An Empirical Study of Classifier Combination for Cross-Project Defect Prediction

Yun Zhang*, David Lo†, Xin Xia*‡ , Jianling Sun*
*College of Computer Science and Technology, Zhejiang University, Hangzhou, China
†School of Information Systems, Singapore Management University, Singapore
yunzhang28@zju.edu.cn, davidlo@smu.edu.sg, {xxia, sunjl}@zju.edu.cn

***Abstract*—To help developers better allocate 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 past history of buggy classes. These techniques work well as long as a sufficient amount of data is available to train a prediction model. However, there is rarely enough training data for new software projects. To deal with this problem, cross-project defect prediction, which transfers a prediction model trained using data from one project to another, has been proposed and is regarded as a new challenge for defect prediction. So far, only a few cross-project defect prediction techniques have been proposed. To advance the state-of-the-art, in this work, we investigate 7 composite algorithms, which integrate multiple machine learning classifiers, to improve cross-project defect prediction. To evaluate the performance of the composite algorithms, we perform 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 compare the composite algorithms with CODEP$_{Logistic}$, which is the latest cross-project defect prediction algorithm proposed by Panichella et al. [1], in terms of two standard evaluation metrics: cost effectiveness and F-measure. Our experiment results show that several algorithms outperform CODEP$_{Logistic}$: Max performs the best in terms of F-measure and its average F-measure outperforms that of CODEP$_{Logistic}$ by 36.88%. Bagging$_{J48}$ performs the best in terms of cost effectiveness and its average cost effectiveness outperforms that of CODEP$_{Logistic}$ by 15.34%.**

***Keywords*—Defect Prediction, Cross-Project, Classifier Combination**

I. Introduction

To build high quality software, developers need to invest much testing and debugging efforts. However, developers often have limited resources and need to prioritize such efforts. To help developers prioritize testing and debugging efforts, software defect prediction techniques have been proposed in the literature. A software defect predicting technique will identify more likely defect-prone software components by constructing a predictive classification model constructed from features such as lines of code, code complexity and number of symbols [2], [3], [4], [5]. 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 work well when a sufficient amount of training data is available [6]. Unfortunately, training data is often limited for new projects with little or no historical bug data. For such cases, engineers need to use data from other projects and companies [7]. Cross-project 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 [8], [9].

Recently, Panichella et al. proposed a composite approach, referred to as CODEP (COmbined DEfect Predictor), which combines different and complementary classifiers learned by different machine learning algorithms, for cross-project defect prediction [1]. Their experiment results show that CODEP outperforms many existing cross-project defect prediction techniques. However, in the machine learning literature, many composite techniques have been proposed to combine multiple classification models. In this work, we want to investigate the applicability of existing composite techniques proposed in the machine learning literature for cross-project defect prediction and whether these composite techniques can outperform the state-of-the-art work for cross-project defect prediction namely CODEP.

In this paper, We use two well-known metrics to evaluate the performance of a predictive algorithm: cost effectiveness [10], [11], [12], [13] and F-measure [14], [15], [16], [12]. We compare the composite algorithms against the best variant of CODEP which uses logistic regression as a meta-learner – referred to as CODEP$_{Logistic}$. We evaluate the algorithms on defect datasets from 10 projects (i.e., ant, camel, ivy, jedit, log4j, lucene, poi, prop, tomcat, xalan) which are part of the PROMISE data repository[1]. The datasets contain a total of 5,305 instances along with their labels (i.e., defective or not). The experiment results show that several of the composite algorithms outperform CODEP$_{Logistic}$ in terms of F-measure and cost effectiveness. Among them, Max performs the best in terms of F-measure, and achieves an average score of 0.412 which improves that of CODEP$_{Logistic}$ by 36.88%; Bagging$_{J48}$ performs the best in terms of cost effectiveness, and achieves an average NofB20 score of 40.6 which improves that of CODEP$_{Logistic}$ by 15.34%.

In summary, the main contributions of this paper are:

1) We investigate the effectiveness of 7 different composite algorithms proposed in the machine learning literature for cross-project defect prediction in terms of cost effectiveness and F-measure.

‡Corresponding author.

