Soft Comput (2018) 22:1959–1980
https://doi.org/10.1007/s00500-016-2456-8

METHODOLOGIES AND APPLICATION

Multi-objective cross-version defect prediction

Swapnil Shukla[1] · T. Radhakrishnan[1] · K. Muthukumaran[1] · Lalita Bhanu Murthy Neti[1]

Published online: 9 December 2016

Abstract Defect prediction models help software project teams to spot defect-prone source files of software systems. Software project teams can prioritize and put up rigorous quality assurance (QA) activities on these predicted defect-prone files to minimize post-release defects so that quality software can be delivered. Cross-version defect prediction is building a prediction model from the previous version of a software project to predict defects in the current version. This is more practical than the other two ways of building models, i.e., cross-project prediction model and cross- validation prediction models, as previous version of same software project will have similar parameter distribution among files. In this paper, we formulate cross-version defect prediction problem as a multi-objective optimization problem with two objective functions: (a) maximizing recall by minimizing misclassification cost and (b) maximizing recall by minimizing cost of QA activities on defect prone files. The two multi-objective defect prediction models are compared with four traditional machine learning algorithms, namely logistic regression, naïve Bayes, decision tree and random forest. We have used 11 projects from the PROMISE repository consisting of a total of 41 different versions of these projects. Our findings show that multi-objective logistic regression is more cost-effective than single-objective algorithms.

Keywords Cross-version defect prediction · Multi-objective optimization · Search-based software engineering · Misclassification cost · Cost-effectiveness

1 Introduction

Software project teams adopt various techniques such as testing and expert reviews of software artifacts to improve quality. Due to limited resources and tight schedules, it may not be possible to take up these activities on all files. The defect prediction model predicts defect-prone files using trained models built from the historical data. Software project teams can prioritize and focus on rigorous QA activities of these predicted defect-prone files to minimize post-release defects. The defect prediction work has evolved in many different directions like finding effective predictor metrics, efficient prediction algorithms and performing defect prediction at different granularity levels like method, file and project. In the past, defect prediction models were built using process metrics, product metrics or bug history of software systems.

Prediction models are built with training data, and they are evaluated for performance on the testing data. In the literature, defect prediction models are built using three prediction techniques that differ based on the source of training data and testing data. The different prediction techniques are: (1) cross-validation prediction: In this approach, training and testing data are taken from the same version of a project. The defect prediction models are built by taking 70% of data as the training data and tested on remaining 30% of the data, or by using tenfold cross-validation technique; (2) cross-version

Communicated by V. Loia.

✉ Lalita Bhanu Murthy Neti
bhanu@hyderabad.bits-pilani.ac.in

Swapnil Shukla
h2014103019@hyderabad.bits-pilani.ac.in

T. Radhakrishnan
h2014103026@hyderabad.bits-pilani.ac.in

K. Muthukumaran
p2011415@hyderabad.bits-pilani.ac.in

[1] Department of Computer Science and Information Systems, BITS Pilani Hyderabad Campus, Shameerpet, Hyderabad 500078, India

prediction: In this approach, the data of the previous version of a software project are taken as the training data to build prediction model and it is tested on the data of current version of the same project; and (3) cross-project prediction: The prediction models are built by taking data from different projects as the training data and tested on the data of the project under study.

It is our intuitive understanding that using data of the previous version of the same project is more appropriate and can provide better results instead of using data from different projects to build defect prediction models. The main architectural or design characteristics of the project will remain more or less the same across different releases. The pattern of bug occurrences of the current version might also get influenced by the bug occurrences in the previous version. Hence, for predicting defect-prone files of the current version, the most suitable training data are the data of previous version of the same project. These factors motivated us to explore further into cross-version defect prediction model.

Harman suggested that the defect prediction problem can be viewed as a search problem, which can be solved using evolutionary algorithms (Harman 2010). The defect prediction problem can be formulated as a multi-objective optimization problem with contrasting objectives. We have attempted to solve two multi-objective defect prediction problems with competing objective functions in cross-version setup.
We formulate our first multi-objective defect prediction problem with the following contrasting objectives

– Maximize effectiveness of the model
– Minimize misclassification cost.

Effectiveness is the ratio of the number of components correctly predicted as defective to the number of actual defective components (recall). The misclassification cost is the cost incurred due to quality assurance activities required on wrongly classified files, i.e., cost of reviewing/testing the false-positive (predicted to be defective but not actually defective) files and cost of the false- negative (predicted to be non-defective but actually defective) files during post-release phase. Ideally any good model attempts to minimize the misclassification cost and maximize the effectiveness of model. Hence, we have chosen them as objective functions.

There are many evolutionary algorithms which can solve multi-objective optimization problems. In this study, we have used NSGA-II, a multi-objective genetic algorithm proposed by Deb et al. (2000). Any multi-objective genetic algorithm requires a fitness function that guides the search process to find optimal or near optimal solutions. We have considered logistic regression function as fitness function to find out cost and effectiveness. Hence, we are denoting this approach as multi-objective logistic regression (MOLR).

We build a cross-version defect prediction model using multi-objective logistic regression (MOLR) for our first problem and this model is denoted by M1. We also build prediction models using four traditional single-objective algorithms, namely logistic regression, naïve Bayes, decision tree and random forest. We try to answer the following research question.
RQ_1: How does the proposed M1 perform as compared to traditional single-objective defect prediction models in the cross-version defect prediction?
We formulate second multi-objective defect prediction problem with the following contrasting objectives.

– Maximize effectiveness of the model
– Minimize LOC cost.

Canfora et al. (2013, 2015) proposed this objective function to solve multi-objective defect prediction problem in cross-project setup. Effectiveness is the same as it is defined in M1. The cost borne by project team to perform review/test/any QA activity on all defect-prone files is taken to be another objective. As the effort required is directly proportional to the sum of lines of code of all defect-prone files, cost is taken as the sum of lines of code (LOC) of all the files which are predicted defective. This includes both true positives (predicted to be defective and actually defective) and false positives. Canfora et al. (2013, 2015) showed that the multi-objective cross- project prediction is more cost-effective than single-objective algorithms when there is lack of training data for a project, which is true for relatively newer projects. We consider these objective functions to implement multi-objective defect prediction models across versions of the same project.

We build cross-version defect prediction model using multi-objective logistic regression (MOLR) for second problem, and this model is denoted by M2. We try to answer the following research question.
RQ_2: How does the proposed M2 perform as compared to traditional single-objective defect prediction models in the cross-version defect prediction?

We have used 11 software projects from the PROMISE repository (Menzie et al. 2015), having a total of 41 different versions to answer RQ1 and RQ2. We had 30 pairs of training and testing versions. For each pair, we compared multi-objective logistic regression with the traditional classification algorithms in terms of cost-effectiveness.

The rest of the paper is structured as follows. Related work on defect prediction is discussed in Sect. 2. Datasets that consist of features and projects used in experiments are discussed in Sect. 3. We formulate problems in Sect. 4. And the process of building prediction models with multi-objective algorithm is discussed in Sect. 5. Experimental setup and analysis carried out to compare MOLR with traditional algorithms are described in Sect. 5. Section 6 discusses the results and threats to validity is explained in Sect. 7. Conclusions are presented at the end.

2 Related work

Defect prediction models have been built using various sets of static code attributes and process metrics. Metrics like lines of code (LOC), Halstead, CK and OO, change metrics and past bugs are mined from the code base, version control system and bug trackers. They are used to predict defect-prone files either within a project or across the projects. D'Ambros et al. (2010) tried to consolidate the defect prediction work, using the metrics mentioned above. Though they found that WCHU (weighted churn of source code metrics) and LDHH (linearly decayed entropy of source code metrics) are better performing metrics for defect prediction, they concluded that there is not a single metric that predicts defect-prone files well across software projects. But they agree to many other past works which say that past bugs and source code metrics are right alternatives in terms of overall prediction and computational requirements. Researchers often say most of the source code metrics are proxies of size. Zhang et al. (2009) investigated the relationship between size of files and defect, with an assumption that large code base may correlate with more defects. Their study concludes that the defect proneness increased with the size of the classes, but they suggested spending more resources on smaller classes which were found to be more problematic than larger classes. Kim et al. (2007) predicted defects using cached history. They assumed that the defects will not occur alone, but rather in bursts of several related faults. So they cached locations that are likely to have bugs. Basili et al. (1996) found that CK metrics are useful in finding defect proneness in early phases of the software development. Subramanyam and Krishnan (2003) showed that CK metrics are associated with defects even after controlling for the size of the software. A few other studies endorsed the defect prediction models built with change metrics, as they gave better prediction performance in classifying defect-prone files (Hassan 2009; Krishnan et al. 2011; Moser et al. 2008; Muthukumaran et al. 2015).

Apart from finding better prediction metrics, coming up with competitive prediction techniques for defect prediction is also an interesting research area. Classifiers such as logistic regression, naïve Bayes, decision tree, support vector machine and random forest are applied by different researchers in the past (He et al. 2013; Kamei et al. 2010; Kim et al. 2007; Mende and Koschke 2009; Peters et al. 2013). Czibula et al. used relational association rule mining to predict defective modules in software systems. Their model, defect prediction using relational association rules (DAPR) gives better predictive performance compared to the existing defect prediction techniques (Czibula et al. 2014). Marian et al. (2015) proposed unsupervised machine learning method based on self-organizing maps which puts defective and non-defective files in two clusters. Lessmann et al. performed a comparative study on defect prediction experiments with 22 classifiers applied on 10 public domain datasets from NASA repository and concluded that though the metrics-based classification was useful in this domain, the importance of classification algorithm was not significant. They did not find any significant performance difference between top 17 out of 22 classifiers used in the study (Lessmann et al. 2008). But Ghotra et al. (2015) argue that by making use of cleaned versions of NASA, PROMISE corpus and different classification algorithms, it is possible to produce defect prediction models with significant differences in performance.

With a perspective different from traditional approaches, Harman suggested to reformulate the classic software engineering problems as a search problem. This will help the community in finding solutions to difficult problems with competing constraints (e.g., quality, cost) (Harman 2010). Harman and Clark (2004) showed that many metrics can be used as the guiding force behind the search for optimal or near optimal solutions to many software engineering problems. Taking a clue from Harman's work, some researchers have come up with multi-objective defect prediction techniques. Canfora et al. (2013, 2015) proposed MODEP (multi-objective defect predictor) based on multi-objective forms of machine learning techniques, logistic regression and decision tree, trained using genetic algorithm. Carvalho et al. (2010) came up with multi-objective particle swarm optimization (MOPSO), which generates a model composed of rules with specific properties, which are intuitive and comprehensible.

Research studies mentioned above were focused on either within project cross-validation or cross-project defect prediction to build defect prediction models. According to our understanding, there is not much comprehensive work done on the defect prediction across multiple versions of a software project. Zimmermann et al. (2007) showed that defect prediction models learned from earlier releases can be used to predict defects for future releases. For instance, the model trained from release 1.0 can be used to predict defects in release 2.0. But they concluded that their results were far from being perfect. Yang et al. (2015) built a regression model based on the data of previous version to predict defect proneness of components in the current version and ranked them based on their defect proneness. They compared many traditional regression algorithms with their newly proposed learning-to-rank (LTR) approach and concluded that LTR performed better as compared to other algorithms. One of the recent works in cross-version defect prediction found that network measures are more effective in cross- version defect prediction. But they conclude that it does not improve

Table 1 Metrics (Canfora et al. 2015)

Name	Description
Lines of code (LOC)	Number of non-commented lines of code for each software component (e.g., in a class)
Weighted methods per class (WMC)	Number of methods contained in a class including public, private and protected methods
Coupling between objects (CBO)	Number of classes to which a class is coupled
Depth of inheritance (DIT)	Maximum inheritance path from the class to the root class
Number of children (NOC)	Number of immediate subclasses of a class
Response for a class (RFC)	Number of methods that can be invoked for an object of given class
Lack of cohesion among methods (LCOM)	Number of methods in a class that are not related through the sharing of some of the class fields

the prediction performance in a bigger way, especially when ranking fault-prone modules (Ma et al. 2016). Our work treats cross-version defect prediction within a project as a multi-objective optimization problem, with competing constraints like cost and effectiveness. We have presented a comprehensive comparison of multi-objective algorithm with traditional algorithms. We have conducted experiments in a large number of projects with multiple releases, with the motivation of providing more generalizable results.

