Effort-Aware Defect Prediction Models

Thilo Mende, Rainer Koschke
Fachbereich Mathematik und Informatik
University of Bremen
Bremen, Germany
{*tmende,koschke*}*@informatik.uni-bremen.de*

***Abstract*—Defect Prediction Models aim at identifying error-prone modules of a software system to guide quality assurance activities such as tests or code reviews. Such models have been actively researched for more than a decade, with more than 100 published research papers.**

However, most of the models proposed so far have assumed that the cost of applying quality assurance activities is the same for each module. In a recent paper, we have shown that this fact can be exploited by a trivial classifier ordering files just by their size: such a classifier performs surprisingly good, at least when effort is ignored during the evaluation. When effort is considered, many classifiers perform not significantly better than a random selection of modules.

In this paper, we compare two different strategies to include treatment effort into the prediction process, and evaluate the predictive power of such models. Both models perform significantly better when the evaluation measure takes the effort into account.

***Keywords*-Defect Prediction Models, Evaluation, Cost-Benefits.**

I. Introduction

Quality assurance activities, such as tests or code reviews, are an expensive, but vital part of the software development process. Any support that makes this phase more effective may thus improve software quality or reduce development costs. It has been observed that the distribution of defects follows a Pareto-principle, that is, that most bugs are located in only few files [1]. This lead to the development of defect prediction models, aiming at identifying error-prone parts of a system so that quality assurance activities can be focused.

This task is often seen as a classification problem with the goal to label files (or modules) as defective or non-defective. Since the prediction is most often based on data for files or modules, the evaluation is then performed at the file or module level as well.

As pointed out by Turney, different kinds of costs are associated with such classification models [2]. First, there are misclassification costs, when a non-defective file is labeled as defective or vice versa. This type of costs has already been considered for defect prediction models, e.g., by Khoshgoftaar et al. [3] and Jiang et al. [4].

Turney additionally identifies *Cost of Intervention*, in our case the costs of additional quality assurance activities. We refer to this type of cost as *effort* in the remainder of this paper. It is safe to assume that the effort is not uniformly distributed across files, so that some files are more expensive to test or review than others.

This was also pointed out by Arisholm et al. [5]–[7], who were interested in defect prediction models to support unit testing and code reviews. They argue that the costs for both activities are approximately proportional to the size of the file. They conclude that defect prediction models should always be evaluated with the amount of source code relative to the expected treatment effort in mind, otherwise the cost-effectiveness of such models are unclear. When defect prediction models are evaluated solely on the file level, this aspect is ignored and can be exploited by such a model: We recently demonstrated that a trivial model ordering files just by their size performs surprisingly well when evaluated on the file level. [8]

Therefore, we have to include the notion of effort into the evaluation. The measure CE introduced by Arisholm et al. [6] does that by comparing the performance of a defect prediction model with a random selection of files, using lines of code as a surrogate measure for effort. Many classification algorithms that were good according to traditional performance measures are actually relatively bad according to CE. [6]–[8] This is not too surprising, since the algorithms are tuned to optimize one specific measure, namely misclassification rate, that has only an indirect influence on CE. It may thus may be possible that the classifiers would perform better if they were aware of the changed performance measure.

Contributions. We propose two strategies to include the notion of effort awareness into defect prediction models and evaluate these strategies on fifteen publicly available data sets, twelve from the NASA Metrics Data Program (MDP)[1] and three from the Eclipse IDE. Both strategies improve the cost effectiveness of defect prediction models significantly, in the statistical and a practical sense.

Overview. The remainder of this paper is organized as follows: First, we discuss related work in Section II. Afterwards, in Section III, we describe two strategies to make defect prediction models effort-aware. Afterwards, we describe the experimental setup in Section IV, and

[1] http://mdp.ivv.nasa.gov

present our results on fifteen publicly available data sets in Section V. Finally, Section VI concludes.

II. Related Work

Predicting defective parts of a software system has been actively researched for more than a decade. The task can be seen as a classification problem: The goal is to predict the outcome of a *dependent variable* with a *classification technique* using several *independent variables*. The dependent variable is often a binary classification whether a file or module is defective within a certain time frame. Classification techniques used for defect prediction vary from regression models to data mining algorithms. Independent variables include code complexity measures [9], complexity of code changes [10], object-oriented metrics [11], dependencies [12], [13], or organizational factors [14].

Instead of directly predicting class labels, many modeling techniques can be used as probabilistic classifiers [15], which assign scores to each file. These scores can be used to build *module-order models* [16], where modules are ordered by descending predicted error-proneness. This approach is attractive in practice, as it enables project managers to select a fixed percentage of files (determined by the available budget) for further treatment. Such a model is, for instance, used by Ostrand et al. [17]: they select 20 % of the files for further treatment and thereby identify up to 84 % of the defects, if the QA activity were perfect.

Module-order models are also used by Arisholm et al. [5], [6], however, they point out that the comparison of defects identified in n % of the files may be misleading. They assume that treatment effort is roughly proportional to the size of a file, and thus the percentage of defects found in n % percent of the files should be compared to the relative amount of source code contained in these files, since a random selection of m % of the source code is able to identify m % of the defects. We confirmed this recently by showing how effective a trivial model — ordering files just by the size — is, at least when evaluated on the file level. [8], [18] Arisholm et al. proposed a performance measure for classifiers taking the effort into account, which is described in detail in Section IV.

