Cross-Project Defect Prediction Models: L'Union Fait la Force

Annibale Panichella[1], Rocco Oliveto[2], Andrea De Lucia[1]
[1]Department of Management & Information Technology, University of Salerno, Fisciano (SA), Italy
[2]Department of Bioscience and Territory, University of Molise, Pesche (IS), Italy
apanichella@unisa.it, rocco.oliveto@unimol.it, adelucia@unisa.it

***Abstract*—Existing defect prediction models use product or process metrics and machine learning methods to identify defect-prone source code entities. Different classifiers (*e.g.,* linear regression, logistic regression, or classification trees) have been investigated in the last decade. The results achieved so far are sometimes contrasting and do not show a clear winner. In this paper we present an empirical study aiming at statistically analyzing the equivalence of different defect predictors. We also propose a combined approach, coined as** CODEP **(COmbined DEfect Predictor), that employs the classification provided by different machine learning techniques to improve the detection of defect-prone entities. The study was conducted on 10 open source software systems and in the context of cross-project defect prediction, that represents one of the main challenges in the defect prediction field. The statistical analysis of the results indicates that the investigated classifiers are not equivalent and they can complement each other. This is also confirmed by the superior prediction accuracy achieved by** CODEP **when compared to stand-alone defect predictors.**

I. Introduction

During software development and maintenance, software engineers need to manage time and resources. Improper management of these factors could lead to project failure. Given its importance, in recent years a lot of effort has been devoted to provide software engineers with tools supporting management activities. Defect prediction is one of these tools. Knowing the parts of a software system that are more defect-prone may aid in scheduling testing or refactoring activities. In fact, it is reasonable to allocate more resources on more critical and defect-prone source code entities.

Existing prediction models use different information (*i.e.,* source code metrics, process metrics, or past defects) and machine learning methods to identify defect-prone source code entities. A plethora of classifiers (*e.g.,* linear regression, logistic regression, or classification trees) have been proposed and empirically investigated in the last decade [1], [2], [3]. Generally, the evaluation of these classifiers is based only on metrics—such as precision and recall—able to capture the prediction accuracy of the experimented approaches. The results achieved are sometimes contrasting and generally do not allow to identify a method that sensibly overcomes the others (see *e.g.,* [1]). Indeed, looking at the prediction accuracy (captured by metrics), the impression is that all the classifiers exploited so far are able to capture the same information when used to detect software entities having a high likelihood to exhibit faults. Thus, selecting a specific classifier does not seem to play an important role [4], [1].

However, metrics do not tell the whole story. They only provide an overview of the prediction accuracy without emphasizing peculiarities (if any) of each investigated method. What is missing is a thorough analysis of the predictions of each approach aiming at verifying whether or not different classifiers are able to identify different defect-prone entities. In other words, while different classifiers achieve the same accuracy, the following question remains still unanswered:

Do different classifiers identify the same defect-prone entities or different entities with the same defect-proneness?

In this paper we try to bridge this gap. We conjecture that the investigated classifiers are not equivalent even if they provide the same prediction accuracy. In other words, we believe that different classifiers could be able to identify different defect-prone entities. This calls for a combined approach that employs different (and complementary) classifiers aiming at improving the prediction accuracy of stand-alone approaches.

To verify our conjecture we conducted an empirical study aiming at statistically analyzing the equivalence of different classifiers. The study is based on Principal Component Analysis (PCA) and overlap metrics. With such analyses we are able to identify classifiers that complement each others, *i.e.,* they identify different sets of defect-prone entities. The experimentation was conducted on 10 open-source software systems from the Promise repository. In the context of our study we investigated six different machine learning techniques used for defect prediction in many previous works [5], [6], [7], [8], [9], [2]. These techniques belong to four different categories: regression functions, neural networks, decision trees, and rules models. All these approaches were experimented in the context of cross-project defect prediction, that represents one of the main challenges in the defect prediction field [10].

The achieved results confirmed our conjecture. *The investigated classifiers are not equivalent despite the similar overall prediction accuracy. This means that they are able to identify different sets of defect-prone entities*. This is confirmed by our statistical analysis but especially by the superior performances of a novel combined approach, coined as CODEP (**CO**mbined **DE**fect **P**redictor), that uses machine learning techniques to combine different and complementary classifiers.

In summary, the contributions of this paper are:

- an empirical study on 10 software systems aimed at analyzing the complementarity of six different classifiers used in the context of cross-project defect prediction;
- an approach to combine different and complementary classifiers to overcome the prediction accuracy of stand-alone classifiers.

Structure of the paper. Section II discusses the related literature, while Section III provides background information on the investigated classifiers. Section IV presents the design of our study, while Section V reports the achieved results and Section VI discusses the threats that could affect their validity. Finally, Section VII concludes the paper highlighting future research directions.

II. Related Work

In the last decade a lot of effort has been devoted to the definition of approaches to predict defect-prone source code components. Such approaches use different predictors, such as object-oriented metrics, *e.g.,* the Chidamber and Kemerer (CK) metrics [11], process metrics [12], history based metrics [13], [14], structural metrics [15]. An extensive analysis of the different software metrics and approaches used for defect prediction can be found in the survey by D'Ambros *et al.* [3].

Once a set of software metrics is chosen, machine learning techniques can be used to predict the defect proneness of software modules by training a classification/regression model on past data of the same software project (*within-project strategy*). Several machine learning techniques have been used in literature for defect prediction, such as logistic regression [5], [16], [17], support vector regression (SVR) [6], Radial Basis Function Network (RBF Network) [7], multi-layer perceptron (MLP) [8], Bayesian network [9], [18], decision trees [19] and decision tables [2].

These different machine learning models have been compared aiming at identifying the best model in the context of within-project defect prediction. However, results are sometimes contrasting demonstrating no clear winner among the experimented techniques. For example, Bezerra *et al.* [7], [20] proposed the usage of the RBF Network and its modified version, named RBF-eDDA, demonstrating the superiority of the proposed model against two variants of the *k-means* algorithm [7], [20]. Mahmoud [21] compared six machine learning models, namely MLP, radial basis function (RBF), k-nearest neighbor (KNN), regression tree (RT), dynamic evolving neuro-fuzzy inference system (DENFIS), and SVR. The results achieved in their empirical study indicated that the DENFIS, SVR, and RT models were more accurate in predicting defect density than MLP, RBF, and KNN models. Pai and Bechta [4] empirically evaluated Bayesian networks highlighting that Bayesian networks perform no worse than existing techniques. Nelson *et al.* [19] considered four machine learning models, namely Naive Bayes, J48, ADTree, One-R, that are trained using discretized and not discretized data for a total of eight combinations. In their study the ADTrees trained on discretized data yielded the best performance. Lessman *et al.* [1] conducted an empirical study with 22 classification models to predict the defect proneness of 10 public domain software development data sets from the NASA repository. The results highlighted that there is no statistical differences between the top-17 models. Hence, it is impossible to establish which is the best model for defect prediction.

