Front. Comput. Sci.
https://doi.org/10.1007/s11704-017-6015-y

RESEARCH ARTICLE

Combined classifier for cross-project defect prediction: an extended empirical study

Yun ZHANG[1], **David LO**[2], **Xin XIA** (✉)[1], **Jianling SUN**[1]

1 College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China
2 School of Information Systems, Singapore Management University, Singapore 641674, Singapore

Abstract To facilitate developers in effective allocation of their testing and debugging efforts, many software defect prediction techniques have been proposed in the literature. These techniques can be used to predict classes that are more likely to be buggy based on the past history of classes, methods, or certain other code elements. These techniques are effective provided that a sufficient amount of data is available to train a prediction model. However, sufficient training data are rarely available for new software projects. To resolve this problem, cross-project defect prediction, which transfers a prediction model trained using data from one project to another, was proposed and is regarded as a new challenge in the area of defect prediction. Thus far, only a few cross-project defect prediction techniques have been proposed. To advance the state of the art, in this study, we investigated seven composite algorithms that integrate multiple machine learning classifiers to improve cross-project defect prediction. To evaluate the performance of the composite algorithms, we performed experiments on 10 open-source software systems from the PROMISE repository, which contain a total of 5,305 instances labeled as defective or clean. We compared the composite algorithms with the combined defect predictor where logistic regression is used as the meta classification algorithm (CODEP$_{Logistic}$), which is the most recent cross-project defect prediction algorithm in terms of two standard evaluation metrics: cost effectiveness and F-measure. Our experimental results show that several algorithms outperform CODEP$_{Logistic}$: Maximum voting shows the best performance in terms of F-measure and its average F-measure is superior to that of CODEP$_{Logistic}$ by 36.88%. Bootstrap aggregation (Bagging$_{J48}$) shows the best performance in terms of cost effectiveness and its average cost effectiveness is superior to that of CODEP$_{Logistic}$ by 15.34%.

Keywords defect prediction, cross-project, classifier combination

1 Introduction

To build high quality software, developers need to invest a considerable amount of effort in testing and debugging. However, their resources are frequently limited and thus they need to prioritize these efforts. Software defect prediction techniques to help developers prioritize testing and debugging efforts have been proposed in the literature. Such a technique identifies the software components that are more likely to be defect-prone by constructing a predictive classification model constructed from features such as lines of code (LOCs), code complexity, and number of symbols [1–6]. Such predictions can be used to optimize the allocation of testing and debugging resources: more resources should be allocated to more defect-prone modules.

Defect prediction techniques are effective when a sufficient amount of training data is available [7]. Unfortunately, training data are frequently limited for new projects having little or no historical bug data. For such cases, engineers need to use data from other projects and companies [8]. Cross-project

Received January 7, 2016; accepted December 29, 2016

E-mail: {yunzhang28, xxkidd, sunjl}28@zju.edu.cn; davidlo@smu.edu.sg

defect prediction is a strategy that trains a generalized prediction model on data belonging to other projects and uses the model to predict the defect proneness of components belonging to the target project [9, 10]. Recently, effort has been invested in solving cross-project defect prediction, which is a challenging problem because of the heterogeneity of data taken from various projects [10].

Recently, Panichella et al. proposed a composite approach, the combined defect predictor, referred to as CODEP, which combines different and complementary classifiers learned by different machine learning algorithms, for cross-project defect prediction [11]. Their experimental results show that CODEP outperforms many existing cross-project defect prediction techniques. However, in the machine learning literature, many composite techniques have been proposed for combining multiple classification models. In this study, our objective was to investigate the applicability of existing composite techniques proposed in the machine learning literature for cross-project defect prediction and determine whether they can outperform the state-of-the-art method for cross-project defect prediction, namely CODEP.

We used four well-known metrics that are utilized to evaluate the performance of a predictive algorithm, in particular in the context of defect prediction: cost effectiveness [12–15], F-measure [14, 16–18], mean average precision (MAP) [19, 20], and the area under the receiver operator characteristic curve (AUC-ROC) [6, 14]. Cost effectiveness evaluates an algorithm's performance for a given cost threshold, such as a certain LOC inspection percentage. For example, when a team has limited time and resources to inspect source code, it is crucial to inspect the top percentage of code that is predicted to be buggy, which can help developers discover as many bugs as possible. In this study, we used the cost effectiveness setting proposed by Jiang et al. [3], which measures the number of bugs that can be discovered by inspecting 20% of the LOCs (NofB20). Furthermore, we also used F-measure [14, 16–18], which is a summary measure that combines precision and recall, for the evaluation. F-measure evaluates whether an increase in precision (recall) outweighs a reduction in recall (precision). MAP is a single-figure measure of quality and has been shown to have particularly good discrimination and stability properties for evaluating ranking techniques. AUC-ROC measures the probability that a randomly chosen defective entity ranks higher than a randomly chosen clean entity.

In this study, we compared the composite algorithms against the best variant of CODEP, which uses logistic regression as a meta-learner, referred to as $\text{CODEP}_{Logistic}$. We evaluated the algorithms on defect datasets from ten projects (ant, camel, ivy, jedit, log4j, lucene, poi, prop, tomcat, xalan), which are part of the PROMISE data repository. The datasets contain a total of 5,305 instances together with their labels, i.e., defective or clean. The experimental results show that several of the composite algorithms outperform CODEP where logistic regression is used as the meta classification algorithm ($\text{CODEP}_{Logistic}$) in terms of F-measure and cost effectiveness. Among them, the performance of maximum voting (Max) is best in terms of F-measure, and achieves an average score of 0.412, which improves on that of $\text{CODEP}_{Logistic}$ by 36.88%; the performance of Bagging_{J48} is the best in terms of cost effectiveness, achieving an average NofB20 score of 40.6, which improves on that of $\text{CODEP}_{Logistic}$ by 15.34%.

This paper presents an extension of a preliminary study published in a research paper presented at a conference [21]. The preliminary paper is extended in various ways: 1) we describe the process of the defect prediction techniques and list the features used in our study; 2) we examine the cost effectiveness of the composite algorithms when different percentages of LOCs are inspected; 3) we add two evaluation metrics, MAP and AUC-ROC, in the experiments and results section; 4) we add a baseline approach for answering research question 1 (RQ1), which is a common approach for cross-project defect prediction; 5) we add a Discussion section, in which the performance of the composite algorithms on NASA datasets is discussed, the performance of CODEP with meta classifier J48 is examined, the time complexity of the composite algorithms is analyzed, and threats to validity are introduced; 6) we expand the related work section to include more classical defect prediction work; and 7) we improve the exposition by adding more examples to make the paper clearer for readers.

In summary, the main contributions of this paper are as follows.

1) We examine the effectiveness of seven different composite algorithms proposed in the machine learning literature for cross-project defect prediction in terms of cost effectiveness, F-measure, AUC-ROC, and MAP.
2) We describe experiments conducted using 10 defect datasets to demonstrate the effectiveness of the algorithms and highlight promising algorithms, the performance of which is better than that of CODEP and the common approach proposed by Zimmermann et al. [9], denoted by LR.

The remainder of the paper is organized as follows. We describe several classical classification algorithms and CODEP in Section 2. We then present a number of composite algorithms that can be used to combine multiple classical classification algorithms in Section 3. We present our experiments and their results in Section 4. Section 5 discusses some issues about the performance, efficiency, and threats to validity of the composite classifiers in defect prediction. We discuss related work in Section 6. We conclude and mention future work in Section 7.

2 Background

In this section, we first briefly introduce the steps that defect prediction techniques have in common. We then present several classical classification techniques used in the composite algorithms that we investigated in this work. Finally, we describe a state-of-the-art cross-project defect prediction approach named CODEP.

2.1 Defect prediction techniques: common steps

A defect prediction technique learns a prediction model from a set of classes/files/modules that are known to be defective or clean. The steps that are common to such techniques include the following.

• **Training data extraction** Collect classes/files/modules that are defective or clean. These classes can be identified by manual code inspection, or, if a project has been ongoing for some time, by mining the bug tracking and version control systems of the project.

• **Feature extraction** Extract the needed features from the training set of defective and clean classes/files/modules. Various features have been used in many previous defect prediction studies. Table 1 [22–27] shows the 20 features used in our study, which were proposed by Jureczko and Madeyski [28].

• **Model training phase** Train a classification model with a particular algorithm based on the extracted features.

• **Model testing phase** Test the model on new classes/files/modules, for which the defect proneness needs to be predicted. First, the values of relevant features need to be extracted from these new classes/files/modules. These values serve as inputs to the trained model, which then predicts whether the new classes/files/modules are defective or not.

2.2 Classical classification techniques

Several machine learning techniques have been used to predict defect-prone source code classes/files/components, such as logistic regression [7, 29, 30], radial basis function network (RBF network) [31], multi-layer perceptron (MLP) [32], Bayesian network (BN) [33], decision trees [34], and decision tables (DTs) [5].