If effort is related to lines of code, predicting defect densities is one way to make effort-aware predictions, as we will see in Section III. Only a few studies so far have predicted defect densities: Knab et al. [19] predict discretized defect densities using decision trees, while Nagappan and Ball build regression models to directly predict defect densities [20], [21]. However, neither of them evaluated the influence on the treatment effort. Ostrand et al. [17] also predict defect density, and evaluate their prediction for one specific cutoff. They conclude that a prediction using defect densities is able to find more defects in a fixed percentage of code, but argue that testing costs are, at least for system tests, not related to the size of a file. In this paper, we replicate their experiment and conduct a more thorough analysis over all possible cutoff values.

Two recent papers specifically address the evaluation of defect prediction models and are particularly important in the following. Both compare different classifiers on data sets from the NASA MDP repository. The methodology to compare classifiers in both of them is based on work by Demšar [22]: He describes a set of non-parametric hypothesis tests to compare the performance of two or more classifiers over multiple data sets. Demšar's approach is described in Section IV.

Lessmann et al. [23] identify the need for a common evaluation framework for defect prediction models. They propose to use Area under the ROC curve (AUC) — a representation of a classifier's performance independent of thresholds — to assess the performance of prediction models, and to use the process described by Demšar to compare the performance of different classification algorithms. They conclude that sophisticated data mining techniques, such as Random Forests, are performing best, although many simpler algorithms are not significantly worse.

Jiang et al. [24] evaluate different classification techniques on eight data sets from the NASA MDP. They compare several performance measures, among them AUC and lift charts, and conclude that different performance measures are suitable for different application scenarios, that is, advocate the choice of different classification techniques for different data sets. In a subsequent study, they explore the performance of defect prediction models from the perspective of misclassification costs, that is, the ratio of costs for false positives to the costs of false negatives [25]. They conclude that different misclassification costs have a huge impact on the selection of appropriate prediction models, but also point out that they assume the same misclassification costs for each module, which might be unreasonable in practice.

III. Making Effort-Aware Predictions

The goal of a defect prediction model is to determine modules for further quality assurance activities. As described in the previous section, module-order models as defined by Khoshgoftaar et al. [26] are most useful for this purpose. Such a model assigns a score $R(x)$ to each module representing the predicted risk, i.e., relative error-proneness. This score is then used to order modules, and the QA team can select the modules with the highest score (representing highest risk) for further *treatment*, such as code reviews or unit tests.

The effort for this treatment depends on the type of treatment. Even though it may be hard to quantify, the assumption that treatment effort is the same for each file is unreasonable. Arisholm et al. [7] prososed an effort-aware performance measure for defect prediction models and showed that many models which are quite good according to confusion-matrix based metrics are in fact not cost-effective,

or offer only small cost-benefits when evaluated with an effort-aware performance measure.

One reason for this may be that the classifiers did not know about the changed evaluation criteria, i.e. are not cost-sensitive. In the following, we present two ways to incorporate treatment effort into a module-order model, and evaluate both ways in Section V.

Cost-sensitive learning has been identified as an important research topic in the machine learning community, foundations are summarized by Elkan [27]. Most of the resulting learning algorithms are based on misclassification costs depending only on the class of an object, in our case whether a file is defective or not. However, the costs of a *false positive*, labeling a file as defective when it is in fact not defective, vary from object to object. There are only few learning algorithms that take such example-dependent costs directly into account, and none of them is available in off-the-shelf data mining toolkits.

A naïve approach is to adjust the predicted risk $R(x)$ of a module by its relative treatment effort $\frac{E(x)}{E_{max}}$ as follows (let $E(x)$ be the effort required to treat a module and E_{max} be the maximum effort among all modules within a project):

$$\begin{aligned} R_{adj}(x) = &\quad R(x) \cdot RAF \\ RAF = &\quad \left(1 - \tfrac{E(x)}{E_{max}}\right) \end{aligned}$$

where *RAF* is a risk adjustment factor sensitive to the treatment effort. If we rank modules according to $R_{adj}(x)$, we want high-risk modules with less treatment effort—the low-hanging fruits—to appear at the top to spend the effort most cost-effective. Consequently, the role of the term $1 - \frac{E(x)}{E_{max}}$ is to rate less costly modules higher than more costly modules.

One disadvantage of $R_{adj}(x)$ is that it does not take the distribution of defects relative to the effort into account. If the modelling technique used is able to perform regression, i.e., predict numerical values instead of just class labels, we can use the MetaCost [28] approach and directly predict the relative risk $R_{dd}(x)$, which is in our case

$$R_{dd}(x) = \frac{\#errors(x)}{E(x)}$$

where $\#errors(x)$ is the number of errors in module x and $E(x)$ is defined as above. If lines of code is used as a proxy for effort, this is equivalent to predicting defect-density.

Both strategies consider only costs associated with *false positives* and ignore the costs of false negative predictions. The costs of false negatives appear as damages caused by faults in the field experienced by the user. The above two rankings are based solely on the budget available for testing. If not all errors cause the same damage costs, a test engineer can first use a risk-based testing strategy prioritizing the modules to be tested based on likelihood and impact of failure. This results in a partitioning of modules according to their risks of damages. Then our technique can be used to steer testing effort for modules at the same level of risk. Because these damages are not known to us in our case studies, we cannot take them into account in the evaluation.

In the following, we compare both strategies with a regular classification algorithm using the experimental setup described in the next section. But first of all, we have to define a way to quantify treatment effort. Defect prediction models can be used to support different quality assurance activities, partly depending on the granularity of their predictions. File-based predictions are useful for unit testing and code reviews, while other activities may require coarser predictions, e.g. based on subsystems [29]. In this paper, we use file-based predictions and thus focus on the effort for unit testing and code reviews. For the latter, the effort is probably approximately proportional to the size, as pointed out by Arisholm et al. [5], [6].

