A Search-based Training Algorithm for Cost-aware Defect Prediction

Annibale Panichella
Delft University of Technology
The Netherlands
a.panichella@tudelft.nl

Carol V. Alexandru
University of Zurich
Switzerland
alexandru@ifi.uzh.ch

Sebastiano Panichella
University of Zurich
Switzerland
panichella@ifi.uzh.ch

Alberto Bacchelli
Delft University of Technology
The Netherlands
a.bacchelli@tudelft.nl

Harald C. Gall
University of Zurich
Switzerland
gall@ifi.uzh.ch

ABSTRACT

Research has yielded approaches to predict future defects in software artifacts based on historical information, thus assisting companies in effectively allocating limited development resources and developers in reviewing each others' code changes. Developers are unlikely to devote the same effort to inspect each software artifact predicted to contain defects, since the effort varies with the artifacts' size (cost) and the number of defects it exhibits (effectiveness). We propose to use Genetic Algorithms (GAs) for training prediction models to maximize their cost-effectiveness. We evaluate the approach on two well-known models, Regression Tree and Generalized Linear Model, and predict defects between multiple releases of six open source projects. Our results show that regression models trained by GAs significantly outperform their traditional counterparts, improving the cost-effectiveness by up to 240%. Often the top 10% of predicted lines of code contain up to twice as many defects.

Keywords

Defect prediction, genetic algorithm, machine learning

1. INTRODUCTION

Statistical modeling has been frequently used in empirical software engineering to analyze software projects [7]. One of its leading applications is to create prediction models to anticipate where defects will occur in a software system. Such models are valuable in different contexts: For example, Kim *et al.* demonstrate their importance in efficient API testing, where prediction models increase the testing effectiveness in industrial environment [20]. Researchers and practitioners underline their importance to effectively allocate human and computing resources [11], for example during code review [5].

In early efforts, researchers (*e.g.*, [41]) had investigated prediction models to provide a *binary* classification of each software artifact: Likely or not likely to incur in future defects. Commonly used evaluation metrics were precision and recall [41] or the Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC) curve. The AUC plots the classes correctly classified as defective against those incorrectly classified as defective, as the prediction model's discrimination threshold varies.

Recently, researchers noted that the effort required by developers towards inspecting artifacts suggested by binary classification models varies depending on the artifact [25]. Larger and more complex software artifacts require additional inspection effort, thus hindering both the usefulness and the effectiveness of the prediction. As a solution, researchers have proposed to rethink prediction in terms of *cost-effectiveness*: Artifacts should be inspected in the order that maximizes the ratio between the number of defects found and the effort spent (effort usually approximated by the size of the artifacts) [11]. In this context, commonly used evaluation metrics are: (i) the *cost-effective* AUC (AUC-CE) [11], which represented a weighted version of the more traditional AUC metric; and (ii) the P_{effort} metric [11].

In every effort-aware prediction model presented so far—regardless of the employed statistical mechanism—the model is not directly trained to find the best fit to rank on the cost-effectiveness, rather to predict the *raw number of defects*, *i.e.*, an *approximation* of it.

The idea we propose and evaluate in this paper is to use genetic algorithms (GAs) to automatically tweak the coefficients of a prediction model such that the cost-effectiveness on the training set is maximized. Menzies *et al.* [28] were the first to propose the idea of manually tuning the internal parameters of a rule learner to find the proper settings, thus leading to a learner that significantly outperforms standard learning methods [28]; here, we aim to *automatically* train statistical models. We use GAs to evolve the coefficients of regression algorithms to build a model optimizing the cost-effectiveness on the training set, under the assumption that it will also predict cost-effectiveness better on the test set.

We assess our idea by implementing it and conducting an empirical evaluation on a number of distinct software systems and releases. As our baseline, we consider widespread statistical regression models (*i.e.*, generalized linear regression model (GLM) and regression trees (RT)) and metrics (*i.e.*, Chidamber and Kemerer (CK) metrics and Lines of Code (LOC)). Our results show that our approach significantly outperforms traditional models.

GECCO '16, July 20–24, 2016, Denver, Colorado, USA
© 2016 ACM. ISBN 978-1-4503-4206-3/16/07. . . $15.00
DOI: http://dx.doi.org/10.1145/2908812.2908938

2. BACKGROUND AND PROBLEM

Researchers have studied the relation between characteristics of the source code or the development process of a project and its evolution for more than two decades.

2.1 Previous work

Prediction approaches. Researchers devised a number of defect prediction approaches to guide software maintenance and evolution by identifying more defect-prone software artifacts [11]. These approaches are based on statistical models, whose main difference is the diverse sets of predicting metrics and the underlying algorithms that learn from these metrics and make predictions [15, 37]. Examples of metrics are the Chidamber and Kemerer's object-oriented (CK) metrics [10, 6], structural metrics [3] or process metrics [29]. Examples of algorithms are logistic regression used by Zimmermann *et al.* [40]; Multi-Layer Perceptron (MLP), radial basis function (RBF), k-nearest neighbor (KNN), regression tree (RT), dynamic evolving neuro-fuzzy inference system (DENFIS), and Support Vector Regression (SVR) used by Elish [14]; Bayesian networks used by Bechta [31]; and Naive Bayes, J48, Alternative Decision Tree (ADTree), and One-R considered by Nelson *et al.* [30]. Recently, other researchers have proposed further advanced machine learning techniques, such as ensemble learning [23], clustering algorithms [36], and combined techniques [32]. Lessman *et al.* [22] evaluated 22 classification models and showed that there is no statistical difference between the top-17 models when classifying software modules as defect prone. Meta-heuristics have been also investigated, such as using genetic algorithms (GAs) [13, 19, 23] or genetic programming (GP) [1] to build prediction model aimed at optimizing traditional performance metrics for classification problems, such as *precision*, *recall*, and *f-measure*.