Table 1 Extracted features

Attribute	Description
wmc	Number of methods used in a given class [22]
dit	Maximum distance from a given class to the root of an inheritance tree [22]
noc	Number of children of a given class in an inheritance tree [22]
cbo	Number of classes that are coupled to a given class [22]
rfc	Number of distinct methods invoked by code in a given class [22]
lcom	Number of method pairs in a class that do not share access to any class attributes [22]
lcom3	Another type of lcom metric proposed by Henderson-Sellers [23]
npm	Number of public methods in a given class [24]
loc	Number of LOCs in a given class [24]
dam	Ratio of the number of private/protected attributes to the total number of attributes in a given class [24]
moa	Number of attributes in a given class that are of user-defined types [24]
mfa	Number of methods inherited by a given class divided by the total number of methods that can be accessed by the member methods of the given class [24]
cam	Ratio of the sum of the number of different parameter types of every method in a given class to the product of the number of methods in the given class and the number of different method parameter types in the entire class [24]
ic	Number of parent classes to which a given class is coupled [25]
cbm	Total number of new or overwritten methods to which all inherited methods in a given class are coupled [25]
amc	Average size of methods in a given class [25]
ca	Afferent coupling, which measures the number of classes that depends on a given class [26]
ce	Efferent coupling, which measures the number of classes on which a given class depends [26]
max_cc	Maximum McCabe's cyclomatic complexity (CC) score [27] for methods in a given class
avg_cc	Arithmetic mean of McCabe's CC scores [27] for methods in a given class

In this study, we investigated six classification algorithms, namely, logistic regression, BN, RBF network, MLP, alternating decision tree (ADTree), and DT. We used these classification algorithms to construct various underlying classifiers for our composite classification algorithms. We chose these six because they were successfully used for defect prediction in many previous studies [1,5,29,31,32]. Furthermore, these classifiers belong to four different families: regression functions, neural networks, decision trees, and rule models. We describe these six classification algorithms succinctly in the following paragraphs.

2.2.1 Logistic regression

Logistic regression [35] models the relationship between features and labels as a parametric distribution $P(y|x)$, where y refers to the label of a data point and x refers to the data point represented as a set of features. The parameters of this distribution are directly estimated from the training data. Let $x = \{x_{f_1}, x_{f_2}, \ldots, x_{f_m}\}$ denote the vector representation of the features of a data point x, x_{f_i} denote the value of the i-th feature of x, and $W = \{w_0, w_1, w_2, \ldots, w_m\}$ denote the weight vector associated with the features in x, where w_0 is a bias parameter and $w_i, i \in \{1, 2, \ldots, m\}$ is the weight of the i-th feature of x, i.e., x_{f_i}. Consider binary classification, where y takes two values, 0 or 1 (in our case, 0 represents clean and 1 represents defective). We derive $p(y = 1|x)$ and $p(y = 0|x)$ as

$$p(y = 1|x) = \frac{1}{1 + \exp(w_0 + \sum_{i=1}^{m} w_i \times x_{f_i})}, \tag{1}$$

$$p(y = 0|x) = \frac{\exp(w_0 + \sum_{i=1}^{m} w_i \times x_{f_i})}{1 + \exp(w_0 + \sum_{i=1}^{m} w_i \times x_{f_i})}. \tag{2}$$

To evaluate the label of a new instance x_{new}, we can compute $ratio(x_{new}) = \frac{p(y=1|x_{new})}{p(y=0|x_{new})}$. If $ratio(x_{new}) > 1$, we predict the label of x_{new} as 1; else, the predicted label is 0. The main learning task for logistic regression is to estimate the parameter W. Various methods can be used for this purpose, such as gradient ascent.

2.2.2 Bayesian network

BN is a graphical model of probabilistic relationships representing the input feature space and label space [36]. It is a directed acyclic graph (DAG) and each node in a BN represents a feature (in our case, one of the features in Table 1) or a label (in our case, defective or clean). A directed edge between two nodes denotes that there is a causal relationship between them. During the model training phase, BN constructs a Bayesian network from the training set. Then, during the prediction phase, this network is used to predict the label of a new unlabeled instance.

2.2.3 Radial basis function network

An RBF network is an artificial neural network that uses radial basis functions as activation functions [37]. It typically contains three different layers: an input layer, a hidden layer with a non-linear RBF activation function, and a linear output layer. The output of the network is a linear combination of the radial basis functions of the inputs and neuron parameters.

2.2.4 Multi-layer perceptron

MLP is an additional type of artificial neural network model, which is trained using a supervised learning technique, called a back-propagation algorithm, which maps sets of input data to a set of appropriate outputs. An MLP consists of multiple layers of nodes in a directed graph: one input layer, one output layer, and one or more hidden layers [11]. The output of a layer is used as the input of the nodes in the subsequent layer. MLP can distinguish data that are not linearly separable, which is an improvement on the standard linear perceptron.

2.2.5 Alternating decision trees

An ADTree consists of a tree structure with decision nodes and prediction nodes in an alternating order. Decision nodes specify conditions (e.g., $feature_1 < 0.5$) and each is connected to two prediction nodes, one of which corresponds to the case where the condition is evaluated as true and the second to the case where the condition is evaluated as false. A prediction node contains a single decimal value. An instance (in our case, a class) is classified by an ADTree by finding paths in the tree from the root node to leaf nodes, where all the decision nodes in between the root and the leaf nodes are evaluated as true based on the feature values of the instance. The values of the in-between prediction nodes along the corresponding paths are then summed. This sum is used to determine the class label (in our case, defective or clean) of an instance; i.e., if the sum is positive, then an instance is defective, and otherwise it is clean.

2.2.6 Decision table

A DT can be regarded as an extension of a one-valued decision tree [35]. It is a rectangular table, the columns of which are features and the rows sets of decision rules. Each decision rule consists of two parts: (i) a pool of conditions that are linked through "and" and "or" logical operators and (ii) an outcome that reflects the classification of an instance according to the corresponding rule into one of the class labels (in our case, defective or clean). In order to eliminate equivalent rules and reduce the likelihood of over-fitting, DT attempts to find a good subset of features by running a feature reduction algorithm.

2.3 Combined defect predictor (CODEP)

CODEP is a two-level composite algorithm that predicts the label of an instance, i.e., it predicts whether a class is defective or clean [11]. On the first level, CODEP builds six underlying classifiers on a training set. These six classifiers are built by running each of the six classical classification algorithms described in Section 2. Then, the confidence scores output by each classifier on each instance in the training set are collected to create a new dataset. On the second level, another classifier is built on the new dataset, which is referred to as the meta classifier. In this study, we used logistic regression as the meta classifier, since Panichella et al. showed that its performance is the best. To predict the label of an instance, first CODEP outputs the confidence scores of the six underlying classifiers, and then, these confidence scores are used as input to the meta classifier to predict the label of the instance.

3 Composite algorithms

Panichella et al. showed that the composite algorithm CODEP that they proposed outperforms many other approaches [11]. On the basis of their work, we investigated several other composite algorithms proposed in the machine learning literature, aiming at finding one or more that perform better than CODEP. The following subsections provide a detailed description of the composite algorithms for cross-project defect prediction that we investigated in this study.

3.1 Overall framework

Figure 1 presents the overall framework, which describes the usage of the composite algorithms for cross-project defect prediction. The framework contains two phases: model building and prediction. In the model building phase, our goal is to build a composite classifier by leveraging several underlying classifiers built using one or more of the classical classification algorithms presented in Section 2. In the prediction phase, this combined classifier is used to predict whether a new instance, i.e., a class/file/component, is defect-prone or not.

Our framework first extracts features from training instances (Step ①). Then, it applies a feature selection technique to select a subset of relevant features to further improve the prediction performance (Step ②). Using these selected features, we next construct a composite prediction model by combining several underlying classifiers (Step ③). We investigated various composite classification techniques, including average voting (Ave), Max, CODEP$_{Logistic}$, Bagging$_{J48}$, Bagging$_{Naive}$, Boosting$_{J48}$, Boosting$_{Naive}$, and Random Forest (RF). These techniques are used to create composite classification models that can process an instance and predict whether it is defective or not based on its features.

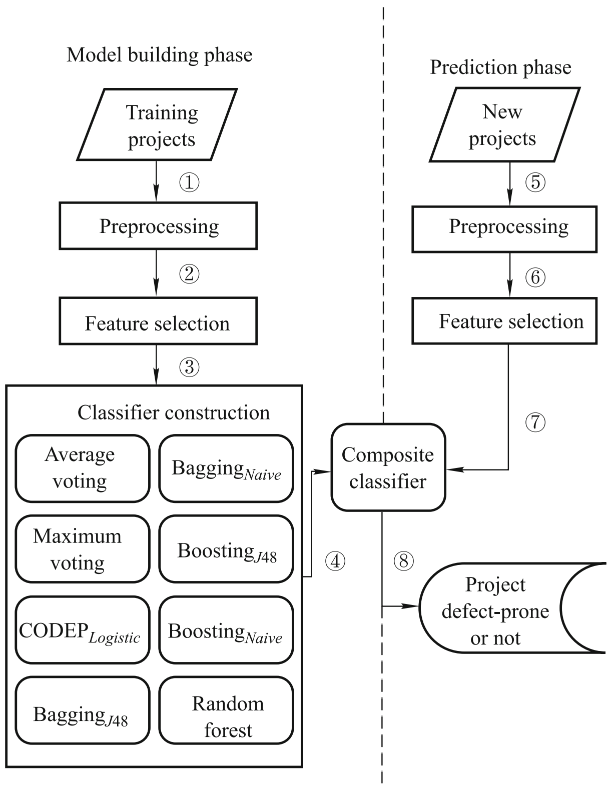


Fig. 1 Proposed overall cross-project defect prediction framework

After the composite classifier has been built, in the prediction phase, it is used to predict whether a new instance is defective or not. From each such new instance, our framework first preprocesses and extracts features (Step ⑤), and then represents them by using the features selected in the model building phase (Step ⑥). Next, these features are input into the composite classifier in the classifier application step (Step ⑦). Finally, the classifier outputs the prediction result: defective or clean (Step ⑧).