Hence, we could use lines of code. An alternative is McCabe's cyclomatic complexity. McCabe's cyclomatic complexity [30] is a lower bound for the number of linear independent paths through a program, and an upper bound for the number of test cases required to achieve complete branch coverage. It can thus serve as a proxy for unit-test effort. The cyclomatic complexity has often been criticized, especially because of its strong correlation to lines of code, making it more of a size measure than a complexity measure. This, however, is desirable to quantify testing effort, as shown by Bruntink et al. [31], [32]. They investigate the relationship between static code metrics and the effort for unit tests, and also conclude that the size of a module has a large influence on the testing effort.

To summarize, even though cyclomatic complexity is not a direct measure of testing effort, we argue that it is a better proxy for effort in case of unit testing and code reviews than assuming equal treatment costs — and thus use it to approximate effort in the following. Another advantage of this measure is that it is available for publicly available data sets, thus enabling independent reproduction of our results.

IV. Experimental Setup

We assume that the inclusion of effort-awareness into defect prediction models improve their performance, and investigate this hypothesis with a comparison of three prediction models on 15 publicly available data sets.

The experimental setup closely follows Lessmann et al. [23] and Jiang et al. [24] in the selection of data sets and evaluation methodology.

Datasets: We use fifteen data sets from the PROMISE repository[2] shown in Figure 1. This includes 12 data sets from the NASA MDP, representing the union of data sets

[2]http://promisedata.org

Name	Nr. of Modules	% Defective
kc1	2107	0.15
kc3	458	0.09
kc4	125	0.49
jm1	10878	0.19
pc1	1107	0.07
pc2	5589	0.00
pc3	1563	0.10
pc4	1458	0.12
pc5	17186	0.03
cm1	505	0.10
mc2	161	0.32
mw1	403	0.08
Eclipse 2.0	6729	0.14
Eclipse 2.1	7888	0.11
Eclipse 3.0	10593	0.15

Figure 1. Data sets used for evaluation.

used by Lessmann et al. and Jiang et al.[3] A detailed description of these systems can be found elsewhere [23], [24]. Additionally, we include the file-level data for three versions of the Eclipse IDE[4] provided by Zimmermann et al. [33]. We use all input attributes available for each data set as independent variables, except module identifiers and metrics related to error count and density. These attributes include static code metrics, such as Halstead's [34] or McCabe's [30] complexity measures. The size of each system and the ratio of defective modules (i.e., modules with at least one defect) can be found in Figure 1.

Modelling: We use the RandomForest algorithm proposed by Breiman [35] as our underlying modelling technique. This algorithm uses a majority voting of 500 decision trees to generate classification (predicting, often binary, class labels) or regression (predicting numerical values) results. The algorithm offers good out-of-the-box performance and has performed very good in different defect prediction benchmarks, e.g., by Lessmann et al. [23]. We use the R [36] package RandomForest [37] for all our experiments.

We use RandomForests for three different models: First, to predict defective files without any effort-awareness, which we refer to as *RF*. Second, a model to predict R_{adj} using the predictions of RF is abbreviated as *AD*, and third, a model predicting R_{dd} as *DD*. RF is a binary prediction, i.e., the dependent variable is a binary one encoding whether a module has either none or at least one defect. DD is based on defect density and, thus, the only classifier (indirectly) seeing the number of defects in each module during training. We investigated whether the performance of RF is better or worse when the number of defects is predicted instead of the binary class label. When an effort-sensitive evaluation measure is used, the performance is either almost the same or much worse on almost all data sets, so we use RF for binary classification.

Performance Measures: Appropriate performance measures for defect prediction models are a long debated topic, since each measure captures only certain aspects [7], [24], [38], [39]. We have used the following criteria to select appropriate measures for our comparison:

1) The measure should be sensitive to the treatment effort.
2) The measure should be a single scalar value to make comparisons between different classifiers on the same data set feasible.
3) The measure should be easily interpretable from a practical perspective, i.e., it should reflect the benefit one could get by using a defect prediction model.
4) The measure should be independent of thresholds, i.e., represent all possible budgets available for treatment.

Unfortunately, requirement 3 and 4 are contradictory. We therefore decided to evaluate two aspects separately: the practical cost-benefits for one specific cutoff are discussed in Section V-A, while a cutoff-independent measure is used in Section V-B.

It is generally accepted that the most important aspect of a defect prediction model is to determine what percentage of defects could actually be found by the QA activities using the prediction model. Therefore, one generally uses *recall*, i.e., the percentage of defective modules that are detected, out of all defective modules. A slight variation of this is to use the *defect detection rate* (ddr), which measures the ratio of number of detected defects compared to the total amount of defects. We prefer ddr over recall, since it better captures the cost-effectiveness of a model.

The values for ddr depends on a cut-off value, for an ordering model, this means the relative effort one would be able to spend for additional QA activities. The distribution for a prediction model can then be depicted in a *cumulative lift-chart*, as exemplified in Figure 2, where the ddr value associated with 20 % of the code is, for this prediction model, around 40 %.

However, by flagging all modules as defective, it is easy to get recall or ddr of 1, so we have to take the number of false positives—i.e., modules that are flagged as defective but actually are non-defective—into account. Measuring false positives is a more controversial topic [38], [39]. Either one uses *precision*, which determines the ratio of actually defective files compared to the number of files flagged as defective, or we can use the *false-positive ratio* (fpr) denoting the percentage of non-defective files flagged as defective. The main problem of fpr is that, when the data set is very imbalanced, that is, there are much more non-defective files than defective ones, a low fpr may be misleading, since the amount of false positives in the files

[3]Dataset kc2 was excluded, since there is no version with error count available at the Promise repository.

