Revisiting the Evaluation of Defect Prediction Models

Thilo Mende, Rainer Koschke
University of Bremen, Germany
http://www.informatik.uni-bremen.de/st/
{tmende,koschke}@informatik.uni-bremen.de

ABSTRACT

Defect Prediction Models aim at identifying error-prone parts of a software system as early as possible. Many such models have been proposed, their evaluation, however, is still an open question, as recent publications show.

An important aspect often ignored during evaluation is the effort reduction gained by using such models. Models are usually evaluated per module by performance measures used in information retrieval, such as recall, precision, or the area under the ROC curve (AUC). These measures assume that the costs associated with additional quality assurance activities are the same for each module, which is not reasonable in practice. For example, costs for unit testing and code reviews are roughly proportional to the size of a module.

In this paper, we investigate this discrepancy using optimal and trivial models. We describe a trivial model that takes only the module size measured in lines of code into account, and compare it to five classification methods. The trivial model performs surprisingly well when evaluated using AUC. However, when an effort-sensitive performance measure is used, it becomes apparent that the trivial model is in fact the worst.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Complexity measures, Performance measures, Product metrics*

Keywords

Defect Prediction, Cost-Sensitive Performance Measures

1. INTRODUCTION

Defect Prediction Models aim at identifying error-prone parts of a software system as early as possible, so that these parts can be scheduled for additional quality assurance activities, such as testing or code review. Many such models have been proposed, for example, based on static code metrics, change information, or fault history.

The evaluation of defect prediction models is still an open debate, as recent publications show [16, 11]. Often, performance measures from machine learning or information retrieval are used, but as pointed out by Arisholm et al. [1], these measures may not represent the cost effectiveness of the prediction models. One reason for this discrepancy is a skewed distribution of source code: a small percentage of modules may contain a large portion of the code, at least when measured in lines of code. However, costs of quality assurance treatment are often not fixed per module, but dependent upon the size of the module.

In a recent paper, we proposed a performance measure that takes these considerations into account and assesses defect prediction models by comparing them to the optimal performance of an imaginary best classifier [20]. However, this metric has been empirically validated only on confidential industrial data but not on publicly available data sets yet.

Contributions. In this paper, we investigate the usability of this performance measure on publicly available defect data sets from the NASA metrics data program (MDP)[1]. First, we evaluate a trivial classifier based only on the size – measured in lines of code – using classical performance measures. This classifier performs surprisingly well. Afterwards, we compare five different classification algorithms and our trivial model on several data sets, using a traditional performance measure and our newly proposed one. Our trivial model performs surprisingly well when evaluated using the former, while our newly proposed metric identifies it as an insufficient model.

Overview. The remainder of this paper is organized as follows: First, we discuss related work in Section 2. Afterwards, in Section 3, we describe a trivial model and evaluate its performance briefly. In Section 4, we compare our trivial model with several data mining algorithms on thirteen data sets from the NASA MDP. Finally, Section 5 concludes.

2. 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 tech-*

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

nique 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, reuse information, or churn metrics.

Most of the early work in defect prediction used proprietary data, which made comparisons of different approaches and performance results difficult. With the availability of public defect data sets from the NASA MDP or the PROMISE repository[2], this situation has improved. Comparisons of different classifications algorithms [22], different input attributes [14], or the influence of input data transformation [12] can now be published in a repeatable way.

However, the evaluation of defect prediction models is still a debated topic as the discussion between Zhang et al. [30] and Menzies et al. [21], or the recent work by Lessmann et al. [16] and Jiang et al. [11] show.

One of the main problems is to find suitable performance measures. Scalar values, such as accuracy, precision, or recall that are derived from a confusion matrix are commonly used in information retrieval, and have been used in defect prediction as well. It is well known that accuracy is a bad performance measure for imbalanced data [29] – when one class is much less likely than the other – and thus not suited for defect predictors [7, 18]. Other performance measures derived from the confusion matrix, such as precision and recall, are more appropriate for imbalanced data. Recall is defined as the ratio of detected defective modules among all defective modules, while precision is the ratio of actually defective modules within the modules predicted as defective. These measures are sensible only in combination – increasing one of them independently usually decreases the other. While recall is often used and seems to be generally accepted, precision is not, as the discussion between Zhang et al.[30], and Menzies et al.[21] shows: On the one hand, high precision is desirable to achieve models that are cost effective. But since precision is also influenced by the ratio of defective to non-defective files (*pos/neg ratio*), a comparison of recall and precision across data sets with different pos/neg ratios is difficult.