All these results motivate our work. Specifically, in all these studies the comparison is only based on metrics able to evaluate the prediction of the experimented techniques. A deeper analysis of the list of candidate defect-prone classes is missed. In this paper we statistically analyze the complementarity of different techniques aiming at emphasizing the peculiarities of each machine learning technique.

In the last years, a substantial effort has been devoted to investigate the possibility to predict the defect proneness of software entities using the *cross-project strategy*. In such a strategy the prediction model is trained on past data belonging to different software projects with different domains and it is used to predict the defect proneness of entities belonging to a new project [22], [10], [23], [24]. While cross-project strategy is particularly useful for new projects with limited or no past defect data available, it turns out to be more challenging than the within-project strategy because of the heterogeneity of data coming from different projects. Zimmermann *et al.* [10] analyzed several factors that should be taken into account when selecting the projects to build cross-project predictors. Turhan *et al.* [22] used nearest-neighbor algorithm to filter the data points to be used for building cross-project defect prediction models. Cruz *et al.* [25] applied the log transformations to build logistic regression models when the test set predictors are not as spread as those of the training set. Nam *et al.* [26] proposed to apply the z-score normalization to reduce the differences between data coming from different projects. Further studies have suggested to use "local prediction" by clustering data according to the set of predictors and training classifiers for each clusters separately [23], [24].

Liu *et al.* [2] experimented 17 different machine learning models (including logistic regression, neural networks, decision table, trees and evolutionary algorithms) in the context of cross-project defect prediction. They suggested to use a given machine learning technique to build a prediction model for each system and use an *ensemble learning* mechanism, named *Validation-and-Voting*, to combine the defect proneness obtained by each model when used on a new system. Specifically, the prediction on the entities of the new system is obtained by voting. If the majority of models (obtained using the same techniques on different training systems) predict an entity as defect-prone, then it is predicted as defect-prone; otherwise, it is predicted as non defect-prone. Other *ensemble learning* mechanisms have been also used by Kim *et al.* [27] and Misirli *et al.* [28] to combine different classifiers obtained by a given machine learning technique on multiple training data in the context of within-project defect predictions. However, such approaches do not combine different machine learning techniques. They are strategies to combine heterogeneous data belonging to different projects with different domains (Liu *et al.* [2]) or data belonging to different sub-sets of the same training set obtained by random sampling (Kim *et al.* [27]).

While all these studies helped to reduce the gap between the accuracy of within- and cross-project predictions, cross-project defect prediction still represents one of the main challenges in the defect prediction field. For this reason, in the context of our study we experimented six different defect prediction approaches using the cross-project strategy.

Recent works have also suggested to evaluate the performance of defect prediction models using both effectiveness (*e.g.,* precision and recall) and inspection cost [3], [29], [15], [30]. Brahman *et al.* [31] approximated the inspection cost in terms of number of source code lines that need to be inspected to detect the defects, as previously done by Arisholm *et al.* [29], [15]. When using cost and effectiveness, they found that the prediction performance of cross-project defect prediction are comparable to within-project prediction. Moreover, such metrics are much more useful than the traditional metrics since they approximate the effort required to analyze the predicted software modules.

Canfora *et al.* [30] explicitly took into account the trade-off between effectiveness and inspection cost proposing a multi-objective approach for cross-project defect prediction. The approach is based on a multi-objective logistic regression model built using a genetic algorithm. Specifically, instead of providing the software engineer with a single predictive model, the multi-objective approach allows software engineers to choose the model achieving a compromise between number of likely defect-prone entities (effectiveness) and LOC to be analyzed/tested (which can be considered as a proxy of the cost of code inspection). The results of an empirical study provide evidence of the benefits of the multi-objective approach with respect to single-objective predictors especially in the context of cross-project prediction.

In our study we also analyze the cost-effectiveness of the experimented classifiers. We observed that, while the combined approach proposed in this paper (CODEP) is not designed to find a compromise between effectiveness and inspection cost, it is able to achieve a better compromise between cost and effectiveness than stand-alone approaches.

III. Background

Cross-project defect prediction models are trained on data coming from past projects for which predictors (*e.g.,* product metrics) and actual defects are available. Then, supervised machine learning techniques are used to build prediction models with the conjecture that the chosen pool of predictors can be also used to predict the defect proneness of software entities of a new project.

A general prediction model can be viewed as a function $F : \mathbb{R}^n \to \mathbb{R}$, which takes as input a set of predictors and returns a scalar value—ranging within the interval $[0; 1]$—that measures the likelihood that a specific software entity is defect-prone. The estimated defect proneness is used to rank the source code entities of a system in order to identify the most defect-prone entities to be investigated in quality assurance activities. In some cases a fixed cut point μ is used to cut the ranked list and provide the software engineer with the top-μ ranked entities (these entities are the estimated defect prone entities).

The difference between the various machine learning techniques depends on the specific function F that is used to map the predictors to the estimated defect proneness. In this paper we experimented six different machine learning techniques, namely Logistic Regression, Bayes Network, Radial Basis Function Network, Multi-layer Perceptron, Alternating Decision Trees, and Decision Table. The choice of these techniques is not random. We selected them since they (i) were successfully used for defect prediction in many previous works [2], [5], [6], [7], [8], [9]; and (ii) they belong to four different categories (*i.e.,* regression functions, neural networks, decision trees, and rules models) allowing to increase the generalizability of our findings. The following subsections provide a brief description of the chosen machine learning techniques.

A. Logistic Regression

One of the most used machine learning techniques for predicting the defect proneness of software entities is multivariate logistic regression, a generalization of the linear regression to binary classification, *i.e.,* either a file is defect-prone or it is not in our case. Its general equation is the following:

$$P(c_i) = \frac{e^{\alpha+\beta_1 \dot{x}_1+\beta_2 \dot{x}_2+\cdots+\beta_n \dot{x}_n}}{1+e^{\alpha+\beta_1 \dot{x}_1+\beta_2 \dot{x}_2+\cdots+\beta_n \dot{x}_n}} \qquad (1)$$

where $P(c_i)$ is the estimated defect proneness—within the interval $[0, 1]$—of the entity c_i , the scalars $\alpha, \beta_1, \beta_2, \dots, \beta_n$ denote the linear combination coefficients, and $x_1, x_2, \dots, x_n$ are the predictors, *i.e.,* software metrics in our study. The larger the absolute value of a coefficient β_i, the stronger the effect of the corresponding predictor x_i on the likelihood of a defect being detected in the entity c_i [5]. Since the equation (1) cannot be solved analytically, the maximum likelihood procedure is used to estimate the coefficients that minimize the prediction error, *i.e.,* the difference between the predicted probability $P(c_i)$ and the observed outcome values.