Effort-aware prediction. Mende *et al.* [25], Kamei *et al.* [18], Menzies *et al.* [28], and D'Ambros *et al.* [11] are among the first to warn of the importance of taking into account the *effort* needed to review the files suggested by prediction models. Traditional performance metrics used for binary predictions (precision, recall, f-measure, AUC [32], error sum, median error, error variance, and correlation [11]) are not well-suited to evaluate prediction since they give the same priority/importance to all defect-prone software components. Instead, in a practical scenario engineers would benefit from identifying those software components likely to contain more defects earlier, or requiring lower inspection cost at the same number of defects. Consequently, prediction methods should be *cost-effective*, where the *effectiveness* is number of defects to predict and the inspection *cost* is approximated by the lines of code (LOC) metric, relying on the intuition that larger files require more time and effort to review than smaller files [11, 25, 18].

Previous work proposed performance metrics (*e.g.*, AUC-CE and P_{effort}) designed for evaluating the cost-effectiveness of prediction models [11, 34, 17, 32, 33, 28]. However, the models were still built using traditional training algorithms; for example, D'Ambros *et al.* [11] trained traditional linear regression models using the classical *iteratively re-weighted least square* algorithm; Rahman and Devanbu [33] used four different machine learning techniques (*i.e.*, Logistic Regression, J48, SVM, and Naive Bayes) that were trained using the corresponding classical training algorithms.

2.2 Problem statement

Even if previous work [25, 18, 11] proposes to evaluate prediction model using new *effort-aware* metrics based on the required inspection effort, these metrics have not been used to train the statistical models. In fact, statistical and machine learning techniques have been used to find models that minimize the prediction error when computing the number of defects in a software artifact, not to maximize the cost-effectiveness (which is a different problem). Moreover, cost-effective metrics have been used to assess the final quality of a model (*e.g.*, as a post training process). Therefore, models are built onto a training set optimizing some performance metric (*e.g.*, relative error and precision with respect to number of defects), but they are evaluated on the test set using *different* metrics (*e.g.*, AUC-CE).

Menzies *et al.* [28] were the first to propose the idea of considering the *effort* (LOC) in the inner loop of a rule learner (*i.e.*, WHICH). The WHICH parameters were manually tuned to find the settings leading to the best cost-effectiveness and this lead WHICH to significantly outperform standard learning methods [28]. In practice, the proposed solution requires to (i) *manually* tune the parameters and (ii) *manually* inspect the results on the test set to verify whether the performance is improved with respect to standard learning methods (*i.e.*, using the oracle).

Our goal. In this work, we aim at *automatically* finding the internal parameter values that optimize the cost-effectiveness (on the training set), instead of manually tuning them. To this aim, we present a general framework to train any regression model with GAs to effectively explore the search space of possible parameters, where the quality of each parameters' setting (represented as a GA individual) is evaluated based on its cost-effectiveness detected on the training set.

In the following we present the two statistical models that we use to instantiate our general framework (*i.e.*, GLM and RT). These statistical models make different assumptions over the training data and have been widely used in a number of defect prediction scenarios [11, 21], thus making them good candidates as subjects for this study.

2.3 Generalized linear regression

GLM is a generalization of the traditional linear regression model that relaxes some of the traditional assumptions, such as the normal distribution of data points and identical variance of the predictors. A GLM consists of three main components: (i) independent variables, (ii) a linear function, and (iii) a link function. In our case the (i) independent variables $M = \{m_1, \ldots, m_k\}$ are the software metrics used as explanatory variables of the scalar dependent variable Y, *i.e.*, the number of defects. The (ii) linear function condenses the independent variable into a scalar value η with a set of linear coefficients $B = \{\alpha, \beta_1, \ldots, \beta_k\}$ such that

$$\eta = \alpha + \beta_1 \times m_1 + \cdots + \beta_k \times m_k \tag{1}$$

The (iii) link function is a smooth and invertible linearizing function f that provides the relationship between the expectation of the outcome $\mu = E(Y)$ and the linear function:

$$f(\mu) \ = \ \eta \ = \ \alpha + \beta_1 \times m_1 + \cdots + \beta_k \times m_k \tag{2}$$

If the link function is the *identity function* with the underlying assumption that the data points are normally distributed, then Equation 2 corresponds to the traditional

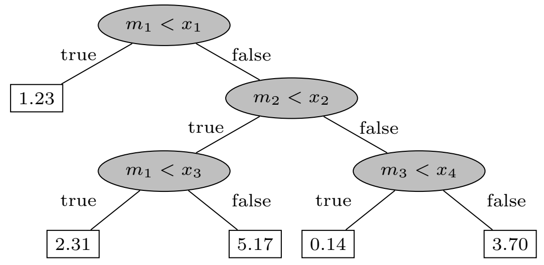


Figure 1: Regression tree for defect prediction.

multinomial linear regression $Y = \alpha + \beta_1 \times m_1 + \cdots + \beta_k \times m_k$, as used in previous work (*e.g.*, [41]). Given the general model represented by Equation 2, the problem is to find the set of coefficients $B = \{\alpha, \beta_1, \ldots, \beta_k\}$ such that the corresponding generalized linear model $f(\mu)$ minimizes the Mean Squared Error (MSE) between the predicted value and the actual outcome Y. The traditional algorithm to solve this problem is the *iteratively re-weighted least square* procedure.