Additionally, the underlying classification technique is often a *probabilistic* classifier, assigning scores instead of class labels to each observation [8]. It can be turned into a binary classifier by user-defined thresholds, but then measures based on the confusion matrix are assessing only the performance of one specific threshold. One way to evaluate probabilistic classifiers are receiver operating characteristic (ROC) curves. These ROC curves plot the probability of detection on the y-axis and probability of false alarm on the x-axis. According to Fawcett[8], ROC curves are not sensitive to changes in the pos/neg ratio, which makes them particularly suited to compare classifiers over different data sets[16]. A scalar performance measure derived from ROC curves is the area enclosed by the curve and the x-axis. This scalar is known as *area under curve (AUC)*. A perfect classifier has AUC=1, while a random classifier is expected to achieve AUC=0.5. As a scalar value, AUC is well suited to compare the performance of different classifiers, and is often used for that purpose.

An advantage of probabilistic classifiers is that the scores can be used to rank modules according to their (predicted) fault proneness, and thus support prioritizing their treatment. This approach is, for example, advocated by Ohlsson et al. [23], Ostrand et al. [24] or Khoshgoftaar et al. [15], and coined Module-Order-Model (MOM) by the latter. It enables to select a fixed percentage of modules for further treatment – a more realistic scenario for projects with a fixed quality assurance budget. MOMs can be evaluated by assessing which percentage of defects is detected at fixed percentages of modules. For example, Ostrand et al. [24] found up to 83% of the defects in 20% of the files

One way to graphically evaluate MOMs are lift charts, sometimes known as Arberg diagrams [23]. They are created by ordering modules according to the score assigned by a prediction model, and denoting for each ratio of modules on the x-axis which cumulative ratio of defects has been identified on the y-axis. Thus for any selected percentage of modules, one can easily identify the percentage of correctly predicted defective modules.

Two recent papers specifically address the evaluation of defect prediction models, and are thus 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[5]: 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 4.

Lessmann et al. [16] identify the need for a common evaluation framework for defect prediction models. They propose to use AUC 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. [11] 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 [13]. 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.

These imbalanced test costs for modules are an important aspect that is often neglected during the evaluation, as pointed

[2]http://www.promisedata.org

out by Arisholm et al. [1]: The costs of treatment of modules predicted as defective, for example, by testing or manual inspection, is not the same for all modules, but roughly proportional to the size of a module, for example measured in lines of code. They propose to use a variation of lift charts where the x-axis contains the ratio of lines of code instead of modules, and evaluate various data mining algorithms by measuring the performance improvement provided by a classification technique over a random selection of modules. In a recent paper, we have extended this approach to take the characteristics of the data set into account by measuring the deviation from an optimal model [20]. The resulting performance measure is described in Section 3.

The aspect of cost effectiveness is also investigated by Ostrand et al. [25]. They define four different models and again consider the upper 20% of the files as defective. One of their models is a trivial one, ordering files solely by their size measured in lines of code. They note that this model performs surprisingly well. Contrary to their previous publications, they also record the percentage of the source code measured in lines of code. For all models, the 20% of the files predicted as defective contain much more than 20% of the source lines, concretely, between 59.5% and 69.9%. For the trivial model, the percentage of defects found is even smaller than the percentage of source lines predicted as defective. However, they argue that testing, especially integration and system testing, is not closely related to the number of lines, and thus evaluating on the file level is superior.

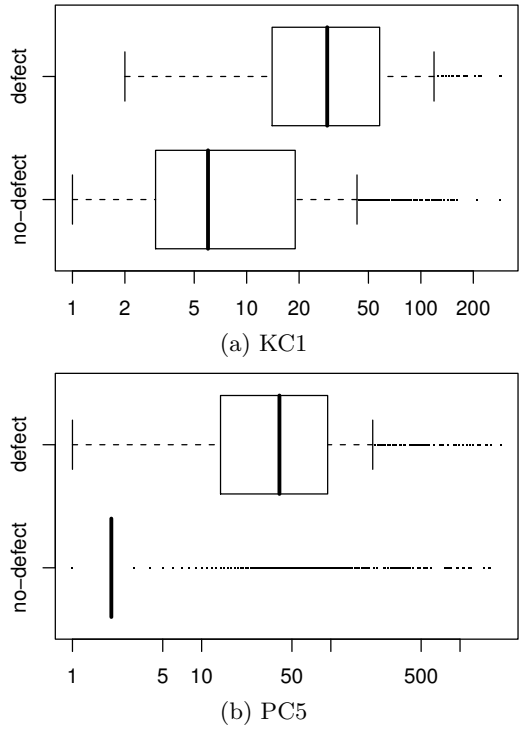


Figure 1: Log-scaled distribution of Lines of Code for data sets KC1 and PC5.