3 Datasets and metrics

We are using six Chidamber and Kemerer metrics (CK) (Chidamber and Kemerer 1994) and LOC as features to build defect prediction models. The choice of the metrics is based on the fact that all projects used in our study are object-oriented projects and CK metrics has been widely used as quality indicators for object oriented softwares (D'Ambros et al. 2010; He et al. 2013; Herbold 2013; Jureczko and Madeyski 2010; Peters et al. 2013). Table 1 gives brief description about predictors used in our study.

We have considered 11 open-source projects that are having data for 3 or more versions in PROMISE repository (Menzie et al. 2015). The details of these versions, namely number of classes and percentage of defective classes, are presented in Table 2. As the table shows, different versions of the projects have 109–965 classes with average number of classes being 385. The choice of projects is done with a view of evaluating performance of multi-objective algorithm across projects having wide diversity, so that the results can be generalized.

Table 2 Projects under study

Project	Version	Number of classes	% of Defective classes
Ant	1.3	125	16
	1.4	178	22.47
	1.5	293	10.92
	1.6	351	26.21
	1.7	745	22.28
Camel	1.0	339	3.83
	1.2	608	35.53
	1.4	872	16.63
	1.6	965	19.48
Ivy	1.1	111	56.76
	1.4	241	6.64
	2.0	352	11.36
Jedit	3.2	272	33.09
	4.0	306	24.51
	4.1	312	25.32
	4.2	367	13.08
	4.3	492	2.23
Log-4j	1.0	135	25.18
	1.1	109	33.94
	1.2	205	92.19
Lucene	2.0	195	46.67
	2.2	247	58.3
	2.4	340	59.7
Poi	1.5	237	59.49
	2.0	314	11.78
	2.5	385	64.41
	3.0	442	63.57

Table 2 continued

Project	Version	Number of classes	% of Defective classes
Synapse	1.0	157	10.19
	1.1	222	27.03
	1.2	256	33.59
Velocity	1.4	196	75
	1.5	214	66.35
	1.6	229	34.06
Xalan	2.4	723	15.21
	2.5	803	48.19
	2.6	885	46.44
	2.7	909	98.79
Xerces	init	162	47.53
	1.2	440	16.14
	1.3	453	15.23
	1.4	588	74.32

4 Formulation of multi-objective defect prediction models

We formulate defect prediction problem as multi-objective optimization problem with contrasting objectives. We propose two multi-objective prediction problems each with distinct set of objective functions.

Most of the machine learning algorithms are single objective in nature. That is, their final goal is to estimate one solution that minimizes the prediction error. For example, logistic regression minimizes prediction error, i.e., root-mean-square error (RMSE) where RMSE is defined as follows,

$$RMSE = \sqrt{\sum_{i=1}^{n} (f(c_i) - dp(c_i))^2} \tag{1}$$

where $f(c_i)$ and $dp(c_i)$ take either 1 or 0, $f(c_i)$ represents whether c_i is defective class or not and $dp(c_i)$ represents whether c_i is predicted to be defective or not.

It is always good to have defect prediction models with high recall so as to minimize post-release defects. And it is easy to build models with recall value of 1. A dummy model which predicts all files to be defect prone will have a recall value of 1. But this model is of no use because recall is maximized without considering cost of misclassification. Hence, we would like to view defect prediction problem as multi-objective optimization problem rather than single-objective optimization problem. There will be two types of costs associated with defect prediction models. We will discuss building multi-objective defect prediction models that maximizes effectiveness and minimizes cost(misclassification cost/LOC cost) in the next two subsections.

4.1 Optimize misclassification cost and effectiveness

There are two types of errors for any prediction model. For defect prediction problem, type I error is predicting non-defective file to be defect prone and type II error is predicting defective file to be non-defective. In fact, the number of type I errors and type II errors is number of false-positive files and number of false- negative files, denoted by FP and FN. The cost incurred due to Type I errors is the effort spent by project team on quality assurance (QA) activities such as reviewing and testing of false-positive files. The cost incurred to fix type II errors is the effort spent by project team to fix the defective file in post-release phase.

$$Misclassification\ Cost = CostofType\ I\ errors + Cost\ of\ Type\ II\ errors \tag{2}$$

It is evident that cost of type II error is much more than the cost of type I error for our problem as fixing defective file during post release phase takes huge effort as compared to reviewing / testing a file before release. It is difficult to say how much type I error is costly as compared to type II error. Cost factor denotes how much is the cost of misclassifying a defective class as non-defective compared to misclassifying a non-defective class as defective. The cost factor α can be written as follows:

$$\alpha = Cost\ of\ Type\ II\ error\ /\ Cost\ of\ Type\ I\ error \tag{3}$$

If cost of $Type\ I$ error is c, then cost of $Type\ II$ error is αc.

$$Misclassification\ Cost = FPc + (FN)\alpha c \tag{4}$$

We can normalize misclassification cost as follows:

$$Normalized\ Misclassification\ Cost = (FP + \alpha(FN))c/nClasses \tag{5}$$

where $nClasses$ is number of classes in given version. It is always recommended to have defect prediction model that maximizes the effectiveness and minimizes misclassification cost. As $nClasses$ and c are constants for a project, minimizing the misclassification cost is same as minimizing normalized misclassification cost. And minimizing normalized misclassification is same as minimizing $(FP + \alpha(FN))/nclasses$. Hence, c can be taken to be 1 for these kinds of optimization problems. Now, we formulate our first multi-objective defect prediction problem with the following contrasting objectives.

– ***maximize recall and minimize normalized misclassification cost.***

We take effectiveness of model to be recall, misclassification cost as normalized misclassification cost where

$$Recall = \frac{TP}{TP + FN} \tag{6}$$

$$Normalized\ Misclassification\ Cost = \frac{FP + \alpha \cdot FN}{nClasses} \tag{7}$$

4.2 Optimize cost of QA activities and effectiveness

In cross-version defect prediction, the defect-prone files of the current version will be predicted by making use of prediction models built using previous version data. The project team performs QA activities on the files which are predicted defective. The predicted defective files involve both TP and FP. Hence, the cost incurred by project team is cost of QA activities on TP and FP files. The cost of QA activities is proportional to LOC of TP and FP files. Thus, cost is measured in terms of lines of code and effectiveness in terms of recall. The main motive behind these objectives is to find set of models such that they minimize cost of reviewing/testing files while achieving best possible effectiveness. And among these models, the most effective model can be selected based on the cost borne by project team.
We formulate second multi-objective defect prediction problem with the following contrasting objectives.

– ***maximize recall and minimize LOC cost.***

We take effectiveness of model to be recall, and it is the same as defined in Eq. 6. We believe that the cost of QA activities of defect-prone class(c_i) is proportional to number of lines of $c_i(LOC(c_i))$ and this is also confirmed by Rahman et al. (2012). We can find LOC cost using the following equation,

$$LOC\ cost = \sum_{i=1}^{n} P(c_i) \cdot LOC(c_i) \tag{8}$$

where $P(c_i)$ denotes whether i^{th} class, c_i, is defect prone or not. $P(c_i)$ is set to 1 if the predicted probability $p(c_i) > 0.5$, otherwise it is set to 0.

5 Proposed approach

In this section, we illustrate our approach to build multi-objective prediction models. Let $C = \{c_1, c_2, \ldots, c_n\}$ be classes of a given version V_k of a project P with each class having 'm' attributes. So training data take form of $n \times m$ matrix. A matrix entry $x_{i,j}$ denotes the value of j^{th} attribute for i^{th} class. The mathematical formulation of logistic regression is given as,

$$p(c_i) = \frac{e^{w_0 + w_1 x_{i,1} + w_2 x_{i,2} + \cdots + w_m x_{i,m}}}{1 + e^{w_0 + w_1 x_{i,1} + w_2 x_{i,2} + \cdots + w_m x_{i,m}}} \tag{9}$$

where $p(c_i)$ denotes probability of i^{th} class being defective, while the set of scalars
$W = \{w_0, w_1, w_2, \ldots, w_m\}$ represents the coefficients for the attributes $\{x_{i,1}, \ldots, x_{i,m}\}$. The objective of traditional logistic regression approach is to find out a set of coefficients $W = \{w_0, w_1, w_2, \ldots, w_m\}$ that minimize the prediction error. It is usually done with the help of gradient descent algorithm. This model is built using training data, and it is used to predict defective classes in the test data. A class c_i is predicted to be defective if $p(c_i) > 0.5$, based on the coefficients found during training process and the metrics $\{x_{i,1}, x_{i,2}, \ldots, x_{i,m}\}$.

The goal of multi-objective problem stated above is to find out a set of coefficients W, which optimizes two objectives. As there are two contrasting objectives, we will get multiple models with different trade-offs between two objectives. This problem of multi-objective optimization can be solved by using genetic algorithm.

Now we briefly describe some basic concepts used to solve multi-objective optimization problem. The definitions are presented here for the sake of continuity. The set of all possible values that can be taken by solutions is defined as the *feasible region*. In our case, the set of values that can be taken by the coefficients W forms the feasible region, which is the set of real numbers, as there are no constraints on the values that coefficients can take.

In multi-objective optimization problems, concepts of *Pareto dominance* and *Pareto optimality* are used to define the optimality of solutions (Coello et al. 2007). These terms are defined for two contrasting objectives.

A solution x dominates another solution y (also written $x <_p y$) if and only if the values of the objective functions satisfy the following conditions:

$$\begin{aligned} &objective1(x) \leq objective1(y)\ and \\ &\quad objective2(x) > objective2(y) \\ &\quad\quad or \\ &objective1(x) < objective1(y)\ and \\ &\quad objective2(x) \geq objective2(y) \end{aligned}$$

Let us assume that objective1 is the cost (LOC cost or misclassification cost) and objective2 is effectiveness (recall) of a prediction model. In simple words, above definition indicates that *x is better than y* if and only if, at the same level of cost x achieves greater effectiveness than that of y, or at the same level of effectiveness x incurs lower cost than that of y. A solution x is Pareto optimal if and only if it is not dominated by any other solution in the feasible region. The set of solutions (coefficient vectors W_i) that are not dominated by any other solutions is said to form *Pareto optimal set*, while the corresponding objective vectors, cost and effectiveness values of the solution set W, said to form *Pareto frontier*.

The final set of solutions on the Pareto frontier gives different cost-effectiveness trade-offs. It is up to software project team to decide upon how much amount of time (cost) they can spend and then choose appropriate model. For example, with one month away from the release, owing to shortage of time, the software project manager typically needs to plan the cost of quality assurance activities on topmost $n\%$ of defective components. The value of n can vary based on quality requirements, cost borne by project and organization. In this case, there is a need of having a model which can provide multiple solutions with different cost-effectiveness trade-offs. So multi-objective approach presented here can be of great help in similar situations.

To search for coefficient vectors in the solution space, we have used a multi-objective genetic algorithm (MOGA) presented by Goldberg (2006). In general terms, a genetic algorithm works as follows:

- It starts with random set of solutions called *population* of size p. Each individual in the population is known as *chromosome*.
- The population evolves through set of iterations, known as *generations*.
- From the given generation, new population called *offspring* is created using *crossover* and *mutation* operations. Crossover operator combines two individuals to generate new offspring. Mutation operator modifies the internal structure of an individual.
- A fitness function is applied on each individual of the current population to find the values of objective functions. From the current population, fittest set of p individuals are selected to be part of next generation using a selection operator. The fittest set of individuals are selected based on their objective values. The selection process follows the concepts of Pareto dominance and crowding distance. This is done to keep the size of population as constant in each generation.
- This process is repeated until termination criteria are met.

The intuition behind the genetic algorithm is that at the end of every generation only the fittest set of individuals make it to the next generation. So there is some improvement in the solution at the end of every generation. After many generations, the population approaches to an ideal solution or approximately ideal solution.

In our implementation, we have used NSGA-II proposed by Deb et al. (2000). For multi-objective logistic regression used in our study, one chromosome represents the coefficient vector $W = \{w_0, w_1, w_2, \ldots, w_m\}$, which forms one logistic regression model. Initially, a process starts with a population size of say p. For each model, the fitness function computes the values of objectives. Based on the definition of Pareto optimality and crowding distance, the best p set of coefficient vectors are selected for the next generation. The process of selecting best chromosomes depends on implementation of multi-objective genetic algorithm. NSGA-II uses fast non-dominated sorting algorithm and the concept of crowding distance for selection process. Complete explanation of the selection process, used in NSGA-II, is beyond the scope of this paper. One can refer to the original paper by Deb et al. (2000). The process of generating new population and selection of fittest set of individuals repeats in each iteration, until optimal set of coefficients are found or maximum number of generations are reached. At the end of training process, we get optimal set of coefficients, i.e., logistic regression models.