B. Bayes Network

A Bayesian network is a directed graph composed of V vertexes and E edges, where each vertex represents a predictor (*i.e.,* software metric) and each edge denotes the causal relationship of one predictor to its successor in the network. Since the search space tends to grow exponentially when the number of nodes increases, heuristic algorithms are generally used to find the network configuration that best fits the training data [18]. One of the most used algorithm is the $K2$ algorithm introduced by Cooper and Herskovits [32] which iteratively looks for parents for each node whose addition increases the score of the Bayesian network on the basis of the ordering of the nodes.

C. Radial Basis Function Network

The Radial Basis Function (RBF) Network is a type of neural network which uses radial basis function as activation functions [33]. It has three different layers: (i) the input layer which corresponds to the predictors, *i.e.,* software metrics; (ii) the output layer which maps the outcomes to predict, *i.e.,* the defect proneness of entities; (iii) the hidden layer used to connect the input layer with the output layer. The radial basis function used to activate the hidden layer is a Gaussian function. The prediction model is defined as follows [33]:

$$P(c_i) = \sum_{k=1}^{n} \alpha_k \, e^{\frac{-\|x-\gamma_k\|}{\beta}} \tag{2}$$

where $P(c_i)$ is the predicted defect proneness of the entity c_i, $\alpha_1, \alpha_2, \ldots, \alpha_n$ is the set of linear weights, and the points γ_k are the centers of the radial basis function. Finally, the function $e^{\|x-\gamma_k\| \cdot 1/\beta}$ is the radial basis function which is the core of the RBF network.

D. Multi-layer Perceptron

The Multi-layer Perceptron (MLP) is another type of neural network that is trained using a back-propagation algorithm. In general multi-layer perceptron consists of multiple layers of nodes in a directed graph: an input layer, one or more hidden layers, and one output layer. The output from a layer is used as input to nodes in the subsequent layer. The formal definition of the model has the following mathematical formulation:

$$P(c_i) = \sum_{k=1}^{n} w_k \, \frac{1}{1 + e^{\alpha + \beta_1 \dot{x}_1 + \beta_2 \dot{x}_2 + \cdots + \beta_n \dot{x}_n}} \tag{3}$$

where $P(c_i)$ is the predicted defect proneness of the entity c_i, $\alpha, \beta_1, \beta_2, \ldots, \beta_n$ is the set of linear combination coefficients while w_k are the weights of the each layers.

E. Alternating Decision Trees

An Alternating Decision Tree (ADTree) consists of a tree structure with leaf and decision nodes: leaf nodes contain the predicting outcomes, *i.e.,* entity defect-proneness, while the other nodes contain the decision rules (*decision nodes*). Let p_i a given predictor and let a_i be a *decision coefficient*, each decision node contains a specific rule (*e.g., if* $p_i < a_i$), while each leaf node contains one of the two possible outcomes: either the software entity is defect prone or not. When a given instance has to be classified, the tree is traversed from the root node to bottom until a leaf node is reached. The decision about which branch to follow is performed at each node of the tree on the basis of the test condition on the predictor p_i corresponding to that node. Finally, each leaf node is a linear regression model associated to obtain the predicting outcome. Hence, the classification of a given instance is performed by following all paths for which all decision nodes are true and summing the predicting values that are traversed along the corresponding path.

F. Decision Table

A Decision Table (DT) can be viewed as an extension of one-valued decision tree [34]. It is a rectangular table T where the columns are labeled with predictors, *i.e.,* software metrics [34], and rows are sets of decision rules. Each decision rule of a decision table is composed of (i) a pool of *conditions*, linked through *and* and *or* logical operators which are used to reflect the structure of the if-then rules; and (ii) an outcome, generally classed as *action*, which mirrors the classification of a software entity respecting the corresponding rule as defect-prone or non defect-prone. DT uses an attribute reduction algorithm to find a good subset of predictors aiming at (i) eliminating equivalent rules and (ii) reducing the likelihood of over-fitting the data.

TABLE I: Software projects used in our study.

Name	Release	Classes	Defect-prone Classes	(%)
Ant	1.7	745	166	22%
Camel	1.6	965	188	19%
Ivy	2.0	352	40	11%
Jedit	4.0	306	75	25%
Log4j	1.2	205	189	92%
Lucene	2.4	340	203	60%
Poi	3.0	442	281	64%
Prop	6.0	661	66	10%
Tomcat	6.0	858	77	9%
Xalan	2.7	910	898	99%

IV. Study Design

The *goal* of our study was twofold. We analyzed (i) whether different machine learning techniques are equivalent or complementary to each other when used to predict defects in source code entities, *i.e.,* they classify different sets of classes as defect-prone; and (ii) whether the prediction accuracy improves when combining different classifiers.

The *quality focus* of the study was to improve the effectiveness of defect prediction approaches in the context of cross-project defect prediction, while the *perspective* is of a researcher who is interested to understand to what extent different classifiers complement each other when used to predict defect-prone software entities.

The *study context* consists of 10 Java projects where both (i) information on defects and (ii) software metrics are available. All the projects belong to the Promise repository[1]. The dataset provides, for each system, the actual defect-proneness of its classes and a pool of metrics used as independent variables for the predictors, *i.e.,* LOC and Chidamber & Kemerer (CK) metrics. Table I summarizes the characteristics of each project.

A. Research Questions

In the context of our study we formulated the followings research questions:

- **RQ$_1$**: *Are different classifiers equivalent to each other when applied to cross-project defect prediction?* Since previous work (*e.g.,* [1]) demonstrated that there is no clear winner among several machine learning models used for defect prediction, this research question aimed at providing a deeper insight on the actual equivalence (or complementarity) of different models. Specifically, the goal was to verify whether different classifiers are able to capture the same information, *i.e.,* all the models predict as defect prone the same set of source code entities,

[1]https://code.google.com/p/promisedata/

or specific models are able to capture some important information missed by other models.

- **RQ**$_2$: *Does the combination of different defect prediction models outperform stand-alone models?* The complementarity of machine learning models may provide the opportunity to improve the prediction accuracy of stand-alone models through their combination. Hence, the main goal of this research question is to verify whether the prediction accuracy of stand-alone models can be improved by combining individual predicted defect proneness in the context of cross-project defect prediction.