[1] http://promisedata.googlecode.com/

2) We experiment on 10 defect datasets to demonstrate the effectiveness of the algorithms, and highlight promising algorithms with better performance than CODEP.

The remainder of the paper is organized as follows. We describe several classical classification algorithms and CODEP in Section II. We then present a number of composite algorithms in Section III. We present our experiments and results in Section IV. We discuss related work in Section V. We conclude and mention future work in Section VI.

II. Background

A. Classical Classification Techniques

Several machine learning techniques have been used to predict defect-prone source code classes/files/components, such as logistic regression [6], Radial Basis Function Network (RBF Network) [17], Bayesian network [18]. In this paper, we investigate six classification algorithms, namely Logistic Regression, Bayes Network, Radial Basis Function Network, Multi-layer Perceptron, Alternating Decision Trees, and Decision Table. We will use these classification algorithms to construct various underlying classifiers for our composite classification algorithms.

1) ***Logistic Regression:*** Logistic regression [19] 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 is directly estimated from the training data. Let $x = \{x_{f_1}, x_{f_2}, ...x_{f_m}\}$ denotes the vector representation of features of a data point x, and x_{f_i} denotes the value of the i-th feature of x, and $W = \{w_0, w_1, w_2, ...w_m\}$ denotes the weight vector associated to the features in x, w_0 is a bias parameter, and $w_i, i \in \{1, 2, ...m\}$ 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, 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. There are various methods to do this, such as gradient ascent.

2) ***Bayesian Network:*** Bayesian Network (BN) is a graphical model of probabilistic relationships representing the input feature space and label space [20]. It is a directed acyclic graph (DAG) and each node in a BN represents a feature 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 would construct a Bayesian network from the training set. And then during the prediction phase, this Bayesian network is used to predict the label of a new unlabeled instance.

3) ***Radial Basis Function Network:*** The Radial Basis Function (RBF) Network is an artificial neural network which uses radial basis function as activation functions [21]. 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 radial basis functions of the inputs and neuron parameters.

4) ***Multi-Layer Perceptron:*** Multi-Layer Perceptron (MLP) is another type of artificial neural network model that is trained using a supervised learning technique called back-propagation algorithm, which maps sets of input data onto a set of appropriate outputs. A MLP consists of multiple layers of nodes in a directed graph: one input layer, one output layer, and one or more hidden layers [1]. The output of a layer is used as the input of nodes in the subsequent layer. MLP can distinguish data that are not linearly separable, which is better than the standard linear perceptron.

5) ***Alternating Decision Trees:*** An Alternating Decision Tree (ADTree) consists of a tree structure with decision nodes and prediction nodes in an alternating order. Decision nodes specify conditions (e.g., $feature_1 < 0.5$, etc.) and each of them is connected to two prediction nodes – one corresponds to the case when the condition is evaluated to true, and another corresponds to the case when the condition is evaluated to 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 to true based on the feature values of the instance. The values of the in-between prediction nodes along the corresponding paths are then summed up. This sum is used to decide the class label (in our case: defective or clean) of an instance – i.e., if the sum is positive then an instance is defective, else it is clean.

6) ***Decision Table:*** A Decision Table (DT) can be regarded as an extension of a one-valued decision tree [19]. It is a rectangular table where the columns are features and rows are sets of decision rules. Each decision rule consists of two parts: (i) a pool of conditions which are linked through "and" and "or" logical operators; and (ii) an outcome which 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 try to find a good subset of features by running a feature reduction algorithm.

B. COmbined DEfect Predictor (CODEP)

CODEP is a two level composite algorithm which predicts the label of an instance (i.e., predicts if a class is defective or clean) [1]. In the first level, CODEP builds 6 underlying classifiers on a training set. These six classifiers are built by running each of the 6 classical classification algorithms described in Section II-A. Then, the confidence scores output by each classifier on each instances in the training set are collected to create a new dataset. In the second level, another classifier is built on the new dataset, which is referred to as the meta classifier. In this paper, we use logistic regression as the meta classifier since Panichella et al. have shown that it performs the best. To predict the label of an instance, CODEP first outputs the confidence scores of the 6 underlying classifiers, then these confidence scores are used as input to the meta classifier to predict the label of the instance.

