Smart Learning: a Search-based Approach to Rank Change and Defect Prone Classes

Carol V. Alexandru*, Annibale Panichella†, Sebastiano Panichella*, Alberto Bacchelli†, Harald C. Gall*
* University of Zurich, Switzerland
† Delft University of Technology, The Netherlands

Abstract—Research has yielded approaches for predicting future changes and defects in software artifacts, based on historical information, helping developers in effectively allocating their (limited) resources. Developers are unlikely to focus on all predicted software artifacts, hence the ordering of predictions is important for choosing the right artifacts to concentrate on. We propose using a Genetic Algorithm (GA) for tailoring prediction models to prioritize classes with more changes/defects. We evaluate the approach on two models, regression tree and linear regression, predicting changes/defects between multiple releases of eight open source projects. Our results show that regression models calibrated by GA significantly outperform their traditional counterparts, improving the ranking of classes with more changes/defects by up to 48%. In many cases the top 10% of predicted classes can contain up to twice as many changes or defects.

Keywords—code change, defect prediction, genetic algorithm

I. INTRODUCTION

Statistical modeling is an important technique that has been frequently used in empirical software engineering to analyze software projects [1]. One of the leading applications of statistical modeling is to create prediction models to anticipate where changes and defects will take place in a software system. Such prediction models are valuable in different contexts: For example, Bavota et al. make the case for their importance in public API consumption, calling for “monitoring systems aimed at providing [information about the change- and fault-proneness of APIs] to developers” [2]; while many researchers and practitioners underline their importance to effectively allocate human and computing resources [3].

Initially, researchers (e.g., [4]) have investigated prediction models to be able to classify each software artifact as either likely to have changes/defects in the future or not. This is a binary classification problem and commonly used evaluation metrics are precision and recall [5] or the Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC) curve, which plots the classes correctly classified as defective (true positives) against the classes incorrectly classified as defective (false positives) as the prediction model’s discrimination threshold is varied.

More recent work on prediction models raised the issue that the effort that developers have to put into inspecting artifacts suggested by binary classification models varies depending on the artifact [6]. Larger and more complex software artifacts require additional inspection effort, which might limit both the usefulness and the effectiveness of the prediction, especially when the prediction is used as a means to better allocate resources. As a consequence, researchers have proposed to rethink prediction as a ranking problem, in which artifacts should be ordered by maximizing the ratio of effort spent for a number of defects/changes found [3]. In other words, prediction models should be effort-aware and try to rank software artifacts based on the changes/defects they will exhibit, placing those with most changes/defects at the top. In this case, commonly used evaluation metrics are: the Spearman’s rank correlation between predicted and actual ranking, and the \( P_{opt} \) metric [6], defined as the difference between the AUC of the optimal ranking (considering the inspection of \( n\% \) of artifacts) and the AUC of the prediction model.

Interestingly, all the presented effort-aware prediction models—regardless of the underlying statistical mechanism they employ—use the same approach for ranking: The model is trained to find the best fit to predict the raw number of changes/defects, based on the given metrics; then the predicted numbers are used to rank the artifacts, and on this ranking the evaluation is conducted. We notice how this approach does not directly tackle the main target, i.e., the ranking, rather an approximation of it. We claim that this approximation can hinder the effectiveness of prediction models, which should be instead trained for the task on which they are evaluated.

In this paper, we address this problem by proposing an approach that transforms the way in which statistical models are trained, so that they are tailored to their actual target, i.e., ranking. The underlying idea is to use genetic algorithms (GAs) to tweak the coefficients of a prediction model such that it yields results in the desired order (for example change size or number of defects).

We assess our approach by conducting an empirical evaluation involving both change and defect prediction tasks 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 tree (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. Moreover, although the model is trained to better fit the target, we did not encounter overfitting issues. Overall, our results also corroborate our underlying hypothesis that prediction models should be trained on the task under which they are evaluated, thus hinting at the generalizability of our approach to different prediction scenarios.
II. Background and Problem Description

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

Prediction approaches. Researchers devised a number of defect and change prediction approaches to guide software maintenance activities by identifying the software artifacts that are more prone to being changed or to defects in the future [3]. All 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[7], [8]. Examples of metrics that have been used for this prediction task are the Chidamber and Kemerer’s object-oriented (CK) metrics [9], [10], [11], structural metrics [12] or process metrics [13]. Examples of algorithms are logistic regression used by Zimmermann et al. [14] to predict the defect proneness of classes using complexity, interaction and change metrics as predictors; 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 [15] for defect prediction; Bayesian networks used by Bechta [16]; and Naïve Bayes, 48, Alternative Decision Tree (ADTree), and One-R considered by Nelson et al. [17]. Recently, other researchers have proposed further advanced machine learning techniques for defect prediction, such as the Multivariate Adaptive Regression Spline (MARS) [18], Personalized Change Classification (PCC) [19], Logistic Model Trees (LMT) [20], ensemble learning [21], clustering algorithms [22], and combined techniques [23]. Interestingly, Lessman et al. [24] evaluated 22 classification models and showed that there is no statistical differences between the top-17 models when classifying software modules as defect prone.