To respond to our research questions we computed the defect-proneness of classes[2] by using six different machine learning techniques (see Section III), *i.e.,* producing six different probability distributions, one for each model.

B. Planning and Analysis Method

As for **RQ**$_1$ we preliminary evaluated the defect prediction accuracy of the experimented methods using *precision* and *recall*. Given a threshold μ used to cut the list of classes ranked on the basis of their defect proneness, we can compute:

$$Precision(\mu) = \frac{TP}{TP + FP}$$

$$Recall(\mu) = \frac{TP}{TP + TN}$$

where TP is the number of classes containing defects that are correctly classified as defect-prone; TN denotes the number of defect-free classes classified as non defect-prone classes; FP and FN measure the number of classes for which a prediction model fails to identify defect-prone classes by declaring defect-free classes as defect-prone (FP) or identifying actually defected classes as non defect-prone ones (FN). This preliminary analysis was necessary to corroborate the findings achieved in the literature, *i.e.,* different classifiers exhibit similar recall and precision.

After that, we analyzed the equivalence of different techniques using Principal Component Analysis (PCA). Such an analysis was necessary to analyze whether different techniques assign the same defect proneness to the same set of classes. Specifically, PCA is a statistical technique able to identify various orthogonal dimensions (principal components) captured by the data (defect-proneness of classes in our case) which measure contributes to the identified dimensions. Through the analysis of the principal components and the contributions (scores) of each model to such components, it is possible to understand whether different models contribute to the same principal components. Specifically, two models are complementary if they contribute or capture different principal components. Hence, the analysis of the principal components provides insights on the complementarity between models.

However, PCA does not tell the whole story. Indeed, using PCA it is not possible to identify to what extent a technique complements another and *vice versa*. This is the reason why we complemented the PCA with an analysis of the overlap between the predictions of different techniques. Specifically, given a cut point μ, we computed:

$$\begin{aligned} A(\mu) \cap B(\mu) &= \mid TP_{A,\mu} \cap TP_{B,\mu} \mid \\ A(\mu) \setminus B(\mu) &= \mid TP_{A,\mu} \setminus TP_{B,\mu} \mid \end{aligned}$$

where $TP_{A,\mu}$ and $TP_{B,\mu}$ represent the sets of defect-prone classes correctly classified by the prediction models A and B, respectively, at a given cut-point. Note that $A \cap B$ measures the overlap between the set of true positive (TP) correctly identified by both the two prediction models, while $A \setminus B$ gives an indication of how many true positive (TP) are identified by A and missed by B.

Turning to the second research question (**RQ**$_2$) we combined the defect probabilities of the six experimented machine learning models. We assumed that *the defect-proneness of a class c_i can be computed as combination of all the defect-proneness probabilities obtained by the different models*. In the context of our study we decided to combine the different classifiers using machine learning techniques. Specifically, we build a prediction model where defect predictors represent the predictors itself (*e.g.,* logistic regression). In other word, we suggest to combine the six experimented classifiers as follows:

- apply the six stand-alone machine learning models on the CK software metrics to obtain for each class the estimated defect proneness;
- use the output of the previous step as predictors and build a new prediction model using a machine learning technique.

In the context of our study, we considered two combined approaches that differ by the underlined machine learning technique. In the first approach we use the logistic regression model, while in the second approach we experimented the Bayesian network.

Note that we are not claiming that these are the best ways to combine different classifiers. We are aware that more sophisticated approaches could be used. However, in the context of this study we prefer to use very simple approaches since our main goal was to provide some evidence that the combination is worthwhile. Identifying the best strategy to combine different classifiers is out of the scope of this paper and will be part of our future research agenda.

To compare the prediction accuracy of stand-alone models and of the two combined models, we use as alternative performance metrics the Receiver Operating Characteristic (ROC). ROC is a non-parametric method to evaluate models which plots the precision/recall values reached for all possible cutoff values raging within the interval [0; 1]. Hence it is independent of the cutoff unlike the precision and recall metrics [3]. ROC plots the *True Positive Rate* on the y-axis and the *False Positive Rate* on the x-axis, where

$$TPR = \frac{TP}{TP + FN}$$

$$FPR = \frac{FP}{FP + TN}$$

[2]In our study we considered classes as source code entities.

TABLE II: Precision and Recall values achieved using as cut-points $\mu = 10, 50, 100$.

Method	μ=10		μ=50		μ=100	
	Prec	**Rec**	**Prec**	**Rec**	**Prec**	**Rec**
Logistic	0.90	0.05	0.84	0.25	0.69	0.42
RBFNetwork	1.00	0.06	0.80	0.24	0.64	0.39
ADTree	1.00	0.06	0.76	0.23	0.71	0.43
DecisionTable	0.90	0.05	0.84	0.25	0.74	0.45
MultilayerPerc.	1.00	0.06	0.86	0.26	0.70	0.42
BayesNet	0.20	0.01	0.12	0.05	0.19	0.11

(a) Ant

Method	μ=10		μ=50		μ=100	
	Prec	**Rec**	**Prec**	**Rec**	**Prec**	**Rec**
Logistic	0.80	0.11	0.68	0.45	0.46	0.61
RBFNetwork	0.80	0.11	0.64	0.43	0.44	0.59
ADTree	0.7	0.09	0.74	0.49	0.6	0.8
DecisionTable	0.90	0.12	0.72	0.48	0.53	0.71
MultilayerPerc.	0.90	0.12	0.66	0.44	0.51	0.68
BayesNet	0.4	0.05	0.40	0.27	0.34	0.45

(b) jEdit

The best possible model is the one with a ROC curve close to the line $y = 1$, while a random model will be close to the diagonal $y = x$. To have a single scalar value that facilitates the comparison across models, we report the Area Under the ROC Curve (AUC). An area of 1 represents a perfect classifier, whereas for a random classifier an area of 0.5 would be expected. We used ROC and AUC as performance metrics because, as suggested by recent works [3], [31], the traditional evaluation metrics, namely *precision* and *recall*, are sensitive to the thresholds used as cutoff parameters [3].

Besides effectiveness, we also analyze the inspection cost. Specifically, we use the LOC to approximate the effort needed to analyze the classes classified as defect-prone following the intuition that larger classes require a longer time to review than smaller ones [3]. Hence, we used a cost-effective ROC (ROC-CE) which plots on the x-axis the total number of lines of codes of the classes labeled as defect prone and on the y-axis the corresponding number of defects reached by a specific model for all possible cutoff values raging within the interval $[0; 1]$. To have a single cost-effective scalar value that facilitates the comparison across models, we report the Area Under the cost-effective ROC Curve (AUC-CE). This metric assume values within the range $[0; 1]$ and its optimal value is equal to 1.

