Against the Mainstream in Bug Prediction

Haidar Osman
Software Composition Group
University of Bern, Switzerland
osman@inf.unibe.ch

Abstract

Bug prediction is a technique used to estimate the most bug-prone entities in software systems. Bug prediction approaches vary in many design options, such as dependent variables, independent variables, and machine learning models. Choosing the right combination of design options to build an effective bug predictor is hard. Previous studies do not consider this complexity and draw conclusions based on fewer-than-necessary experiments.

We argue that each software project is unique from the perspective of its development process. Consequently, metrics and AI models perform differently on different projects, in the context of bug prediction.

We confirm our hypothesis empirically by running different bug predictors on different systems. We show that no single bug prediction configuration works globally on all projects and, thus, previous bug prediction findings cannot generalize.

1 Introduction

A bug predictor is an intelligent system (model) trained on data derived from software (metrics) to make a prediction (number of bugs, bug proneness, etc.) about software entities (packages, classes, files, methods, etc.).

Bug prediction helps developers focus their quality assurance efforts on the parts of the system that are more likely to contain bugs. Bug prediction takes advantage of the fact that bugs are not evenly distributed across the system but they rather tend to cluster [19]. The distribution of the bugs of the bugs follow the Pareto principle, i.e., 80% of the bugs are located in 20% of the files [17]. An effective bug predictor locates the highest number of bugs in the least amount of code.

Over the last two decades, bug prediction has been a hot topic for research in software engineering and many approaches have been devised to build effective bug predictors. Among the scientific findings, two are agreed upon the most: (i) different machine learning models do not differ in predicting bugs [6][12][4][14][5], and (ii) change metrics are better than source code metrics at predicting bugs [16][10][15][20][1][9][5].
However, these studies do not consider the complexity of building a bug predictor, a process that has many design options to choose from:

1. The prediction model (e.g., Neural Network, Support Vector Machines, Random Forest, Linear Regression).
2. The independent variables (i.e., the metrics used to train the model like source code metrics, change metrics, etc.).
3. The dependent variable or the model output (e.g., bug proneness, number of bugs, bug density).
4. The granularity of prediction (e.g., package, class, binary).
5. The evaluation method (e.g., accuracy measures, percentage of bugs in percentage of software entities).

Most previous approaches vary one design option, which is the studied one, and fix all others. This affects the generalizability of the findings because every option affects the others and, consequently, the overall outcome, as shown in Figure 1.

![Diagram of the design aspects of Bug prediction.](image)

**Figure 1:** The design aspects of Bug prediction. The diagram shows the effects aspects have on each other.

We hypothesise that bug prediction findings are inherently non-generalizable. A bug prediction configuration that works with one system may not work with another because software systems have different teams, development methods, frameworks, and architectures. All these factors affect the correlation between different metrics and software defects.

To confirm our hypothesis, we run an extended empirical study where we try different bug prediction configurations on different systems. We show that no single configuration generalizes to all our subject systems and every system has its own “best” bug prediction configuration.

## 2 Experimental Setup

**Dataset**

We run the experiments on the “bug prediction data set” provided by D’Ambros et al. [3] to serve as a benchmark for bug prediction studies. This data set contains software metrics on the class level for five software systems (Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, Lucene, and Mylyn). Using this data set constrains the level of prediction to be on the class level. We compare source code metrics and version history metrics (change metrics) as the independent variables.
Dependent Variable

All bug-prediction approaches predict one of the following: (1) the classification of the software entity (buggy or bug-free), (2) the number of bugs in the software entity, (3) the probability of a software entity to contain bugs (bug proneness), (4) the bug-density of a software entity (bugs per LOC), or (5) the set of software entities that will contain bugs in the near future (e.g., within a month). In this study, we consider two dependent variables: number of bugs and classification.