Evaluation approaches. In recent years there has been an increasing interest in designing proper evaluation metrics for measuring the performance of defect and change prediction models. Traditional performance metrics used in most previous work are precision, recall, f-measure, AUC [23], error sum, median error, error variance, and correlation [3], Mende and Koschke [6], Arisholm et al. [25], and Rahman et al. [26] suggested that traditional performance metrics are not well-suited for evaluating defect prediction approaches in a practical scenario. Indeed, for traditional metrics, all defect prone software artifacts have the same priority while software engineers would benefit from identifying those software components containing more defects earlier. As pointed out by Mende and Koschke [6] and D’Ambros et al. [3] the scenario that is more useful in practice is to rank the classes by the predicted number of defects they will exhibit. In the context of defect prediction, prediction models assign a defect probability to all classes, according to which they can be ranked. Similarly, when using regression models to predict the number of defects [27], [3], source code classes can be ranked by the predicted number of defects they will show in the future. Given a list of ranked classes, a software engineer is expected to focus resources on as many items in the beginning of the list as possible. Thus, the main goal of defect prediction approaches is to maximize the percentage of defects that can be encountered when reviewing only the top \( k \% \) of classes [3]. This can be shown visually via a cumulative lift chart [28], [29], where the classes are ordered according to the prediction model on the \( x \) axis while the cumulative number of actual defects is plotted on the \( y \) axis. Similarly, in the context of change prediction, Koru et al. [30] proposed the usage of regression tree to predict the number of future changes in two releases of two software projects. As an evaluation metric they proposed to analyze the percentage of changes that will be applied to classes in the top positions of the ranked list, where classes are ordered by their predicted number of changes [30].

Reflection. Even if previous work [28], [29], [3], [30] proposes to perform a ranking or prioritization of the predicted defect or change prone classes, these metrics have been, so far, only used to assess the quality of a model (e.g., as a post training process), while machine learning techniques are used to find models that minimize the prediction error when computing the number of changes or defects in a software artifact. Therefore, models are built onto a training set optimizing some performance metric (e.g., relative error and precision with respect to number of changes/defects) while they are evaluated on the test set using different metrics (e.g., ranking and lift chart). In this paper, we claim that this situation may hinder the effectiveness of these models and we try to fill the gap between training and evaluation processes, showing that regression techniques can be adapted to build prediction models that optimize the ranking assigned to software models, such that classes that are likely to contain more defects or larger changes are prioritized.

In the following we present the two statistical models that we use as a baseline on which we build our approach, namely generalized linear regression (GLM) and regression tree (RT). These statistical models make different assumptions over the training data and have been widely used in a number of change and defect prediction scenarios [3], [31], thus making them good candidates as subjects for this study.

A. Generalized linear regression

GLM constitutes 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 [32]. 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 (defect prediction) or the number of changed lines of code (change prediction). The (ii) linear function condenses the independent variable into a scalar value \( \eta \) with a set of linear coefficients \( B = \{\alpha, \beta_1, \ldots, \beta_k\} \) such that

\[
\eta = \alpha + \beta_1 \times m_1 + \ldots + \beta_k \times m_k
\]

Finally, the (iii) link function is a smooth and invertible linearising function \( f \), which provides the relationship between the expectation of the outcome \( \mu = E(Y) \) and the linear function:

\[
f(\mu) = \eta = \alpha + \beta_1 \times m_1 + \ldots + \beta_k \times m_k
\]

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 multinomial linear
regression $Y = \alpha + \beta_1 \times m_1 + \ldots + \beta_k \times m_k$, as used in previous work on defect prediction (e.g. [33]). Other canonical link functions are the distributions poisson, gamma, and binomial [32]. Given the general model represented by Equation 2, the problem consists in finding 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 used to solve this problem is the iteratively re-weighted least square procedure [32].

### B. 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 or changed lines in 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 change/defect prediction problem consists in finding the regression tree model which minimizes the Mean Squared Error (MSE) [34]. 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. In particular, starting from the root node, the greedy algorithm uses the MSE to assign a given software metric $m_i$ to the current node and to then split each node in two children, partitioning the data in two subsets containing instances with similar predictor values (node purity). The process continues iteratively until no further improvements can be obtained in terms of MSE by further splitting [34]. Finally, the greedy algorithm assigns a prediction score to each leaf node in the tree that is equal to the mean of the scores (e.g., number of defects or number of changed lines of code) for the software artifacts falling into such a leaf.

![Fig. 1. Regression tree for change/defect prediction.](image)

III. PROPOSED SOLUTION

The target of defect/change prediction models is to provide an ordered set of artifacts by maximizing the ratio of effort spent for the number of defects/changes found. Although researchers use a ranking-based evaluation for prediction models to reflect this target, the models are still built using traditional training algorithms, which are designed to optimize for a different target and evaluation metric. For example, as described in sections II-A and II-B, regression algorithms are designed to predict the number of defects by using a specific mathematical model which condenses multiple software metrics (e.g., CK metrics) in only one scalar value. Among the set of all possible models, a specific training algorithm looks for the one that minimizes the Mean Squared Error (MSE), i.e., the difference between the predicted scalar values (obtained by condensing the software metrics) and the actual number of defects (or changed lines of code). This follows the general assumption that if a specific model minimizes the MSE on a training set, it will also provide better performance (lower MSE) when used to predict the raw number of defect for the test set.