[4]http://eclipse.org

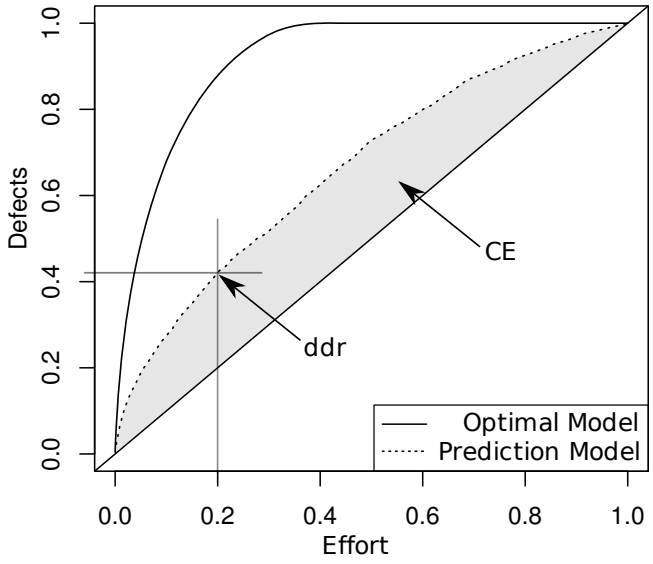

Figure 2. Example for the calculation of ddr and CE

flagged as defective is overwhelming. We therefore prefer precision over fpr.

In the following, we use mainly ddr at a fixed percentage of effort to evaluate the cost-effectiveness of a model. By selecting a fixed percentage of code, an unreasonably high ddr with an unacceptable high amount of false positives is mitigated. Using ddr in this way has the advantage of having a single scalar value that can be used to compare classifiers. However, in Section V-A, we also provide and analyze precision and recall for the selected cutoff to enable a comparison with earlier studies.

The selection of a fixed cutoff value for the effort, in our case 20 %, is somewhat arbitrary but allows a comparison with earlier studies, in which this value was chosen. To compare different models over the whole range of possible cutoff values, we use the measure *Cost Effectiveness* (CE) introduced by Arisholm et al. [6]. It is defined in a cumulative, effort-based lift-chart as the area below the defect predictors line, but above a line of slope 1. The line of slope 1 represents the random selection of source lines. This area is shaded grey in the example in Figure 2.

Evaluation Procedure: We use ten-times ten-fold cross-validation to train and test all algorithms on all data sets and average results.

Comparing classifiers by just comparing scalar performance measures may be misleading due to inherent variance, so statistical hypothesis tests are necessary. One approach, described by Demšar [22], uses non-parametric statistics to evaluate whether the performance of several classifiers over multiple data sets is significantly different. This approach is used by both Lessmann et al. and Jiang et al., so we also adopt it here.

Demšar uses the Friedman test [40] to check whether the null hypothesis, namely, that all classifiers perform equal on the selected data sets, can be rejected. The Friedman test is a non-parametric statistical test using only relative rankings, and not performance values directly, thus making no assumptions on the distribution of performance values. It can be calculated using the following formulas from Demšar [22], where k denotes the number of classifiers, N the number of data sets, and R_j the average rank of classifier j on all data sets:

$$\chi_F^2 = \frac{12N}{k(k+1)} \left(\sum_j R_j^2 - \frac{k(k+1)^2}{4} \right)$$

and

$$F_F = \frac{(N-1)\chi_F^2}{N(k-1) - \chi_F^2}$$

F_F is distributed according to the F-Distribution with $k-1$ and $(k-1)(N-1)$ degrees of freedom. Once computed, we can check F_F against critical values for the F-Distribution and then accept or reject the null hypothesis.

When the Friedman test rejects the null hypothesis, we can use the Nemenyi post-hoc test to check whether the performance of two classifiers is significantly different. The test uses the average ranks of each classifier and checks for each pair of classifiers whether the difference between their ranks is greater than the critical difference $CD = q_\alpha \sqrt{\frac{k(k+1)}{6N}}$, where k and N are the same as above, and q_α is a critical value depending on the number of classifiers and the significance level α. For our setup with $k = 3$ and $\alpha = 0.05$, $q_{0.05} = 3.34$.

We use Lessmann et al.'s [23] modified version of Demšar's significance diagrams to depict the results of Nemenyi's post-hoc test: For each classifier on the y-axis, the average rank is plotted on the x-axis, together with a line segment whose length encodes *CD*. All classifiers that do not overlap in this plot perform significantly different.

V. Evaluation

The evaluation of defect prediction models is, as already mentioned, a difficult tasks. In this section, we evaluate our results from two different perspectives: First, in Section V-A, we investigate the practical usability of our models. Afterwards, in Section V-B, we compare the different models from a statistical perspective. A discussion of our results follows in Section V-C, and threats to validity are discussed in Section V-D.

A. Effort Reduction

In this section, we assume that an organization has a fixed budget for QA activities and wants to spend this as cost-effective as possible, i.e., find as many bugs as possible. Since QA is expensive, the budget is typically not sufficient to investigate the whole code base, but only a fraction of it.

	RF	AD	DD	Optimal
kc1	0.32	0.37	0.45	0.92
kc3	0.34	0.47	0.50	0.98
kc4	0.32	0.39	0.51	0.76
jm1	0.24	0.27	0.42	0.88
pc1	0.59	0.61	0.69	1.00
pc2	0.66	0.69	0.69	1.00
pc3	0.50	0.52	0.51	1.00
pc4	0.85	0.88	0.84	1.00
pc5	0.44	0.53	0.69	0.98
cm1	0.21	0.34	0.45	1.00
mc2	0.28	0.39	0.49	0.86
mw1	0.40	0.39	0.44	0.99
Eclipse 2.0	0.31	0.36	0.39	0.86
Eclipse 2.1	0.26	0.29	0.37	0.95
Eclipse 3.0	0.27	0.29	0.35	0.87

Figure 3. ddr at 20 % cutoff

We use 20 % in the following investigations, simply because this number has often been used in the past, e.g., by Ostrand et al. [17]. Additionally, we expect that this number is a realistic budget in practice.