2.4 Regression Tree

A regression tree (RT) is based on a tree-like structure where the internal nodes (*i.e.*, *decision nodes*) contain *decision rules* on software metrics (*e.g.*, number of classes) while the leaf nodes are the prediction outcomes, *i.e.*, number of defects a given class (see Figure 1). A decision rule is based on a software metric m_i and a *decision coefficient* x_i and it verifies whether a specific condition (*e.g.*, *if* $m_i < x_i$) is reached or not, thus partitioning the decision in two branches (*true* and *false*). Given a specific class instance, the classification is performed by traversing a specific path in the tree according to the set of satisfied conditions (rules) until reaching a leaf node, which contains the final predicted score, *e.g.*, number of defects. For example, a software artifact whose metrics traverse the first *true* branch in Figure 1 will be classified as *defect-prone* and its predicted number of defects will be 1.23. During the training process of the tree, a building algorithm is used to find the tree structure that provides the best prediction of the outcome for the training set. Specifically, given the set of software metrics $M = \{m_1, \ldots, m_k\}$, the traditional prediction problem consists in finding the regression tree model which minimizes the Mean Squared Error (MSE) [35]. One of the most used algorithms to solve this problem is the CART *greedy algorithm*, which applies a top-down strategy to derive the best structure of the tree through a subsequent splitting process.

3. PROPOSED SOLUTION

We hypothesize that if the target of prediction models is the cost-effectiveness and the models are *evaluated* differently with respect to traditional classification and regression problems, then models *trained* using a different, more appropriate training algorithms than traditional ones would show better results. Therefore, we propose to modify the training algorithm such that artifacts likely to have higher defect density are given higher priority.

To this end, instead of minimizing the MSE, we wish to maximize the ratio between the cumulative number of defects (effectiveness) and the total amount of code to inspect (cost), with regard to the list of predicted artifacts, ordered by their defect-proneness. Specifically, let $O = \langle a_1, \ldots, a_n \rangle$ be the list of artifacts in the training set ordered by their predicted scores produced by a regression model f_B, where

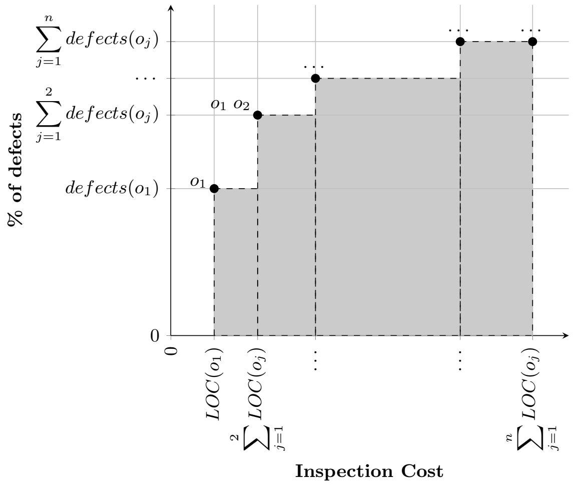


Figure 2: Graphical interpretation of the proposed fitness function

$B = \{\beta_1, \ldots, \beta_k\}$ is the set of regression coefficients. We reformulate the defect prediction problem as follows:

Problem 1. *For the regression model f_B, find the set of coefficients $B = \{\beta_1, \ldots, \beta_k\}$ that maximizes its cost-effectiveness, i.e., maximizing the cumulative number of defects encountered when inspecting the LOC for the predicted artifacts in the ordering $O = \langle o_1, \ldots, o_n \rangle$:*

$$\max \varphi(F_B) = \sum_{i=2}^{n} \left(\sum_{j=1}^{i-1} defects(o_j) \times LOC(o_i) \right) \tag{3}$$

In the equation above, $\sum_{j=1}^{i-1} actual(o_j)$ denotes the cumulative number of defects for the first $i-1$ artifacts in the ordering O. Finally, $LOC(o_i)$ measures the lines of code of the i-th artifact in the ordering O. We use the LOC metric as a proxy for effort as previously done [25, 4, 11, 8, 32, 9].

The fitness function reported in Equation 3 measures the AUC-CE using the rectangle rule, i.e., by summing-up the areas of rectangles forming the grey-area in Figure 2. This Figure plots the cost-effective ROC (ROC-CE) drawn by a given ordering $O = \langle a_1, \ldots, a_n \rangle$ of software artifacts produced by a regression model f_B. Specifically, ROC-CE plots on the y axis the cumulative number of actual defects encountered when analyzing the first i artifacts, whose cumulative inspection cost (approximated by artifacts size) is reported on the x axis. Hence, the function φ is equal to the sum of the rectangles with the following height and width:

$$Height_i = \sum_{j=1}^{i-1} defects(o_j) \tag{4}$$

$$Width_i = \sum_{j=1}^{i-1} LOC(o_j) - \sum_{j=1}^{i} LOC(o_j) = LOC(o_i) \tag{5}$$

The higher the grey-area in Figure 2, *i.e.*, the higher the function $\varphi(F_B)$, the higher the defect density for the first LOC to inspect according to the ordering O.