In this paper, we claim that if the target of prediction models is ranking and if the models are evaluated differently with respect to traditional classification and regression problems, then these models should also be trained using a different, more appropriate training algorithm. Therefore, we propose to modify the training algorithm such that artifacts likely to contain more defects/changes have higher priorities within the ordered list of artifacts.

To this aim, we consider the cumulative number of defects (or changed lines of code) encountered when inspecting the predicted artifacts, ordered by their proneness to defects/changes, as the function to be optimized (maximized) instead of the MSE (to be minimized). Specifically, let $O = \{o_1, \ldots, o_n\}$ be the list of artifacts in the training set ordered by their predicted scores produced by a regression model $f_B$, where $B = \{\beta_1, \ldots, \beta_k\}$ is the set of regression coefficients. We reformulate the defect/change 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 the cumulative number of defects (or modified lines of code) encountered when inspecting the artifacts in $A$ using the ordering $O = \{o_1, \ldots, o_n\}$:

$$\max \varphi = \sum_{i=1}^{n} \left( \sum_{j=1}^{i} actual(o_j) \right)$$  \hspace{1cm} (3)

For defect prediction, $\sum_{j=1}^{i} actual(o_j)$ denotes the cumulative number of defects that can be detected inspecting the first $i$ artifacts in the ordering $O$; for change prediction, $\sum_{j=1}^{i} actual(o_j)$ measures the cumulative number of modified lines of code for the first $i$ artifacts in the ordering $O$.

Figure 2 provides a graphical interpretation of the proposed function as computed for defect prediction: It plots the cumulative lift chart for a given ordering $O = \{o_1, \ldots, o_n\}$ of software artifacts produced by a generic regression model $f_B$. This chart plots on the $y$ axis the cumulative number of actual defects encountered when analyzing the first $i$ artifacts.
reported on the $x$ axis. The proposed optimization function $\varphi$ corresponds to the gray area reported in Figure 2, i.e., it measures the area-under-curve delimited by the cumulative points obtained when inspecting the artifacts in the training set in the ordering specified by $O$. The function $\varphi$ is equal to to the sum of the rectangles of height $\sum_{j=1}^{n} \text{defects}(o_i)$ (more in general $\sum_{j=1}^{n} \text{actual}(o_i)$) and width equal to one. The higher the curve in Figure 2 (i.e., the higher the function $\varphi$), the better the corresponding regression model prioritizes artifacts containing more defects (or undergoing larger changes).

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

**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 $\varphi$.

**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 $\varphi$.

Therefore, we use the proposed function $\varphi$ as the measure for predicting the performances of GLM and RT in the context of both change and defect prediction.

### A. Re-calibrating Regression Models With Genetic Algorithms

To solve the aforementioned optimization problems, we apply Genetic Algorithms (GAs), i.e., stochastic search algorithms inspired by natural selection and natural genetics. Genetic Algorithms 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 $\varphi$ 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 GAs can be found in specific literature (e.g., [35], [36]).

In our approach, we use the Roulette wheel selection as the selection operator, which assigns to chromosomes with higher fitness a higher chances to be selected. The crossover operator is the blend crossover (BLX-α) [35], i.e., one of the most used and efficient crossover operators for real-coded (float) chromosomes. As the mutation operator, we use the polynomial mutation operator [37], which randomly changes a gene (i.e., one of the regression coefficients) of an individual, with a different parameter value. The fitness function that drives the evolution of GAs is the function $\varphi$ described in Problem 1.

In the related literature, previous work applies evolutionary algorithms, and in particular GAs, to defect prediction. However, GAs are used to calibrate machine learning techniques as stand-alone prediction algorithms to optimize traditional performance metrics for classification problems [38], [39], [40], [21], such as precision, recall, and $f$-measure. For example, Di Martino et al. [38] proposed the usage of GAs for calibrating the parameters of Support Vector Machines (SVM) to optimize the three aforementioned metrics. Liu et al. [21] use an evolutionary algorithm as a stand-alone model for predicting defect prone classes with the goal of maximizing the accuracy of the prediction. Hochman et al. [39] investigated GAs for calibrating (enhancing) artificial neural networks (ENN) to obtain the optimal configuration of the artificial neural network, maximizing the accuracy. Khosghoftaar et al. proposed multi-objective genetic programming to automatically generate classification trees that optimize the accuracy of the prediction, controlling the size of the decision tree. Recently, Canfora et al. [41], [42] proposed the application of multi-objective genetic algorithms to generate a set of classification models considering file size and recall as functions to optimize.

All previous GA-based approaches for defect prediction use 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 that predict the defect proneness as a binary outcome (i.e., defective or non-defective artifact) while we use regression models that, by definition, predict continuous values (e.g., number of defects). Another important difference with respect to previous approaches is that most of them use traditional performance metrics as fitness functions [38], [39], [40], [21] for classification problems. We use a different fitness function, which measures the ability of regression models to prioritize (rank) software artifacts containing more defects. Finally, this paper provides a wider investigation, as we evaluate our GA-based approach not only for defect prediction but also for change prediction, where the goal is to rank artifacts that will undergo larger (more complex) changes.
IV. EMPIRICAL EVALUATION

In this section, we present the design of the study we conduct to empirically evaluate the effectiveness of our approach in real scenarios and compared to traditional defect/change prediction approaches.