C. Pre-processing and Validation Method

Our experimentation was performed in the context of cross-project defect prediction. In this context, the main challenge is represented by the data heterogeneity. Indeed, software projects are often heterogeneous because they exhibit different software metric distributions [30]. Hence, when evaluating prediction models on a software project with a software metric distribution that is different with respect to the data distribution used to build the models themselves, the prediction accuracy can be compromised [23].

To mitigate the data heterogeneity problem, we performed a *data standardization*, *i.e.,* converting software metrics into a z distribution, in order to reduce the effect of heterogeneity between different projects. A similar approach was applied by Gyimothy *et al.* [16] and Canfora *et al.* [30]. The usefulness of data normalization for cross-project prediction was also shown by Nam *et al.* [26].

It is important to note that the data normalization procedure was performed only on the software metrics, which are used as predictors for the stand-alone models. It was not applied to the defect probabilities used as predictors for the combined approaches.

As for the validation method, since we are using supervised classifiers, we applied a 10-fold cross validation by removing each time a project from the set, training on 9 projects and predicting on the 10^{th} one. All the pre-processed training and test sets are available for replications [35].

V. Analysis of the Results

In this section we discuss the results achieved aiming at answering our research questions formulated in Section IV-A.

A. ***RQ***$_1$*: Are different classifiers equivalent to each other when applied to cross-project defect prediction?*

As planned, we first analyzed the accuracy of the different machine learning techniques to test whether one technique provides better prediction accuracy than the others. To this aim, we first computed recall and precision metrics obtained by the different techniques at various cut-points levels, *i.e.,* when considering 10, 50 and 100 classes having the highest defect proneness.

For sake of space limitation, Table II summarizes the results obtained on `Ant` and `jEdit`. Similar results have been achieved on the other systems (see our technical report for the complete results [35]). The analysis reveals that there is no technique that outperforms the others. For example, on `Ant`, when $\mu = 10$ the best techniques are RBF, ADT and MLP. Increasing the cut-point to $\mu = 50$ the best precision/recall values are provided by MLP while at $\mu = 100$ better performance is yielded by DT. Similarly, on `jEdit` the best techniques when $\mu = 10$ are DT and MLP, but when considering the first 100 top-most predicted defect-prone classes ADT turned out to be the best one. Summarizing, with the only exception of BN, the experimented techniques provide quite similar prediction accuracy.

Thus, we analyzed the complementarity of the experimented classifiers by using the PCA. Table III reports the results obtained by comparing the distributions of defect proneness provided by the experimented techniques on `Ant` and `Log4j`. As we can see, the PCA identifies 4 principal components with a proportion greater than the 2% on average. The first component (C1) cover the majority of the proportion ranging between 72% and 85%. The second component (C2) has a proportion ranging between 6% and 20%, while the third component and forth components show a proportion between 4% and 6% and between 1% and 5%, respectively. Looking at

TABLE III: Results of the Principal Component Analysis (PCA). Values in bold face show the elements (classifiers) that best captured the main components identified by PCA.

Method	C1	C2	C3	C4	C5	C6
Log.	0.46	-0.24	0.07	-0.44	**0.73**	0.00
RBF	0.37	-0.30	0.40	-0.44	-0.65	0.00
ADT	0.44	**0.88**	-0.05	-0.16	-0.07	0.00
DT	0.37	-0.25	**-0.87**	0.01	-0.22	0.00
MLP	**0.56**	-0.12	0.28	**0.77**	0.04	0.00
BN	0.00	0.00	0.00	0.00	0.00	**1.00**
Proportion	**72.00%**	**12.51%**	**7.46%**	**5.55%**	**2.29%**	**0.19%**

(a) Ant

Method	C1	C2	C3	C4	C5	C6
Log.	0.24	**0.73**	-0.48	0.43	0.00	0.00
RBF	0.18	0.44	**0.87**	0.12	0.00	0.00
ADT	**0.94**	-0.34	-0.02	0.01	0.00	0.00
DT	0.00	0.00	0.00	0.00	**0.80**	0.60
MLP	0.15	0.41	-0.12	**-0.89**	0.00	0.00
BN	0.00	0.00	0.00	0.00	-0.60	**0.80**
Proportion	**73.68%**	**20.02%**	**5.62%**	**0.68%**	**0.00%**	**0.00%**

(b) Log4j

TABLE IV: Overlap analysis results.

Models		**μ = 10**			**μ = 50**			**μ = 100**		
		AnB	**A-B**	**B-A**	**AnB**	**A-B**	**B-A**	**AnB**	**A-B**	**B-A**
A = Log.	B = RBF	4	5	6	34	8	6	58	11	6
A = Log.	B = ADT	2	7	8	27	15	11	50	19	21
A = Log.	B = DT	0	9	9	20	22	22	52	17	22
A = Log.	B = MLP	5	4	5	36	6	7	68	1	2
A = Log.	B = BN	0	9	2	2	40	4	6	63	13
A = RBF	B = ADT	5	5	5	27	13	11	48	16	23
A = RBF	B = DT	0	10	9	21	19	21	54	10	20
A = RBF	B = MLP	4	6	6	36	4	7	57	7	13
A = RBF.	B = BN	0	10	2	2	38	4	6	58	13
A = ADT	B = DT	0	10	9	19	19	23	55	16	19
A = ADT	B = MLP	1	9	9	26	12	17	51	20	19
A = ADT	B = BN	0	10	2	3	35	3	9	62	10
A = DT	B = MLP	0	9	10	23	19	20	52	22	18
A = DT	B = BN	0	9	2	2	40	4	6	68	13
A = MLP	B = BN	0	10	2	2	41	4	7	63	12

(a) Ant

Models		**μ = 10**			**μ = 50**			**μ = 100**		
		AnB	**A-B**	**B-A**	**AnB**	**A-B**	**B-A**	**AnB**	**A-B**	**B-A**
A = Log.	B = RBF	7	3	3	37	13	13	77	20	18
A = Log.	B = ADT	0	10	10	23	27	27	62	35	38
A = Log.	B = DT	0	10	10	9	41	37	48	49	45
A = Log.	B = MLP	4	6	6	42	8	8	91	6	6
A = Log.	B = BN	0	10	10	9	41	37	48	49	45
A = RBF	B = ADT	0	10	10	22	28	28	65	30	35
A = RBF	B = DT	0	10	10	7	43	39	46	49	47
A = RBF	B = MLP	4	6	6	35	15	15	72	23	25
A = RBF	B = BN	0	10	10	7	43	39	46	49	47
A = ADT	B = DT	0	10	10	9	41	37	46	54	47
A = ADT	B = MLP	0	10	10	22	28	28	61	39	36
A = ADT	B = BN	0	10	10	9	41	37	46	54	47
A = DT	B = MLP	0	10	10	11	35	39	46	47	51
A = DT	B = BN	10	0	0	46	0	0	93	0	0
A = MLP	B = BN	0	10	10	11	39	35	46	51	47

(b) Log4j

the scores reached by the different classifiers, we can observe that they seem to capture different components.