By selecting 20 % of the code base at random, we can assume that our QA activities should be able to find 20 % of the bugs. This can serve as a baseline, no classifier should perform worse. On the other hand, if the defects are spread across files that require more than 20 % of the effort, our file-based prediction model would not be able to catch all bugs. In this case, an optimal ordering can serve as the upper bound.

The results for our three models RF, AD and DD and the optimal model for each data set by selecting 20 % of the code according to the cyclomatic complexity in terms of ddr can be found in Figure 3. Additionally, they are plotted in Figure 4. As we can see, both AD and DD perform better than RF on ten data sets, and DD performs only for two data sets worse than AD. On average, AD is able to detect 5.3 % more defects than RF, with a minimum of -1.4 % and a maximum of 13 %. DD improves, compared to RF, the prediction by 11.9 % on average, with a minimum of -1.5 % and a maximum of 24.9 %.

We can thus see that the difference between DD and RF is large for many data sets, especially when we take into account that a random selection of source code would already find 20 % of the defects. For example, for dataset jm1, RF offers only 4 % advantage over a random selection of files, while DD is able to identify 22 % additional defects.

For some data sets, DD performs about the same or even worse than RF, which is surprising to us. We investigate this issue in Section V-C.

In Figure 5, we provide the results measured in precision and recall for our three predictors on all data sets. As we can see (and expect) the results for recall are very similar to the ones measured with ddr; both AD and DD perform better on most data sets. However, this comes at a price: the values for precision are lower on most data sets. Whether this is acceptable is a tradeoff between bugs not identified during the treatment and the costs of treatment itself.

B. Statistical Comparison

The selection of a fixed cutoff threshold is obviously somewhat arbitrary, therefore we compare all classifiers in a cutoff-independent manner in this section. We use the performance metric CE [6] explained in Section IV for all comparisons in this section.

The detailed results of the three classifiers on our data sets in terms of CE and the average rank of each classifier can be found in Figure 6. As we can see, both effort-adjusted classifiers outperform the traditional prediction model RF on most data sets. This confirms our results from the previous section, and shows that DD and AD outperform RF not only for one specific cutoff.

For data sets pc3 and pc4, the CE values of RF are very close to or better than the ones of AD and DD which we already observed in the previous section. We investigate these data sets in Section V-C.

We can now test whether the differences between the three classifiers overall are statistically significant using the procedure by Demšar: The Friedman test can be calculated using the formula described above and yields $F_F = 45.43$. The critical value for the F-Distribution and $\alpha = 0.05$ with 2 and 28 degrees of freedom is 3.34, so the null hypothesis that all classifiers perform equally well can be rejected. Nemenyi's critical difference can be calculated as $CD = 0.856$.

The results of the Nemenyi post-hoc test can be found in Figure 7. Both effort-aware classifiers perform significantly better than RF. Although DD is performing better in terms of CE than AD, the difference is not statistically different, according to the Nemenyi test.

C. Discussion

As we have seen in Section V-A and Section V-B, both strategies to incorporate effort perform better than the regular classifier RF on most data sets. Thus when the type of treatment and approximate costs associated with it are known when evaluating a defect prediction model, it is beneficial to use one of these strategies. On some data sets, the performance of RF is very close to a random selection of modules, while DD is able to provide practically valuable predictions. Nevertheless, on most data sets, the difference between DD and the optimal prediction is still large, leaving much room for improvements. Whether this is feasible, especially when one is restricted to the currently available metrics, is an open research question.

The difference between RF and DD is much larger than the difference between RF and AD, both in terms of CE and