3.2 Average voting

Ave is a voting method that combines confidence scores from different underlying classifiers [35]. We used the six classical classification algorithms described in Section 2 to build the underlying classifiers. Each underlying classifier outputs a confidence score, which ranges from 0 to 1, for an instance. In total, we obtained six confidence scores, which correspond to the six underlying classifiers. Next, Ave averages the six confidence scores and outputs the final confidence score (Ave score), which also ranges from 0 to 1. To decide whether an instance is of a particular class label (in our case, defective), we compare the Ave score with 0.5. If it is larger than 0.5, then we predict it as defective; else, it is clean. For example, consider an instance in a project where the scores computed by the six underlying classifiers are 0.60, 0.90, 0.40, 0.55, 0.73, and 0.81, respectively; then, the confidence score of Ave is 0.67. Since $0.67 > 0.5$, we predict it as defective.

3.3 Maximum voting

Max is also a voting method, which outputs the maximum confidence scores of different underlying classifiers [35]. We used the same six classical classification algorithms to build the same the underlying classifiers as for Ave. Unlike average voting, Max outputs the confidence score of an instance by selecting the maximum confidence score of the six underlying classifiers. For example, consider an instance in a project where the scores computed by the six underlying classifiers are 0.60, 0.90, 0.40, 0.55, 0.73, and 0.81, respectively; then the confidence score of Max is 0.9. Since it is larger than 0.5, we predict the instance as defect-prone.

3.4 Bagging

Bootstrap aggregation (Bagging) [38] is a robust ensemble algorithm, which can be combined with other supervised learning algorithms to improve the overall performance and avoid overfitting. Given a dataset D of size n, Bagging first performs bootstrap sampling from D, i.e., random sampling with replacement, to generate m new datasets $D'_i, i \in \{1, 2, \ldots, m\}$. The size of D'_i is denoted by n'_i, and $n'_i < n$. Next, Bagging trains a weak classifier (also called an underlying classifier) from each dataset D'_i. For the prediction phase, all the outputs of m classifiers are combined into a single prediction using majority voting. In this study, we used decision tree (J48) and naive Bayes as the underlying classifiers of Bagging, denoted by Bagging$_{J48}$ and Bagging$_{Naive}$, respectively. We note that Bagging does not combine classifiers built by different basic algorithms, but rather classifiers built by one algorithm trained with different training subsets, which means it does not use the six basic classifiers described in Section 2.

3.5 Boosting

Boosting [38] is used to generate strong classifiers from weak classifiers. It can be combined with many other supervised learning algorithms to improve overall accuracy and performance. Boosting generates and calls a new weak classifier in a series of rounds. For each round t, it updates the weights of instances in a dataset, which indicates their different levels of importance. In general, instances that have been misclassified in the previous round are assigned a higher weight, while instances that have been correctly classified are assigned a lower weight. This re-weighting strategy causes the weak classifier in the current round to focus more on the misclassified instances. In this study, as for Bagging, we use decision tree (J48) and naive Bayes as the underlying classifiers of Boosting, denoted as Boosting$_{J48}$ and Boosting$_{Naive}$, respectively. Similar to Bagging, Boosting also does not use the six basic classifiers described in Section 2; for Boosting$_{J48}$, it trains J48 many times with weighted training instances to generate strong classifiers.

3.6 Random forest

RF [39] combines an ensemble of decision trees. It takes advantage of both bagging and random feature selection for the tree building; each decision tree is built using a bootstrap sample of the data, and RF selects a subset of features randomly to split at each node when growing a tree instead of using all the features. Multiple decision trees are learned and the output of the decision trees is combined into a single prediction using majority voting.

4 Experiments and results

In this section, we evaluate the effectiveness of the seven composite algorithms and CODEP. The experimental environment was an Intel(R) Core(TM) T6570 2.10 GHz CPU, 4GB RAM desktop computer running Windows 7 (32 bits). We first present our experimental setup and evaluation metrics in Sections 4.1 and 4.2, respectively. We then present three research questions and the experimental results that answer them in Section 4.3.

4.1 Experimental setup

We evaluated the composite algorithms on defect datasets from ten Java projects, i.e., ant, camel, ivy, jedit, log4j, lucene, poi, prop, tomcat, and xalan, that belong to the Promise repository. The ten projects are all in the Java domain and contain the same set of features as that listed in Table 1, which improves the feasibility of cross-project defect prediction. Each dataset contains a set of classes labeled as defective or clean and their corresponding metrics, e.g., LOC or the Chidamber and Kemerer (CK) metric; see Table 1. Table 2 summarizes the statistics of each project. The columns correspond to the project name (Name), the release version of each project (Release), the total number of classes in each project (Instances), the number of defective classes in each project (Defective Instances), and the percentage of defective classes (%).

Table 2 Software projects used in our study

Project	Release	Instances	Defect-prone instances	Percentage/%
ant	1.7	745	166	22
camel	1.6	965	188	19
ivy	2	352	40	11
jedit	4	306	75	25
log4j	1	135	34	25
lucene	2.2	247	144	58
poi	2	314	37	12
prop	6	660	66	10
tomcat	6	858	77	9
xalan	2.4	723	110	15

Our experiments were performed in the context of cross-project defect prediction, which means we trained a model on one dataset and applied the model to another dataset. Our experiments proceeded in ten iterations. In the first iteration, we took classes from the first project “ant” as a testing set and combined instances from the other nine projects as a training set. We learned a model from the training set and used it to predict the defect labels of instances in the testing set. In the second iteration, we took instances of the second project “camel” as a testing set and combined the instances of the other projects as a training set. We repeated the same process eight additional times, each time taking a different project as the testing set. We report the average performance of a prediction technique across the ten iterations.

We used the implementations of the six classification techniques and seven composite algorithms in Weka [40]. For Ave, Max, CODEP, and RF, we used their default settings in Weka. For the bagging and boosting techniques, we set the number of iterations to ten.

4.2 Evaluation metrics

We used four performance metrics for our evaluation: cost effectiveness, F-measure, MAP, and AUC-ROC.

4.2.1 Cost effectiveness

Cost effectiveness is widely used in defect prediction as an evaluation metric [12–14]. It aims at maximizing benefits under the condition of incurring the same amount of cost. In the context of defect prediction, the cost is the LOCs inspected and the benefit is the number of buggy classes found. The cost effectiveness setup we used is the same as that used by Jiang et al. [3]. Our objective was to count the number of buggy classes found when a developer inspects the first 20% of the LOCs. This number is referred to as NofB20.

To evaluate cost effectiveness, we sort instances in the test data based on their probabilities of being defective. For each instance, a prediction technique outputs not only its predicted defect label (i.e., defective or clean) but also the probability that the instance is defective. We then simulate an inspection process where a developer examines the instances serially, starting from those that are the most likely to be defective. We stop the process when 20% of the LOCs have been inspected and output the number of buggy classes that are identified. This number is the NofB20 cost effectiveness score. An increase in the cost effectiveness score indicates that a developer can discover more bugs when inspecting a certain number of LOCs.

4.2.2 F-measure

F-measure, which is the harmonic mean of precision and recall, is a standard and widely used measure for evaluating classification algorithms [35]. There are four possible outcomes for an instance in a target project: an instance can be classified as buggy when it truly is buggy (true positive, TP), as buggy when it is in fact clean (false positive, FP), as clean when it is in fact buggy (false negative, FN), as clean when it truly is clean (true negative, TN). Based on these possible outcomes, precision, recall, and F-measure are defined as follows.

• Precision: the proportion of instances that are correctly labeled as buggy among those labeled as buggy.

$$\text{Precision} = TP/(TP + FP). \tag{3}$$

• Recall: the proportion of buggy instances that are correctly labeled.

$$\text{Recall} = TP/(TP + FN). \tag{4}$$

• F-measure: a summary measure that combines precision and recall; it evaluates whether an increase in precision (recall) outweighs a reduction in recall (precision)

$$\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{5}$$

There is a trade-off between precision and recall. One can increase precision by sacrificing recall (and vice versa). The trade-off causes difficulties in comparing the performance of several prediction models by using precision or recall alone [35]. For this reason, we compared the prediction results using F-measure, which is a harmonic mean of precision and recall. F-measure is frequently used to judge whether an increase in precision outweighs a loss in recall (and vice versa). This follows the setting used in many classification and defect prediction studies [3, 17] and various software analytics studies [41,42].