Let us consider for example the project *Log4j*. It can be noted that the first component, which represents about 73% of the phenomenon, is captured by ADT only (score greater then the 90%). Such a component is also poorly captured, or not captured by the remaining techniques. The second component, which mirrors 20% of the proportion, is captured by Logistic, RBF, ADT and MLP but not captured by DT and BN. Instead, the third component (about 6% of proportion) is captured by RBF and Logistic models only. Similar analysis can be performed for the other projects, with the difference that the techniques showing the highest scores for each components vary from project to project (see [35]). Moreover, for all the projects there is at least one technique that is not able to capture a component captured by the other techniques. Finally, the bayesian networks always capture only the last component, *i.e.,* the component showing the lower proportion.

In summary, the PCA confirms that different techniques tend to cover different principal components. From a practical point of view this means that different techniques assign different defect proneness to the analyzed classes.

To evaluate the degree of the overlap among the set of estimated defect prone classes we computed the overlap metrics defined in Section IV. Table IV reports the results of the overlap analysis. In particular, for each pair of techniques the table reports the number of classes that are correctly classified as defect-prone by both the compared techniques (labeled as A and B) in the top-most μ positions (where $\mu \in 1, 50, 100$). The table also reports the number of classes correctly classified by one model (A) and not correctly classified by the other one (B) within the top most μ classes and the opposite, *i.e.,* number of classes correctly classified by B and not by A.

From the analysis of the results, we can observe that the overlap between pairs of techniques is low, while the number of classes correctly classified by one technique against another one is high. For example, consider the overlap between each pair of techniques obtained on the system `Ant` when $\mu = 10$. The overlap between different techniques is always lower than the 50% and in several cases it is equal to 0. Conversely, we can also observe that for each pair the majority of defect-prone classes correctly classified by only one technique is high (always greater than the 50% and often equals to the 100%). This means that each technique correctly classifies as defect-prone a set of classes that is quite disjoint to the set of classes correctly identified by the other techniques. For increasing μ value, the scenario is the same. Specifically, $A \setminus B$ and $B \setminus A$ are always greater than 0 in all the cases for both $\mu = 50$ and $\mu = 100$. Once again the results turned out to be stable among the different projects (see [35]).

RQ1 Summary. *The six experimented machine learning techniques are not equivalent to each others. Different techniques assign different defect probability to the same software classes. Moreover, the overlap between pairs of techniques is quite low.*

*B. **RQ**$_2$: Does the combination of different defect prediction models outperform stand-alone models?*

Our goal was to evaluate whether it is possible to improve the performance of machine learning techniques by their combination. To this purpose, Table V reports the prediction performance according to the AUC metrics (see section IV) obtained by the experimented stand-alone techniques and their combinations based on logistic regression (CODEP$_{Log}$) and Bayesian network (CODEP$_{Bayes}$), respectively. The last column contains the average AUC metric values yielded by the different techniques across the 10 projects.

The best prediction accuracy is obtained by the two combined approaches. The mean AUC values (last column in Table V) reached by CODEP$_{Log}$ and CODEP$_{Bayes}$ are much greater than those reached by the stand-alone techniques. The improvements range between 6% and 37% for CODEP$_{Log}$ and between 10% and 41% for CODEP$_{Bayes}$. Instead, the mean accuracy —in terms of the mean of the AUC values—of the stand-alone techniques are quite comparable to each others. All techniques have a mean AUC value ranging between 70% and 76% except for the Bayesian networks that provide substantially lower AUC values across all the projects.

To provide a graphical interpretation of the achieved results, Figure 1 plots the ROC curve achieved on the `Ant` system by stand-alone techniques and their combinations. As we can see, the two best ROC curves—*i.e.,* the highest curves—are achieved by CODEP$_{Log}$ and CODEP$_{Bayes}$. This means that at same percentage of classes to be analyzed they allow to correctly identify a higher percentage of actual defect-prone classes. Moreover, with both combinations it is possible to identify all the defect-prone classes by considering only 60% of all the classes belonging to `Ant`. To reach the same percentage of correctly identified defect-prone classes, the software engineer should analyze more than 90% of all classes of `Ant` when using the stand-alone techniques. Similar curves can be also obtained for all the other projects (see [35]).

Such improvements mirror a better ability of CODEP to assign defect probabilities that are near to the actual defect proneness of the predicted classes. For example, let us to consider the defect-prone class `ant.taskdefs.Available` extracted from the `Ant` project. For such a class all the stand-alone techniques assign a defect proneness ranging between 22% and 71%, more precisely $Log = 55\%$, $RBF = 57\%$, $ADT = 55\%$, $DT = 71\%$, $MLP = 64\%$, $BN = 22\%$. For the same class the two combined approaches, *i.e.,*CODEP$_{Log}$ and CODEP$_{Bayes}$, provide higher probabilities, *i.e.,* 96% and 80%, respectively. Hence, they assign a better and more accurate prediction for such a defect-prone class. Similarly, the combined approaches assign lower—*i.e.,* better—defect probabilities than the stand-alone techniques for classes that are defect free. For example, the class `ant.types.resources.CompressedResource` from the `Ant` project is a defect-free class. For such a class the stand-alone techniques provide a defect-proneness ranging between 22% and 44%, namely $Log = 21\%$, $RBF = 14\%$, $ADT = 40\%$, $DT = 17\%$, $MLP = 23\%$, $BN = 22\%$. The combined approaches assign a defect probabilities that are much more near to the actual defect-proneness, *i.e.,*CODEP$_{Log} = 8\%$ and CODEP$_{Bayes} = 1\%$.

According to the results shown in Table V and Figure 1, we can assert that the prediction accuracy—measured using the AUC metric—of stand-alone machine learning techniques can be improved through their combination. Hence, different classifiers complement each other.