III. Composite Algorithms

Panichella et al. have shown that the composite algorithm CODEP that they proposed outperforms many other approaches [1]. Based on their work, we investigate several other composite algorithms proposed in the machine learning literature aiming at finding one or more which perform better than CODEP.

A. *Overall Framework*

Figure 1 presents the overall framework that describes how we use the composite algorithms for cross-project defect prediction. The framework contains of two phases: model building phase and prediction phase. 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 II. In the prediction phase, this composite classifier would be used to predict if a new instance (i.e., class/file/component) would be defect-prone or not.

Our framework first extracts features from training instances (Step 1). Then, our framework applies a feature selection technique to select a subset of relevant features to further improve the prediction performance (Step 2). With these selected features, we next construct a composite predicting model by combining several underlying classifiers (Step 3). We investigate various composite classification techniques which are used to create composite classification models.

After the composite classifier has been built, in the prediction phase, it is used to predict whether a new instance would be defective or not. For each of such new instances, our framework first preprocesses and extracts features from it (Step 5), and represents it 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 would output the prediction result: defective or not (Step 8).

B. *Average Voting*

Average voting (Ave) is a voting method which combines confidence scores from different underlying classifiers [19]. We use the 6 classical classification algorithms described in Section II-A to build the underlying classifiers. Each underlying classifier outputs a confidence score for an instance which ranges from 0 to 1. In total, we have 6 confidence scores which corresponds to the 6 underlying classifiers. Next, Ave averages the 6 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.

C. *Maximum Voting*

Maximum voting (Max) is also a voting method which outputs the maximum confidence scores of different underlying classifiers [19]. We use the same 6 classical classification algorithms to build the underlying classifiers as Ave. Different from average voting, Max outputs the confidence score of an instance by selecting the maximum confidence score of the 6 underlying classifiers.

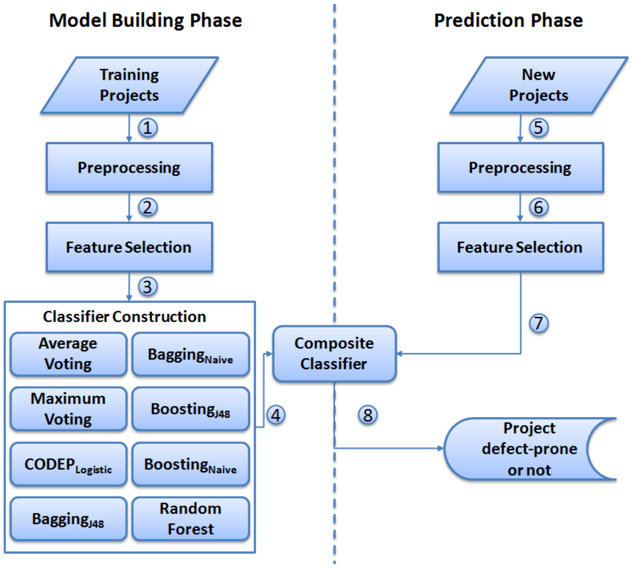

Fig. 1. Our Overall Cross-Project Defect Prediction Framework

D. *Bagging*

Bootstrap Aggregating (Bagging) [22] 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 bootstrapping sampling from D (i.e., random sampling with replacement) to generate m new datasets $D'_i, i \in \{1, 2, ...m\}$. The size of D'_i is denoted as n'_i, and $n'_i < n$. Next, Bagging trains a weak classifier (aka. an underlying classifier) from each dataset D'_i. For the prediction phase, all the output of m classifiers are combined to a single prediction using majority voting. In this paper, we use decision tree (J48) and naive Bayes as the underlying classifier of Bagging – denoted as Bagging$_{J48}$ and Bagging$_{Naive}$.

E. *Boosting*

Boosting [22] is used to generate strong classifiers out of weak classifiers. It can be combined with many other supervised learning algorithms to improve the 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 different importance of the instances. Generally speaking, instances which have been misclassified in the previous round would be assigned a higher weight, while instances which have been correctly classified would be assigned a lower weight. This re-weighting strategy makes the weak classifier in the current round focuses more on the misclassified instances. In this paper, similar to Bagging, we use decision tree (J48) and naive Bayes as the underlying classifier of Boosting – denoted as Boosting$_{J48}$ and Boosting$_{Naive}$.