The study goal is evaluating the benefits introduced by the proposed GA-based approach when ranking change-prone and defect-prone software artifacts. The quality focus of the study is the ability of the proposed approach in prioritizing change-prone (on change size) and defect-prone artifacts (on the number of bugs), compared to traditional machine learning approaches, namely GLM and RT. The perspective is on researchers aiming at developing better prediction models that could be used, for example, by software project managers to better allocate available resources focusing on those artifacts that will require more maintenance (change) and bug fixing effort. The study context consists of data from 8 unrelated Java open-source projects (a summary of the characteristics of the projects is reported in Table I and Table II). Specifically, for each project we computed 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 either change-prone (in DERBY, GUAVA, LUCENE) and defect-prone (in IVY, POI, LOG4J, PROP) classes.

### A. Research questions

We structure our empirical evaluation around the following research questions:

**RQ1.** Does the prediction improve when existing models to rank change-prone classes are optimized with GAs?

This research question aims at evaluating the benefits of using GAs to calibrate RT and GLM in prioritizing change-prone classes according to the change size, compared to the performances of traditional regression techniques.

**RQ2.** Does the prediction improve when existing models to rank defect-prone classes are optimized with GAs?

Analogously, this research question aims at evaluating the benefits of using GAs to calibrate RT and GLM in prioritizing defect-prone classes according to the number of defects.

### B. Data Extraction

For the change prediction task, we compute Chidamber and Kemerer (CK) metrics and the lines of code (LOC) metric for each class in releases $n-1$ as well as the changes (lines added and removed) between releases $n-1$ and $n$ for each class in the DERBY, GUAVA, and LUCENE projects. Specifically, the data used for change prediction is collected from three sources:

1. The CK metrics are extracted using `ckjm` [43], which computes the metrics from Java class files. The class files are obtained from the binary releases published by each project. `ckjm` calculates the metrics for inner classes separately, but we recombine these results such that the data for each regular class contains the accumulated results for all inner classes contained within the class.

2. The LOC of each class is derived from the source code by counting the number of lines of code (source code without blank lines and comments). For each binary release we retrieve the matching git commit id and then check out the code using git and run `cloc` to collect the LOC metric.

3. Finally, we compute the number of lines added/removed by running `git diff` between the commit ids representing each pair of subsequent releases.

Since `ckjm` produces output for class names, while the other two extractions produce output for file names, we transform the class names back into file names (the Java file structure dictates that the path of a class in a package must match the path in the file system). Then, we combine the data so that every line in the data set contains a Java file name, the CK metrics and the LOC metric for each binary release.

---

The dataset we use for the defect prediction task also contains LOC and CK metrics for releases \( n - 1 \), plus the number of bugs in releases \( n \) for each class of the IVY, POI, LOG4I, PROP and SYNAPSE projects. We retrieve this data without modification from the PROMISE repository.\(^6\)

C. Research settings

We study the effect of the following independent factors:

**Prediction algorithms.** We use the implementations of GLM and RT available in MATLAB [44]. Specifically, for GLM, we use the `glmfit` routine to train the linear regression model using the identity function as the linking function. 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.

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. Since such a problem is unconstrained (i.e., there are no upper/lower bounds for the values that the regression coefficients can assume), 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 300.

**Crossover function.** As previously explained, we use the blend crossover (BLX-\(\alpha\) with \(\alpha \in \{1; 0\}\)) [35]. 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 \(\alpha\) values [35].

**Mutation function.** We use the polynomial mutation function with distribution index \(p_m = 20\), which is one of the most used mutation operators for real-coded GAs [37].

**Selection function.** As selection function we use 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.

**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 (or changes) for a new release. In other words, to predict the defects/changes 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).

D. Evaluation

To compare the prediction performance of traditional models and the proposed GA-based approach, we use cumulative lift charts, as proposed and employed by previous work [3], [28], [29]. These charts plot the cumulative number of defects (or changed lines of code) on the y axis, when varying the number of inspected classes ordered by their predicted number of defects (or changed lines of code). In our evaluation we use the number of changed (added/deleted/modified) lines of code to approximate the size of future changes (with the intuition that larger changes require more maintenance effort than smaller ones). Similarly, for the prediction of defects we consider the total number of defects affecting each class of a given release.

To ease the comparison across models, we use the \( P_{opt} \) metric [28], [3]. This metric measures the area-under-curve (AUC) delimited by the cumulative lift charts with respect to the AUC of the optimal classifier, i.e., the classifier that sorts the defect-prone classes in the test set (or change-prone classes) in descending order of the number of defects (or modified lines of code). As described by D’Ambros et al. [3], \( P_{opt} = 1 - \Delta_{opt} \), where \( \Delta_{opt} \) is the difference between the AUC of the optimal classifier and the AUC of the prediction model. Th \( P_{opt} \) 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 \( P_{opt} \) scores produced by the compared algorithms over 30 independent runs are statistically significant. We use the Wilcoxon Rank Sum test to determine whether the following null hypotheses could be rejected:

- \( H_{loc} \): There is no significant difference between the (ranked) predictions of change-prone classes performed by traditional regression models and their optimised versions performed by GA.
- \( H_{eq} \): There is no significant difference between the (ranked) predictions of defect-prone classes performed by traditional regression models and their optimised versions performed by GA.