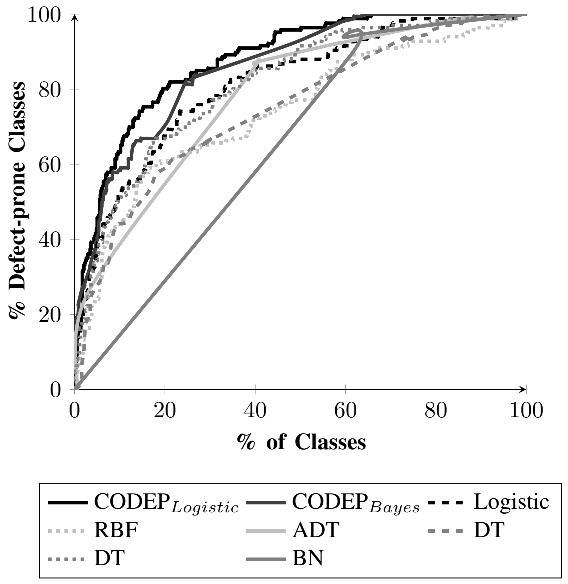

Fig. 1: AUC curve for *Ant*

We also evaluated the different techniques using the AUC-CE metric (see Table VI). Once again, it is possible to note that the best performances are obtained by the two combined approaches. The mean AUC-CE values reached by CODEP$_{Log}$ and CODEP$_{Bayes}$ are much greater than those reached by the stand-alone techniques. The improvements range between 5% and 12% for CODEP$_{Log}$ and between 6% and 13% for CODEP$_{Bayes}$. Instead, the mean AUC-CE values obtained using the stand-alone techniques are quite comparable to each others. In particular, all the techniques have a mean AUC value ranging between 42% and 49%.

This represents an important result because better AUC-CE values mean that CODEP$_{Log}$ and CODEP$_{Bayes}$ allow to early identify defect-prone classes with higher density of defects at same inspection cost (which is approximated by the total number of lines of code to be analyzed [15], [3], [29], [30]). It is also interesting to highlight that even if the AUC values reached by Bayesian network (BN) are substantially lower than those achieved by the other stand-alone techniques (see Table V), from a cost-effectiveness point of view such a technique is competitive with the other stand-alone techniques (see Table VI). This confirms the results of previous work. Generally, better prediction accuracy does not necessary mirror better performance in terms of cost-effectiveness [31], [30] .

RQ2 Summary. *The six experimented machine learning techniques can complement each others. Indeed, two simple combined models based on logistic regression and Bayesian networks allow to achieve better prediction accuracy than the stand-alone techniques. The combined approaches also allow to obtain better performance from a cost-effectiveness point of view.*

TABLE V: AUC values reached for all the projects by the stand-alone methods an their combinations.

Model	Systems										
	Ant	Camel	Ivy	jEdit	Log4j	Lucene	Poi	Prop	Tomcat	Xalan	Mean
CODEP$_{Log}$	**0.88**	**0.71**	0.86	**0.85**	0.5	0.80	0.88	0.88	0.87	**0.96**	0.82
CODEP$_{Bayes}$	0.86	0.70	**0.90**	0.84	**0.88**	**0.81**	**0.88**	**0.89**	**0.87**	**0.98**	0.86
Log.	0.81	0.64	0.83	0.77	0.58	0.76	0.79	0.78	0.81	0.81	0.76
RBF	0.74	0.61	0.82	0.71	0.55	0.75	0.73	0.76	0.76	0.76	0.72
ADT	0.75	0.62	0.79	0.74	0.55	0.74	0.78	0.78	0.79	0.71	0.72
DT	0.75	0.61	0.75	0.78	0.60	0.59	0.67	0.78	0.78	0.69	0.70
MLP.	0.81	0.64	0.81	0.78	0.59	0.74	0.80	0.77	0.82	0.85	0.76
BN	0.65	0.46	0	0.49	0.50	0.42	0.50	0.5	0.51	0.50	0.45

TABLE VI: AUC-CE value reached by the experimented methods an their combinations.

Model	Systems										
	Ant	Camel	Ivy	jEdit	Log4j	Lucene	Poi	Prop	Tomcat	Xalan	Mean
CODEP$_{Log}$	**0.59**	0.54	**0.64**	0.61	**0.54**	**0.51**	**0.51**	**0.51**	**0.60**	0.36	0.54
CODEP$_{Bayes}$	0.58	**0.56**	0.57	**0.62**	0.47	0.48	0.47	0.47	0.58	**0.69**	0.55
Log.	0.55	0.48	0.50	0.46	0.30	0.42	0.41	0.41	0.54	0.45	0.45
RBF	0.52	0.43	0.52	0.46	0.30	0.41	0.40	0.38	0.51	0.27	0.42
ADT	0.54	0.45	0.48	0.47	0.32	0.45	0.47	0.47	0.56	0.46	0.47
DT	0.55	0.48	0.50	0.61	0.35	0.41	0.46	0.44	0.55	0.52	0.49
MLP	0.49	0.49	0.51	0.46	0.30	0.47	0.43	0.43	0.54	0.40	0.45
BN	0.49	0.53	0.52	0.38	0.45	0.51	0.42	0.48	0.50	0.49	0.48

VI. Threats to Validity

This section discusses the threats that might affect the validity of the reported case study.

Evaluation method. Recall, precision, ROC and AUC are widely used metrics for evaluating the performances of defect prediction techniques [3]. The overlap metrics are used to provide an indication on the overlap between the sets of the defect-prone classes predicted by different machine learning techniques. Moreover, the complementarity is statistically analyzed using PCA to verify the presence of machine learning techniques that provide complementary defect probabilities. To analyze the cost-effectiveness of each experimented techniques, we used the lines of code to be analyzed. This metric was previously used to approximate the analysis/testing cost required to test/inspect the predicted classes [31], [30], [3].

Dataset and oracle. An important issue regards the set of software metrics and defect data sets. All the datasets are open-source Java projects and they come from the Promise repository. Hence, they can be prone to imprecision and incompleteness. However, such datasets have been also widely used in previous work on cross-project defect prediction [23], [36], [30]. Finally, the projects arise from very different application domains and present different characteristics (developers, size, number of classes, etc.).

Generalizability of the results. We considered data from 10 projects and experimented 6 stand-alone machine learning techniques most used in previous studies on defect prediction [16], [17], [6], [7], [2]. It is also important to highlight that the goal of our study was to show that different machine learning techniques are complementary when applied to cross-project defect prediction and their complementarity can be used to build better combined models. Hence, other techniques, such as *local models* [23], [24], can be viewed as components—*i.e.,* as further stand-alone techniques—for the combined approaches proposed in this paper. Another important issue is about the machine learning techniques used to combine the stand-alone techniques. In this paper we just consider only two approaches based on logistic regression and bayesian network to combine the six experimented stand-alone techniques. Clearly, further combined mechanisms could be experimented. However, the main goal of this paper is to show that there is still room to improve cross-project defect prediction by exploiting the complementarity of the stand-alone classifiers.