Evaluation Method

An effective bug predictor should locate the highest number of bugs in the least amount of code. Recently, researchers have drawn the attention to this principle and proposed evaluation schemes to measure the cost or effort of using a bug prediction model [13][1][9][11][8][18][2]. Cost-aware evaluation schemes rely on the fact that a bug predictor should produce an ordered list of the software entities, then they measure the maximum percentage of predicted faults in the top $k\%$ of lines of code of a system. These schemes take the number of lines of code (LOC) as a proxy for the effort of unit testing and code reviewing.

In this study, we use an evaluation scheme called cost-effectiveness ($CE$), proposed by Arisholm et al. [1]. $CE$ ranges between $-1$ and $+1$. The closer $CE$ gets to $+1$, the more cost-effective the bug predictor is. A value of $CE$ around zero indicates that there is no gain in using the bug predictor. Once $CE$ goes below zero, it means that using the bug predictor costs more than not using it.

Machine Learning Model

For classification, we use RandomForest (RF), K-Nearest Neighbour (KNN), Support Vector Machine (SVM), and Neural Networks (NN). To predict the number of bugs (regression), we use linear regression (LR), SVM, KNN, and NN. We used the Weka data mining tool [7] to build these prediction models.

Procedure

For every configuration, we randomly split the data set into a training set (70%) and a test set (30%) in a way that retains the ratio between buggy and non-buggy entities. Then we train the prediction model on the training set and run it on the test set and calculate the $CE$ of the bug predictor. For each configuration, we repeat this process 30 times and take the mean $CE$.

3 Results

First, we compare the different machine learning models. To see if machine learning models perform differently, we apply the analysis of variance, ANOVA, and the post-hoc analysis, Tukey’s HSD (honest significant difference), among the different machine learning models in classification and regression. The tests were carried out at 0.95 confidence level. Only when the ANOVA test is statistically significant, we carry out the post-hoc test. Otherwise, we only report the best preforming model. Statistically significant results are in bold. As can be seen from the results in Table 1, It is clear that different machine learning models actually perform differently. Also there is not dominant model that stands out as the best model throughout the experiments.

Second, to compare the two different types of metrics, we compare the most performing model using source code metrics and the most performing model using change metrics using the t-student test at the 95% confidence level. We do the comparison in both classification and regression. Table 2 shows the results of the test, where bold text indicates statistically significant results. It can be deduced from the results that source code metrics are better than change metrics in some projects and worse in others. No type of metrics is constantly the best for all projects in the dataset.

1 We use the Weka default configuration values for the models
Table 1: This table shows the results of the analysis of variance, ANOVA, and the post-hoc analysis, Tukey’s HSD (honest significant difference). Bold text indicates statistically significant results at 95% confidence level.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>JDT</th>
<th>PDE</th>
<th>Equinox</th>
<th>Mylyn</th>
<th>Lucene</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version History</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$NN &gt; KNN$</td>
<td>$SVM &gt; KNN$</td>
<td>$SVM &gt; RF$</td>
<td>$SVM &gt; KNN$</td>
<td>$SVM &gt; RF$</td>
<td>$SVM &gt; NN$</td>
</tr>
<tr>
<td>$NN &gt; RF$</td>
<td>$SVM \geq RF$</td>
<td>$SVM \geq NN$</td>
<td>$SVM &gt; RF$</td>
<td>$SVM \geq RF$</td>
<td>$SVM \geq NN$</td>
</tr>
<tr>
<td>$NN \geq SVM$</td>
<td>$SVM \geq NN$</td>
<td>$SVM \geq NN$</td>
<td>$SVM \geq NN$</td>
<td>$SVM \geq NN$</td>
<td>$SVM \geq NN$</td>
</tr>
<tr>
<td><strong>Source Code</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$KNN &gt; SVN$</td>
<td>$KNN &gt; SVN$</td>
<td>$NN &gt; SVM$</td>
<td>$RF &gt; NN$</td>
<td>$NN &gt; SVM$</td>
<td></td>
</tr>
<tr>
<td>$KNN &gt; NN$</td>
<td>$KNN \geq NN$</td>
<td>$KNN$</td>
<td>$NN \geq KNN$</td>
<td>$RF \geq SVM$</td>
<td></td>
</tr>
<tr>
<td>$KNN &gt; RF$</td>
<td>$KNN \geq RF$</td>
<td>$KNN \geq RF$</td>
<td>$KNN \geq RF$</td>
<td>$KNN \geq RF$</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: This tables shows the t-student test between the best performing model trained on source code metrics (SM) and the best performing model trained on change metrics (CM). Bold text indicates the statistically significant results at 95% confidence level.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>JDT</th>
<th>PDE</th>
<th>Equinox</th>
<th>Mylyn</th>
<th>Lucene</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Classification</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$CM &lt; SM$</td>
<td>$CM &lt; SM$</td>
<td>$CM &gt; SM$</td>
<td>$CM &gt; SM$</td>
<td>$CM &gt; SM$</td>
<td></td>
</tr>
<tr>
<td>$CM \geq SM$</td>
<td>$CM \leq SM$</td>
<td>$CM \leq SM$</td>
<td>$CM &gt; SM$</td>
<td>$CM &gt; SM$</td>
<td></td>
</tr>
</tbody>
</table>