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. For all tests we assume a significance level \( \alpha = 0.05 \), i.e., a 5% probability of rejecting the null hypothesis when it should not be rejected. Following the guidelines provided by Arcuri and Briand [45] to assess the performances of random algorithms, such as GAs, we use the Vargha-Delaney (\( \hat{A}_{12} \)) statistical test: a non-parametric test for measuring the magnitude of the difference between the \( P_{opt} \) scores achieved with different algorithms. An effect size \( \hat{A}_{12} > 0.5 \) means that the \( P_{opt} \) scores 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 \( P_{opt} \), while \( \hat{A}_{12} = 0.5 \) means there is no difference between the compared algorithms.

\(^6\)https://code.google.com/p/promisedata/
V. Results

In this section, we present the empirical results to our research questions. A replication package for our study is publicly available for download\(^7\). In the replication package, we provide: (i) the scripts for the extraction process on a specific dataset, (ii) the datasets used in our experimentation, and (iii) the raw data for the experimented predictors.

A. Does our approach improve change prediction models?

Our first research question seeks to evaluate whether change predictions improve when traditional models are optimized with GAs.

Table III shows that GA generally improves the ranking predictions of change-prone classes made by both models (bold values are statistically significant improvements as indicated in Table IV). The improvement in terms of \( P_{opt} \) when using RT varies between 0 and 36% (14% on average). For 6 out the 15 releases we considered, the improvement is greater than 20% and it is greater than 5% in further 5 out of 15 releases. When using GLM, the improvement ranges from 0 to 48% (8% on average), although in most cases it is lower than 12%.

The improvement is much larger when the traditional models achieve poor performance. For example the improvement of the predictions for Guava’s releases R.12 is +13% and R.13 is +24% for release R.11 R.12 → R.13 (Figure 3-(a)). The correlation is also notable in Derby: Traditional RT can well predict changes in release R.17, where only 15% of classes were changed. The correlation is also noticeable in Derby: Traditional RT can well predict changes in release R.17, where 99% of the classes were changed, yet it performs poorly predicting changes for release R.10.2.2.0, where 2% of classes were changed. This seems to indicate that the proposed approach gives precise predictions even when the number of changed classes is small in relation to the total number of classes.

![Comparison between Genetic Algorithms and Regression Algorithm](https://dx.doi.org/10.7287/peerj.preprints.1160v1)

The ranking quality is most important at the beginning of the list of predictions, as developers are only able to focus on a limited number of classes. Improvements in this initial range are best expressed visually and are apparent in Figure 3. The curves for the GA approach have a steeper incline at the beginning of each graph. For example, the overall improvement of the predictions for Guava’s release 12 is 8%, but in Figure 3-(a), we can see that the percentage of

\(^7\)http://www.ifi.uzh.ch/seal/people/alexandru/downloads/smart-learning-rp.html

\[\text{Fig. 3. Comparison between Genetic Algorithms and Regression Algorithm when evaluating the achieved ranking for change prediction}\]

Consider the average changed lines (last column of Table I), we see that in all cases where the traditional approach scores similarly to the GA one, there is a high portion of changed classes. For example, traditional RT performs well for predicting changes in GUAVA’s releases R.13, R.14 and R.15

\[\text{where around 50% of all classes were changed; in contrast, it}\]

\[\text{performs poorly for predicting changes in release R.17, where only 15% of classes were changed. The correlation is also noticeable in Derby: Traditional RT can well predict changes in release R.10.2.2.0, where 99% of the classes were changed, yet it performs poorly predicting changes for release R.10.2.2.0, where 2% of classes were changed. This seems to indicate that the proposed approach gives precise predictions even when the number of changed classes is small in relation to the total number of classes.}\]
changes contained at the beginning of the list differs between the two approaches. When looking at the first 10% of classes predicted by a traditional RT, only 20% of the actual changes are included. In contrast, the GA-based RT reveals 40% of the changes. The same holds for GLM. The overall improvement for Derby’s release 10.1.1.0 is only 6%. When looking at Figure 3-(b), the first 10% of classes predicted by a traditional model is 33% of all actual changes, while the predictions by GA reveal 50% of changes. In Figure 3-(b), we also see a vertical line segment in both curves, which represents a class that contains roughly 20% of all change between the two releases (‘org.apache.derby.iapi.db.OnlineCompress’). GLM reveals this class only when looking at over 60% of classes, while the GA approach includes this class in the first 25%. This finding corroborates the hypothesis on (i) the limits of traditional machine learning approaches in the ranking task and (ii) the potential of the proposed GA-based approach on overcoming such limitations.