5.1 Data preprocessing

We describe the steps required to build and evaluate the prediction models built using single-objective and multi-objective algorithms. We preprocess the data of training version and testing version using data standardization. We have used CK metrics (Chidamber and Kemerer 1994) and lines of code (LOC) as predictor metrics. Since the values of different metrics have different ranges, metrics are standardized to reduce the heterogeneity. Mathematically, a metric m is normalized as follows:

$$m_z(i, c_j, V_k, P) = \frac{m(i, c_j, V_k, P) - \mu(i, V_k, P)}{\sigma(i, V_k, P)} \tag{10}$$

where $m(i, c_j, V_k, P)$ is the value of i^{th} metric computed on class c_j of version V_k of project P. Mean $\mu(i, V_k, P)$ and standard deviation $\sigma(i, V_k, P)$ have been calculated on all the classes of version V_k of the project P for the i^{th} metric. $m_z(i, c_j, V_k, P)$ is the normalized value of i^{th} metric. We apply this normalization on data of both the training and testing versions, while building and predicting the use of any modeling technique.

5.2 Training process

1. We train MOLR for our first multi-objective optimization problem as mentioned above on normalized data of the training version V_{k-1}. At the end of training, we get multiple MOLR models and four single-objective models. From the final set of MOLR models, we find out the best model. The best model is the one which is closest to the

ideal model. The ideal model has misclassification cost = 0 and recall = 1. We find the closest model with help of Euclidean distance measure as shown in Algorithm 1. If more than one models are equidistant to the ideal model, then we chose the model with least misclassification cost.

2. We train MOLR for our second problem as mentioned above and single-objective models on normalized data of the training version V_{k-1}. At the end of training, we get multiple MOLR models and four single-objective models. We retain all the MOLR models and apply it on the data of the test version.

Algorithm 1 Algorithm to select best MOLR model among several models of final population produced by genetic algorithm.

```
function FINDBESTMODEL($X, train\_data, \alpha$)
    /* $X$ is a $p \times (m+1)$ matrix which represents set of coefficient vectors obtained at the end of execution of multi-objective genetic algorithm. Here $p$ is the number of coefficient vectors in X (final population) and $m$ is the number of predictor metrics (6 CK metrics and LOC).*/
    /* train_data represents $n \times m$ matrix denoting predictor metrics values for $n$ classes in the training data.*/
    /* $\alpha$ is the cost factor.*/

    $Y$ = CALCULATEMISCLASSCOSTRECALL($X, train\_data, \alpha$)
    /* calculateMisclassCostRecall function will find out misclassification cost and recall for each of the coefficient vectors in $X$ based on the current training data and the cost factor. $Y$ is $p \times 2$ matrix.*/

    /* Initializing $min\_distance$, $max\_recall$, $min\_misclass\_cost$ variables to find out best model which is closest to ideal model with misclassification cost 0 and recall 1. */
    $best\_model\_index \leftarrow (-1)$
    $min\_dist \leftarrow \infty$
    $min\_misclass\_cost \leftarrow \infty$
    $max\_recall \leftarrow 0$

    for i=0 to p do
        $dist \leftarrow$ SQRT($(Y[i,1])^2 + (1 - Y[i,2])^2$)
        if $dist < min\_dist$ then
            $best\_model\_index \leftarrow i$
            $min\_dist \leftarrow dist$
            $min\_misclass\_cost \leftarrow Y[i,1]$
            $max\_recall \leftarrow Y[i,2]$

        else if $dist = min\_dist$ & $Y[i,1] < min\_misclass\_cost$ then
            /* Choosing the model with lesser misclassification cost in case the distance is same as current minimum distance */
            $best\_model\_index \leftarrow i$
            $min\_dist \leftarrow dist$
            $min\_misclass\_cost \leftarrow Y[i,1]$
            $max\_recall \leftarrow Y[i,2]$
        end if
    end for

    $best\_model \leftarrow X[best\_model\_index]$
    return $best\_model$
end function
```

5.3 Testing process

We illustrate testing process for both of our problems in this subsection.

5.3.1 Testing process for M1

The best possible MOLR model and other four single-objective models are applied on the data of test version. We compare all the single objective algorithms with MOLR in terms of misclassification cost, recall and F-measure.

Misclassification cost changes based on the value of cost factor α. So, for both MOLR and single-objective algorithms, misclassification cost is calculated each time cost factor changes. For any model (single-objective or MOLR model) misclassification, cost and recall can be calculated as per Eqs. 6 and 7. As recall is one of the objectives in MOLR, its value changes based on cost factor. F-measure is the harmonic mean of recall and precision. It denotes the balance achieved by the model between recall and precision values. This, in turn, shows how a model achieves the balance between false negatives and false positives. This is also a useful measure to find out effectiveness of the prediction model. For MOLR, F-measure changes as cost factor value changes, recall being one of the objectives of MOLR. For single-objective models, it remains the same. F-measure can be calculated using the following equation.

$$F - measure = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{11}$$

As explained earlier, after choosing the best MOLR model from the training process we apply it on the data of test version. The evaluation measures (recall, misclassification cost and F-measure) are calculated for the MOLR and single-objective models.

We perform two-tailed Wilcoxon signed-rank test for each pair of results between MOLR and other single-objective algorithms to determine whether the following null hypothesis can be rejected.

– H_{01}: There is no significant difference between evaluation measures of MOLR and single-objective models.

This comparison is different for all four single-objective algorithms and for different cost factors. We have conducted experiments with different cost factors—5, 10, 15, 20. One can choose appropriate cost factor as required by project, company and past experiences. For all the tests, the significance level is assumed to be 0.05, i.e., probability of rejecting the null hypothesis is 5%, when it should not be rejected.

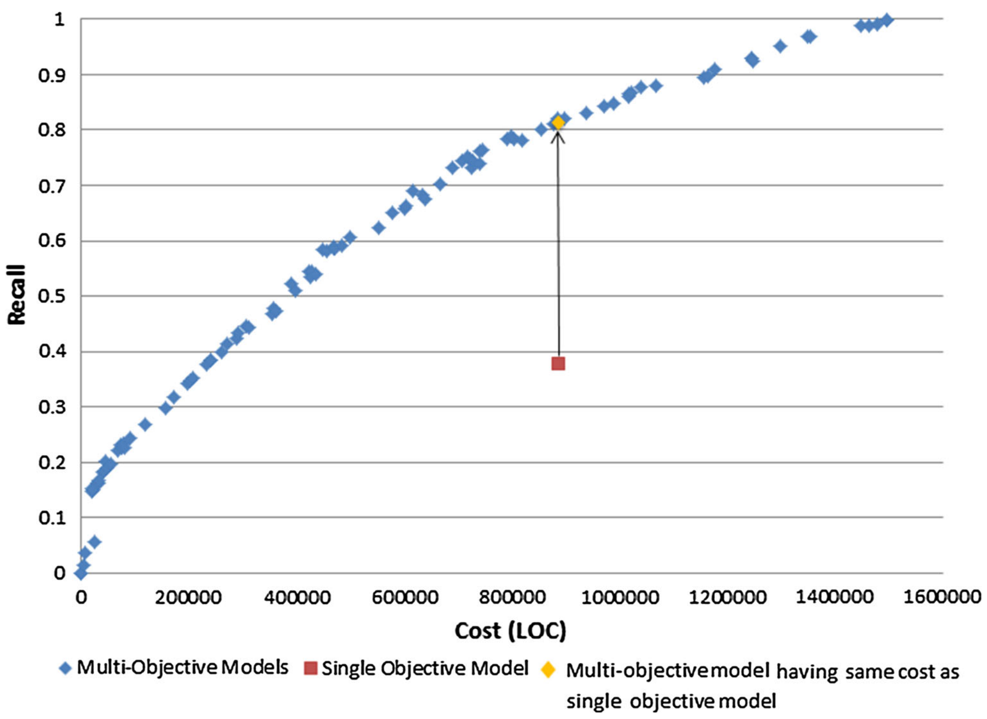

Fig. 1 Example showing selection of multi-objective objective model having same cost as single-objective model

5.3.2 Testing process for M2

From the training process, we get multiple models for MOLR. In this approach, we apply all the training models on the data of test version. We compare all the single-objective algorithms with multi-objective algorithm in terms of LOC cost and effectiveness. LOC cost and effectiveness can be found using Eqs. 6 and 8. After complete training and testing process, we plot LOC cost and effectiveness obtained from data of testing version on the same graph for comparison purpose.

From the definition of Pareto dominance described in the previous section, for each single-objective model, we try to find out one MOLR model that has the same or lesser LOC cost than that of the single-objective model. In particular, we try to compare single- and multi-objective models in terms of effectiveness at the same LOC cost that we have to spend with the single-objective model. Figure 1 depicts the process of selecting multi-objective model corresponding to single-objective model, having same LOC cost as that of single objective model. For the situation shown in Fig. 1, multi-objective model has more effectiveness than single-objective model.

Due to inherent randomness of NSGA-II algorithm, we may not get a multi-objective model having the same LOC cost as that of the single objective model. So we try to find nearest multi-objective model having lesser cost than single-objective model. We find out how multi-objective model performs compared to the single-objective model by spending lesser LOC cost than that to be spent with use of single-objective model. The results show that multi-objective model is more effective even at lesser LOC cost. One example for this situation is depicted in Fig. 2. Here multi-objective model has more effectiveness than single-objective model even at lesser LOC cost than that of the single-objective model.

For M2, we compare single-objective and multi-objective models in terms of effectiveness. As explained above, we find out MOLR model having same or lesser LOC cost as compared to given single-objective model. After finding such a MOLR model for each of the single objective algorithm, we compare recall values of both the models. We find out how much effective the MOLR model is compared to single objective model at the same LOC cost.

To prove significance of the results statistically, we compare recall values of MOLR and single-objective model using two-tailed Wilcoxon paired test to determine whether the following null hypotheses could be rejected.

- H_{02}: There is no significant difference between recall values of MOLR and single-objective predictors for cross-version defect prediction at the same LOC cost.

This comparison is different for all four single-objective algorithms. In other words, we find out closest multi-objective model with respect to each single-objective models,

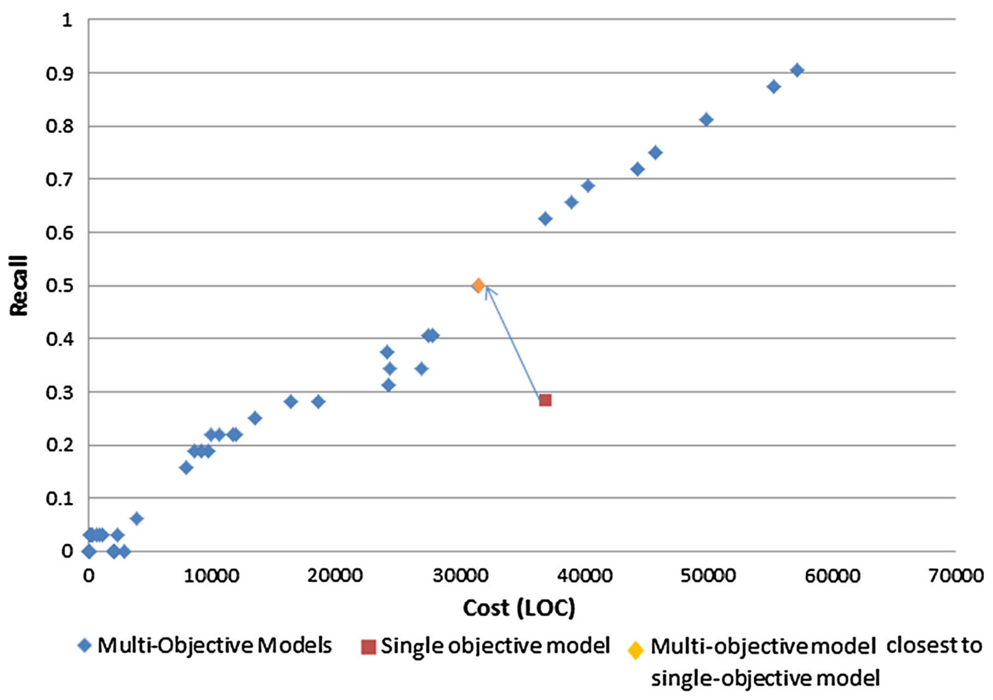

Fig. 2 Example showing selection of closest multi-objective objective model having lesser cost than single objective model

using the process described above. For all tests, the significance level is assumed to be 0.05, i.e., probability of rejecting the null hypothesis is 5%, when it should not be rejected.