TABLE I. SOFTWARE PROJECTS USED IN OUR STUDY

Project	Release	Instances	Defect-Prone Instances	(%)
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%

F. Random Forest

Random Forest [23] combines an ensemble of decision trees. RF takes advantage of both bagging and random feature selection for the tree building; each of the decision trees 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 are combined to a single prediction using majority voting.

IV. EXPERIMENTS AND RESULTS

In this section, we evaluate the effectiveness of the 7 composite algorithms and CODEP. The experimental environment is an Intel(R) Core(TM) T6570 2.10 GHz CPU, 4GB RAM desktop running Windows 7 (32-bit).

A. Experiment Setup

We evaluate the composite algorithms on defect datasets from 10 Java projects, i.e., ant, camel, ivy, jedit, log4j, lucene, poi, prop, tomcat, and xalan, that belong to the Promise repository. Each of the dataset contains a set of classes labeled as defective or clean and their corresponding metrics (e.g., LOC, Chidamber & Kemerer (CK) metric, etc.). Table I 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 (%).

Our experiments are performed in the context of cross-project defect prediction. Our experiments proceed in ten iterations. In the first iteration, we take classes from the first project "Ant" as a testing set, and combine instances from the other 9 projects as a training set. We learn a model from the training set and use it to predict the defect labels of instances from the test set. In the second iteration, we take instances of the second project "camel" as a testing set and combine the instances of the other projects as a training set. We repeat the same process eight more times, each time considering a different project as the testing set. We report the average performance of a prediction technique across the ten iterations.

We use the implementations of the 6 classification techniques and 7 composite algorithms in Weka [24][2]. For the average voting, maximum voting, CODEP and random forest, we use their default settings in Weka. And for the bagging and boosting, we set the number of iterations to 10.

B. Evaluation Metrics

We use two performance metrics for our evaluation: cost effectiveness and F-measure. These two measures are useful in different situations.

1) Cost Effectiveness: Cost effectiveness is widely used in defect prediction as an evaluation metric [12], [11], [10]. It aims at maximizing benefits in the condition of spending the same amount of cost. In the context of defect prediction, the cost is the lines of code to inspect, and the benefit is the number of buggy classes found. The cost effectiveness setup we use is the same as the one used by Jiang et al. [3]. We want to count the number of buggy classes found when a developer inspect the first 20% lines of code – this number is referred to as NofB20.

2) F-measure: F-measure, which is the harmonic mean of precision and recall, is a standard and widely used measure to evaluate classification algorithms [19], [25]. 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); it can be classified as buggy when it is actually clean (false positive, FP); it can be classified as clean when it is actually buggy (false negative, FN); or it can be classified as clean and it truly is clean (true negative, TN). Based on these possible outcomes, precision, recall and F-measure are defined as:

Precision: the proportion of instances that are correctly labeled as buggy among those labeled as buggy, i.e., $Precision = TP/(TP + FP)$.

Recall: the proportion of buggy instances that are correctly labeled, i.e., $Recall = TP/(TP + FN)$.

F-measure: a summary measure that combines both precision and recall - it evaluates if an increase in precision (recall) outweighs a reduction in recall (precision), i.e., $F-measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$.

There is a trade-off between precision and recall. One can increase precision by sacrificing recall (and vice versa). The trade-off causes difficulties to compare the performance of several prediction models by using only precision or recall alone [19], [25]. For this reason, we compare the prediction results using F-measure, which is a harmonic mean of precision and recall.

C. Research Questions

How effective are the 7 composite algorithms? How much improvement could these composite algorithms achieve over CODEP$_{Logistic}$?

Motivation. We need to investigate the effectiveness of the 7 composite algorithms and compare them against CODEP$_{Logistic}$ [1]. Answer to this research question would shed light to whether and to what extent the composite algorithms improve over CODEP$_{Logistic}$, which is the state-of-the-art cross-project defect prediction technique.