4.2.3 Mean average precision

MAP is a single-figure measure of quality and has been shown to have especially good discrimination and stability properties to evaluate ranking techniques [19, 20, 43]. For a project p, when a classifier returns a sorted list of instances, the average precision is defined as the mean of the precision values obtained for the different sets of top k instances that were retrieved before each defect instance was retrieved, which is computed as

$$AvgP(p) = \frac{\sum_{j=1}^{M} P(j) \times Rel(j)}{\text{Defect instances in } p}. \tag{6}$$

In the above equation, M is the number of top ranked instances a developer needs to inspect, $Rel(j)$ indicates whether the instance at position j is defect-prone or not, and $P(j)$ is the precision at the given cut-off position j and is computed as

$$P(j) = \frac{\text{Defect instances in top } j \text{ positions}}{j}. \tag{7}$$

Then, the MAP for a set of projects Ps is the mean of the average precision scores for all the projects:

$$\text{MAP} = \frac{\sum_{p \in Ps} AvgP(p)}{|\, Ps \,|}. \tag{8}$$

We used MAP to measure the average performance of the composite classifiers. The higher the MAP value, the better is the classifiers performance.

4.2.4 Area under the receiver operator characteristic curve

ROC is a non-parametric method used to evaluate models. It plots the precision/recall values reached for all possible cutoff values ranging within the interval [0;1]. Hence, it is independent of the cutoff, unlike the precision and recall metrics [6]. A curve of the false positive rate is plotted against the true positive rate; the best possible model is that with a ROC curve close to the line $y = 1$, while a random model will be close to the diagonal $y = x$. To show a single scalar value that facilitates the comparison across models, we report the AUC-ROC values. The AUC-ROC value measures the probability that a randomly chosen defective entity ranks higher than a randomly chosen clean entity. An area of 1 represents a perfect classifier, whereas for a random classifier an area of 0.5 is expected. We used AUC-ROC as the performance metrics, because the traditional evaluation metrics, namely, precision and recall, are sensitive to the thresholds used as cutoff parameters [6, 14].

4.3 Research questions

We were interested in answering the following research questions.

RQ1 How effective are the seven composite algorithms? How much improvement can these composite algorithms achieve as compared to CODEP$_{Logistic}$ and LR?

• **Motivation** We needed to investigate the effectiveness of the seven composite algorithms and compare them against CODEP$_{Logistic}$ [11] and LR in terms of cross-project defect prediction. The answer to this research question would shed light on whether and to what extent the combined algorithms improve on CODEP$_{Logistic}$, which is the state-of-the-art cross-project defect prediction technique, and on LR.

• **Approach** To answer this research question, we computed the F-measure, NofB20, MAP, and AUC-ROC scores of the seven composite algorithms and CODEP$_{Logistic}$ when they were applied to ten datasets from the PROMISE repository. We then compared the results achieved by each of the seven combined algorithms with the results of CODEP$_{Logistic}$. We also compared the results with the LR approach proposed by Zimmermann et al. [9], who built a logistic regression model from other projects and tested its effectiveness for classifying elements as defect-prone in a current project.

• **Results** Table 3 presents the F-measure scores of CODEP$_{Logistic}$ and LR as compared with those of Ave, Max, Bagging$_{J48}$, Bagging$_{Naive}$, Boosting$_{J48}$, Boosting$_{Naive}$, and RF. The F-measure scores of CODEP$_{Logistic}$ vary from 0.053 to 0.435. Across the ten datasets, the average F-measure of CODEP$_{Logistic}$ is 0.301. In Table 3, we can observe that the average F-measure scores of Max, Boosting$_{J48}$, and RF are 0.412, 0.302, and 0.308, respectively, which are superior to the average F-measure of CODEP$_{Logistic}$ by 36.88%, 0.33%, and 2.33%, respectively. The F-measure scores of LR vary from 0.0 to 0.492. Across the ten datasets, the average F-measure of LR is 0.217, which is lower than that of all the composite algorithms and CODEP$_{Logistic}$. Max achieves the best F-measure scores: its F-measure scores vary from 0.286 to 0.608 and the average score is 0.412. Meanwhile, the other four composite algorithms that we investigated in this study do not perform as well as CODEP$_{Logistic}$ in terms of F-measure. The average F-measure scores of Ave, Bagging$_{J48}$, Bagging$_{Naive}$, and Boosting$_{Naive}$ are 0.299, 0.245, 0.298, and 0.29, respectively, which are lower than that of CODEP$_{Logistic}$ by 0.67%, 22.86%, 1.01%, and 1.01%, respectively. To the best of our knowledge, Max achieves higher F-measure scores than other composite algorithms because the datasets we used in the experiments are imbalanced, and Max outputs the confidence score of an instance by selecting the maximum confidence score of the six underlying classifiers; therefore, it tends to predict the clean instances as defective instances, and can achieve a high recall score.

Table 4 presents the NofB20 scores of CODEP$_{Logistic}$ and LR as compared with those of Ave, Max, Bagging$_{J48}$, Bagging$_{Naive}$, Boosting$_{J48}$, Boosting$_{Naive}$, and RF. The NofB20 scores of CODEP$_{Logistic}$ vary from 8 to 88. Across the 10 datasets, the average NofB20 score of CODEP$_{Logistic}$ is 35.2. In Table 4, we can see that the average NofB20 scores of Ave, Max, Bagging$_{J48}$, Boosting$_{J48}$, and RF are 38.1, 37.1, 40.6, 35.4, and 37.2, respectively, which are superior to the NofB20 score of CODEP$_{Logistic}$ by 8.24%, 5.40%, 15.34%, 0.57%, and 5.68%, respectively. The NofB20 scores of LR vary from 7 to 87. Across the 10 datasets, the average NofB20 score of LR is 31.3, which is lower than that of all the composite algorithms and CODEP$_{Logistic}$. Bagging$_{J48}$ achieves the highest NofB20 score; its NofB20 scores vary from 6 to 92 and the average score is 40.6. Meanwhile, the other two composite algorithms that we investigated in this study do not perform as well as CODEP$_{Logistic}$ in terms of NofB20. The average scores of Bagging$_{Naive}$ and Boosting$_{Naive}$ are 34.4 and 22.8, respectively, which are lower than that of CODEP$_{Logistic}$ by 2.33% and 54.39%, respectively.

Table 5 presents the AUC-ROC scores of CODEP$_{Logistic}$ and LR as compared with those of Ave, Max, Bagging$_{J48}$, Bagging$_{Naive}$, Boosting$_{J48}$, Boosting$_{Naive}$, and RF. The AUC-ROC scores of CODEP$_{Logistic}$ vary from 0.633 to 0.82. Across the ten datasets, the average AUC-ROC score of CODEP$_{Logistic}$ is 0.752, and the average AUC-ROC scores of Ave and Max are close to the results of CODEP$_{Logistic}$. The AUC-ROC scores of LR vary from 0.617 to 0.805. Across the 10 datasets, the average AUC-ROC score of LR is 0.723 and the average AUC-ROC scores of Ave, Max, and Bagging$_{Naive}$ are superior to those of LR by 3.18%, 3.6%, and 0.69%, respectively.

Table 6 presents the MAP scores of CODEP$_{Logistic}$ and LR as compared with those of Ave, Max, Bagging$_{J48}$, Bagging$_{Naive}$, Boosting$_{J48}$, Boosting$_{Naive}$, and RF. Across the ten datasets, the MAP score of CODEP$_{Logistic}$ is 0.169 and the MAP score of LR is 0.154. In Table 6, we can see that the MAP scores of Ave and Max are 0.173 and 0.171, respectively, which are superior to the MAP scores of CODEP$_{Logistic}$

Table 3 F-measure scores of the seven composite algorithms and baseline approaches

Algorithms	ant	camel	ivy	jedit	log4j	lucene	poi	prop	tomcat	xalan	Average
Ave	0.343	0.112	0.444	0.516	0.205	0.066	0.237	0.222	0.44	0.402	0.299
Max	0.554	0.306	0.439	0.608	0.5	0.319	0.286	0.295	0.38	0.439	0.412
CODEP$_{Logistic}$	0.321	0.127	0.43	0.435	0.293	0.053	0.296	0.239	0.415	0.404	0.301
Bagging$_{J48}$	0.284	0.127	0.27	0.441	0.205	0.115	0.217	0.184	0.234	0.376	0.245
Bagging$_{Naive}$	0.421	0.188	0.383	0.492	0.211	0.116	0.254	0.171	0.379	0.365	0.298
Boosting$_{J48}$	0.343	0.22	0.362	0.397	0.356	0.231	0.282	0.202	0.306	0.322	0.302
Boosting$_{Naive}$	0.407	0.183	0.414	0.481	0.211	0.128	0.229	0.168	0.396	0.366	0.298
RF	0.43	0.175	0.313	0.434	0.293	0.186	0.267	0.217	0.392	0.376	0.308
LR	0.283	0.04	0.367	0.492	0.0	0.079	0.217	0.105	0.368	0.222	0.217