| **Regression** | | | | | |
| $CM > SM$ | $CM > SM$ | $CM > SM$ | $CM > SM$ | $CM > SM$ |
Table 3: This table shows the t-student test between the classifier (CLA) and the best regressor (REG). The test is carried out at the 0.95 confidence level. Bold text indicates the statistically significant results at 95% confidence level.

<table>
<thead>
<tr>
<th>System</th>
<th>Independent Variables</th>
<th>Prediction Model</th>
<th>Dependent Variable</th>
<th>Mean CE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse JDT Core</td>
<td>Change Metrics</td>
<td>SVM</td>
<td>Number of Bugs</td>
<td>0.356</td>
</tr>
<tr>
<td>Eclipse PDE UI</td>
<td>Source Code Metrics</td>
<td>KNN</td>
<td>Number of Bugs</td>
<td>0.246</td>
</tr>
<tr>
<td>Equinox</td>
<td>Source Code Metrics</td>
<td>SVM</td>
<td>Number of Bugs</td>
<td>0.429</td>
</tr>
<tr>
<td>Mylyn</td>
<td>Change Metrics</td>
<td>SVM</td>
<td>Number of Bugs</td>
<td>0.484</td>
</tr>
<tr>
<td>Lucene</td>
<td>Change Metrics</td>
<td>LR</td>
<td>Number of Bugs</td>
<td>0.588</td>
</tr>
</tbody>
</table>

Third, we compare the two types of response variables (classification vs regression) by comparing the best performing model from each using also the t-test statistical test at the 95% confidence level. Table 3 shows that the comparisons are in favor of regression all projects in our dataset but with statistical significance only in case of Equinox. This means that treating bug prediction as a regression problem is more cost effective than classification.

Finally, we compare configurations with the highest $CE$ for the five projects in the data set. In Table 4, we report the highest mean $CE$ and the configuration of the bug predictor behind. The results show that there is no global configuration of settings that suits all projects.

To summarize the results of the experiments, we make the following observations:

1. Different machine learning models actually perform differently in predicting bugs and there is no dominant model that stands out as the best for all projects.
2. There is no general rule about which metrics are better at predicting bugs.
3. The configurations of the most cost-effective bug predictor vary from one project to another.
4. The cost effectiveness of bug prediction is different from one system to another.

4 Conclusions

Building a software bug predictor is a complex process with many interleaving design choices. In the bug prediction literature, researchers have overlooked this complexity, suggesting generalizability where none is warranted. We argue that bug prediction studies cannot be generalized because software systems are different. Among the five subject systems we have, no type of metrics stands out as the best and no machine learning algorithm prevails when building a cost-effective bug predictor. This indicates a need for more research to revisit literature findings while taking bug prediction complexity into account. Also, the direction of research in this domain should change from looking for silver bullets to looking for project-specific most effective configurations.
References