5.4 Implementation settings

MOLR is implemented using MATLAB Global Optimization Toolbox (MATLAB 2015). Other single-objective algorithms are also implemented in MATLAB. The implementation settings are kept the same for all experiments and for both the approaches, so that we can compare results in the same conditions. Otherwise, one can choose appropriate parameters according to project requirement.

The implementation details about single-objective algorithms are as follows:

- **Logistic Regression:** *glmfit* function has been used for logistic regression implementation. The *distribution* parameter has been set to *binomial* and *link* parameter has been set to *logit*.
- **Naïve Bayes:** *NaïveBayes.fit* function has been used for naïve Bayes implementation. Implementation of naïve Bayes function requires unnormalized data as input. Rest of the functions requires normalized data as described in data preprocessing process (Sect. 5.1).
- **Decision Tree:** *classregtree* function has been used for decision tree implementation. The *method* parameter has been set to *classification*.
- **Random Forest:** *TreeBagger* function has been used for random forest implementation. The *method* parameter has been set to *classification*. Number of trees (*ntrees* parameter) has been set to 100. We experimented with different number of trees and chose the one having the least variation in results.

gamultiobj function has been used for NSGA-II implementation. Selection of parameters for NSGA-II implementation has been done with reference to some of the previous studies like (Canfora et al. 2013, 2015; Coello et al. 2007; Krall et al. 2015) and experimentation. For the parameters chosen by experimentation, we have taken *spread* value, defined by Deb, as the evaluation criteria (Deb 2001). Spread value gives an indication of how well the solutions are spaced on the final Pareto front. The greater the spread value, the better the solutions on Pareto front. Krall et al. (2015) have also used spread as one of the parameters to evaluate the multi-objective algorithm proposed by them. Spread value has been calculated on the Pareto fronts obtained after each run of NGSA-II on the data of training version. The parameters that are used for NSGA-II implementation are described in Table 3.

For the first multi-objective defect prediction problem M1, NSGA-II algorithm has been executed 30 times during training process. This is done to account for the inherent randomness of GAs. After each run, we choose a best model as described with algorithm 1. At the end of 30 runs, we take best model among all 30 models obtained during each run. This is also done with help of Algorithm 1, i.e., final logistic

Table 3 Multi-objective genetic algorithm parameter configuration

Parameter	Value
Population size	200
Initial population	Random values between [-10,10]
Number of generations	500
Crossover function	Arithmetic crossover with crossover probability $p_c = 0.9$
Mutation function	Uniform Mutation with probability $p_m = 1/n$ where n is size of chromosomes
Stopping criteria	If the average Pareto spread is $<10^{-6}$ in the subsequent 50 generations, then the execution of GA is terminated

regression model is the one which is closest to ideal model (0 misclassification cost, 1 recall) among all the 30 models obtained in 30 runs. Final logistic regression model obtained from the training process is applied on the data of the testing version.

For our second problem, NSGA-II algorithm has been executed 31 times during training process. The reason behind choosing an odd number (31 runs) is to choose coefficient vectors with respect to median Pareto front. We chose Pareto front with median spread value as the final solution, i.e., the coefficient vectors corresponding to median spread value among these 31 runs. These coefficient vectors represent final logistic regression models obtained from the training process, and these models will be applied on the data of the testing version.

6 Results and discussion

We discuss the results obtained for both the multi-objective defect prediction models and four single-objective approaches in this section. We have conducted 30 experiments in 11 projects among 41 versions. We have taken projects which had at least 3 versions to put more strength to cross-version study, with the hope of achieving more generalizable results. Each experiment will be referred as *'project_name train_version-test_version.'* For example, 'Ant 1.3–1.4' denotes that data of version 1.3 of Ant project are used as training data and data of version 1.4 are used as testing data. We use this notation to denote each experiment throughout this section.

RQ_1: How does the proposed M1 perform as compared to traditional single-objective defect prediction models in the cross-version defect prediction?

In this section, we discuss about the results obtained from our experiments and try to answer research question RQ_1. As explained earlier, we compare the performance in terms of misclassification cost, recall and F-measure.

We have built multi-objective defect prediction model MOLR with cost factors being 5, 10, 15, 20, and the results are presented in Tables 4, 5, 6 and 7, respectively. At the end of training phase of MOLR, we will have the best possible MOLR model that has (misclassification cost, recall) closest to (0,1) as explained in Sect. 5.2. For each of the experiments, misclassification cost and recall values of MOLR and four single-objective prediction models are recorded. For each of the experiment, we have boldfaced misclassification value if it is the lowest among all other misclassification values. Similarly, we have boldfaced recall value if it is the largest among all other recall values. For example, misclassification cost is the lowest and recall is the highest for MOLR for the experiment 'Ant 1.3–1.4' as shown in Table 4. And in this scenario, we can always claim that MOLR is preferred to other four single objective models.

From Table 4, we can observe that MOLR achieves lesser misclassification cost and higher recall as compared to other single objective algorithms in most of the experiments. Overall, in 20 out of 30 experiments, MOLR achieve better performance in terms of misclassification cost and recall combined. As cost factor increases, the performance of MOLR keeps improving. We can observe that, for the cost factors 10, 15 and 20, MOLR model dominates 24, 25 and 28 out of 30 experiments, respectively, in terms of both evaluation measures (misclassification cost and recall).

With increase in cost factors, FN file is penalized more than FP file. Minimizing misclassification cost takes care of controlling false-negative files in prediction, and hence, recall improves. Hence, MOLR performs better than other single-objective models with increase in cost factor value.

There are only 2 experiments, namely Ivy 1.1–1.4 and Jedit 4.2–4.3, where MOLR is not able to achieve least misclassification cost. One of the reasons can be that the test versions of both experiments have very few classes which are actually defective.

For Camel 1.2–1.4, Ivy 1.1–1.4, Poi 1.5–2.0 And Velocity 1.5–1.6 experiments, recall values achieved by MOLR are 1.

Table 4 Misclassification cost and recall comparison for $\alpha = 5$

Project	Train	Test	MOLR		LR		NB		DT		RF	
			MC cost	Recall	MC cost	Recall	MC cost	Recall	MC cost	Recall	MC cost	Recall
Ant	1.3	1.4	**0.9551**	**0.3000**	1.0169	0.1500	1.0674	0.1250	0.9719	0.2250	1.0843	0.1250
	1.4	1.5	0.6894	**0.9063**	0.4676	0.1563	0.7884	0.6563	**0.3686**	0.5625	0.5256	0.1250
	1.5	1.6	**0.8262**	**0.4239**	1.1225	0.1522	1.0342	0.2500	1.1909	0.1087	1.0712	0.1957
	1.6	1.7	**0.5557**	**0.8855**	0.6765	0.4398	0.8040	0.3735	0.6832	0.5121	0.6268	0.5121
Camel	1.0	1.2	**1.4227**	**0.2824**	1.7516	0.0139	1.5526	0.1574	1.7204	0.0370	1.7549	0.0139
	1.2	1.4	0.7924	**1.0000**	0.7397	0.1517	0.7144	0.2345	0.7592	0.3931	**0.7099**	0.4483
	1.4	1.6	**0.6705**	**0.7234**	0.9098	0.0798	0.8860	0.1489	0.8642	0.2394	0.8342	0.1702
Ivy	1.1	1.4	0.9253	**1.0000**	0.5560	0.8125	**0.4398**	0.3125	0.6805	0.6875	0.6141	0.7500
	1.4	2.0	**0.3977**	**0.7750**	0.4858	0.1750	0.5341	0.1250	0.5483	0.1250	0.5653	0.0250
Jedit	3.2	4.0	**0.5784**	**0.8667**	0.7745	0.4667	1.0523	0.2000	0.6928	0.5333	0.6569	0.5733
	4.0	4.1	**0.5064**	**0.8734**	0.8237	0.3671	1.0417	0.2152	0.6058	0.6329	0.6795	0.5063
	4.1	4.2	0.4823	**0.9375**	0.3924	0.5417	0.5259	0.3125	**0.3760**	0.8542	0.4005	0.6042
	4.2	4.3	0.3862	**0.6364**	**0.1077**	0.4546	0.1240	0.2727	0.1728	0.3636	0.1280	0.3636
Log4j	1.0	1.1	**0.5780**	**0.7568**	0.9083	0.4865	0.9908	0.4595	0.6147	0.6757	0.7798	0.5676
	1.1	1.2	**1.3854**	**0.7090**	3.5659	0.2275	3.2488	0.2963	3.4732	0.2487	3.7122	0.1958
Lucene	2.0	2.2	**0.6559**	**0.9028**	1.7692	0.4236	2.2915	0.2292	1.7692	0.4306	1.5830	0.4861
	2.2	2.4	**0.4412**	**0.9803**	0.7265	0.8670	2.2353	0.2906	1.1676	0.6897	0.9147	0.7636
Poi	1.5	2.0	0.8599	**1.0000**	0.8567	0.5946	**0.6879**	0.3243	0.8280	0.7838	0.8280	0.8919
	2.0	2.5	**2.5403**	**0.2581**	3.1351	0.0282	2.8649	0.1169	2.9558	0.0927	2.9870	0.0766
	2.5	3.0	**0.3778**	**0.9893**	0.6538	0.8577	2.5317	0.2135	1.1380	0.7011	0.8869	0.7829
Synapse	1.0	1.1	**0.9505**	0.4167	1.3514	0.0000	1.0991	**0.4333**	1.1486	0.1667	1.2072	0.1167
	1.1	1.2	**0.7344**	**0.7791**	1.3594	0.2093	1.5352	0.1279	1.1836	0.3488	1.2930	0.2558
Velocity	1.4	1.5	**0.5701**	**0.9225**	0.6729	0.8873	1.4112	0.6620	0.8037	0.8451	0.6121	0.9085
	1.5	1.6	0.6332	**1.0000**	**0.5721**	0.9615	1.2882	0.3205	0.6900	0.8590	0.5852	0.8718
Xalan	2.4	2.5	**1.2304**	**0.5736**	2.2864	0.0543	2.1644	0.1137	2.1270	0.1344	2.2379	0.0801
	2.5	2.6	**0.5356**	**0.9903**	1.4147	0.4453	2.0362	0.1484	0.8429	0.7348	1.1119	0.6034
	2.6	2.7	**0.7701**	**0.8463**	3.3333	0.3252	3.6084	0.2695	2.6095	0.4733	2.7393	0.4454
Xerces	1.0	1.2	0.8295	**0.9718**	0.7614	0.4085	0.8932	0.4507	**0.7159**	0.2535	0.7591	0.3380
	1.2	1.3	0.7550	**0.6232**	0.7174	0.0580	**0.6313**	0.3188	0.7638	0.1739	0.7395	0.1449
	1.3	1.4	**1.9711**	**0.4783**	3.4201	0.0801	3.1207	0.1625	3.1667	0.1510	3.4456	0.0732
Average				**0.7603**		**0.3625**		**0.2774**		**0.4346**		**0.4005**

This means MOLR is able to identify all the defective classes accurately.

F-measure is the harmonic mean of recall and precision, and it explains how the model is balanced against false positives and false negatives. For each value of cost factor—5, 10, 15, 20, F-measure achieved by the respective models is reported in Table 8. And also F-measure of four single-objective prediction models is also shown in the same table. F-measure of MOLR models is relatively better than other single-objective algorithms.

For Xalan 2.6–2.7 experiment, F-measure is greater than 0.9 for all values of cost factors. Unlike other experiments, F-measure does not always increase with increase in cost factor. We can observe that on an average, for different values of cost factor, MOLR achieves better F-measure as compared to four single-objective prediction models.

Wilcoxon signed test is applied to test whether MOLR model is significantly better than single-objective optimization model. If p value is <0.05, we can conclude that MOLR is significantly better than the single-objective prediction model. For each evaluation measure (recall, misclassification cost, F-measure) and cost factor value (5, 10, 15, 20), the MOLR model is tested to check whether it is significantly better than the four single-objective prediction models. And p values are shown in Table 9.