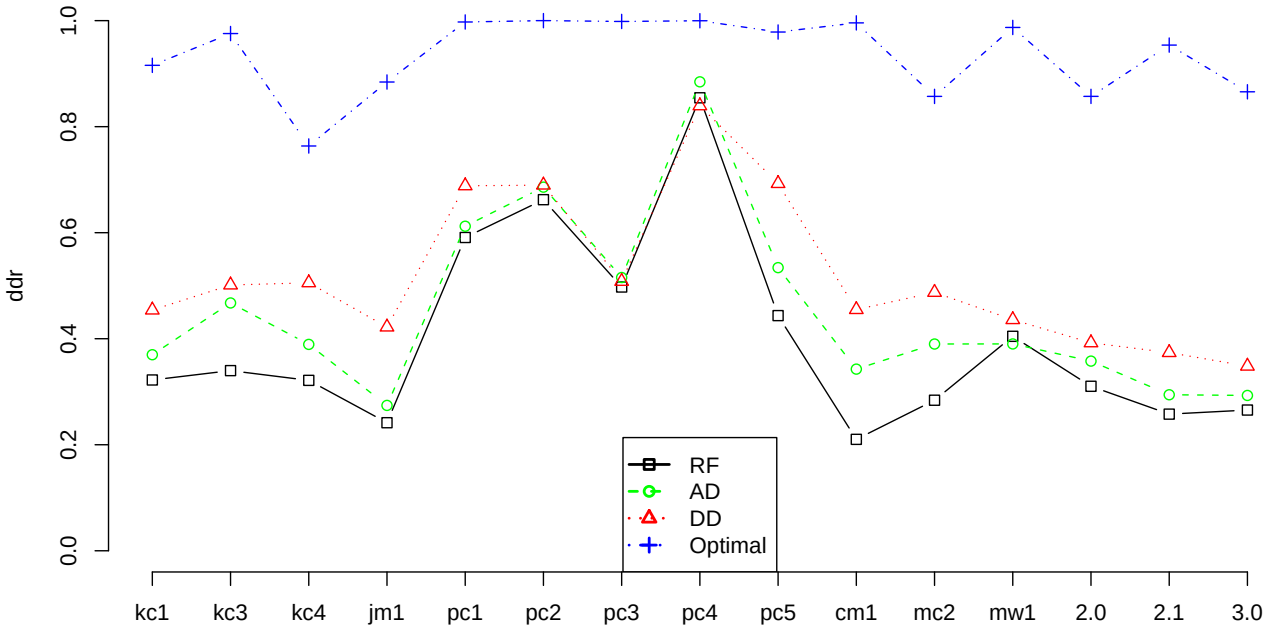

Figure 4. ddr at 20 % cutoff, the Eclipse data sets are abbreviated by their version number.

	RF		AD		DD	
	Re.	Pr.	Re.	Pr.	Re.	Pr.
kc1	0.28	0.63	0.35	0.53	0.44	0.33
kc3	0.31	0.35	0.45	0.38	0.46	0.30
kc4	0.21	0.79	0.41	0.70	0.57	0.59
jm1	0.16	0.68	0.22	0.63	0.43	0.26
pc1	0.53	0.40	0.59	0.31	0.67	0.26
pc2	0.64	0.05	0.67	0.04	0.68	0.03
pc3	0.47	0.40	0.49	0.40	0.54	0.25
pc4	0.81	0.52	0.85	0.51	0.80	0.45
pc5	0.33	0.75	0.43	0.66	0.68	0.32
cm1	0.23	0.23	0.35	0.25	0.46	0.17
mc2	0.20	0.65	0.30	0.51	0.41	0.44
mw1	0.38	0.30	0.40	0.22	0.40	0.18
Eclipse 2.0	0.22	0.83	0.29	0.78	0.38	0.34
Eclipse 2.1	0.19	0.59	0.23	0.52	0.38	0.15
Eclipse 3.0	0.17	0.79	0.22	0.70	0.32	0.20

Figure 5. Recall (Re.) & Precision (Pr.) at 20 % cutoff

	RF	AD	DD
kc1	0.12	0.15	0.19
kc3	0.14	0.17	0.20
kc4	0.02	0.07	0.12
jm1	0.04	0.06	0.17
pc1	0.28	0.28	0.31
pc2	0.32	0.33	0.34
pc3	0.24	0.26	0.24
pc4	0.40	0.41	0.40
pc5	0.28	0.30	0.36
cm1	0.08	0.14	0.19
mc2	0.05	0.10	0.15
mw1	0.15	0.15	0.18
Eclipse 2.0	0.11	0.13	0.16
Eclipse 2.1	0.04	0.06	0.13
Eclipse 3.0	0.04	0.06	0.13
Average Rank	2.93	1.87	1.20

Figure 6. Performance of all classifiers on fifteen data sets measured using CE and average rank per classifier.

ddr. This is ignored by the non-parametric Nemenyi test, but should be taken into account when choosing a strategy.

However, there are two data sets where the ddr performance of RF is very close to or better than AD or DD, namely pc3 and pc4. First of all, we have to keep in mind that all data sets are from real-world projects, and for some of them it is well known that they contain implausible data for certain modules.[5]

When we take a closer look at the pc4 data set, we can see such problems as well:

- There are 111 modules with a value 0 for lines of code, and none of them contains an error.
- The two modules with the highest error count (25 and

[5]A discussion of one such issue, namely missing values for lines of code, can be found at http://promisedata.org/?p=30.

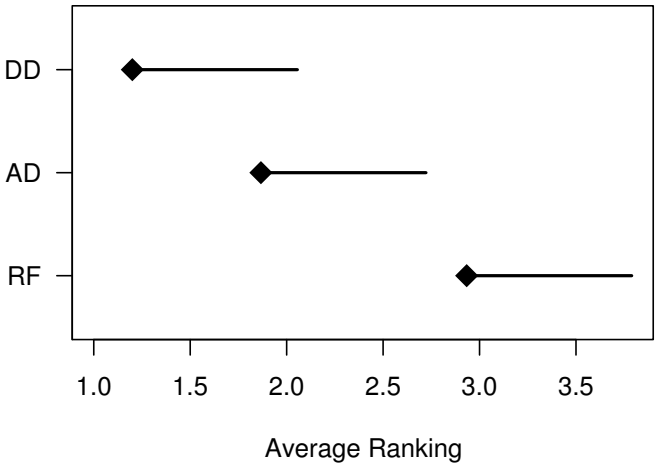

Figure 7. Nemenyi's Critical-Difference Diagram for the evaluation using CE.

9 errors totalling 25 % of the errors) have a cyclomatic complexity of 1, but the highest and third-highest values for lines of code. This is either a long list of statements with high error-proneness, or an error in the data. In both cases, McCabe is not suitable to approximate the testing effort, thus we have to use another surrogate measure, such as lines of code.

When we remove all modules with lines of code of 0 and use lines of code as our surrogate measure for effort, we get the following results with a 10-times 10-fold cross-validation:

	RF	AD	DD
ddr	0.69	0.71	0.75
CE	0.34	0.34	0.36

This results in the ordering between the classifiers observed on most of the other data sets, although the differences between our three classifiers are still small compared to the other data sets.