Table 4 NofB20 scores of the seven composite algorithms and baseline approaches

Algorithms	ant	camel	ivy	jedit	log4j	lucene	poi	prop	tomcat	xalan	Average
Ave	87	73	13	82	15	33	8	13	30	27	38.1
Max	79	77	13	81	15	33	6	11	32	24	37.1
CODEP$_{Logistic}$	88	77	12	23	15	76	8	12	22	19	35.2
Bagging$_{J48}$	80	92	10	58	12	82	6	11	27	28	40.6
Bagging$_{Naive}$	93	76	13	0	20	80	6	10	24	22	34.4
Boosting$_{J48}$	66	74	10	26	20	101	4	14	19	20	35.4
Boosting$_{Naive}$	54	42	7	31	15	26	3	7	23	20	22.8
RF	69	64	11	65	16	84	4	9	25	25	37.2
LR	81	87	11	15	18	26	7	13	31	24	31.3

Table 5 Area under curve-receiver operating characteristic scores of the seven composite algorithms and baseline approaches

Algorithms	ant	camel	ivy	jedit	log4j	lucene	poi	prop	tomcat	xalan	Average
Ave	0.811	0.63	0.813	0.773	0.824	0.675	0.649	0.681	0.815	0.786	0.746
Max	0.804	0.623	0.814	0.768	0.841	0.687	0.651	0.702	0.813	0.782	0.749
CODEP$_{Logistic}$	0.798	0.633	0.82	0.769	0.798	0.68	0.709	0.701	0.816	0.796	0.752
Bagging$_{J48}$	0.684	0.615	0.752	0.733	0.764	0.677	0.612	0.629	0.733	0.703	0.69
Bagging$_{Naive}$	0.791	0.616	0.802	0.759	0.819	0.639	0.632	0.673	0.807	0.738	0.728
Boosting$_{J48}$	0.689	0.578	0.722	0.637	0.764	0.548	0.606	0.606	0.729	0.695	0.657
Boosting$_{Naive}$	0.712	0.568	0.778	0.691	0.763	0.61	0.661	0.649	0.74	0.736	0.691
RF	0.741	0.634	0.698	0.706	0.78	0.636	0.605	0.665	0.782	0.705	0.695
LR	0.805	0.617	0.782	0.764	0.751	0.632	0.669	0.676	0.781	0.755	0.723

Table 6 Mean average precision scores of the seven composite algorithms and baseline approaches

Algorithms	MAP
Ave	0.173
Max	0.171
CODEP$_{Logistic}$	0.169
Bagging$_{J48}$	0.12
Bagging$_{Naive}$	0.131
Boosting$_{J48}$	0.103
Boosting$_{Naive}$	0.056
RF	0.11
LR	0.154

by 2.37% and 1.18%, and to the MAP scores of LR by 12.34% and 11.04%.

The most effective algorithms, measured in terms of F-measure and NofB20, are Max and Bagging$_{J48}$, respectively. Max, Boosting$_{J48}$, and RF outperform CODEP$_{Logistic}$ in terms of both F-measure and NofB20.

RQ2 In terms of cost effectiveness, how effective are the composite algorithms for different LOC inspection percentages?

• **Motivation** By default, we set the LOC inspection percentage as 20%, which is the setting used for RQ1. To answer this research question, we investigated the effectiveness of the seven composite algorithms for different LOC inspection percentages. The answer to this research question can shed light on whether some of the composite algorithms outperform CODEP when considering other cost settings.

• **Approach** To answer this research question, we calculated the cost effectiveness scores of three composite algorithms: Max (the algorithm that achieves the best F-measure scores), Bagging$_{J48}$ (the algorithm that achieves the best NofB20 scores), and CODEP$_{Logistic}$. We investigated different LOC inspection percentages, from 5% to 95%, at 5% intervals. Next, we plotted the cost effectiveness graphs, which show the number of buggy classes that can be detected by inspecting different percentages of LOCs.

• **Results** Figure 2 presents the cost effectiveness graphs of Max, Bagging$_{J48}$, and CODEP$_{Logistic}$ for the ant, camel, ivy, jedit, log4j, lucene, poi, prop, tomcat, and xalan datasets. In the graphs, for most of the datasets, we notice that Max and Bagging$_{J48}$ perform better than or as well as CODEP$_{Logistic}$ for a wide range of LOC inspection percentages. For camel, the percentage range for which Max achieves a better performance than CODEP$_{Logistic}$ is from around 15% to 80%, and the performance of CODEP$_{Logistic}$ is better than that of Bagging$_{J48}$ when the percentage is between 60% and 90%. For ivy, the percentage range for which Max achieves the best performance is narrower, i.e., from around 5% to 45%. For jedit, Max and Bagging$_{J48}$ perform better than CODEP$_{Logistic}$ for a wide range of percentages from around 5% to 65%, and for log4j, Max and Bagging$_{J48}$ perform better than CODEP$_{Logistic}$ for a range of percentages from around 55% to 95%. For lucene, the percentage range for which Bagging$_{J48}$ achieves the best performance is wide, i.e., from around 20% to 95%. For prop, Bagging$_{J48}$ performs better than CODEP$_{Logistic}$ for a percentage range from around 35% to 85%. For tomcat, for the percentage ranges from around 10% to 30% and 55% to 75%, Max achieves better cost effectiveness scores than CODEP$_{Logistic}$. For xalan, the percentage range for which Bagging$_{J48}$ achieves a better performance than CODEP$_{Logistic}$ is from around 5% to 65%. However, for ant and poi, CODEP$_{Logistic}$ performs better than Max and Bagging$_{J48}$ for a wide range of percentages.

In general, the composite algorithm Max performs better than or as well as CODEP$_{Logistic}$ for a wide range of LOC percentages for most projects.

5 Discussion

5.1 NASA dataset

To evaluate the performance of the composite classifiers better, we also applied the algorithms to NASA datasets, which

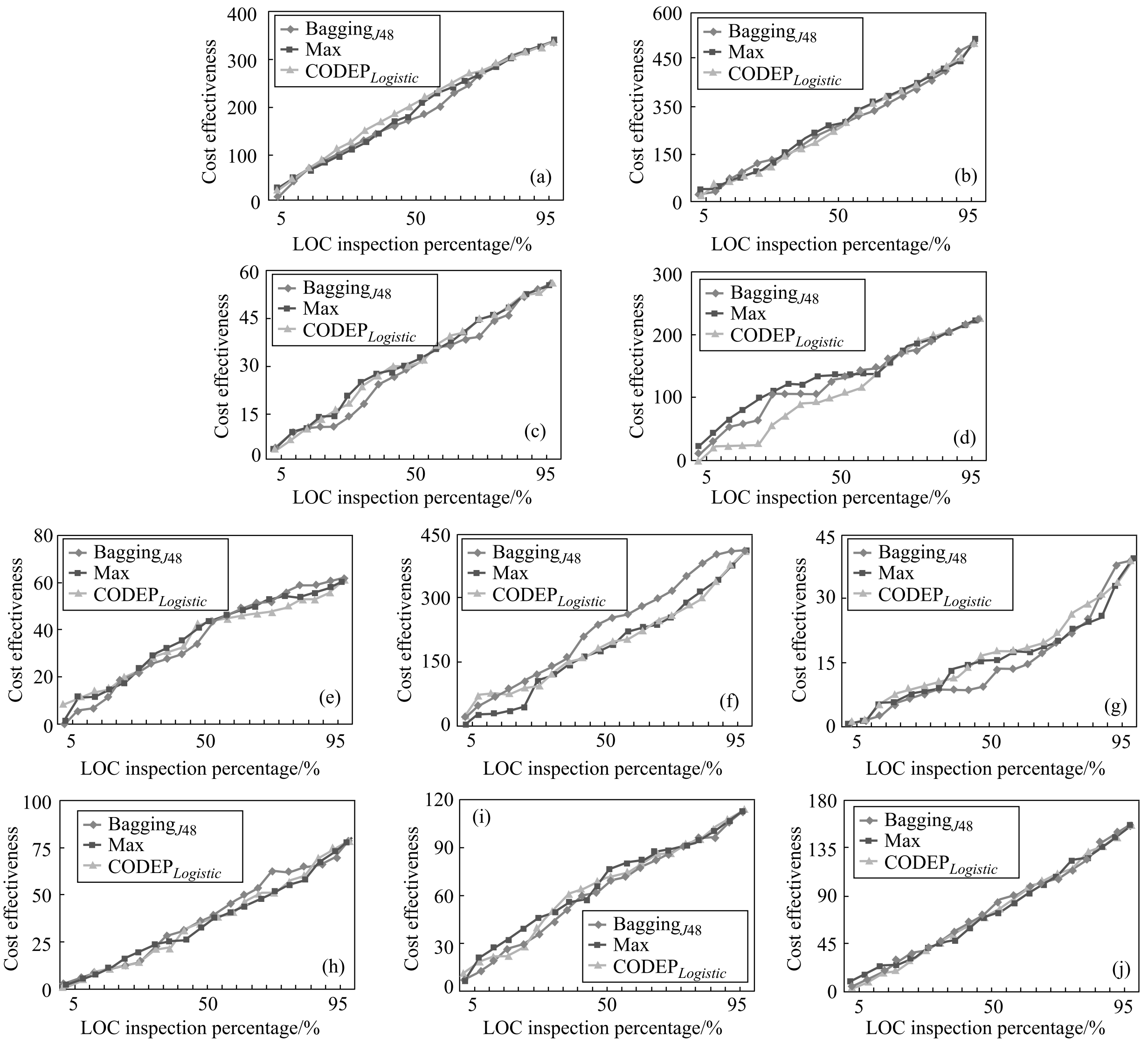


Fig. 2 Cost effectiveness graphs for the ten datasets. (a) ant; (b) camel; (c) ivy; (d) jedit; (e) log4j; (f) lucene; (g) poi; (h) prop; (i) tomcat; (j) xalan

are also a part of the PROMISE repository. We chose five projects that contain the same features: CM1, MW1, PC1, PC3, and PC4. Each of the projects contains a set of classes labeled as defective or clean and their corresponding features, e.g., LOC and the CK metric. Table 7 summarizes the statistics of each project.