From the results, one can observe that p value is <0.05 for all evaluation measures and cost factors. In fact, for comparison in terms of recall value, p value is zero for all the

Table 5 Misclassification cost and recall comparison for $\alpha = 10$

Project	Train	Test	MOLR		LR		NB		DT		RF	
			MC cost	Recall	MC cost	Recall	MC cost	Recall	MC cost	Recall	MC cost	Recall
Ant	1.3	1.4	**1.4326**	**0.5000**	1.9719	0.1500	2.0506	0.1250	1.8427	0.2250	2.0506	0.1250
	1.4	1.5	0.7509	**0.9688**	0.9283	0.1563	0.9761	0.6563	**0.6075**	0.5625	0.9727	0.1563
	1.5	1.6	**0.7664**	**0.8261**	2.2336	0.1522	2.0171	0.2500	2.3590	0.1087	2.1538	0.1848
	1.6	1.7	**0.6349**	**0.9157**	1.3007	0.4398	1.5020	0.3735	1.2268	0.5121	1.0966	0.5482
Camel	1.0	1.2	**2.5707**	**0.3287**	3.5033	0.0139	3.0493	0.1574	3.4309	0.0370	3.4885	0.0185
	1.2	1.4	**0.7947**	**1.0000**	1.4450	0.1517	1.3509	0.2345	1.2638	0.3931	1.0940	0.4621
	1.4	1.6	**0.6984**	**0.9362**	1.8062	0.0798	1.7150	0.1489	1.6052	0.2394	1.6435	0.1702
Ivy	1.1	1.4	0.8755	**1.0000**	**0.6183**	0.8125	0.6680	0.3125	0.7842	0.6875	0.7261	0.8125
	1.4	2.0	**0.4773**	**0.8000**	0.9545	0.1750	1.0313	0.1250	1.0455	0.1250	1.0938	0.0500
Jedit	3.2	4.0	**0.6993**	**0.8800**	1.4281	0.4667	2.0327	0.2000	1.2647	0.5333	1.1078	0.6133
	4.0	4.1	**0.7115**	**0.8861**	1.6250	0.3671	2.0353	0.2152	1.0705	0.6329	1.2468	0.5317
	4.1	4.2	0.5995	**0.9583**	0.6921	0.5417	0.9755	0.3125	**0.4714**	0.8542	0.5886	0.6458
	4.2	4.3	0.4817	**0.6364**	**0.1687**	0.4546	0.2053	0.2727	0.2439	0.3636	0.1850	0.4546
Log4j	1.0	1.1	**1.0183**	**0.7568**	1.7798	0.4865	1.9083	0.4595	1.1651	0.6757	1.5138	0.5676
	1.1	1.2	**2.4341**	**0.7407**	7.1268	0.2275	6.4927	0.2963	6.9366	0.2487	7.4683	0.1905
Lucene	2.0	2.2	**0.9393**	**0.9028**	3.4494	0.4236	4.5385	0.2292	3.4292	0.4306	3.2186	0.4653
	2.2	2.4	**0.5294**	**0.9754**	1.1235	0.8670	4.3529	0.2906	2.0941	0.6897	1.6794	0.7537
Poi	1.5	2.0	0.8822	**1.0000**	1.0955	0.5946	1.0860	0.3243	0.9554	0.7838	**0.8758**	0.8919
	2.0	2.5	**4.3818**	**0.3427**	6.2649	0.0282	5.7091	0.1169	5.8779	0.0927	6.0104	0.0686
	2.5	3.0	**0.4887**	**0.9715**	1.1063	0.8577	5.0317	0.2135	2.0882	0.7011	1.6176	0.7758
Synapse	1.0	1.1	**1.6757**	**0.4500**	2.7027	0.0000	1.8649	0.4333	2.2748	0.1667	2.3604	0.1333
	1.1	1.2	**0.9453**	**0.8488**	2.6875	0.2093	3.0000	0.1279	2.2773	0.3488	2.5078	0.2674
Velocity	1.4	1.5	**0.5187**	**0.9718**	1.0467	0.8873	2.5327	0.6620	1.3178	0.8451	0.8178	0.9225
	1.5	1.6	0.6419	**1.0000**	**0.6376**	0.9615	2.4454	0.3205	0.9301	0.8590	0.7860	0.8718
Xalan	2.4	2.5	**1.9738**	**0.6408**	4.5654	0.0543	4.3001	0.1137	4.2130	0.1344	4.4060	0.0904
	2.5	2.6	**0.5458**	**0.9951**	2.7028	0.4453	4.0136	0.1484	1.4588	0.7348	1.8000	0.6545
	2.6	2.7	**0.3311**	**0.9677**	6.6667	0.3252	7.2167	0.2695	5.2112	0.4733	5.5006	0.4432
Xerces	1.0	1.2	**0.8500**	**0.9718**	1.2386	0.4085	1.3364	0.4507	1.3182	0.2535	1.1818	0.4085
	1.2	1.3	**1.0088**	**0.6812**	1.4349	0.0580	1.1501	0.3188	1.3929	0.1739	1.4062	0.1304
	1.3	1.4	**3.1412**	**0.5835**	6.8384	0.0801	6.2330	0.1625	6.3214	0.1510	6.8912	0.0732
Average				**0.8146**		**0.3625**		**0.2774**		**0.4346**		**0.4161**

experiments. Even for misclassification cost, p value is zero for the cost factor greater than or equal to 10. This shows that MOLR is working quite dominantly. As p value is <0.05 for all evaluation measures and cost factors, we can easily reject null hypothesis H_{01}. This confirms domination of MOLR over single-objective prediction models.

RQ_2 How does the proposed M2 perform as compared to traditional single-objective defect prediction models in the cross-version defect prediction?

In this subsection, we discuss about results of our experiments to answer our second research question RQ_2. As explained earlier, we compare the performance of MOLR and four single objective prediction models in terms of cost and recall.

At the end of the training phase of building multi-objective defect prediction model, there are several models with varying cost and effectiveness. As described in Sect. 5.3.2, for each single objective prediction model we select the closest multi-objective model having the same or lesser LOC cost. We find out how effective (in terms of recall) the multi-objective model is compared to single objective model at the same or lesser LOC cost.

The comparative results of single-objective logistic regression model and the corresponding multi-objective prediction model are shown in Table 10. We also report LOC cost difference along with recall difference of two models since we compare effectiveness at the same or lesser LOC cost. From Table 10, it can be observed that, for the experiment Ant 1.6–

Table 6 Misclassification cost and recall comparison for $\alpha = 15$

Project	Train	Test	MOLR		LR		NB		DT		RF	
			MC cost	Recall	MC cost	Recall	MC cost	Recall	MC cost	Recall	MC cost	Recall
Ant	1.3	1.4	**2.0787**	**0.4750**	2.9270	0.1500	3.0337	0.1250	2.7135	0.2250	3.1124	0.1000
	1.4	1.5	**0.8464**	**0.9375**	1.3891	0.1563	1.1638	0.6563	0.8464	0.5625	1.4437	0.1563
	1.5	1.6	**1.0399**	**0.8152**	3.3447	0.1522	3.0000	0.2500	3.5271	0.1087	3.2678	0.1739
	1.6	1.7	**0.8148**	**0.9036**	1.9248	0.4398	2.2000	0.3735	1.7705	0.5121	1.5960	0.5482
Camel	1.0	1.2	**3.6266**	**0.3565**	5.2549	0.0139	4.5461	0.1574	5.1414	0.0370	5.2566	0.0139
	1.2	1.4	**0.7936**	**1.0000**	2.1502	0.1517	1.9874	0.2345	1.7683	0.3931	1.6594	0.4207
	1.4	1.6	**0.7741**	**0.9628**	2.7026	0.0798	2.5440	0.1489	2.3461	0.2394	2.3648	0.2021
Ivy	1.1	1.4	0.8589	**1.0000**	**0.6805**	0.8125	0.8963	0.3125	0.8880	0.6875	0.8589	0.7500
	1.4	2.0	**0.6278**	**0.8000**	1.4233	0.1750	1.5284	0.1250	1.5426	0.1250	1.6364	0.0500
Jedit	3.2	4.0	**0.9183**	**0.8667**	2.0817	0.4667	3.0131	0.2000	1.8366	0.5333	1.5196	0.6267
	4.0	4.1	**0.8301**	**0.9241**	2.4263	0.3671	3.0288	0.2152	1.5353	0.6329	1.8397	0.5317
	4.1	4.2	0.5940	**0.9792**	0.9918	0.5417	1.4251	0.3125	**0.5668**	0.8542	0.8965	0.6042
	4.2	4.3	0.5346	**0.6364**	**0.2297**	0.4546	0.2866	0.2727	0.3150	0.3636	0.2663	0.3636
Log4j	1.0	1.1	**1.2569**	**0.8108**	2.6514	0.4865	2.8257	0.4595	1.7156	0.6757	2.1009	0.5946
	1.1	1.2	**4.5805**	**0.6720**	10.6878	0.2275	9.7366	0.2963	10.4000	0.2487	11.0537	0.2011
Lucene	2.0	2.2	**1.2834**	**0.8958**	5.1296	0.4236	6.7854	0.2292	5.0891	0.4306	4.6599	0.4792
	2.2	2.4	**0.5559**	**0.9803**	1.5206	0.8670	6.4706	0.2906	3.0206	0.6897	1.9147	0.8128
Poi	1.5	2.0	0.8822	**1.0000**	1.3344	0.5946	1.4841	0.3243	1.0828	0.7838	**0.8089**	0.9730
	2.0	2.5	**6.1195**	**0.3871**	9.3948	0.0282	8.5532	0.1169	8.8000	0.0927	8.9351	0.0766
	2.5	3.0	**0.5045**	**0.9822**	1.5588	0.8577	7.5317	0.2135	3.0385	0.7011	2.2285	0.7865
Synapse	1.0	1.1	**2.6126**	0.4000	4.0541	0.0000	2.6306	**0.4333**	3.4009	0.1667	3.5901	0.1167
	1.1	1.2	**0.5977**	**0.9884**	4.0156	0.2093	4.4648	0.1279	3.3711	0.3488	3.8477	0.2442
Velocity	1.4	1.5	**0.6028**	**0.9718**	1.4206	0.8873	3.6542	0.6620	1.8318	0.8451	1.1589	0.9155
	1.5	1.6	**0.6245**	**1.0000**	0.7031	0.9615	3.6026	0.3205	1.1703	0.8590	1.0087	0.8718
Xalan	2.4	2.5	**2.4620**	**0.7003**	6.8443	0.0543	6.4359	0.1137	6.2989	0.1344	6.6389	0.0853
	2.5	2.6	**0.8203**	**0.9586**	3.9910	0.4453	5.9910	0.1484	2.0746	0.7348	2.3480	0.6910
	2.6	2.7	**0.6062**	**0.9599**	10.0000	0.3252	10.8251	0.2695	7.8130	0.4733	8.1683	0.4488
Xerces	1.0	1.2	**0.8886**	**0.9155**	1.7159	0.4085	1.7795	0.4507	1.9205	0.2535	1.6318	0.4225
	1.2	1.3	**1.2450**	**0.6812**	2.1523	0.0580	1.6689	0.3188	2.0221	0.1739	2.0839	0.1449
	1.3	1.4	**4.5340**	**0.5973**	10.2568	0.0801	9.3452	0.1625	9.4762	0.1510	10.3078	0.0755
Average				**0.8186**		**0.3625**		**0.2774**		**0.4346**		**0.4160**

1.7, MOLR gained recall of 0.1446 by incurring lesser cost (−1050) than that of single-objective logistic regression.

The comparative results of multi-objective prediction model with single-objective prediction models, naïve Bayes classifier, decision tree, random forest, are shown in Tables 11, 12 and 13, respectively. We take a deeper look on each comparison table. For each comparison table, we discuss the gain in recall values achieved by MOLR and the special cases where MOLR did not achieve any gain compared to single-objective algorithm. From the tables, it can be seen that recall values of MOLR are better than single-objective algorithms for most of the cases.

The comparative results of MOLR and single-objective logistic regression are shown in Table 10. For 25 out of 30 experiments, MOLR is able to achieve better recall than single objective logistic regression. MOLR is able to achieve 1–65% gain in recall. In Xalan project, all 3 experiments achieved more than 40% gain in recall, with Xalan 2.6–2.7 experiment achieving highest recall gain of 65.14%. The LOC cost difference is highest for Xerces 1.3–1.4 experiment (LOC difference of 11412), but recall gain achieved by MOLR model is 63.39%.