This may be due to the good performance for the five pc data sets, where all three classifiers offer relatively good performance in terms of ddr and CE compared to the remaining data sets. When we take a look at the cumulative lift chart for pc4 in Figure 8, we can see that all three classifiers are quite close to a perfect prediction, and thus it is hard for one classifier to perform much better than one of the other.[6]

Even though pc3 is the data set with the worst CE results among the pc data sets, the performance of all three classifiers is higher than any other CE value on the non-pc data sets. This may be an explanation for the similar performance of RF, AD and DD. However, when we look

[6]The results shown in Figure 8 and Figure 9 are for one 10-fold cross-validation run, while the results in Figure 6 are averaged over 10 runs, which explains the small differences in the performance of classifiers.

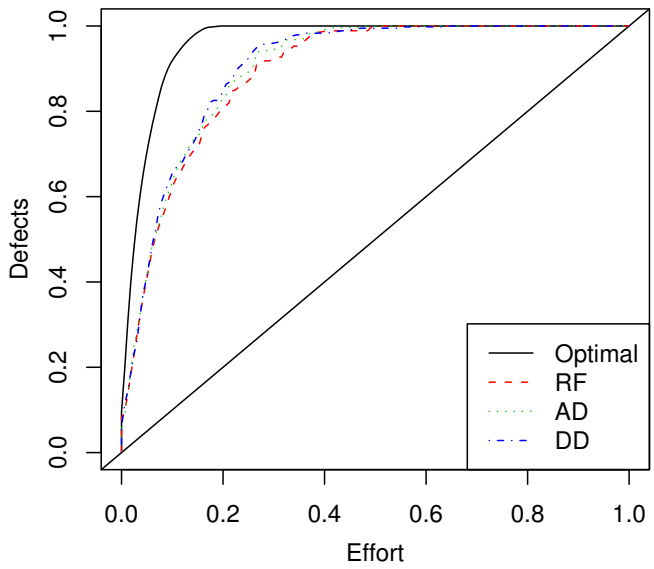

Figure 8. Cumulative Lift Chart for three classifiers and the optimal prediction on pc4.

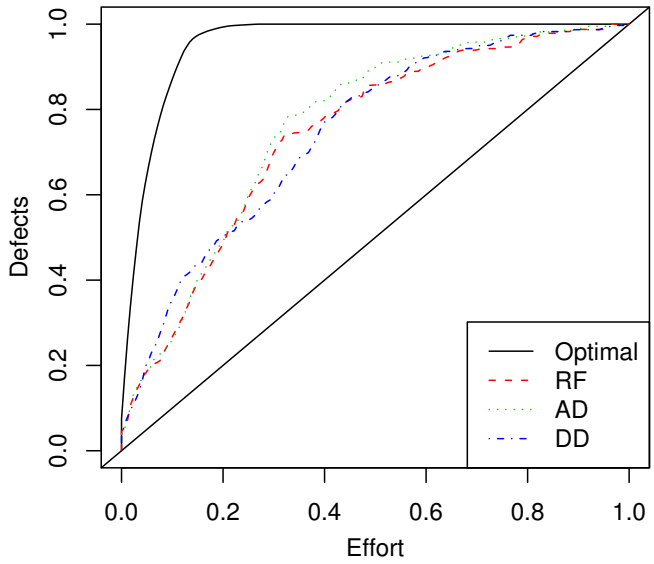

Figure 9. Cumulative Lift Chart for three classifiers and the optimal prediction on pc3.

at the lift chart for pc3 in Figure 9, we can see that the difference between all classifiers and the optimal prediction is still large. Another possible explanation for the results on pc3 and pc4 is the following: randomForest is an ensemble technique, using a voting of several (500 in our case) decision trees to predict R_{dd}. When we use rpart, which is the classifier used by randomForest, to predict R_{dd}, we were not able to achieve CE values above 0.0 for data sets pc3 and pc4, so that we assume that regression seems to be,

at least for rpart, harder on these data sets.[7] We did not yet find a difference in the data sets that might explain this, but assume that it is at least one of the reasons for the results on the pc data sets.

D. Threats to Validity

As every empirical study, ours is subject to some threats to validity. First of all, we cover only a small number of data sets from two different source, namely, NASA MDP and the Eclipse project. We cannot necessarily generalize to other data sets from the current study, since the characteristics of these data sets may not be representative. Additionally, we use only one classification algorithm and only static code metrics in this study. While Random Forest proved to be effective in the past, other algorithms may lead to different results. And previous work has shown that information about the history of files can lead to better defect prediction models. [33] Such data is not available for all data sets used in this study, thus an investigation of this aspect, together with a comparison of different classification algorithms, is left for future work.

Another threat is the chosen proxy for effort: Cyclomatic complexity may not be appropriate to measure the true effort associated with QA activities. On the one hand, this can be mitigated in future studies where detailed effort estimates for testing is available. On the other hand, assuming a uniform effort for all modules is even more unrealistic, and size plays an important role for testing effort, as shown by Bruntink et al. [31], [32].

When defect prediction models are used to optimize activities other than unit testing, such as system tests, a file-based effort estimation becomes less appropriate. Nevertheless, even for these types of treatment it is safe to assume that the effort is not uniformly distributed across the system, although we are not aware of studies relating file characteristics to testing effort in that case. Additionally, as pointed out by Leszak [29], a subsystem-based prediction model may be more appropriate for this usage scenario.

Finally, we have investigated defect data for single releases, or treated the data that way in case of Eclipse. In reality, for a multi-release software system, unit tests will accumulate over time, so that an approximation of testing effort based only on the cyclomatic complexity becomes less appropriate.

VI. Conclusions

In this paper, we have presented and evaluated two strategies to incorporate the treatment effort into defect prediction models. The first strategy, AD, is applicable to any probabilistic classifier, while DD is applicable only for regression algorithms.

[7]We use the default parameters of rpart, so tuning of parameters might improve the results.