Table 7 Software projects used from NASA datasets

Project	Instances	Defect-prone instances	Percentage/%
CM1	327	42	12.8
MW1	253	27	10.7
PC1	705	61	8.7
PC3	1,077	134	12.4
PC4	1,458	178	12.2

Our experiments were performed in the context of cross-project defect prediction. We combined the instances of four projects and trained a model, and then applied the model to the remaining project. We report the average performance of the prediction techniques.

Table 8 presents the F-measure scores of CODEP$_{Logistic}$ and LR as compared with those of Ave, Max, Bagging$_{J48}$, Bagging$_{Naive}$, Boosting$_{J48}$, Boosting$_{Naive}$, and RF. The F-measure scores of CODEP$_{Logistic}$ vary from 0.0 to 0.296. Across the five datasets, the average F-measure of CODEP$_{Logistic}$ is 0.109. In Table 8, we can see that the average F-measure scores of Max, Bagging$_{Naive}$, Boosting$_{J48}$, Boosting$_{Naive}$, and RF are 0.329, 0.313, 0.146, 0.312, and 0.112, respectively, which are superior to the average F-measure of CODEP$_{Logistic}$ by 201.83%, 187.16%, 33.94%, 186.24%, and 2.75%, respectively. The F-measure scores of LR vary from 0.0 to 0.243. Across the five datasets, the average F-measure of LR is 0.107, which is lower than that of all the composite algorithms and CODEP$_{Logistic}$, except Bagging$_{J48}$. Max achieves the best F-measure scores; its F-measure scores vary from 0.306 to 0.393 and the average score is 0.329. Meanwhile, Ave and Bagging$_{J48}$ do not perform as well as CODEP$_{Logistic}$ in terms of F-measure.

Table 8 F-measure scores of the classifiers

Algorithms	CM1	MW1	PC1	PC3	PC4	Average
Ave	0.154	0.0	0.312	0.063	0.0	0.106
Max	0.306	0.325	0.314	0.393	0.307	0.329
CODEP$_{Logistic}$	0.176	0.0	0.296	0.053	0.022	0.109
Bagging$_{J48}$	0.184	0.0	0.141	0.064	0.0	0.078
Bagging$_{Naive}$	0.317	0.483	0.306	0.278	0.18	0.313
Boosting$_{J48}$	0.22	0.0	0.255	0.138	0.118	0.146
Boosting$_{Naive}$	0.309	0.5	0.298	0.24	0.215	0.312
RF	0.19	0.0	0.247	0.076	0.047	0.112
LR	0.217	0.0	0.243	0.075	0.0	0.107

Table 9 presents the NofB20 scores of the algorithms. The NofB20 scores of CODEP$_{Logistic}$ vary from 4 to 31. Across the five datasets, the average NofB20 score of CODEP$_{Logistic}$ is 15.2. In Table 9, we can see that the average NofB20 scores of Max, Bagging$_{J48}$, Boosting$_{J48}$, and RF are 21, 16.2, 18, and 20, respectively, which are superior to the average NofB20 score of CODEP$_{Logistic}$ by 38.16%, 6.58%, 18.3%, and 31.58%, respectively. The NofB20 scores of LR vary from 4 to 44. Across the five datasets, the average NofB20 score of LR is 16.8, the average NofB20 scores of Max, Boosting$_{J48}$, and RF are superior to that of LR by 25%, 7.14%, and 19.05%, respectively. Max achieves the highest NofB20 score; its NofB20 scores vary from 7 to 40 and the average score is 21. Meanwhile, the other three composite algorithms that we investigated in this study do not perform as well as CODEP$_{Logistic}$ in terms of NofB20.

Table 9 NofB20 scores of the classifiers

Algorithms	CM1	MW1	PC1	PC3	PC4	Average
Ave	5	7	19	16	26	14.6
Max	10	7	19	29	40	21
CODEP$_{Logistic}$	4	7	20	14	31	15.2
Bagging$_{J48}$	6	3	22	21	29	16.2
Bagging$_{Naive}$	5	10	13	9	17	10.8
Boosting$_{J48}$	8	0	26	24	32	18
Boosting$_{Naive}$	3	10	19	15	16	12.6
RF	7	8	25	34	26	20
LR	5	4	18	13	44	16.8

Table 10 presents the AUC-ROC scores of CODEP$_{Logistic}$ and LR in comparison with those of Ave, Max, Bagging$_{J48}$, Bagging$_{Naive}$, Boosting$_{J48}$, Boosting$_{Naive}$, and RF. The AUC-ROC scores of CODEP$_{Logistic}$ vary from 0.623 to 0.821. Across the five datasets, the average AUC-ROC score of CODEP$_{Logistic}$ is 0.732. In Table 10, we can see that the average AUC-ROC scores of Ave, Max, Bagging$_{Naive}$, and Boosting$_{Naive}$ are 0.744, 0.735, 0.757, and 0.733, respectively, which are superior to the average AUC-ROC score of CODEP$_{Logistic}$ by 1.64%, 0.41%, 3.42%, and 0.14%, respectively. The AUC-ROC scores of LR vary from 0.59 to 0.768. Across the five datasets, the average AUC-ROC score of LR is 0.702, the average AUC-ROC scores of Ave, Max, Bagging$_{Naive}$, and Boosting$_{Naive}$ are superior to that of LR by 5.98%, 4.7%, 7.83%, and 4.42%, respectively. Bagging$_{Naive}$ achieves the highest AUC-ROC score and its NofB20 scores vary from 0.705 to 0.793 and the average score is 0.757. Meanwhile, the other three composite algorithms that we investigated in this study do not perform as well as CODEP$_{Logistic}$ in terms of AUC-ROC.

Table 10 Area under the receiver operator characteristic curve scores of the classifiers

Algorithms	CM1	MW1	PC1	PC3	PC4	Average
Ave	0.683	0.688	0.83	0.786	0.732	0.744
Max	0.655	0.698	0.813	0.784	0.726	0.735
CODEP$_{Logistic}$	0.623	0.711	0.821	0.763	0.741	0.732
Bagging$_{J48}$	0.577	0.602	0.831	0.735	0.666	0.682
Bagging$_{Naive}$	0.705	0.769	0.793	0.746	0.771	0.757
Boosting$_{J48}$	0.673	0.546	0.794	0.726	0.671	0.682
Boosting$_{Naive}$	0.703	0.763	0.8	0.633	0.767	0.733
RF	0.588	0.677	0.796	0.716	0.655	0.686
LR	0.59	0.67	0.74	0.743	0.768	0.702

Table 11 presents the MAP scores of CODEP$_{Logistic}$ and LR in comparison with those of Ave, Max, Bagging$_{J48}$, Bagging$_{Naive}$, Boosting$_{J48}$, Boosting$_{Naive}$, and RF. Across the five datasets, the MAP score of CODEP$_{Logistic}$ is 0.054, and the MAP score of LR is 0.051. In Table 6, we can see that the MAP scores of Ave, Max, Bagging$_{Naive}$, and Boosting$_{Naive}$ are 0.061, 0.075, 0.101, and 0.109, respectively, which are superior to the MAP score of CODEP$_{Logistic}$ by 12.96%, 38.89%, 87.04%, and 101.85%, and to the MAP score of LR by 19.61%, 47.06%, 98.04%, and 113.73%, respectively. Boosting$_{Naive}$ achieves the highest AUC-ROC score. Meanwhile, the other three composite algorithms that we investigated in this study do not perform as well as CODEP$_{Logistic}$ in terms of MAP.

5.2 Performance of the combined defect predictor with meta classifier J48

We also investigated the performance of CODEP with another meta classifier. We report the result of CODEP with J48 as the meta classifier, referred to as CODEP$_{J48}$. Table

12 shows the F-measure, NofB20, and AUC-ROC scores of CODEP$_{J48}$ when applied to the PROMISE datasets. The average F-measure score of CODEP$_{J48}$ is 0.213, which is lower than that of CODEP$_{Logistic}$ by 41.31%. The average NofB20 score of CODEP$_{J48}$ is 18.5, which is lower than that of CODEP$_{Logistic}$ by 90.27%. The average AUC-ROC score of CODEP$_{J48}$ is 0.668, which is lower than that of CODEP$_{Logistic}$ by 12.57%. The MAP score of CODEP$_{J48}$ is 0.086, which is lower than that of CODEP$_{Logistic}$ by 96.51%. From the above results, we can draw the conclusion that CODEP$_{Logistic}$ performs better than CODEP$_{J48}$ on the PROMISE datasets.