Table IV reports the results of the Wilcoxon test and of the Vargha-Delaney ($A_{12}$) statistical test. In 13 cases out of 15, the proposed GA-based RT statistically outperforms the traditional RT in terms of $P_{opt}$ on GLM only in Guava’s releases R.13 and R.15. The Wilcoxon test did not reveal any statistical difference between GA-RT and GA. Similarly, for GLM the Wilcoxon test revealed that the proposed solution based on GAs statistically outperforms the traditional GLM in 14 cases out of 15, with the only one exception represented by Derby’s release 10.2.1.6. The $A_{12}$ statistics shows that the effect size is large for most cases over both traditional RT and GLM as reported in Table IV. According to the obtained results, we can reject the null hypothesis $H_{0C}$ in favor of the alternative hypothesis, i.e., there is a statistical significant difference between standard regression models (GLM and RT) and their enhanced version calibrated by GAs (GLM-GA and RT-GA) for change prediction.

**Summary RQ1.** GA-optimized models significantly outperform traditional ones when predicting the ranking of change-prone classes, especially when there is a small proportion of changed classes and when traditional models perform less effectively.

**TABLE IV. WILCOXON TEST p-VALUES OF THE HYPOTHESIS GAS > REGRESSION TREE (OR GENERALIZED LINEAR MODEL) IN TERMS OF $P_{opt}$ FOR CHANGE PREDICTION.**

<table>
<thead>
<tr>
<th>System</th>
<th>Test Set</th>
<th>RT-GA &gt; RT $p$-value $A_{12}$ Magn</th>
<th>GLM-GA &gt; GLM $p$-value $A_{12}$ Magn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derby</td>
<td>R.12</td>
<td>&lt; 0.01 0.7 M</td>
<td>&lt; 0.01 0.85 L</td>
</tr>
<tr>
<td></td>
<td>R.13</td>
<td>&lt; 0.01 1.00 L</td>
<td>&lt; 0.01 0.90 L</td>
</tr>
<tr>
<td></td>
<td>R.14</td>
<td>&lt; 0.01 1.00 L</td>
<td>&lt; 0.01 0.90 L</td>
</tr>
<tr>
<td></td>
<td>R.15</td>
<td>&lt; 0.01 0.90 L</td>
<td>&lt; 0.01 0.95 L</td>
</tr>
<tr>
<td>Guava</td>
<td>R.11</td>
<td>&lt; 0.01 1.00 L</td>
<td>&lt; 0.01 1.00 L</td>
</tr>
<tr>
<td></td>
<td>R.12</td>
<td>&lt; 0.01 1.00 L</td>
<td>&lt; 0.01 1.00 L</td>
</tr>
<tr>
<td></td>
<td>R.13</td>
<td>&lt; 0.01 1.00 L</td>
<td>&lt; 0.01 1.00 L</td>
</tr>
<tr>
<td></td>
<td>R.14</td>
<td>&lt; 0.01 1.00 L</td>
<td>&lt; 0.01 1.00 L</td>
</tr>
<tr>
<td></td>
<td>R.15</td>
<td>&lt; 0.01 1.00 L</td>
<td>&lt; 0.01 1.00 L</td>
</tr>
<tr>
<td></td>
<td>3.20</td>
<td>&lt; 0.01 1.00 L</td>
<td>&lt; 0.01 1.00 L</td>
</tr>
<tr>
<td>Lucene</td>
<td>3.30</td>
<td>&lt; 0.01 1.00 L</td>
<td>&lt; 0.01 1.00 L</td>
</tr>
<tr>
<td></td>
<td>3.40</td>
<td>&lt; 0.01 1.00 L</td>
<td>&lt; 0.01 1.00 L</td>
</tr>
</tbody>
</table>

Fig. 4. Comparison between Genetic Algorithms and Regression Algorithm when evaluating the achieved ranking for defect prediction

**B. Does our approach improve defect prediction models?**

Our second research question seeks to evaluate whether defect prediction improves when traditional models are optimized with GAs.

Table V shows that GA-based regression models (GA-RT and GA-GLM) significantly outperform their traditional counterparts in terms of $P_{opt}$ (bold values indicate statistically significant differences as reported in Table VI). The $P_{opt}$ metric improves by up to 48% for RT, with the improvement being at or above 10% in 6 out of 13 releases (15.9% on average), and up to 33% for GLM, with typical improvements below 6% (7.7% on average). Improvements are larger when the traditional model performs less effectively, similarly to the results obtained for change prediction (RQ1). For example, the improvement obtained for RT is large for PoI’s releases 2.5 (+34%) and 3.0 (+44%), while the traditional model scores below 0.60; while the improvement is small for Log4j’s release 1.2 (+1%) where the traditional RT scores at 0.82.

We analyze the $P_{opt}$ scores reached by traditional models with an eye on the percentage of defective classes (displayed per each project in Table II). We see that the traditional RT does not perform very well for IVY’s release 2.0, where only 11% of classes are defective, while the GA-based approach makes more accurate predictions. The traditional RT performs well for Log4j’s release 1.2, where 92% of classes contained defects, and the corresponding improvement provided by our approach is minimal. For the Prop project, both the traditional and GA-based models achieve lower performances with respect to other projects. We hypothesize that this is caused by the combination of the large number of classes and the low proportion of defective ones: This combination makes the
TABLE V. Average $P_{opt}$ scores by models in terms of number of defects for test sets over 30 independently runs.