Approach. To answer this research question, we compute F-measure and NofB20 scores of the 7 composite algorithms and

[2] http://www.cs.waikato.ac.nz/ml/weka/

TABLE II. F-MEASURE SCORES OF THE 7 COMPOSITE ALGORITHMS

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

TABLE III. NOFB20 SCORES OF THE 7 COMPOSITE ALGORITHMS

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

CODEP$_{Logistic}$ when they are applied to 10 datasets from the PROMISE repository. We then compare the results achieved by each of the 7 composite algorithms with the results of CODEP$_{Logistic}$.

Results. Table II presents the F-measure scores of CODEP$_{Logistic}$ 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 0.053-0.435. Across the 10 datasets, the average F-measure of CODEP$_{Logistic}$ is 0.301. From Table II, we can note that the average F-measure scores of Max, Boosting$_{J48}$ and RF are 0.412, 0.302 and 0.308 respectively, which outperform the average F-measure of CODEP$_{Logistic}$ by 36.88%, 0.33% and 2.33% respectively. Max achieves the best F-measure scores; its F-measure scores vary from 0.286-0.608 and the average score is 0.412. Meanwhile, the other four composite algorithms that we investigate 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.

Table III presents the NofB20 score of CODEP$_{Logistic}$ 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 8-88. Across the 10 datasets, the average NofB20 score of CODEP$_{Logistic}$ is 35.2. From Table II, we can note 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 outperform the NofB20 score of CODEP$_{Logistic}$ by 8.24%, 5.40%, 15.34%, 0.57% and 5.68% respectively. Bagging$_{J48}$ achieves the highest NofB20 score; its NofB20 scores vary from 6-92 and the average score is 40.6. Meanwhile, the other two composite algorithms that we investigate 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.

D. Threats to Validity

Threats to internal validity relate to errors in our experiments. We have double checked our experiments and implementation. Still, there could be errors that we did not notice. Threats to external validity relate to the generalizability of our results. We have 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 use cost effectiveness and F-measure which are also used by past software engineering studies to evaluate the effectiveness of various prediction techniques [26], [27], [12], [15], [28], [27], [29]. Thus, we believe there is little threat to construct validity.

V. RELATED WORK

In the last few years, a substantial effort has been devoted to use cross-project strategy in predicting the defect proneness of software entities. This means using defect data from other projects to improve defect prediction for a target project. Zimmermann et al. propose a cross-project defect prediction approach which trains a model on a source project, and uses the model on a target project [8]. They list factors that software engineers should consider before selecting a project as a source project for a given target project. Turhan et al. employ a k-nearest neighbor algorithm for cross-project defect prediction [9]. 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 select 10 nearest instances from source projects. Similar to the work by Turhan et al., Peters et al. propose Peters filter for cross-company defect prediction, which also uses a nearest neighbor approach to select instances from source projects [14]. Nam et al. point out that the poor performance of cross-project defect prediction is largely because of the different feature distribution between the source and target projects [15]. They then propose TCA+, a novel transfer defect learning approach, which make feature distributions in source and target projects similar [15]. Canfora et al. propose a multi-objective approach for cross-project defect prediction, which uses genetic algorithm to build a multi-objective logistic regression model [16]. Zhang et al. build a universal defect prediction model for a large set of projects with diverse contexts [30]. While all of the above studies reduce the gap between the accuracy of within-project and cross-project defect predictions, cross-project defect prediction still represent one of the main challenges in the defect prediction field.

Recently, Panichella et al. propose a state-of-the-art cross-project defect prediction algorithm named CODEP that uses a meta classification algorithm to combine results of six basic classification algorithms [1]. The best results are achieved when logistic regression is used as the meta classification algorithm (i.e., CODEP$_{Logistic}$). In our work, we focus on finding effective composite algorithms for cross-project defect prediction, which can outperform CODEP$_{Logistic}$. We investigate 7 composite algorithms proposed in the machine learning community. These algorithms use different strategies to combine results of a number of basic classifiers. In our experiments, we use the same basic classifiers as CODEP, and the results of the experiments show that 3 out of the 7 composite algorithms perform better than CODEP in terms of both F-measure and cost effectiveness.