For GLM and RT, Problem 1 can be instantiated as reported below:

Problem 2. *Find the set of linear combination coefficients $B = \{\alpha, \beta_1, \ldots, \beta_k\}$ to use in Equation 2 to maximize the function φ.*

Problem 3. *Let t be the decision tree structure obtained using the CART algorithm. Find the set of decision coefficients $X = \{x_1, \ldots, x_k\}$ for the decision nodes in t to maximize the function φ.*

Therefore, we use the proposed function φ as the measure for predicting the performances of GLM and RT in the context of defect prediction.

3.1 Training Regression Models With GAs

To solve the aforementioned optimization problems, we apply GAs, *i.e.*, stochastic search algorithms inspired by natural selection and natural genetics. GAs are a class of a global search algorithms that uses multiple candidate solutions to explore, in parallel, multiple regions of the search space. In our case, the search space is denoted by the set of all possible sets of linear combination coefficients $B = \{\alpha, \beta_1, \ldots, \beta_k\}$ for GLM, and the set of decision coefficients $X = \{x_1, \ldots, x_k\}$ for RT. A candidate solution (*individuals*) is represented as an array with k floats (*chromosome*), where each element represents one regression coefficient in B or a decision coefficient in X. Thus, an individual is a particular GLM configuration or RT configuration, depending on the technique we are training.

GAs start with a random generated set of chromosomes (*population*), *i.e.*, randomly generated GLM or RT configurations. Then, the population is evolved during subsequent iterations (*generations*) using three genetic operators: (i) *selection*, (ii) *crossover*, and (iii) *mutation*. During each generation the chromosomes are first evaluated according to the *fitness* function to be optimized (function φ in our case). The best (fittest) chromosomes are then selected for reproduction using a selection operator. During this phase, new chromosomes (*i.e.*, *offspring*) are generated by recombining *genes* (chromosome elements) between two individuals from the current generation using the *crossover operator* and the *mutation operator*. At the end of each generation, the obtained chromosomes are used as starting points for the next generation. Further details on genetic operators and parameters setting used in this paper can be found in Section 4.

In the related literature, previous work applied evolutionary algorithms, and in particular GAs, to defect prediction. However, GAs have been used in order to optimize traditional performance metrics for classification problems [13, 19, 23, 1], such as *precision*, *recall*, *f-measure*, and *accuracy*. For example, Di Martino *et al.* [13] used GAs for calibrating the parameters of Support Vector Machines (SVM) to optimize the three aforementioned metrics. Liu *et al.* [23] use an evolutionary algorithm as a stand-alone model for predicting defect prone classes with the goal of maximizing the *accuracy* of the prediction. Khoshgoftaar *et al.* proposed multi-objective genetic programming (GP) to automatically generate classification trees that optimize the *accuracy* of the prediction, controlling the size of the decision tree. As reported in the survey by Azfal and Torkar [1] other variants of GPs have been used in order to (i) control the size of evolutionary classification trees, (ii) to penalize misclassified instances in the training set, (iii) to maximize the number of correctly classified defect-prone classes (precision) at a fixed level of recall. Recently, Canfora *et al.* [8, 9] proposed the application of multi-objective genetic algorithms to generate a set of classification models considering *file size* and *recall* as two objectives to optimize. Yang *et al.* [38] used GAs to optimize the ranks of defect-prone software components predicted by simple Linear Regression model (LR) without taking into account the inspection cost.

Previous GA-based and GP-based approaches for defect prediction used classification algorithms (*e.g.*, classification trees and neural networks) and not regression models (our baseline in this paper). Specifically, previous papers use GAs for calibrating algorithms to predict the defect proneness as a binary outcome (*i.e.*, defective or non-defective artifacts) while we use regression models that, by definition, predict *continuous* values (*e.g.*, number of defects) as done in [11] when measuring the cost-effectiveness. The most important difference with respect to previous approaches is that they use traditional performance metrics for classification problem as fitness functions to optimize [13, 19, 23, 38]. As explained in Section 2, traditional performance are not well-suited to evaluate prediction since they give the same priority/importance to all defect prone software components, independently from their size (cost) and the number of bugs (effectiveness).

Unlike previous papers proposing evolutionary algorithms for defect prediction, we use a different fitness function, which measures the cost-effectiveness of regression models, *i.e.*, their ability to identify earlier software artifacts with higher defect density.

4. EMPIRICAL EVALUATION

In this section, we present the design of the study we conduct to empirically evaluate the *cost-effectiveness* of our approach compared to traditional defect prediction approaches.

The study *context* consists of data from Java open-source projects, whose characteristics are summarized in Table 1. All steps of the data collection process are implemented in Bash and are available as part of the replication package[1]. To complete the dataset for the *defect prediction* task we determine the number of bugs in releases n for each class and for each project considering the data in the PROMISE repository [27]. It is important to highlight that we do not use the dataset available in [11]. The main reason is that such dataset do not have many releases and thus, inter-release prediction cannot be performed. Specifically, for each project we use LOC and CK metrics (a reliable set of metrics, well-tested on this task) from several releases and evaluated the effectiveness of our approach in using them to prioritize defect-prone classes in a cost-effective manner. It is important to notice that, while in this paper we used only LOC and the CK metric suite, other software metrics have been used in literature as predictors for building defect prediction models. However, we select the CK suite since it has been widely used to measure the quality of Object-Oriented (OO) software systems. Moreover, the purpose of this paper is not to evaluate which is the best suite of predictors for defect prediction, but to show the benefits of our GA-based cost-aware training process.