There are five cases when recall value of MOLR is lesser than or equal to SOLR values (Ant 1.3–1.4, Ant 1.4–1.5, Ant 1.5–1.6, Jedit 4.2–4.3 and Synapse 1.0–1.1). In Synapse 1.0–1.1 experiment, SOLR achieves zero LOC cost and effectiveness. The reason for this can be that the train version Synapse-1.0 had very few defective classes (10% of 157

Table 7 Misclassification cost and recall comparison for $\alpha = 20$

Project	Train	Test	MOLR		LR		NB		DT		RF	
			MC cost	Recall	MC cost	Recall	MC cost	Recall	MC cost	Recall	MC cost	Recall
Ant	1.3	1.4	**2.2303**	**0.5750**	3.8820	0.1500	4.0169	0.1250	3.5843	0.2250	3.8989	0.1500
	1.4	1.5	**0.8498**	**0.9375**	1.8498	0.1563	1.3515	0.6563	1.0853	0.5625	1.7850	0.2188
	1.5	1.6	**1.2251**	**0.8261**	4.4558	0.1522	3.9829	0.2500	4.6952	0.1087	4.3476	0.1739
	1.6	1.7	**0.9879**	**0.8976**	2.5490	0.4398	2.8980	0.3735	2.3141	0.5121	2.1987	0.5241
Camel	1.0	1.2	**4.8059**	**0.3519**	7.0066	0.0139	6.0428	0.1574	6.8520	0.0370	7.0740	0.0046
	1.2	1.4	**0.7936**	**1.0000**	2.8555	0.1517	2.6239	0.2345	2.2729	0.3931	2.1021	0.4345
	1.4	1.6	**0.7627**	**0.9734**	3.5990	0.0798	3.3731	0.1489	3.0870	0.2394	3.2394	0.1755
Ivy	1.1	1.4	0.8797	**1.0000**	**0.7427**	0.8125	1.1245	0.3125	0.9917	0.6875	0.8423	0.8125
	1.4	2.0	**0.6165**	**0.8500**	1.8920	0.1750	2.0256	0.1250	2.0398	0.1250	2.2898	0.0000
Jedit	3.2	4.0	**0.9444**	**0.9333**	2.7353	0.4667	3.9935	0.2000	2.4085	0.5333	2.0523	0.6133
	4.0	4.1	**0.9840**	**0.9114**	3.2276	0.3671	4.0224	0.2152	2.0000	0.6329	2.3045	0.5570
	4.1	4.2	**0.6294**	**0.9792**	1.2916	0.5417	1.8747	0.3125	0.6621	0.8542	1.0599	0.6458
	4.2	4.3	0.6138	**0.6364**	**0.2907**	0.4546	0.3679	0.2727	0.3862	0.3636	0.3333	0.3636
Log4j	1.0	1.1	**1.2752**	**0.8649**	3.5229	0.4865	3.7431	0.4595	2.2661	0.6757	2.9725	0.5676
	1.1	1.2	**6.1902**	**0.6667**	14.2488	0.2275	12.9805	0.2963	13.8634	0.2487	14.8342	0.1958
Lucene	2.0	2.2	**1.2632**	**0.9236**	6.8097	0.4236	9.0324	0.2292	6.7490	0.4306	6.1741	0.4792
	2.2	2.4	**0.9029**	**0.9557**	1.9176	0.8670	8.5882	0.2906	3.9471	0.6897	3.0941	0.7586
Poi	1.5	2.0	**0.8822**	**1.0000**	1.5732	0.5946	1.8822	0.3243	1.2102	0.7838	1.0223	0.8919
	2.0	2.5	**2.2468**	**0.8468**	12.5247	0.0282	11.3974	0.1169	11.7221	0.0927	12.0130	0.0686
	2.5	3.0	**0.4593**	**0.9893**	2.0113	0.8577	10.0317	0.2135	3.9887	0.7011	2.8190	0.7936
Synapse	1.0	1.1	**3.2658**	**0.4333**	5.4054	0.0000	3.3964	0.4333	4.5270	0.1667	4.7928	0.1167
	1.1	1.2	**0.9375**	**0.9419**	5.3438	0.2093	5.9297	0.1279	4.4648	0.3488	4.9688	0.2674
Velocity	1.4	1.5	**0.7850**	**0.9648**	1.7944	0.8873	4.7757	0.6620	2.3458	0.8451	1.3411	0.9225
	1.5	1.6	**0.6288**	**1.0000**	0.7686	0.9615	4.7598	0.3205	1.4105	0.8590	1.2271	0.8718
Xalan	2.4	2.5	**2.4135**	**0.7830**	9.1233	0.0543	8.5716	0.1137	8.3848	0.1344	8.8406	0.0853
	2.5	2.6	**0.5480**	**0.9951**	5.2791	0.4453	7.9684	0.1484	2.6904	0.7348	3.0927	0.6886
	2.6	2.7	**0.8482**	**0.9577**	13.3333	0.3252	14.4334	0.2695	10.4147	0.4733	10.8691	0.4499
Xerces	1.0	1.2	**0.9341**	**0.9437**	2.1932	0.4085	2.2227	0.4507	2.5227	0.2535	1.8932	0.5070
	1.2	1.3	**1.1280**	**0.8551**	2.8698	0.0580	2.1876	0.3188	2.6512	0.1739	2.6556	0.1449
	1.3	1.4	**4.1395**	**0.7254**	13.6752	0.0801	12.4575	0.1625	12.6310	0.1510	13.8129	0.0709
Average				**0.8573**		**0.3625**		**0.2774**		**0.4346**		**0.4185**

classes) that gave poor performance when tested on Synapse version 1.1. The SOLR predictor classified all classes as non-defective, resulting in zero LOC cost and zero effectiveness. The loss in recall varies from 0 to 10% for these five cases, which is very less compared to gains we achieve for rest of the 25 experiments. For three projects, recall remains the same, but for only two project MOLR was not able to get better recall with lesser cost.

Average recall gain across all experiments is 22.77%. This shows how effectively MOLR model is able to predict defect-prone classes compared to SOLR.

The recall and LOC cost values for MOLR and naïve Bayes classifier are reported in Table 11. MOLR is able to outperform naïve Bayes in all 30 experiments. The gain in recall varies from 1 to 76%. In particular, there are 17 cases when MOLR is able to achieve more than 30% gain in recall compared to naïve Bayes classifier. In this case also, maximum cost difference is reported for Xerces 1.3–1.4 experiment (LOC difference of 13834). The gain in recall value for this experiment is 76.89%, which is maximum among all experiments. MOLR is able to achieve average recall gain of 32.65% across all experiments.

The recall and LOC cost values for MOLR and decision tree are reported in Table 12. In most of the cases, decision tree achieves good recall values. The decision tree is able to achieve more than 60% recall for ten experiments, but still MOLR achieves better recall values in most of the cases. There are five cases where the recall values of MOLR are

Table 8 F-measure comparison for different cost factor values

Project	Train	Test	F-measure							
			MOLR ($\alpha = 5$)	MOLR ($\alpha = 10$)	MOLR ($\alpha = 15$)	MOLR ($\alpha = 20$)	LR	NB	DT	RF
Ant	1.3	1.4	0.29268	0.34783	0.33333	0.38333	0.21053	0.16667	0.26866	0.15873
	1.4	1.5	0.23387	0.22711	0.21429	0.2214	0.25641	0.18341	0.40909	0.16
	1.5	1.6	0.5	0.54874	0.54152	0.54676	0.25455	0.34586	0.18182	0.31034
	1.6	1.7	0.46519	0.46697	0.43924	0.41913	0.52518	0.40391	0.47887	0.54313
Camel	1.0	1.2	0.33243	0.355	0.37288	0.36715	0.0274	0.23944	0.06957	0.02715
	1.2	1.4	0.29562	0.29502	0.29532	0.29532	0.22335	0.2753	0.26887	0.30303
	1.4	1.6	0.38256	0.38344	0.35806	0.36346	0.13889	0.20664	0.25568	0.26122
Ivy	1.1	1.4	0.12549	0.13169	0.13389	0.13115	0.17568	0.13889	0.13253	0.15385
	1.4	2.0	0.37349	0.4	0.36994	0.39766	0.26415	0.17241	0.15873	0.04444
Jedit	3.2	4.0	0.48689	0.49811	0.4797	0.41916	0.47619	0.26786	0.52632	0.54088
	4.0	4.1	0.53906	0.49822	0.45483	0.45283	0.50435	0.30631	0.57803	0.58824
	4.1	4.2	0.35294	0.31293	0.31544	0.30719	0.48148	0.32967	0.42708	0.44961
	4.2	4.3	0.07447	0.06512	0.06335	0.05833	0.25641	0.17143	0.12308	0.18605
Log-4j	1.0	1.1	0.6747	0.65116	0.60606	0.59259	0.61017	0.54839	0.72464	0.66667
	1.1	1.2	0.80723	0.8284	0.78154	0.77778	0.3691	0.45528	0.39496	0.32599
Lucene	2.0	2.2	0.71038	0.71038	0.70685	0.72087	0.53744	0.35106	0.53219	0.59574
	2.2	2.4	0.74812	0.74576	0.74953	0.74046	0.7169	0.39073	0.65882	0.72261
Poi	1.5	2.0	0.21512	0.21083	0.21083	0.21083	0.17391	0.17143	0.2028	0.2129
	2.0	2.5	0.34595	0.4359	0.45714	0.746	0.05447	0.20351	0.16197	0.13971
	2.5	3.0	0.782	0.7913	0.78298	0.79202	0.78887	0.33803	0.70232	0.7483
Synapse	1.0	1.1	0.41322	0.4186	0.3871	0.39695	0	0.325	0.26667	0.2
	1.1	1.2	0.54472	0.53875	0.55016	0.52769	0.32143	0.1913	0.43165	0.36975
Velocity	1.4	1.5	0.77059	0.78632	0.79083	0.78963	0.75904	0.63087	0.74074	0.76558
	1.5	1.6	0.51827	0.51485	0.52174	0.52	0.55762	0.37594	0.54032	0.5913
Xalan	2.4	2.5	0.57513	0.59759	0.60559	0.63924	0.10145	0.19383	0.22034	0.14253
	2.5	2.6	0.63994	0.63757	0.61755	0.64664	0.51841	0.23282	0.66083	0.59903
	2.6	2.7	0.91127	0.9775	0.97346	0.9723	0.49076	0.42456	0.6391	0.61633
Xerces	1.0	1.2	0.27879	0.27935	0.29748	0.28571	0.25778	0.21262	0.25899	0.24742
	1.2	1.3	0.26543	0.26629	0.26857	0.26879	0.10959	0.30986	0.16901	0.16807
	1.3	1.4	0.62857	0.70932	0.72099	0.80457	0.14799	0.2768	0.25882	0.13617
Average			0.4761	0.4876	0.4800	0.4931	0.3436	0.2946	0.3814	0.3658

Table 9 Results of Wilcoxon signed-rank test

Cost factor	Misclassification cost				Recall				F-measure			
	LR	NB	DT	RF	LR	NB	DT	RF	LR	NB	DT	RF
5	0.00096	0.0002	0.00128	0.0015	0	0	0	0	0.00278	0	0.00496	0.00298
10	0	0	0	0	0	0	0	0	0.00168	0	0.00528	0.00466
15	0	0	0	0	0	0	0	0	0.00386	0	0.01174	0.01108
20	0	0	0	0	0	0	0	0	0.00528	0	0.01552	0.01242

Table shows p value obtained by comparing performance measures of MOLR and other algorithms

less than or equal to decision tree (Ant 1.4–1.5, Ant 1.5–1.6, Jedit 4.2–4.3, Log4j 1.0–1.1 and Synapse 1.0–1.1). The loss in recall varies from 0 to 13% for these five cases. For rest of the 25 cases, recall values of MOLR are higher than decision tree. The gain in recall value varies from 2 to 72%. Maximum recall gain of 72.31% is reported for Xerces 1.3–1.4 experiment. And maximum cost difference is reported for Xerces init-1.2 case (LOC difference of 25530). In this case, MOLR achieves recall gain of 36.62%. MOLR is able to achieve average recall gain of 21.08% across all experiments.