VI. Conclusion and Future Work

In this paper, we investigate the effectiveness of 7 composite algorithms proposed in the machine learning community for cross-project defect prediction, aiming at finding one or more algorithms that perform better than CODEP. We evaluate the composite algorithms using two metrics: F-measure and cost effectiveness. We perform experiments on defect datasets from 10 different open-source software projects containing a total 5,305 instances. The results show that Max performs the best in terms of F-measure and achieves an average F-measure score of 0.412, which outperforms the average F-measure of CODEP$_{Logistic}$ by 36.88%; Also, Bagging$_{J48}$ performs the best in terms of cost effectiveness and achieves an average NofB20 score of 40.6, which outperforms the average NofB20 score of CODEP$_{Logistic}$ by 15.34%. In addition to these two algorithms, several other algorithms also outperform CODEP$_{Logistic}$ in terms of F-measure and/or cost effectiveness.

In the future, we plan to investigate additional composite algorithms or create a custom composite algorithm that performs better for cross-project defect prediction.

Acknowledgment. This research was supported by the National Basic Research Program of China (the 973 Program) under grant 2015CB352201, National Key Technology R&D Program of the Ministry of Science and Technology of China under grant 2014BAH24F02, and the Fundamental Research Funds for the Central Universities.

References

[1] A. Panichella, R. Oliveto, and A. De Lucia, "Cross-project defect prediction models: L'union fait la force," in *CSMR-WCRE*. IEEE, 2014, pp. 164–173.

[2] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *ICSE*. ACM, 2008, pp. 531–540.

[3] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *ASE*. IEEE, 2013, pp. 279–289.

[4] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *TSE*, vol. 34, no. 4, pp. 485–496, 2008.

[5] Y. Liu, T. M. Khoshgoftaar, and N. Seliya, "Evolutionary optimization of software quality modeling with multiple repositories," *TSE*, vol. 36, no. 6, pp. 852–864, 2010.

[6] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *ICSE*. ACM, 2006, pp. 452–461.

[7] B. A. Kitchenham, E. Mendes, and G. H. Travassos, "Cross versus within-company cost estimation studies: A systematic review," *TSE*, vol. 33, no. 5, pp. 316–329, 2007.

[8] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *ESEC-FSE*. ACM, 2009, pp. 91–100.

[9] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[10] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom java software," in *ISSRE*. IEEE, 2007, pp. 215–224.

[11] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *ICSE*. IEEE Press, 2013, pp. 432–441.

[12] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in *FSE*. ACM, 2012, p. 61.

[13] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size vs. bias in defect prediction," in *ESEC-FSE*. ACM, 2013, pp. 147–157.

[14] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *MSR*. IEEE, 2013, pp. 409–418.

[15] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *ICSE*. IEEE Press, 2013, pp. 382–391.

[16] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *ICST*. IEEE, 2013, pp. 252–261.

[17] M. E. Bezerra, A. L. Oliveira, and S. R. Meira, "A constructive rbf neural network for estimating the probability of defects in software modules," in *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*. IEEE, 2007, pp. 2869–2874.

[18] A. Okutan and O. T. Yıldız, "Software defect prediction using bayesian networks," *Empirical Software Engineering*, vol. 19, no. 1, pp. 154–181, 2014.

[19] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan kaufmann, 2006.

[20] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[21] M. D. Buhmann, "Radial basis functions," *Acta Numerica 2000*, vol. 9, pp. 1–38, 2000.

[22] J. R. Quinlan, "Bagging, boosting, and c4. 5," in *AAAI/IAAI, Vol. 1*, 1996, pp. 725–730.

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

[24] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[25] X. Xuan, D. Lo, X. Xia, and Y. Tian, "Evaluating defect prediction approaches using a massive set of metrics: An empirical study," in *SAC*. ACM, 2015.

[26] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *TSE*, vol. 34, no. 2, pp. 181–196, 2008.

[27] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, high accuracy prediction of reopened bugs," *Automated Software Engineering*, pp. 1–35, 2014.

[28] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "Elblocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, 2015.

[29] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, "Cross-project build co-change prediction," in *SANER*. IEEE, 2015, pp. 311–320.

[30] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *MSR*. ACM, 2014, pp. 182–191.