We structure our empirical evaluation around the following research questions:

> **RQ$_1$:** *Does our cost-aware training process improve the cost-effectiveness of GLM?*
>
> **RQ$_2$:** *Does our cost-aware training process improve the cost-effectiveness of RT?*

[1]http://tiny.uzh.ch/p0

Table 1: Java projects considered in the study.

System	Release	# Classes	% Defective Classes	Average # of Defects
Log4j	1.0	135	25.19	1.79
	1.1	110	33.94	2.32
	1.2	206	92.22	2.63
Lucene	2.0	196	46.43	2.94
	2.2	248	58.06	2.88
	2.4	341	59.53	3.11
Poi	1.5	237	59.49	2.43
	2.0	315	11.78	1.05
	2.5	387	64.42	2.00
	3.0	443	63.57	1.78
Synapse	1.0	157	10.19	1.31
	1.1	222	27.03	1.65
	1.2	256	33.59	1.67
Velocity	1.4	197	74.62	1.42
	1.5	215	66.05	2.33
	1.6	230	33.91	2.44
Xalan	2.4	724	15.19	1.42
	2.5	804	48.13	1.37
	2.6	886	46.39	1.52
	2.7	910	98.68	1.35

4.1 Research settings

We study the following independent factors:

Prediction algorithms. We use the implementations of GLM and RT available in MATLAB [24]. Specifically, for GLM, we use the `glmfit` routine to train the linear regression model using the *identity* function as the linking function [11]. For RT, we use the `fitrtree` routine with *CART* algorithm for building the tree structure. We also employ the *Global Optimization Toolbox*, particularly the `ga` routine, which implements the GAs we use to re-calibrate GLM and RT for the new fitness function proposed in this paper.

Training (release-project prediction). We conduct our empirical evaluation in the context of *release-project prediction.* In a real testing and maintenance context, release-project prediction means using data from the former releases to train the model used to predict faults for a new release. In other words, to predict the defects of a release n of a project, we train the model on the data of the project's previous release $n-1$. For example, for defect prediction we consider as training set CK metrics and LOC (independent variables) for release $n-1$ and the defects (dependent variable) on release $n-1$. As the test set, we apply the trained model to CK metrics and LOC (independent variables) of release n and try to predict the defects affecting release n (dependent variable).

Parameters setting. We employ the standard GA configuration used for real-coded (float) problems:

- *Population size.* We set GAs with a population of 200 individuals, *i.e.*, 200 for GLM and RT configurations.
- *Initial population.* For each release used as training set, the initial population is uniformly and randomly generated within the solution spaces. We randomly and uniformly generate the initial population in the interval $[-10; 10]$, which corresponds to the default configuration for the `ga` routine.
- *Number of generations.* We set the maximum number of generations equal to 400.
- *Crossover function.* We use the *blend crossover* (BLX-α) [16], *i.e.*, one of the most used and efficient crossover operators for real-coded (float) chromosomes. We use the standard configuration of $\alpha = 0.5$ as previous work in numerical optimization reported that BLX-0.5 performs better than BLX operators with other α values [16].
- *Mutation function.* We use the *polynomial mutation* function with distribution index $\eta_m = 20$, which is one of the most used mutation operators for real-coded GAs [12].
- *Selection Function.* For the selection function we used the *roulette wheel* selection schema.

To allow reliable detection of statistical differences between the traditional regression models and GAs-based ones, we run the GAs 30 times on each training set and evaluate the obtained models onto the corresponding test set.

4.2 Evaluation

To compare the prediction performance of traditional models and the proposed GA-based approach, we use the cost-effective ROC (ROC-CE), as proposed and employed by previous work [11, 32]. The ROC-CE plots on the x-axis the cumulative number of LOC to inspect for the predicted classes (*cost*) and on the y-axis the corresponding cumulative number of defects (*effectiveness*) reached by a specific model when considering the classes order by their predicted number of defects. To ease the comparison across models, we use the P_{effort} metric [11]. This metric measures the area under the cost-effective ROC curve (AUC-CE) of the evaluated prediction model with respect to the AUC-CE of the optimal classifier, *i.e.*, the *ideal* classifier that sorts the defect-prone classes in the test set according to the actual number of defects. As described by D'Ambros *et al.* [11], $P_{\text{effort}} = 1 - \triangle_{\text{effort}}$, where $\triangle_{\text{effort}}$ is the difference between the AUC-CE of the optimal classifier and the AUC-CE of the prediction model under analysis. The P_{effort} metric assumes values within the range $[0; 1]$ and its optimal value is equal to 1 (when the corresponding prediction model is equivalent to the optimal curve).