VII. Conclusion and Future Work

In this paper we investigated the equivalence of six machine learning techniques when used to predict the defect proneness of software classes using a cross-project strategy. Specifically, we analyzed the predicted defect proneness provided by the different classifiers aiming at verifying whether they are able to identify different defect-prone classes. Results from an empirical evaluation which involved 10 open-source Java projects demonstrated that *the experimented classifiers are not statistically equivalent, i.e., they are able to identify different sets of defect-prone classes*.

Such a complementarity between classifiers is also confirmed by the superior performances obtained by CODEP , *i.e.,* the novel combined approaches introduced and evaluated in this paper. CODEP turned out to be more accurate than all the stand-alone techniques, *i.e., the combinations are better than their constituents*. As side effect, the combined classifier allows to achieve a better cost-effectiveness than the stand alone-classifiers. In summary, the combination seems to be particularly useful for cross-project defect prediction and its advantages can also be investigated for within-project predictions.

As future work we plan to replicate our study considering more stand-alone machine learning techniques such as the “local prediction” models [23], [24] or approaches based on ensemble learning mechanisms [2], [27]. All these approaches can be considered as further stand-alone components for CODEP. We also plan to consider more strategies to combine the stand-alone machine learning techniques. Last, but not least, we plan to exploit the possibility to combine not only traditional single-objective defect prediction techniques but also multi-objective defect prediction models [30].

References

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

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

[3] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.

[4] G. Pai and J. Bechta Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *Software Engineering, IEEE Transactions on*, vol. 33, no. 10, pp. 675–686, 2007.

[5] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Software Eng.*, vol. 22, no. 10, pp. 751–761, 1996.

[6] T. Zimmermann and N. Nagappan, "Predicting defects with program dependencies," in *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, 2009, pp. 435–438.

[7] M. Bezerra, A. L. I. Oliveira, and S. R. L. 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*, 2007, pp. 2869–2874.

[8] E. Ceylan, F. Kutlubay, and A. Bener, "Software defect identification using machine learning techniques," in *Software Engineering and Advanced Applications, 2006. SEAA '06. 32nd EUROMICRO Conference on*, 2006, pp. 240–247.

[9] N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra, "Predicting software defects in varying development lifecycles using bayesian nets," *Information and Software Technology*, vol. 49, no. 1, pp. 32 – 43, 2007.

[10] 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 *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2009, pp. 91–100.

[11] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[12] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *30th International Conference on Software Engineering (ICSE 2008)*. Leipzig, Germany, May: ACM, 2008, pp. 181–190.

[13] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Software Eng.*, vol. 31, no. 4, pp. 340–355, 2005.

[14] S. Kim, T. Zimmermann, J. E. Whitehead, and A. Zeller, "Predicting faults from cached history," in *29th International Conference on Software Engineering (ICSE 2007)*. Minneapolis, MN, USA, May 20-26, 2007: IEEE Computer Society, 2007, pp. 489–498.

[15] E. Arisholm and L. C. Briand, "Predicting fault-prone components in a java legacy system," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ser. ISESE '06. ACM, 2006, pp. 8–17.

[16] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction." *IEEE Trans. Software Eng.*, vol. 31, no. 10, pp. 897–910, 2005.

[17] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *28th International Conference on Software Engineering (ICSE 2006)*. Shanghai, China, May 20-28, 2006: ACM, 2006, pp. 452–461.

[18] A. Okutan and O. Yldz, "Software defect prediction using bayesian networks," *Empirical Software Engineering*, pp. 1–28, 2012.

[19] A. Nelson, T. Menzies, and G. Gay, "Sharing experiments using open-source software," *Software: Practice and Experience*, vol. 41, no. 3, pp. 283–305, 2011.

[20] M. Bezerra, A. Oliveira, P. Adeodato, and S. Meira, "Enhancing rbf-dda algorithms robustness: Neural networks applied to prediction of fault-prone software modules," in *Artificial Intelligence in Theory and Practice II*, ser. IFIP The International Federation for Information Processing. Springer US, 2008, vol. 276, pp. 119–128.

[21] M. Elish, "A comparative study of fault density prediction in aspect-oriented systems using mlp, rbf, knn, rt, denfis and svr models," *Artificial Intelligence Review*, pp. 1–9, 2012.

[22] B. Turhan, T. Menzies, A. B. Bener, and J. S. 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.

[23] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. R. Cok, "Local vs. global models for effort estimation and defect prediction," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011*. IEEE, 2011, pp. 343–351.

[24] N. Bettenburg, M. Nagappan, and A. E. Hassan, "Think locally, act globally: Improving defect and effort prediction models," in *9th IEEE Working Conference o Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*. IEEE, 2012, pp. 60–69.

[25] A. E. Camargo Cruz and K. Ochimizu, "Towards logistic regression models for predicting fault-prone code across software projects," in *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement (ESEM 2009)*, Lake Buena Vista, Florida, USA, 2009, pp. 460–463.

[26] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 382–391.

[27] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 481–490.

[28] A. Misirli, A. Bener, and B. Turhan, "An industrial case study of classifier ensembles for locating software defects," *Software Quality Journal*, vol. 19, pp. 515–536, 2011.

[29] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, January 2010.

[30] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation*. Luxembourg, Luxembourg: IEEE, 2013, pp. 252–261.

[31] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the "imprecision" of cross-project defect prediction," in *Proceedings of the ACM-Sigsoft 20th International Symposium on the Foundations of Software Engineering (FSE-20)*. Research Triangle Park, NC, USA: ACM, 2012, p. 61.

[32] G. F. Cooper and E. Herskovits, "A bayesian method for the induction of probabilistic networks from data," *Machine Learning*, vol. 9, no. 4, pp. 309–347, Oct. 1992.

[33] M. D. Buhmann and M. D. Buhmann, *Radial Basis Functions*. New York, NY, USA: Cambridge University Press, 2003.

[34] R. Kohavi, "The power of decision tables," in *Machine Learning: ECML-95*, ser. Lecture Notes in Computer Science, N. Lavrac and S. Wrobel, Eds. Springer Berlin Heidelberg, 1995, vol. 912, pp. 174–189.

[35] A. Panichella, R. Oliveto, and A. De Lucia, "Technical report - cross-project defect prediction: L'union fait la force," University of Salerno: Department of Management & Information Technology, Tech. Rep., 10 2013. [Online]. Available: http://www.distat.unimol.it/reports/CODEP/

[36] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. IEEE Press, 2013, pp. 409–418.