Table 11 Mean average precision scores of the classifiers

Algorithms	MAP
Ave	0.061
Max	0.075
CODEP$_{Logistic}$	0.054
Bagging$_{J48}$	0.029
Bagging$_{Naive}$	0.101
Boosting$_{J48}$	0.022
Boosting$_{Naive}$	0.109
RF	0.035
LR	0.051

5.3 Performance of single classifiers as compared with maximum voting

In the experimental results above, we observe that the performance of Max is best among the composite classifiers in most situations. In this subsection, we examine the performance of the six single classifiers, i.e., logistic regression, BN, RBF network, MLP, ADTree, and DT, as compared with Max. Table 13 shows the average F-measure scores of Max and the six single classifiers. In the table, we can see that Max performs better than all six single classifiers, with an F-measure score of 0.412. Among the six single classifiers, BN outperforms the other five by substantial margins, with an F-measure score of 0.405.

5.4 Time complexity

The efficiency of the algorithms affects its practical usage. Thus, we analyze the time complexity of the composite algorithms in this section. We suppose the time complexity for a single classifier is $O(k)$. Then, the time complexity of Ave and Max is $O(N \times k)$, where N denotes the number of underlying classifiers. As CODEP is a two-level composite algorithm, the time complexity of the first level with N underlying classifiers is $O(N \times k)$ and the time complexity of the second level is $O(k)$, and therefore, the total time complexity of CODEP is $O(N \times k^2)$. For bagging and boosting, the time complexity is $O(T \times k)$, where T donates the iteration numbers. The time complexity of RF is $O(M \times F \times k)$, where M is the number of trees that are planned to be built, and F is the number of features planned to be sampled at each node. From the above, we can draw the conclusion that among the composite algorithms, the time complexity of CODEP is highest.

Table 13 Performance of single classifiers as compared with maximum voting

Algorithms	Average F-measure
Max	0.412
Logistic Regression	0.217
Bayes Network	0.405
RBF Network	0.039
Multi-layer Perceptron	0.268
ADTree	0.244
Decision Table	0.224

5.5 Threats to validity

Threats to internal validity are related to the errors in our experiments. We double-checked our experiments and implementation. However, there could be errors that we did not notice. Threats to external validity are related to the generalizability of our results. We analyzed 5,305 instances from 10 open-source software projects. In the future, we plan to reduce this threat further by analyzing even more defect data from more open-source and commercial software projects. Threats to construct validity refer to the suitability of our evaluation metrics. We used cost effectiveness and F-measure, which were also used in previous software engineering studies to evaluate the effectiveness of various prediction techniques [13–18,44]. Thus, we believe there is little threat to construct validity.

6 Related work

In this section, we present related studies on defect predic-

Table 12 Performance of combined defect predictor with meta classifier J48

Algorithms	ant	camel	ivy	jedit	log4j	lucene	poi	prop	tomcat	xalan	Average
F-measure	0.036	0.168	0.441	0.369	0.111	0.054	0.28	0.105	0.37	0.199	0.213
NofB20	19	9	3	21	13	76	8	4	24	8	18.5
AUC-ROC	0.726	0.533	0.766	0.748	0.633	0.563	0.628	0.648	0.728	0.708	0.668

tion, cross-project defect prediction, and composite prediction techniques for defect prediction.

6.1 Classical defect prediction

A number of studies have been conducted on defect prediction, in which different metrics, such as code complexity, object-oriented metrics, process metrics, and code locations, were analyzed to build prediction models to identify defective code elements [17, 18, 45–51]. D'Ambros et al. analyzed different software metrics and approaches used for defect prediction in their survey [6]. Rahman et al. analyzed the applicability and efficacy of code metrics and process metrics from several different perspectives for defect prediction [13]. In most of these studies, machine learning techniques, such as logistic regression [7, 29, 30], RBF network) [31], BN [9], DT [5], MLP [32], and decision trees [52], were leveraged to predict the defect proneness of classes/files/modules in a software project by training a classification model on past data of the same software project, i.e., using a within-project setting. Zimmermann et al. used network analysis on dependency graphs to identify defect-prone central program units [1]. Kim et al. introduced the problem of predicting buggy software changes. They used a machine learning classifier to determine whether a new software change is more to prior buggy changes or clean changes [44]. D'Ambros et al. presented a benchmark for defect prediction and also provided an extensive comparison of well-known defect prediction approaches [47]. In the within-project setting, defect prediction models are trained and applied on classes/files/modules from the same project. However, in practice, it is rare that sufficient training data are available for a new project; however, ample data from other projects exist.

6.2 Cross-project defect prediction

In the last few years, a substantial effort has been devoted to using cross-project strategies for predicting the defect proneness of software entities [53]. This means using defect data from other projects to improve defect prediction for a target project. Zimmermann et al. proposed a cross-project defect prediction approach that trains a model on a source project, and applies the model to a target project [9]. They listed the factors that software engineers should consider before selecting a project as a source project for a given target project. Turhan et al. employed a k-nearest neighbor algorithm for cross-project defect prediction [10]. Their algorithm selects instances from other projects to be used as training data for a target project; for every unlabeled instance in a target project, they selected 10 nearest neighbor instances from source projects. Peters et al. proposed the Peters filter for cross-company defect prediction, which, similarly to Turhan et al.'s method, uses a nearest neighbor approach to select instances from source projects [16]. Nam et al. noted that the poor performance of cross-project defect prediction is in general due to the different feature distribution between the source and target projects [17]. They then proposed TCA+, a novel transfer defect learning approach, which makes feature distributions in source and target projects similar [17]. Canfora et al. proposed a multi-objective approach for cross-project defect prediction, which uses a genetic algorithm to build a multi-objective logistic regression model [18]. Although in all of the above studies the gap between the accuracy of within-project and cross-project defect predictions was reduced, cross-project defect prediction still represents one of the main challenges in the defect prediction field.

Recently, Panichella et al. proposed a state-of-the-art cross-project defect prediction algorithm named CODEP that uses a meta classification algorithm to combine the results of six basic classification algorithms, i.e., logistic regression, BN, RBF network, MLP, ADTrees, and DT [11]. The best results were achieved when logistic regression was used as the meta classification algorithm, i.e., CODEP$_{Logistic}$. In our work, we focused on finding effective composite algorithms for cross-project defect prediction that can outperform CODEP$_{Logistic}$. We investigated seven composite algorithms proposed in the machine learning community, i.e., Ave, Max, Bagging$_{J48}$, Bagging$_{Naive}$, Boosting$_{J48}$, Boosting$_{Naive}$, and RF. These algorithms use different strategies to combine the results of a number of basic classifiers. In our experiments, we used the same basic classifiers as CODEP, and the results of the experiments show that three out of the seven composite algorithms perform better than CODEP in terms of both F-measure and cost effectiveness.

7 Conclusion and future work

In this paper, we examined the effectiveness of seven composite algorithms proposed in the machine learning community, i.e., Ave, Max, Bagging$_{J48}$, Bagging$_{Naive}$, Boosting$_{J48}$, Boosting$_{Naive}$, and RF, for cross-project defect prediction, with the aim of finding one or more algorithms that outperform CODEP, the state-of-the-art cross-project defect prediction technique. We evaluated the composite algorithms using two metrics, F-measure and cost effectiveness. We performed experiments on defect datasets from ten different open-source software projects containing a total 5,305 instances, i.e., classes. The results show that the performance of Max is best in terms of F-measure and it achieves an average F-measure score of 0.412, which is superior to the average F-measure of $\text{CODEP}_{Logistic}$ by 36.88%. In addition, the performance of Bagging_{J48} is best in terms of cost effectiveness and it achieves an average NofB20 score of 40.6, which is superior to the average NofB20 score of $\text{CODEP}_{Logistic}$ by 15.34%. In addition to these two algorithms, several other algorithms also outperform $\text{CODEP}_{Logistic}$ in terms of F-measure and/or cost effectiveness. As the projects used in our experiments belong to different fields, the results obtained when using them showed that the composite algorithms achieve different performance levels on different projects. As compared to all the composite algorithms we examined, we found that the performance of Max is the best, and therefore we recommend that users apply Max in practice.

In the future, we plan to investigate additional composite algorithms or create a custom composite algorithm that yields a better cross-project defect prediction. We also plan to reduce the threats to external validity further by performing experiments with even more instances from more projects.

Acknowledgements This research was supported by the National Natural Science Foundation of China (Grant No. 61602403) and National Key Technology R&D Program of the Ministry of Science and Technology of China (2015BAH17F01).