In our evaluation we have shown that both strategies improve the predictive performance on fifteen publicly available data sets, both from a practical and a theoretical point of view. On some data sets, only our model DD offers a practically significant improvement over a random selection of modules.

In future work, we plan to investigate classifiers that include example-dependent costs, in our case testing effort, directly into the prediction process, such as the extensions to support vector machines by Brefeld et al. [41] and Geibel et al. [42]. Additionally, we want to combine our notion of effort-awareness with misclassification-cost aware classifiers. Finally, we are interested in investigating test-cost metrics applicable to different test scenarios, such as system or integration tests, to cover a broader usage area for defect prediction models.

References

[1] N. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, 2000.

[2] P. D. Turney, "Types of cost in inductive concept learning," *The Computing Research Repository*, vol. cs.LG/0212034, 2002.

[3] T. M. Khoshgoftaar and E. B. Allen, "Classification of fault-prone software modules: Prior probabilities, costs, and model evaluation," *Empirical Software Engineering*, vol. 3, no. 3, pp. 275–298, 1998.

[4] Y. Jiang and B. Cukic, "Misclassification cost-sensitive fault prediction models," in *Proc. of the 5th PROMISE*. New York, NY, USA: ACM, 2009, pp. 1–10.

[5] E. Arisholm and L. C. Briand, "Predicting fault-prone components in a java legacy system," in *International Symposium on Empirical Software Engineering*. New York, NY, USA: ACM, 2006, pp. 8–17.

[6] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom java software," in *Proc. of the 18th ISSRE*. IEEE Press, 2007, pp. 215–224.

[7] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, 2010.

[8] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proc. of the 5th PROMISE*, 2009.

[9] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 2007.

[10] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. of the 31st ICSE*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88.

[11] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, pp. 897–910, 2005.

[12] T. Zimmermann, N. Nagappan, and A. Zeller, *Predicting Bugs from History*. Springer, March 2008, ch. 4, pp. 69–88.

[13] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. of the 30th ICSE*. New York, NY, USA: ACM, 2008, pp. 531–540.

[14] C. Bird, N. Nagappan, P. T. Devanbu, H. Gall, and B. Murphy, "Does distributed development affect software quality? an empirical case study of windows vista," in *Proc. of the 31st ICSE*, 2009, pp. 518–528.

[15] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006.

[16] T. M. Khoshgoftaar, B. Cukic, and N. Seliya, "An empirical assessment on program module-order models," *Quality Technology and Quantitative Management*, vol. 4, no. 2, pp. 171–190, 2007.

[17] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[18] T. Mende, R. Koschke, and M. Leszak, "Evaluating defect prediction models for a large, evolving software system," in *Proc. of the 13th CSMR*. IEEE Press, 2009, pp. 247–250.

[19] P. Knab, M. Pinzger, and A. Bernstein, "Predicting defect densities in source code files with decision tree learners," in *Proc. of the Workshop on Mining software repositories*. New York, NY, USA: ACM, 2006, pp. 119–125.

[20] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. of the 27th ICSE*, 2005, pp. 284–292.

[21] ——, "Static analysis tools as early indicators of pre-release defect density," in *Proc. of the 27th ICSE*. New York, NY, USA: ACM, 2005, pp. 580–586.

[22] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.

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

[24] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for evaluating fault prediction models," *Empirical Software Engineering*, vol. 13, no. 5, pp. 561–595, 2008.

[25] Y. Jiang, B. Cukic, and T. Menzies, "Costs curve evaluation of fault prediction models," in *Proc. of the 19th ISSRE*. IEEE Press, 2008, pp. 197–206.

[26] T. M. Khoshgoftaar and E. B. Allen, "Ordering fault-prone software modules," *Software Quality Journal*, vol. 11, no. 1, pp. 19–37, 2003.

[27] C. Elkan, "The foundations of cost-sensitive learning," in *Proceedings of the 17th international joint conference on artificial intelligence*, 2001, pp. 973–978.

[28] P. Domingos, "Metacost: A general method for making classifiers cost sensitive," in *Proc. of the 5th International Conference on Knowledge Discovery and Data Mining*, 1999, pp. 155–164.

[29] M. Leszak, "Software defect analysis of a multi-release telecommunications system," in *Proc. of the 6th International Conference on Product Focused Software Process Improvement*, 2005, pp. 98–114.

[30] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[31] M. Bruntink and A. van Deursen, "Predicting class testability using object-oriented metrics," in *Proc. of the 4th IEEE International Workshop on Source Code Analysis and Manipulation*, 2004, pp. 136–145.

[32] ——, "An empirical study into class testability," *Journal of Systems and Software*, vol. 79, no. 9, pp. 1219–1232, 2006.

[33] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. of the 3rd PROMISE*, May 2007.

[34] M. H. Halstead, *Elements of Software Science*. New York, NY, USA: Elsevier Science Inc., 1977.

[35] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[36] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2008.

[37] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R News*, vol. 2, no. 3, pp. 18–22, 2002.

[38] H. Zhang and X. Zhang, "Comments on "data mining static code attributes to learn defect predictors"," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 635–637, 2007.

[39] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'"," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 637–640, 2007.

[40] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *Journal of the American Statistical Association*, vol. 32, pp. 675–701, 1937.

[41] U. Brefeld, P. Geibel, and F. Wysotzki, "Support vector machines with example dependent costs," in *14th European Conference on Machine Learning*, 2003, pp. 23–34.

[42] P. Geibel, U. Brefeld, and F. Wysotzki, "Learning linear classifiers sensitive to example dependent and noisy costs," in *5th International Symposium on Intelligent Data Analysis*, 2003, pp. 167–178.