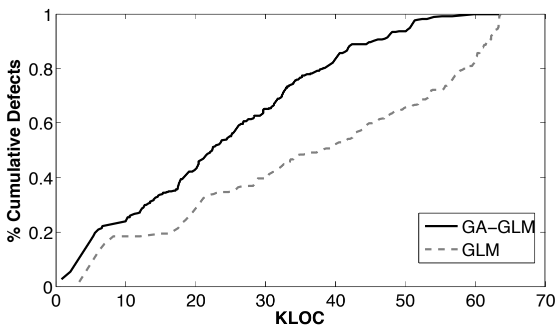
We analyze the results to check whether the differences between the P_{effort} scores produced by the compared algorithms over 30 independent runs are statistically significant. We use the Wilcoxon Rank Sum test with a significance level $\alpha = 0.05$. The Wilcoxon test is non-parametric and does not require any assumption upon the underlying data distribution; we perform a two-tailed test because we do not know *a priori* whether the difference is in favor of GA or the traditional models. Following the guidelines provided in [2] for assessing the performances of randomized algorithms we use the Vargha-Delaney ($\hat{A}_{12}$) statistical test: a non-parametric test for measuring the magnitude of the difference between the P_{effort} scores achieved with different algorithms. An effect size $\hat{A}_{12} > 0.5$ means that the P_{effort} scored yielded by GAs are better than the score produced by RT (or GLM) over 30 independent runs. $\hat{A}_{12} < 0.5$ means RT (or GLM) is better than GAs in terms of P_{effort}, while $\hat{A}_{12} = 0.5$ means there is no difference between the compared algorithms.

5. RESULTS

Table 2 reports the P_{effort} scores achieved by (i) traditional GLM, (ii) traditional RT, (iii) GLM trained with GAs using Equation (3) as fitness function (GLM-GA), and (iv) RT trained with GAs for optimizing Equation (3) (RT-GA). More specifically, for RT-GA and GLM-GA Table 2 report

Table 2: Average P_{effort} scores by models in term of *number of defects* for test sets over 30 independents runs.

System	Training Set	Test Set	RT		RT-GA			GLM		GLM-GA		
			Mean	St. Dev.	Mean		St. Dev.	Mean	St. Dev.	Mean		St. Dev.
Log4j	1.0	1.1	0.6920	-	**0.7461**	(+7.81%)	0.0272	0.7734	-	**0.8407**	(+8.71%)	0.0146
	1.1	1.2	0.4533	-	**0.5938**	(+30.98%)	0.0465	0.5369	-	**0.6728**	(+25.33%)	0.0336
Lucene	2.0	2.2	0.6386	-	**0.7947**	(+24.45%)	0,0174	0.5284	-	**0.7711**	(+45.93%)	0.0190
	2.2	2.4	0.5167	-	**0.6968**	(+34.84%)	0.0378	0.6463	-	**0.7660**	(+18.53%)	0.0036
Poi	1.5	2.0	0.3860	-	**0.5283**	(+36.84%)	0.0241	0.3945	-	**0.6876**	(+74.31%)	0.0028
	2.0	2.5	0.4563	-	**0.6250**	(+36.95%)	0.0252	0.6665	-	**0.8595**	(+28.96%)	0.0684
Synapse	1.0	1.1	0.5448	-	**0.5691**	(+4.45%)	0.0030	**0.6528**	-	0.5231	(-19.87%)	0.0063
	1.1	1.2	0.6440	-	**0.6802**	(+5.61%)	0.0354	0.5356	-	**0.6521**	(+21.74%)	0.0144
Velocity	1.4	1.5	0.8232	-	**0.8698**	(+5.66%)	0.0102	0.7992	-	**0.9113**	(+14.03%)	0.0027
	1.5	1.6	0.4505	-	**0.8082**	(+79.45%)	0.0314	0.3531	-	**0.8162**	(+131.18%)	0.0028
Xalan	2.4	2.5	0.4279	-	**0.7319**	(+71.04%)	0.0925	0.4043	-	**0.8472**	(+109.53%)	0.0070
	2.5	2.6	0.5472	-	**0.8175**	(+49.40%)	0.0068	0.5219	-	**0.8388**	(+60.73%)	0.0007
	2.6	2.7	0.4247	-	**0.9141**	(+115.22%)	0.0302	0.2845	-	**0.9679**	(+240.19%)	0.0035



(a) GLM for Lucene 2.0 → 1.6

Figure 3: Comparison between Genetic Algorithms and Regression Algorithms when evaluating the achieved cost-effectiveness

the mean P_{effort} scores achieved over 30 independent runs as well as the corresponding standard deviation. From the analysis of results, we can observe that GA-based regression models (GA-RT and GA-GLM) significantly outperform their traditional counterparts in terms of P_{effort}. Indeed, for GLM we observe an overall improvement of 58.40% on average, with minimum improvement of +8.71% for `Log4j` (version 1.1) and +240.19% for `Xalan` version 2.7. For RT, the P_{effort} metric improves by up to +115%, with the improvement being at or above 30% in 8 out of 13 releases (38.67% on average). To provide a graphical example for our results, Figure 3, shows the AUC-CE (AUC cost-effectiveness) obtained by GLM and GLM-GA for `Lucene` when using version 2.0 as training set and version 2.2 as test set. We can observe that the first 10% of KLOC to inspect actually contains roughly the same number of actual defects (around 20%). However, when considering the first 30% of KLOC to inspect, the traditional GLM yields roughly 40% of defects, while the GA approach discovers over 65%.

