale empirical study of just-in-time defect prediction [1]. They choose 14 change measures that perform well in traditional defect prediction research and build Logistic Regression models to predict if changes are defective or not.

There are many studies on traditional defect prediction. Zimmermann et al. use network analysis to analyze dependencies between various pieces of code, which can help managers to identify central program units that are more likely to be defective [10]. Zimmermann et al. propose a cross-project defect prediction approach; they train a model on a source project which is selected considering several factors, and use the model on a given target project [48]. Turhan et al. employ a k-nearest neighbor algorithm for cross-project defect prediction, which selects 10 nearest instances from source projects to be used as training data for a target project [11]. D'Ambros et al. present a benchmark for defect prediction and provide an extensive comparison of well-known approaches used for defect prediction in their survey [9]. Rahman et al. analyze code metrics from several different perspectives, and build prediction models across 12 large open source projects to understand the performance, stability, portability and stasis of different sets of metrics for defect prediction [18]. Nam et al. propose TCA+, a novel approach to make feature distributions in source projects similar to that of target projects, which can improve the performance of cross-project defect prediction [21].

6.2. Ensemble learning

In defect prediction, class imbalance is a severe problem. Class imbalance is a situation in which the instances of some classes are much less than those of other classes [36]. Ensemble learning is one of the best solutions to class imbalance problem [49,50]. In addition, ensemble learning can combine strengths of different base learners so that it can achieve much better classification performance [14,15].

There are many studies on applying ensemble learning to defect prediction. Based on the class-imbalance learning method AdaBoost.NC [36], Wang et al. propose a dynamic version of AdaBoost.NC for software defect prediction [2]. The approach uses decision tree as base learner and can adjust its parameters dynamically during the training process. Based on the multiple kernel boosting approach MKBoost [51], Wang et al. propose a multiple kernel ensemble learning approach for software defect prediction [3]. The approach uses boosting method. In each boosting round, different kernels are tried and the SVM with the best kernel is chosen as base ensemble learner. Zheng proposes a boosted neural network with cost-sensitive method to improve the performance of software defect prediction [4]. In the approach the misclassification costs are considered in the weight-update strategy. Sun et al. present a coding-based ensemble learning method for software defect prediction [5]. The approach converts imbalanced binary-class data into balanced multi-class data. Rodriguez et al. suggest a descriptive approach for defect prediction [16]. They use two well-known subgroup discovery algorithms to obtain rules that identify defect prone modules. Different from theirs, our approach is a two-layer ensemble learning approach based on classic classification techniques.

7. Conclusion and future work

In this paper, we propose a two-layer ensemble learning approach TLEL for just-in-time defect prediction. The approach has two layers of ensemble learning technique. In the inner layer, we combine Decision Tree and Bagging to build a Random Forest model. In the outer layer, we use random under-sampling to train many different Random Forest models and ensemble them once more using stacking. We evaluate TLEL on datasets taken from six large open source projects and use two evaluation metrics which are cost effectiveness and F1-score. We compare TLEL with three baselines, i.e., Deeper, DNC and MKEL. The results show that TLEL is the best in terms of the two metrics. For cost effectiveness, our approach can identify over 70% defective changes by reviewing only 20% lines of code, which is much more than the defective changes that can be identified by the three baselines. In addition, our approach achieve an average F1-score of close to 50%.

In the future, we plan to improve the performance of our approach by optimizing parameters of TLEL. We also plan to perform experiments on more datasets to reduce the threats to external validity.

Acknowledgments

This research was supported by NSFC Program (No. 61602403 and 61572426), and National Key Technology R&D Program of the Ministry of Science and Technology of China (No. 2015BAYH17F01).

References

2013, E. 2006
Software
Y. Rahman
Rahman
Proceedings
approaches.


J. Han, M. Kamber, Data mining: concepts and techniques, Morgan kauffmann, 2006.