References

1. Zimmermann T, Nagappan N. Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th International Conference on Software Engineering. 2008, 531–540
2. Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener A. Defect prediction from static code features: current results, limitations, new approaches. Automated Software Engineering, 2010, 17(4): 375–407
3. Jiang T, Tan L, Kim S. Personalized defect prediction. In: Proceedings of the 28th IEEE International Conference on Automated Software Engineering. 2013, 279–289
4. Lessmann S, Baesens B, Mues C, Pietsch S. Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Transactions on Software Engineering, 2008, 34(4): 485–496
5. Liu Y, Khoshgoftaar T M, Seliya N. Evolutionary optimization of software quality modeling with multiple repositories. IEEE Transactions on Software Engineering, 2010, 36(6): 852–864
6. D'Ambros M, Lanza M, Robbes R. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empirical Software Engineering, 2012, 17(4–5): 531–577
7. Nagappan N, Ball T, Zeller A. Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering. 2006, 452–461
8. Kitchenham B A, Mendes E, Travassos G H. Cross versus within-company cost estimation studies: a systematic review. IEEE Transactions on Software Engineering, 2007, 33(5): 316–329
9. Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the European Software Engineering Conference and the ACM Sigsoft International Symposium on Foundations of Software Engineering. 2009, 91–100
10. Turhan B, Menzies T, Bener A B, Di Stefano J. On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering, 2009, 14(5): 540–578
11. Panichella A, Oliveto R, De Lucia A. Cross-project defect prediction models: L'union fait la force. In: Proceedings of Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering. 2014, 164–173
12. Arisholm E, Briand L C, Fuglerud M. Data mining techniques for building fault-proneness models in telecom java software. In: Proceedings of the 18th IEEE International Symposium on Software Reliability. 2007, 215–224
13. Rahman F, Devanbu P. How, and why, process metrics are better. In: Proceedings of the International Conference on Software Engineering. 2013, 432–441
14. Rahman F, Posnett D, Devanbu P. Recalling the imprecision of cross-project defect prediction. In: Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering. 2012, 61
15. Rahman F, Posnett D, Herraiz I, Devanbu P. Sample size vs. bias in defect prediction. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering. 2013, 147–157
16. Peters F, Menzies T, Marcus A. Better cross company defect prediction. In: Proceedings of the 10th IEEE Working Conference on Mining Software Repositories. 2013, 409–418
17. Nam J, Pan S J, Kim S. Transfer defect learning. In: Proceedings of the International Conference on Software Engineering. 2013, 382–391
18. Canfora G, De Lucia A, Di Penta M, Oliveto R, Panichella A, Panichella S. Multi-objective cross-project defect prediction. In: Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation. 2013, 252–261
19. Baeza-Yates R, Ribeiro-Neto B. Modern Information Retrieval. Vol 463. New York: ACM Press, 1999
20. Schütze H. Introduction to information retrieval. In: Proceedings of the International Communication of Association for Computing Machinery Conference. 2008
21. Zhang Y, Lo D, Xia X, Sun J. An empirical study of classifier combination for cross-project defect prediction. In: Proceedings of the 39th IEEE Annual Computer Software and Applications Conference. 2015, 264–269
22. Chidamber S R, Kemerer C F. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 1994, 20(6): 476–493
23. Henderson-Sellers B. Object-Oriented Metrics, Measures of Complexity. Upper Saddle River, NJ: Prentice Hall, 1996
24. Bansiya J, Davis C G. A hierarchical model for object-oriented design

quality assessment. IEEE Transactions on Software Engineering, 2002, 28(1): 4–17

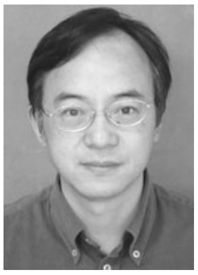
25. Tang M, Kao M, Chen M. An empirical study on object-oriented metrics. In: Proceedings of the 6th International Symposium on Software Metrics. 1999, 242–249
26. Martin R. OO design quality metrics — an analysis of dependencies. IEEE Transactions on Software Engineering, 1994, 20(6): 476–493
27. McCabe T. A complexity measure. IEEE Transactions on Software Engineering, 1976, 2(4): 308–320
28. Jureczko M, Madeyski L. Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. 2010
29. Basili V R, Briand L C, Melo W L. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering, 1996, 22(10): 751–761
30. Gyimothy T, Ferenc R, Siket I. Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Transactions on Software Engineering, 2005, 31(10): 897–910
31. Bezerra M E, Oliveira A L, Meira S R. A constructive rbf neural network for estimating the probability of defects in software modules. In: Proceedings of the International Joint Conference on Neural Networks. 2007, 2869–2874
32. Ceylan E, Kutlubay F O, Bener A B. Software defect identification using machine learning techniques. In: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications. 2006, 240–247
33. Okutan A, Yıldız O T. Software defect prediction using bayesian networks. Empirical Software Engineering, 2014, 19(1): 154–181
34. Nelson A, Menzies T, Gay G. Sharing experiments using open-source software. Software: Practice and Experience, 2011, 41(3): 283–305
35. Han J, Kamber M. Data Mining: Concepts and Techniques. 2nd ed . San Francisco: Morgan Kaufmann, 2006
36. Koller D, Friedman N. Probabilistic Graphical Models: Principles and Techniques. Cambridge: MIT Press, 2009
37. Buhmann M D. Radial basis functions. Acta Numerica, 2000, 9: 1–38
38. Quinlan J R. Bagging, boosting, and c4. 5. In: Proceedings of the 13th National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference. 1996, 725–730
39. Breiman L. Random forests. Machine Learning, 2001, 45(1): 5–32
40. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten I H. The weka data mining software: an update. ACM SIGKDD Explorations Newsletter, 2009, 11(1): 10–18
41. Wu R, Zhang H, Kim S, Cheung S C. Relink: recovering links between bugs and changes. In: Proceedings of the ACM Sigsoft Symposium and the European Conference on Foundations of Software Engineering. 2011, 15–25
42. Tian Y, Lawall J, Lo D. Identifying linux bug fixing patches. In: Proceedings of the 34th International Conference on Software Engineering. 2012, 386–396
43. Rao S, Kak A. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th Working Conference on Mining Software Repositories. 2011, 43–52
44. Kim S, Whitehead E J, Zhang Y. Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering, 2008, 34(2): 181–196
45. Chidamber S R, Kemerer C F. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 1994, 20(6): 476–493
46. Moser R, Pedrycz W, Succi G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th ACM/IEEE International Conference on Software Engineering. 2008, 181–190
47. D'Ambros M, Lanza M, Robbes R. An extensive comparison of bug prediction approaches. In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories. 2010, 31–41
48. Ma Y, Luo G, Zeng X, Chen A. Transfer learning for cross-company software defect prediction. Information and Software Technology, 2012, 54(3): 248–256
49. Xia X, Lo D, Wang X, Yang X. Collective personalized change classification with multiobjective search. IEEE Transactions on Reliability, 2016, 65: 1–20
50. Yang X, Lo D, Xia X, Zhang Y. Deep learning for just-in-time defect prediction. In: Proceedings of the IEEE International Conference on Software Quality, Reliability and Security. 2015, 17–26
51. Xuan X, Lo D, Xia X, Tian Y. Evaluating defect prediction approaches using a massive set of metrics: an empirical study. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing. 2015, 1644–1647
52. Nelson A, Menzies T, Gay G. Sharing experiments using open-source software. Software: Practice and Experience, 2011, 41(3): 283–305
53. Xia X, Lo D, Pan S J, Nagappan N, Wang X. Hydra: massively compositional model for cross-project defect prediction. IEEE Transactions on Software Engineering, 2016, 42: 977–998

Yun Zhang is a PhD candidate in College of Computer Science and Technology, Zhejiang University, China. Her research interests include software analysis, empirical study, and mining software repository.

David Lo received his PhD degree in computer science from the School of Computing, National University of Singapore, Singapore in 2008. He is currently an assistant professor in the School of Information Systems, Singapore Management University (SMU), Singapore. He has close to ten years of experience in software engineering and data mining research and has more than 130 publications in these areas. He received the Lee Foundation Fellow for Research

Excellence from the SMU in 2009. He has won a number of research awards including an ACM distinguished paper award for his work on bug report management. He has published in many top international conferences in software engineering, programming languages, data mining and databases, including ICSE, FSE, ASE, PLDI, KDD, WSDM, TKDE, ICDE, and VLDB. He has also served on the program committees of ICSE, ASE, KDD, VLDB, and many others. He is a steering committee member of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) which is a merger of the two major conferences in software engineering, namely CSMR and WCRE. He will also serve as the general chair of ASE 2016. He is a leading researcher in the emerging field of software analytics and has been invited to give keynote speeches and lectures on the topic in many venues, such as the 2010 Workshop on Mining Unstructured Data, the 2013 Génie Logiciel Empirique Workshop, the 2014 International Summer School on Leading Edge Software Engineering, and the 2014 Estonian Summer School in Computer and Systems Science.

Xin Xia received his PhD degree in computer science from the College of Computer Science and Technology, Zhejiang University (ZJU), China in 2014. He is currently a research assistant professor in the college of computer science and technology at ZJU. His research interests include software analysis, empirical study, and mining software repository.



Jianling Sun received his PhD degree in computer science from the College of Computer Science and Technology, Zhejiang University (ZJU), China in 1992. He is currently a professor in the College of Computer Science and Technology, ZJU. His research interests include database and Web technology.