Cross-comparing the results in Table 2 with the characteristics of projects and releases in Table 1, we observe that the difference in predictive power is especially noticeable between releases where only few classes are defective. In these cases, traditional models fall short and are unable to accurately predict which classes are more defect-prone. We consider some examples. In release 2.0 of Poi, only 11.78% of classes (37 out of 315) have been found to contain bugs. If the developers would pick the first 25% of lines of code predicted to contain defects for further investigation using the traditional RT, this will yield them only 18% of actual bugs (7 out of 39 bugs). Using the GA-based model (GA-RT), developers will actually find 28% of actual bugs (11 out of 39 bugs). Overall, for the scenario of effectively allocating

Table 3: Wilcoxon test p-values of the hypothesis GAs > Regression Tree (or Generalized Linear Model) in terms of P_{effort} for defect prediction.

System	Test Set	GA > RT			GA > GLM		
		p-value	$\hat{A}_{12}$	Magn.	p-value	$\hat{A}_{12}$	Magn.
Log4j	1.1	< 0.01	1.00	L	< 0.01	1.00	L
	1.2	0.01	1.00	L	< 0.01	1.00	L
Lucene	2.2	< 0.01	1.00	L	< 0.01	1.00	L
	2.4	0.01	1.00	L	< 0.01	1.00	L
Poi	2.0	< 0.01	1.00	L	< 0.01	1.00	L
	2.5	< 0.01	1.00	L	< 0.01	1.00	L
	3.0	< 0.01	1.00	L	< 0.01	1.00	L
Synapse	1.1	< 0.01	0.82	L	0.99	0.1	L
	1.2	< 0.01	0.75	L	< 0.01	1.00	L
Velocity	1.5	< 0.01	1.00	L	< 0.01	1.00	L
	1.6	< 0.01	1.00	L	< 0.01	1.00	L
Xalan	2.5	< 0.01	1.00	L	< 0.01	1.00	L
	2.6	< 0.01	1.00	L	< 0.01	1.00	L
	2.7	< 0.01	1.00	L	< 0.01	1.00	L

limited resources with the goal of maximizing the positive impact on the development process, our approach offers a stronger solution that existing models. Table 3 reports the results of the Wilcoxon test and of the Vargha-Delaney ($\hat{A}_{12}$) statistical test. In all cases the GA leads to a significant improvement for both RT and GLM, with the effect size being *large* for all test sets.

Summary. Regression models trained by GAs significantly outperform traditional models in terms of cost-effectiveness, especially when there is a small proportion of classes with defects.

Running Time. In terms of execution time, training RT and GLM with GAs requires more time compared to traditional MSE-based training algorithms (which require a few seconds on average). Specifically, in our study, the GA requires on average 2min 6s to train RT and 2min 36s to train GLM to converge. However, this increase in running time presents a small trade-off compared to the ability to better prioritize defect-prone classes. For this analysis, the execution time was measured using a machine with an Intel Core i5 processor running at 2.4GHz with 16GB RAM and using the MATLAB `cputime` routine, which returns the total CPU time (in seconds) used by a MATLAB script.

6. THREATS TO VALIDITY

This section discusses the threats that could affect the validity of our research and the reported study.

Construct validity. Some of the measures we used to assess the models (P_{effort} metric [25] [11]) are widely adopted in the context of defect prediction. Specifically, we rely on P_{effort} metric because it allows us to perform a better evaluation, since, as pointed out by D'ambros *et al.* [11], it denotes the difference between the area under the curve of the optimal classifier and the area under the curve of the prediction model. We use the amount of LOC to inspect as proxy indicator of the effort required to analyze/test the predicted software modules, as also done in many previous papers [11, 25, 18, 33, 34, 8]. We are aware that such a measure is not necessarily representative of the real effort cost, although proportional to it [11]. In addition, another threat to construct validity can be related to the used metrics and data sets. Although, for our defect prediction analysis we have performed our study on widely used data sets from the PROMISE repository, we cannot exclude that they can be subject to imprecision and incompleteness.

Internal validity. We mitigated the influence of the GA randomness when building the model by repeating the process 30 times and reporting the achieved mean values. Also, it might be possible that the performances of the proposed approach and of the approaches being compared depend on the particular choice of the machine learning technique. In this paper, we evaluated the proposed approach using two statistical models—GLM and RT—that have been extensively used in previous research on defect prediction [11]. We also statistically compared the various model using the Wilcoxon, non-parametric test, to check whether the differences between the P_{effort} scores produced by the compared algorithms (over 30 independent runs) are statistically significant or not.

External validity. The techniques we tried may show different results when applied to other software systems. To alleviate this, we chose 6 systems with unrelated characteristics. All projects we selected are established, have a long history, and involve many different developers in different phases of their life cycle. The sizes of the systems and the number of defects between releases varies significantly. Nevertheless, there is always a threat of bias regarding results stemming from empirical work [26].

7. CONCLUSION

In this paper, we hypothesized that current approaches for defect prediction may not reach their full potential, as they are trained on a task that is different from their final target. Specifically, current approaches based on statistical models are trained to find the best fit to predict the raw number of defects in artifacts, while the actual target is to rank them according to the most cost-effective predictions. To investigate this hypothesis, we presented a novel approach based on GA and assessed it through an empirical evaluation.

The proposed GA-based approach is designed to overcome limitations of traditional approaches, modifying the training algorithm so that artifacts likely to exhibit more defect (at same level of inspection cost) are given higher priority. The results of the empirical evaluation we conducted, involving 6 software projects, show that our GA-based approach significantly outperforms traditional models. In some cases, it can yield classes containing multiple times as many defects in the first 10% or 20% of lines of code to inspected compared to traditional approaches, thus providing a more solid base for resource allocation. Results also show that, in the considered evaluation, the approach improves predictions especially when there are few actual defects occurring, a situation in which traditional models make comparatively inaccurate predictions.