<table>
<thead>
<tr>
<th>System</th>
<th>Training Set</th>
<th>Test Set</th>
<th>RT</th>
<th>RT-GA</th>
<th>GLM</th>
<th>GLM-GA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>St. Dev.</td>
<td>Mean</td>
<td>St. Dev.</td>
<td>Mean</td>
<td>St. Dev.</td>
</tr>
<tr>
<td>Log4</td>
<td>1.0</td>
<td>1.4</td>
<td>0.7561</td>
<td>0.0488</td>
<td>0.6330</td>
<td>0.0364</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>2.0</td>
<td>0.5166</td>
<td>0.0839</td>
<td>0.8046</td>
<td>0.0108</td>
</tr>
<tr>
<td>Log4</td>
<td>1.1</td>
<td>1.2</td>
<td>0.8281</td>
<td>0.0193</td>
<td>0.8403</td>
<td>0.0033</td>
</tr>
<tr>
<td>Log4</td>
<td>1.2</td>
<td>1.7</td>
<td>0.6435</td>
<td>0.0115</td>
<td>0.6435</td>
<td>0.0106</td>
</tr>
<tr>
<td>Poi</td>
<td>2.0</td>
<td>2.5</td>
<td>0.5906</td>
<td>0.0104</td>
<td>0.5354</td>
<td>0.0323</td>
</tr>
<tr>
<td>Prop</td>
<td>1</td>
<td>2</td>
<td>0.6263</td>
<td>0.0089</td>
<td>0.7242</td>
<td>0.0033</td>
</tr>
<tr>
<td>Synapse</td>
<td>1.0</td>
<td>1.1</td>
<td>0.6273</td>
<td>0.0218</td>
<td>0.6849</td>
<td>0.0088</td>
</tr>
<tr>
<td>Synapse</td>
<td>1.2</td>
<td>1.3</td>
<td>0.7218</td>
<td>0.0218</td>
<td>0.6922</td>
<td>0.0170</td>
</tr>
</tbody>
</table>

TABLE VI. Wilcoxon test p-values of the hypothesis GAs > Regression Tree (or Generalized Linear Model) in terms of $P_{opt}$ for defect prediction.

<table>
<thead>
<tr>
<th>System</th>
<th>Test Set</th>
<th>GA &gt; RT</th>
<th>GA &gt; GLM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p-value</td>
<td>Magn.</td>
<td>p-value</td>
</tr>
<tr>
<td>Ivy</td>
<td>1.4</td>
<td>&lt; 0.01</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>0.01</td>
<td>0.70</td>
</tr>
<tr>
<td>Log4</td>
<td>1.1</td>
<td>&lt; 0.01</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>1.2</td>
<td>0.01</td>
<td>0.70</td>
</tr>
<tr>
<td>Poi</td>
<td>2.5</td>
<td>&lt; 0.01</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>3.0</td>
<td>&lt; 0.01</td>
<td>1.00</td>
</tr>
<tr>
<td>Prop</td>
<td>2</td>
<td>&lt; 0.01</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>&lt; 0.01</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>&lt; 0.01</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>&lt; 0.01</td>
<td>1.00</td>
</tr>
<tr>
<td>Synapse</td>
<td>1.1</td>
<td>&lt; 0.01</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>&lt; 0.01</td>
<td>1.00</td>
</tr>
</tbody>
</table>

VI. Discussion

We set out with the goal of improving the ranking of predicted changes and defects provided by traditional prediction models. Given that predictions are often done to better allocate resources (e.g., developers’ time), an effective ranking of predicted artifacts is a very desirable outcome, so that project managers and developers can make better, informed choices regarding which classes to focus on. Using GAs towards this goal significantly improves the quality of predictions overall, but more importantly, the ranking of predictions is improved such that those classes which are ranked higher actually contain much of the actual changes or defects.

We discussed in Section V that the difference in predictive power is especially noticeable between releases where only few classes change. In these cases, traditional models fall short and are unable to accurately predict which classes are more change or defect-prone. This reduces the applicability of these models for those projects with a large code base and relatively few changes that require good prediction performances. In this context, our approach provides a valuable improvement in exactly those cases where they are needed most. Indeed, our results shows that our approach is able to accurately predict changes even if few of them occur, which could mean that the approach can be applied continuously. It could be feasible to continuously predict future changes on the basis of data which is only a few weeks old. Testing this hypothesis is an interesting venue for future research.

We consider some concrete examples. Between Derby’s releases 10.1.2.1 and 10.2.1.6, 6,705 lines have been changed in only 9.71% of the code (114 out of 1,174 classes as shown in Table I). In this scenario predictions have to be accurate in order to be useful, since resources are not likely to be spread over hundreds of files. If we assume that resources (e.g., developers) are allocated to the top 100 predicted classes (8.5% of classes), the traditional model recommends 27% of actual changes (1,810 lines), whereas the improved model recommends 41% of changes (2,749 lines). A similar example can be made for bug prediction. In release 2.0, only 11.36% of classes (40 out of 352) have been found to contain bugs. If the developers would pick the first 50 classes predicted to contain defects for further investigation using the traditional approach, this will yield them only 18% of the classes actually containing defects (7 out of 40 classes). Using the GA-based model, developers will actually find 50% of the defective classes (20 out of 40 classes). Overall, for the scenario of effectively allocating limited resources with the goal of maximizing the prediction more difficult. Nevertheless, the GA significantly improves predictions for this project in all releases.