Table 10 Single-objective logistic regression versus multi-objective logistic regression

Project	Train version	Test version	SOLR		MOLR		Difference	
			Cost (LOC)	Recall	Cost (LOC)	Recall	Cost (LOC)	Recall
Ant	1.3	1.4	15,115	0.1500	**10,091**	**0.1500**	−5024	0
	1.4	1.5	4614	0.1563	4447	0.0625	−167	−0.0938
	1.5	1.6	17,451	0.1522	14,907	0.0978	−2544	−0.0543
	1.6	1.7	102,300	0.4398	**101,250**	**0.5843**	−1050	+0.1446
Camel	1.0	1.2	360	0.0139	**281**	**0.0278**	−79	+0.0139
	1.2	1.4	24,617	0.1517	**23,840**	**0.5172**	−777	+0.3655
	1.4	1.6	18,121	0.0798	**16,678**	**0.4894**	−1443	+0.4096
Ivy	1.1	1.4	49,984	0.8125	**43,924**	**0.8750**	−6060	+0.0625
	1.4	2.0	19,163	0.1750	**17,360**	**0.2000**	−1803	+0.0250
Jedit	3.2	4.0	62,288	0.4667	**61,911**	**0.8000**	−377	+0.3333
	4.0	4.1	54,274	0.3671	**52,027**	**0.6582**	−2247	+0.2911
	4.1	4.2	75,135	0.5417	**68,974**	**0.6667**	−6161	+0.1250
	4.2	4.3	56,442	0.4546	**50,416**	**0.4546**	−6026	0
Log−4j	1.0	1.1	8870	0.4865	**8588**	**0.5946**	−282	+0.1081
	1.1	1.2	19,768	0.2275	**18,850**	**0.6561**	−918	+0.4286
Lucene	2.0	2.2	46,442	0.4236	**45,586**	**0.9097**	−856	+0.4861
	2.2	2.4	89,882	0.8670	**78,787**	**0.9557**	−11095	+0.0887
Poi	1.5	2.0	70,112	0.5946	**69,570**	**0.8649**	−542	+0.2703
	2.0	2.5	28,501	0.0282	**26,640**	**0.4597**	−1861	+0.4315
	2.5	3.0	112,170	0.8577	**104,730**	**0.9751**	−7440	+0.1174
Synapse	1.0	1.1	0	0.0000	0	0.0000	0	0
	1.1	1.2	16,487	0.2093	**13,637**	**0.3954**	−2850	+0.1861
Velocity	1.4	1.5	28,875	0.8873	**28,727**	**0.9366**	−148	+0.0493
	1.5	1.6	55,801	0.9615	**55,062**	**1.0000**	−739	+0.0385
Xalan	2.4	2.5	60,444	0.0543	**55,952**	**0.6021**	−4492	+0.5478
	2.5	2.6	258,200	0.4453	**255,950**	**0.8832**	−2250	+0.4380
	2.6	2.7	354,260	0.3252	**345,740**	**0.9766**	−8520	+0.6514
Xerces	1.0	1.2	112,560	0.4085	**105,950**	**0.6197**	−6610	+0.2113
	1.2	1.3	30,597	0.0580	**28,773**	**0.5797**	−1824	+0.5217
	1.3	1.4	56,835	0.0801	**45,423**	**0.7140**	−11412	+0.6339
Average				0.3625		**0.5902**		+0.2277

The comparative results of MOLR and random forest classifier are shown in Table 13. There are only three cases when recall values of MOLR are lesser than random forest recall values (Camel 1.0–1.2, Synapse 1.0–1.1 and Xerces init-1.2). For rest of the 27 cases, MOLR achieves recall gain up to 77%. Maximum recall gain of 77.12% is achieved for Xerces 1.3–1.4 experiment. MOLR is able to achieve average recall gain of 21.83% across all experiments. Overall recall gains are not as significant as other single objective algorithms like logistic regression and naïve Bayes. One of the reasons might be that, being an ensemble algorithm, random forest forms a model consisting of multiple decision trees. This reduces overfitting and enhances performance compared to single decision tree model.

Overall, there are very few cases when MOLR is achieving lesser effectiveness than single-objective algorithms.

We summarize our findings in Table 14. The average recall of each of single-objective prediction model and MOLR is reported in this table. And also the percentage of increase achieved by MOLR as compared to average recall value of single-objective algorithm is reported. Overall MOLR achieved more than 48% increase in all four cases, with maximum of 117% increase in case of comparison with naïve Bayes algorithm. This shows the dominance of multi-objective approach as compared to single-objective algorithms. Single objective algorithms only optimize prediction error as their objective, and they do not consider cost of prediction. MOLR models are trained to minimize

Table 11 Naïve Bayes versus multi-objective logistic regression

Project	Train version	Test version	NB		MOLR		Difference	
			Cost (LOC)	Recall	Cost (LOC)	Recall	Cost (LOC)	Recall
Ant	1.3	1.4	15,333	0.1250	**10,091**	**0.1500**	−5242	+0.0250
	1.4	1.5	58,017	0.6563	**54,519**	**0.8438**	−3498	+0.1875
	1.5	1.6	38,203	0.2500	**37,861**	**0.4022**	−342	+0.1522
	1.6	1.7	86,944	0.3735	**85,265**	**0.4940**	−1679	+0.1205
Camel	1.0	1.2	12,036	0.1574	**10,890**	**0.1667**	−1146	+0.0093
	1.2	1.4	30,307	0.2345	**29,462**	**0.5724**	−845	+0.3379
	1.4	1.6	30,205	0.1489	**29,745**	**0.6117**	−460	+0.4628
Ivy	1.1	1.4	32,032	0.3125	**30,380**	**0.7500**	−1652	+0.4375
	1.4	2.0	16,486	0.1250	**15,265**	**0.2500**	−1221	+0.1250
Jedit	3.2	4.0	45,376	0.2000	**45,040**	**0.6267**	−336	+0.4267
	4.0	4.1	45,988	0.2152	**44,391**	**0.5443**	−1597	+0.3291
	4.1	4.2	68,278	0.3125	**67,847**	**0.6667**	−431	+0.3542
	4.2	4.3	59,442	0.2727	**50,416**	**0.4546**	−9026	+0.1818
Log−4j	1.0	1.1	7685	0.4595	**7672**	**0.5676**	−13	+0.1081
	1.1	1.2	17,090	0.2963	**16,824**	**0.5450**	−266	+0.2487
Lucene	2.0	2.2	25,216	0.2292	**24,233**	**0.7500**	−983	+0.5208
	2.2	2.4	35,849	0.2906	**34,735**	**0.7291**	−1114	+0.4384
Poi	1.5	2.0	36,223	0.3243	**36,180**	**0.4865**	−43	+0.1622
	2.0	2.5	40,844	0.1169	**35,412**	**0.4758**	−5432	+0.3589
	2.5	3.0	40,658	0.2135	**40,369**	**0.6619**	−289	+0.4484
Synapse	1.0	1.1	28,270	0.4333	**25,356**	**0.5000**	−2914	+0.0667
	1.1	1.2	12,121	0.1279	**11,763**	**0.3023**	−358	+0.1744
Velocity	1.4	1.5	22,826	0.6620	**22,658**	**0.8732**	−168	+0.2113
	1.5	1.6	43,271	0.3205	**42,496**	**0.9872**	−775	+0.6667
Xalan	2.4	2.5	71,233	0.1137	**70,314**	**0.6615**	−919	+0.5478
	2.5	2.6	70,599	0.1484	**68,896**	**0.4818**	−1703	+0.3333
	2.6	2.7	306,030	0.2695	**299,830**	**0.9577**	−6200	+0.6882
Xerces	1.0	1.2	56,364	0.4507	**55,448**	**0.7606**	−916	+0.3099
	1.2	1.3	123,940	0.3188	**114,900**	**0.9130**	−9040	+0.5942
	1.3	1.4	86,894	0.1625	**73,060**	**0.9314**	−13834	+0.7689
Average				0.2774		**0.6039**		+0.3265

LOC cost and maximize effectiveness as their objectives. This is one of the major reasons why MOLR outperformed single-objective algorithms. MOLR is able to identify more defect prone classes than single-objective algorithms at same or lesser LOC cost.

To confirm these findings statistically, we performed Wilcoxon two- tailed paired test (Rahman et al. 2012) for all four cases. p values for all four types of experiments, i.e., SOLR versus MOLR, naïve Bayes versus MOLR, decision tree versus MOLR and random forest versus MOLR, are 0.00000131, 0.00000000, 0.00000117 and 0.00000008, respectively. As p values are significantly lower than threshold value 0.05, we can easily reject null hypothesis H_{02}. This confirms domination of MOLR over single-objective algorithms in terms of LOC cost and recall.

7 Threats to validity

This section discusses various threats to validity that may impact the analysis of the proposed approach and the experimental study presented here.

7.1 Threats to construct validity

The choice of recall as performance measure is widely adopted in previous studies. The choice of cost factor is based

Table 12 Decision tree versus multi-objective logistic regression

Project	Train version	Test version	D. Tree		MOLR		Difference	
			Cost (LOC)	Recall	Cost (LOC)	Recall	Cost (LOC)	Recall
Ant	1.3	1.4	22,266	0.2250	**22,112**	**0.3500**	−154	+0.1250
	1.4	1.5	30,936	0.5625	30,751	0.4375	−185	−0.1250
	1.5	1.6	17,600	0.1087	14,907	0.0978	−2693	−0.0109
	1.6	1.7	104,610	0.5121	**101,250**	**0.5843**	−3360	+0.0723
Camel	1.0	1.2	2937	0.0370	**2626**	**0.0556**	−311	+0.0185
	1.2	1.4	34,238	0.3931	**33,828**	**0.6000**	−410	+0.2069
	1.4	1.6	31,877	0.2394	**31,470**	**0.6277**	−407	+0.3883
Ivy	1.1	1.4	46,099	0.6875	**43,924**	**0.8750**	−2175	+0.1875
	1.4	2.0	19,995	0.1250	**17,360**	**0.2000**	−2635	+0.0750
Jedit	3.2	4.0	51,516	0.5333	**51,047**	**0.6933**	−469	+0.1600
	4.0	4.1	78,999	0.6329	**77,452**	**0.8608**	−1547	+0.2279
	4.1	4.2	118,510	0.8542	**112,650**	**0.9375**	−5860	+0.0833
	4.2	4.3	44,481	0.3636	**39,275**	**0.3636**	−5206	0
Log−4j	1.0	1.1	11,925	0.6757	10,162	0.6216	−1763	−0.0541
	1.1	1.2	20,216	0.2487	**20,019**	**0.6720**	−197	+0.4233
Lucene	2.0	2.2	39,645	0.4306	**38,183**	**0.8958**	−1462	+0.4653
	2.2	2.4	70,512	0.6897	**70,427**	**0.9360**	−85	+0.2463
Poi	1.5	2.0	71,231	0.7838	**69,570**	**0.8649**	−1661	+0.0811
	2.0	2.5	41,642	0.0927	**41,048**	**0.5282**	−594	+0.4355
	2.5	3.0	91,731	0.7011	**90,774**	**0.9537**	−957	+0.2527
Synapse	1.0	1.1	8531	0.1667	4909	0.1000	−3622	−0.0667
	1.1	1.2	20,402	0.3488	**20,390**	**0.4884**	−12	+0.1395
Velocity	1.4	1.5	27,690	0.8451	**25,848**	**0.9085**	−1842	+0.0634
	1.5	1.6	52,329	0.8590	**42,496**	**0.9872**	−9833	+0.1282
Xalan	2.4	2.5	84,023	0.1344	**83,456**	**0.7158**	−567	+0.5814
	2.5	2.6	343,230	0.7348	**336,330**	**0.9562**	−6900	+0.2214
	2.6	2.7	352,990	0.4733	**345,740**	**0.9766**	−7250	+0.5033
Xerces	1.0	1.2	102,530	0.2535	**77,000**	**0.6197**	−25530	+0.3662
	1.2	1.3	29,330	0.1739	**28,773**	**0.5797**	−557	+0.4058
	1.3	1.4	64,336	0.1510	**59,471**	**0.8741**	−4865	+0.7231
Average				0.4346		**0.6454**		+0.2108

on the fact that it is more costly to fix defects during post-release phase as compared to performing QA activities on non-defective files in pre-release phase (Moser et al. 2008). One can choose appropriate cost factor based on the project and organization. And also different values of cost factor may yield different results. For our second problem, the choice of LOC cost is inspired from the fact that it is related to amount of time that will be spent to review or test the code. And the same measure has been used in Canfora et al. (2013, 2015) and Rahman et al. (2012). But one can use other measures as well. One more threat to construct validity can be the choice of metrics and datasets. The CK metrics are one of the standard sets of metrics used for object-oriented projects. We have considered datasets from widely used PROMISE repository (Menzie et al. 2015), but we are not denying the fact that the datasets are prone to imperfection and incompleteness.