Future research will need to investigate if our approach can be applied to other prediction models (*e.g.*, Bayesian or neural networks) and to other software metrics (*e.g.*, historical bug tracking data or socio-technical information). Moreover, since predictions made by both traditional and optimized models degrade if a project contains a very large number of classes or if different releases show different (heterogeneous) characteristics [7, 39], further research should be conducted to improve predictions for very large projects.

8. REFERENCES

[1] W. Afzal and R. Torkar. On the application of genetic programming for software engineering predictive modeling: A systematic review. *Expert Systems with Applications*, 38(9):11984 – 11997, 2011.

[2] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1–10, May 2011.

[3] 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*, ISESE '06, pages 8–17. ACM, 2006.

[4] 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.*, 83(1):2–17, January 2010.

[5] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering*, pages 712–721, 2013.

[6] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.

[7] N. Bettenburg, M. Nagappan, and A. E. Hassan. Towards improving statistical modeling of software engineering data: think locally, act globally! *Empirical Software Engineering*, 20(2):294–335, 2015.

[8] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Multi-objective cross-project defect prediction. In *The 6th International Conference on Software Testing, Verification and Validation*, pages 252–261, 2013.

[9] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella. Defect prediction as a multi-objective optimization problem. *Software Testing, Verification and Reliability*, 25(4):426–459, June 2015.

[10] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, June 1994.

[11] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[12] D. Deb and K. Deb. Investigation of mutation schemes in real-parameter genetic algorithms. In *Swarm, Evolutionary, and Memetic Computing*, volume 7677 of *Lecture Notes in Computer Science*, pages 1–8. Springer Berlin Heidelberg, 2012.

[13] S. Di Martino, F. Ferrucci, C. Gravino, and F. Sarro. A genetic algorithm to configure support vector machines for predicting fault-prone components. In *Product-Focused Software Process Improvement*, volume 6759 of *Lecture Notes in Computer Science*, pages 247–261. Springer Berlin Heidelberg, 2011.

[14] 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*, 42(4):695–703, 2014.

[15] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Software Eng.*, 38(6):1276–1304, Nov 2012.

[16] F. Herrera, M. Lozano, and A. M. Sánchez. A taxonomy for the crossover operator for real-coded genetic algorithms: An experimental study. *International Journal of Intelligent Systems*, 18(3):309–338, 2003.

[17] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 279–289, Nov 2013.

[18] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, 2010.

[19] T. Khoshgoftaar, N. Seliya, and Y. Liu. Genetic programming-based decision trees for software quality classification. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, pages 374–383, Nov 2003.

[20] M. Kim, J. Nam, J. Yeon, S. Choi, and S. Kim. REMI: Defect prediction for efficient api testing. In *Proceedings of ESEC/FSE*, pages 990–993, 2015.

[21] S. Kpodjedo, F. Ricca, P. Galinier, Y. Guéhéneuc, and G. Antoniol. Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering*, 16(1):141–175, 2011.

[22] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Software Eng.*, 34(4):485–496, 2008.

[23] Y. Liu, T. M. Khoshgoftaar, and N. Seliya. Evolutionary optimization of software quality modeling with multiple repositories. *IEEE Trans. Software Eng.*, 36(6):852–864, Nov. 2010.

[24] MATLAB. *version 7.10.0 (R2010b)*. The MathWorks Inc., Natick, Massachusetts.

[25] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, page 7. ACM, 2009.

[26] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Software Eng.*, 33(1):2–13, Jan 2007.

[27] T. Menzies, R. Krishna, and D. Pryor. The promise repository of empirical software engineering data, 2015.

[28] T. Menzies, Z. Milton, B. Turhan, B. Cukic, and Y. J. A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17:375–407, 2010.

[29] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software Engineering*, pages 181–190. ACM, 2008.

[30] A. Nelson, T. Menzies, and G. Gay. Sharing experiments using open-source software. *Software: Practice and Experience*, 41(3):283–305, 2011.

[31] G. Pai and J. Bechta Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Trans. Software Eng.*, 33(10):675–686, 2007.

[32] A. Panichella, R. Oliveto, and A. De Lucia. Cross-project defect prediction models: L'union fait la force. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week*, pages 164–173, Feb 2014.

[33] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 432–441. IEEE Press, 2013.

[34] 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)*, page 61, Research Triangle Park, NC, USA, 2012. ACM.

[35] L. Rokach and O. Maimon. Decision trees. In *The Data Mining and Knowledge Discovery Handbook*, pages 165–192. 2005.

[36] G. Scanniello, C. Gravino, A. Marcus, and T. Menzies. Class level fault prediction using software clustering. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 640–645, Nov 2013.

[37] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Trans. Software Eng.*, 40(6):603–616, June 2014.

[38] X. Yang, K. Tang, and X. Yao. A learning-to-rank approach to software defect prediction. *IEEE Transactions on Reliability*, 64(1), March 2015.

[39] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 182–191. ACM, 2014.

[40] T. Zimmermann, N. Nagappan, and A. Zeller. Predicting bugs from history. *Software Evolution*, pages 69–88.

[41] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 1–9. IEEE Computer Society, 2007.