Figure 4-(a), shows that GA can largely improve the ranking of predicted defect-prone classes. The first 10% of predicted classes actually contain roughly the same number of actual defects (around 15%). However, when considering the first 20% of predicted classes, the traditional RT yields less than 25% of defects, while the GA approach discovers over 60%. Comparing the curves for the predictions made by GLM in Figure 4-(b), we see that the first 10% of classes yielded by the traditional model contains roughly 20% of defects, while the GA approach discovers over 30%. At 20% of predicted classes, the traditional model yields 40% of defects, while the GA yields over 50%.

Table VI reports the results of the Wilcoxon test and of the Vargha-Delaney ($A_{12}$) statistical test. In all the cases the proposed GA-based RT statistically outperforms the traditional RT in terms of $P_{opt}$. For GLM the Wilcoxon test revealed that the proposed solution based on GAs statistically outperforms the traditional GLM in 14 cases out of 15, with the only one exception represented by Prop’s release 4. The numeric and verbal effect size ($A_{12}$) is large for most releases. Therefore, we can reject the null hypothesis $H_{0D}$.

Summary RQ2. GA-optimized models significantly outperform traditional ones when predicting the ranking of defect-prone classes, especially when there is a small proportion of classes with defects and when traditional models perform less effectively.
positive impact on the development process, our approach offers a stronger solution that existing models.

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 1min 45s to train RT and 1min 32s to train GLM for change prediction, while for defect prediction it requires on average 2min 6s for RT and 2min 36s for GLM. However, this increase in running time presents a small trade-off compared to the ability to better prioritize defect- or change-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.

VII. Threats to Validity

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

Threats to construct validity concern the relation between theory and experimentation. Some of the measures we used to assess the models ($P_{\text{opt}}$ metric [28] [3]) are widely adopted in the context of defect prediction. Specifically, we rely on $P_{\text{opt}}$ metric because it allows us to perform a better evaluation, since, as pointed out by D’Ambros et al. [3], it denotes the difference between the area under the curve of the optimal classifier and the area under the curve of the prediction model. In addition, another threat to construct validity can be related to the used metrics and defect/change 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. Regarding the change data sets we compute Chidamber and Kemerer (CK) metrics and lines of code (LOC) metric for each class using widely adopted tools such as ckm and cloc.

For what concerns the threats to 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 change/defect prediction [3], [31]. We also statistically compared the various model using the Wilcoxon, non-parametric test, to check whether the differences between the $P_{\text{opt}}$ scores produced by the compared algorithms (over 30 independent runs) are statistically significant or not.

Threats to external validity concern the generalization of our findings. The approaches we tried may show different results when applied to other software systems. To alleviate this, we chose 8 systems with unrelated characteristics. The systems are developed by separate communities, although we focus largely on projects run by the Apache foundation (with the exception of the Prop data for defect prediction, which stems from an industrial project), since all Apache projects provide well-organized archives of old binary releases required for our analyses. 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 or changed lines of code between releases varies significantly. Nevertheless there is always a threat of bias regarding results stemming from empirical work [46].

While in this paper we use only LOC and the CK metric suite, other software metrics have been used in literature as predictors for building change and defect prediction models. The choice of the CK suite is guided by the wide use of such metrics to measure the quality of Object-Oriented (OO) software systems. However, the purpose of this paper is not to evaluate suites of predictors for defect prediction, but to show the benefits introduced by using GAs to calibrate RT and GLM in prioritizing defect- and change-prone classes, compared to the performances of traditional regression techniques. An extensive analysis of the different software metrics used as predictors can be found in the survey by D’Ambros et al. [3].

VIII. Conclusion

In this paper, we raised the concern that current approaches to change and defect prediction may not reach their full potential, because 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 changes/defects in artifacts, while the target is to predict how these artifacts should be ranked on the number of changes/defects. To investigate and tackle this issue, we present a novel approach based on GA and assess 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 contain more defects/changes have higher priorities within the ordered list of artifacts. The results of the empirical evaluation we conducted, involving 8 software projects, show that our GA-based approach significantly outperforms traditional models when predicting defect- and change-prone classes. Specifically, in some cases, it can yield classes containing multiple times as many changes or defects in the first 10% or 20% of predicted classes 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 changes/defects occurring, a situation in which traditional models make comparatively inaccurate predictions.

Overall, our results corroborate our underlying hypothesis that prediction models should be trained on the task under which they are evaluated, thus hinting at the generalizability of our approach to different prediction scenarios. Future research will need to investigate if our approach can be applied to other prediction models as well, for example Bayesian or neural networks and to software metrics other than code metrics, such as historical bug tracking data or social network parameters surrounding a class. Moreover, since predictions made by both traditional and optimized models degrade if a project contains a very large number of classes, large software projects need to be modularized and split into cohesive, manageable portions which can be individually analyzed, such that more accurate predictions are possible.
REFERENCES


[40] T. Khoshgoftaar, N. Seliya, and Y. Liu, “Genetic programming-based decision trees for software quality classification,” in Proceedings of...