7.2 Threats to internal validity

One of the biggest threat is the choice of parameter settings for implementation. These parameters are chosen from experimentation and past research work (Canfora et al. 2013, 2015; Coello et al. 2007; Krall et al. 2015). For the parameters chosen from experimentations, we have used spread as the evaluation criteria for measuring goodness of Pareto optimal solutions, as defined by Deb (2001). The choice of different evaluation criteria may lead to different parameter settings. The choice of best parameters may vary from one

Table 13 Random forest versus multi-objective logistic regression

Project	Train version	Test version	Random forest		MOLR		Difference	
			Cost (LOC)	Recall	Cost (LOC)	Recall	Cost (LOC)	Recall
Ant	1.3	1.4	18,778	0.1750	**18,699**	**0.3500**	−79	+0.1750
	1.4	1.5	10,176	0.1563	**9342**	**0.1875**	−834	+0.0313
	1.5	1.6	30,515	0.2174	**27,254**	**0.3044**	−3261	+0.0870
	1.6	1.7	112,400	0.5301	**110,760**	**0.6566**	−1640	+0.1265
Camel	1.0	1.2	56	0.0093	22	0.0046	−34	−0.0046
	1.2	1.4	38,565	0.4207	**37,281**	**0.6207**	−1284	+0.2000
	1.4	1.6	22,497	0.2021	**20,718**	**0.5372**	−1779	+0.3351
Ivy	1.1	1.4	48,820	0.7500	**43,924**	**0.8750**	−4896	+0.1250
	1.4	2.0	10135	0.0750	**6557**	**0.1250**	−3578	+0.0500
Jedit	3.2	4.0	65,254	0.5733	**64,131**	**0.8400**	−1123	+0.2667
	4.0	4.1	61,109	0.5570	**59,899**	**0.7215**	−1210	+0.1646
	4.1	4.2	111,950	0.6458	**111,940**	**0.9375**	−10	+0.2917
	4.2	4.3	56,148	0.3636	**50,416**	**0.4546**	−5732	+0.0909
Log−4j	1.0	1.1	9247	0.5405	**9153**	**0.6216**	−94	+0.0811
	1.1	1.2	20,158	0.2169	**20,019**	**0.6720**	−139	+0.4550
Lucene	2.0	2.2	43,632	0.4514	**42,251**	**0.9306**	−1381	+0.4792
	2.2	2.4	86,894	0.7783	**78,787**	**0.9557**	−8107	+0.1773
Poi	1.5	2.0	88,547	0.9189	**83,322**	**0.9730**	−5225	+0.0541
	2.0	2.5	38,128	0.0685	**35,412**	**0.4758**	−2716	+0.4073
	2.5	3.0	106,020	0.7687	**104,730**	**0.9751**	−1290	+0.2064
Synapse	1.0	1.1	5916	0.1167	4909	0.1000	−1007	−0.0167
	1.1	1.2	16,572	0.2558	**13,637**	**0.3954**	−2935	+0.1395
Velocity	1.4	1.5	29,876	0.9155	**29,841**	**0.9578**	−35	+0.0423
	1.5	1.6	50,271	0.8462	**42,496**	**0.9872**	−7775	+0.1410
Xalan	2.4	2.5	73,382	0.0904	**70,314**	**0.6615**	−3068	+0.5711
	2.5	2.6	291,530	0.7470	**281,600**	**0.9027**	−9930	+0.1557
	2.6	2.7	351,450	0.4298	**345,740**	**0.9766**	−5710	+0.5468
Xerces	1.0	1.2	115,990	0.6901	105,950	0.6197	−10040	−0.0704
	1.2	1.3	31,204	0.1594	**28,773**	**0.5797**	−2431	+0.4203
	1.3	1.4	45,145	0.0801	**45,130**	**0.8513**	−15	+0.7712
Average				0.4250		**0.6417**		+0.2167

Table 14 % Increase in average recall achieved by MOLR

Algorithm	Average recall	Difference in average recall	% Difference in recall compared to single-objective algorithm
Logistic Regression	0.3625		
MOLR	0.5902	0.2277	62.81
Naïve Bayes	0.2774		
MOLR	0.6039	0.3265	117.72
Decision Tree	0.4346		
MOLR	0.6454	0.2108	48.51
Random Forest	0.425		
MOLR	0.6417	0.2167	50.98

dataset to another. We chose a uniform set of parameters to compare the results at the same scale. To mitigate inherent randomness of GA, we ran training process multiple times. For the problem M1, we ran NSGA-II 30 times to get the best model of MOLR which is used for testing purpose. For the problem M2, we ran and took coefficients corresponding to median Pareto front. The choice of median Pareto front corresponds to median spread value obtained among all 31 runs. The parameters for other algorithms are standard ones available in MATLAB. We have used logistic regression as fitness function to multi-objective genetic algorithm and four traditional machine learning algorithms for the comparison purpose. All of these algorithms have been used in many of the past works (He et al. 2013; Kamei et al. 2010; Kim et al. 2007; Mende and Koschke 2009; Peters et al. 2013), but the results may not hold true for other machine learning algorithms. Our aim was to compare cost-effectiveness of multi-objective algorithm with traditional algorithms for cross-version defect prediction, rather than comparing different machine learning algorithms with each other.

7.3 Threats to conclusion validity

We have performed two-tailed Wilcoxon paired test to statistically compare the difference in the performance measures obtained from MOLR and traditional single-objective algorithms. Wilcoxon test is a nonparametric test, which does not make any assumptions about input value distributions. We have confirmed our findings at 5% significance level.

7.4 Threats to external validity

We have experimented with 11 open-source projects from the PROMISE repository having different versions. One may find different results with the projects developed in industry, where certain standards are followed. So the findings presented by us may or may not hold good for industrial software projects. We have used CK metrics as predictors in our study. The choice of different metrics may yield different results.

8 Conclusion

In this paper, we formulated cross-version defect prediction as multi-objective optimization problem with two distinct set of objective functions, and the same was solved using multi-objective genetic algorithm. We compared multi-objective logistic regression with four single-objective algorithms for cross-version defect prediction. We applied the proposed approach to 11 projects and a total of 30 train version-test version pairs. Our results indicate following benefits of multi-objective approach:

1. The multi-objective defect prediction model (M1) is able to identify more defects at the same or lesser misclassification cost incurred by all four single-objective defect prediction models. And this observation holds good for four different values of cost factor 5, 10, 15, 20.
2. The multi-objective defect prediction model (M2) incurs the same or lesser LOC cost to achieve better recall as compared to all four single-objective defect prediction models. This proves that M2 is able to identify more defect-prone classes at the same or lesser LOC cost than the cost incurred with single-objective algorithms.

In summary, multi-objective approach can yield better results for cross-version defect prediction compared to traditional single objective approaches.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

Human and animal rights This article does not contain any studies with human participants or animals performed by any of the authors.

References

Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. Softw Eng IEEE Trans 22(10):751–761

Canfora G, De Lucia A, Di Penta M, Oliveto R, Panichella A, Panichella S (2013) Multi-objective cross-project defect prediction. In: 2013 IEEE sixth international conference on software testing, verification and validation (ICST), IEEE, pp 252–261

Canfora G, Lucia AD, Penta MD, Oliveto R, Panichella A, Panichella S (2015) Defect prediction as a multiobjective optimization problem. Softw Test Verif Reliab 25(4):426–459

Chidamber S, Kemerer C (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493

Coello CC, Lamont GB, Van Veldhuizen DA (2007) Evolutionary algorithms for solving multi-objective problems. Springer, Berlin

Czibula G, Marian Z, Czibula IG (2014) Software defect prediction using relational association rule mining. Inf Sci 264:260–278. doi:10.1016/j.ins.2013.12.031

D'Ambros M, Lanza M, Robbes R (2010) An extensive comparison of bug prediction approaches. In: 2010 7th IEEE working conference on mining software repositories (MSR), IEEE, pp 31–41

De Carvalho AB, Pozo A, Vergilio SR (2010) A symbolic fault-prediction model based on multiobjective particle swarm optimization. J Syst Softw 83(5):868–882

Deb K (2001) Multi-objective optimization using evolutionary algorithms. Wiley, New York

Deb K, Agrawal S, Pratap A, Meyarivan T (2000) A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. Lect Notes Comput Sci 1917:849–858

Ghotra B, McIntosh S, Hassan AE (2015) Revisiting the impact of classification techniques on the performance of defect prediction models. In: Proceedings of the 37th international conference on software engineering, ICSE '15—Volume 1. IEEE Press, Piscataway, pp 789–800

Goldberg DE (2006) Genetic algorithms. Pearson Education India, New Delhi

Harman M (2010) The relationship between search based software engineering and predictive modeling. In: Proceedings of the 6th international conference on predictive models in software engineering, PROMISE '10. ACM, New York, pp 1:1–1:13. doi:10.1145/1868328.1868330

Harman M, Clark J (2004) Metrics are fitness functions too. In: Proceedings of 10th international symposium on software metrics. IEEE, pp 58–69

Hassan AE (2009) Predicting faults using the complexity of code changes. In: Proceedings of the 31st international conference on software engineering. IEEE Computer Society, pp 78–88

He Z, Peters F, Menzies T, Yang Y (2013) Learning from open-source projects: an empirical study on defect prediction. In: 2013

ACM/IEEE international symposium on empirical software engineering and measurement, pp 45–54. doi:10.1109/ESEM.2013.20

Herbold S (2013) Training data selection for cross-project defect prediction. In: Proceedings of the 9th international conference on predictive models in software engineering, PROMISE '13. ACM, New York, pp 6:1–6:10. doi:10.1145/2499393.2499395

Jureczko M, Madeyski L (2010) Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th international conference on predictive models in software engineering, PROMISE '10. ACM, New York, pp 9:1–9:10. doi:10.1145/1868328.1868342

Kamei Y, Matsumoto S, Monden A, Matsumoto Ki, Adams B, Hassan A (2010) Revisiting common bug prediction findings using effort-aware models. In: 2010 IEEE international conference on software maintenance (ICSM), pp 1–10. doi:10.1109/ICSM.2010.5609530

Kim S, Zimmermann T, Whitehead EJ Jr, Zeller A (2007) Predicting faults from cached history. In: Proceedings of the 29th international conference on software engineering. IEEE Computer Society, pp 489–498

Krall J, Menzies T, Davies M (2015) GALE: Geometric active learning for search-based software engineering. IEEE Trans Softw Eng 41(10):1001–1018

Krishnan S, Strasburg C, Lutz RR, Goševa-Popstojanova K (2011) Are change metrics good predictors for an evolving software product line? In: Proceedings of the 7th international conference on predictive models in software engineering, Promise '11. ACM, New York, pp 7:1–7:10. doi:10.1145/2020390.2020397

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

Ma W, Chen L, Yang Y, Zhou Y, Xu B (2016) Empirical analysis of network measures for effort-aware fault-proneness prediction. Inf Softw Technol 69:50–70

Marian Z, Czibula IG, Czibula G, Sotoc S (2015) Software defect detection using self-organizing maps. Stud Unive Babes-Bolyai Inform 60(2):55–69

MATLAB (2015) version 8.5.0 (R2015a). The MathWorks Inc., Natick

Mende T, Koschke R (2009) Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th international conference on predictor models in software engineering, PROMISE '09. ACM, New York, pp 7:1–7:10. doi:10.1145/1540438.1540448

Menzie T, Krishna R, Pryor D (2015) The promise repository of empirical software engineering data. http://openscience.us/repo

Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: ACM/IEEE 30th international conference on software engineering, 2008. ICSE'08. IEEE, pp 181–190

Muthukumaran K, Choudhary A, Murthy NB (2015) Mining github for novel change metrics to predict buggy files in software systems. In: 2015 international conference on computational intelligence and networks (CINE). IEEE, pp 15–20

Peters F, Menzies T, Marcus A (2013) Better cross company defect prediction. In: 2013 10th IEEE working conference on mining software repositories (MSR), pp 409–418. doi:10.1109/MSR.2013.6624057

Rahman F, Posnett D, Devanbu P (2012) Recalling the "imprecision" of cross-project defect prediction. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the foundations of software engineering, FSE '12. ACM, New York, pp 61:1–61:11. doi:10.1145/2393596.2393669

Subramanyam R, Krishnan M (2003) Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. IEEE Transa Softw Eng 29(4):297–310. doi:10.1109/TSE.2003.1191795

Yang X, Tang K, Yao X (2015) A learning-to-rank approach to software defect prediction. IEEE Trans Reliab 64(1):234–246

Zhang D, El Emam K, Liu H et al (2009) An investigation into the functional form of the size-defect relationship for software modules. IEEE Trans Softw Eng 35(2):293–304

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: International workshop on predictor models in software engineering, PROMISE'07: ICSE Workshops 2007. IEEE, pp 9–9