3. ISSUES OF MODULE-BASED EVALUATIONS

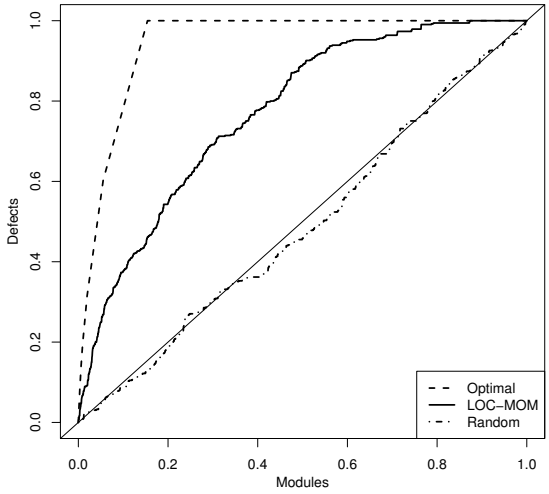
As we have described in the previous section, module-order models are a realistic approach to use defect prediction models in practice, since the budget for validation and verification is typically limited. However, as argued by Arisholm et al. [1], the costs of testing or reviewing a module are not equally distributed, but depend to some extent on the size of a module. Most of the traditional evaluation procedures ignore module size, and in Section 3.1 we show how a classifier can exploit that.

We use two data sets from NASA MDP, since this repository has become one of the standard means to evaluate defect prediction models. Ma and Cukic have observed that many of the defective modules within the MDP repository are small if measured in lines of code (LoC) [18]. This can be seen in Figure 1 as well, where we use boxplots to show the distribution of LoC for defective and non-defective modules in data sets KC1 and PC5 from NASA MDP. As we can see, for both data sets, non-defective modules tend to be smaller than defective ones, although this is much more extreme for PC5 than for KC1. This information can be used to build a simple classifier that orders modules just by decreasing LoC. We use LoC-MOM (lines-of-code based module order model) to refer to this prediction model. Such a model was also used by Ostrand et al. [25] and performed surprisingly well when evaluated on the module level. In the next section, we investigate how our simple model performs when the size is ignored, while Section 3.2 uses a performance measure taking the module size into account.

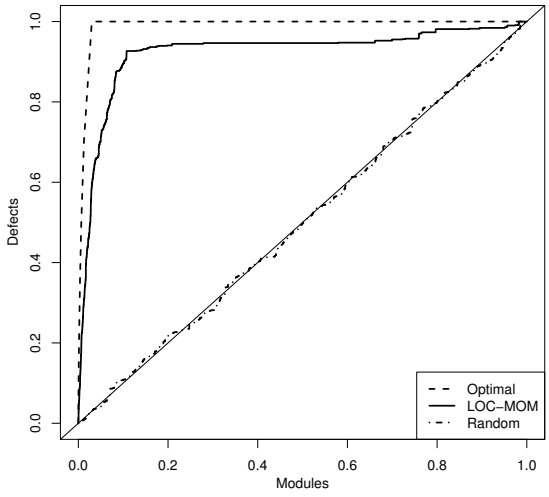
3.1 Module-based Evaluation of a LoC-MOM

The cumulative lift charts in Figure 2 depict the performance of three MOMs, namely an optimal, a trivial, and a random ordering for two data sets, KC1 and PC5, when the module size is ignored. The optimal model is created by ordering modules according to their defect density. Defect prediction models should be as close as possible to this optimal model, and thus it may serve as a benchmark. A random classifier, created by assigning random scores to each module, is depicted as the lower bound. As we can see, our trivial model LoC-MOM performs surprisingly well on both data sets. Although there is still room for improvements compared to the optimal model, it clearly outperforms the random ordering. When considering 20% of the files as defective, as Ostrand et al. [24] do, LoC-MOM is able to identify around 55% of the defects in KC1 and over 90% in PC5.

LoC-MOM performs much better on data set PC5 than on KC1. This effect can be explained by data provided by Jiang et al. [11]: They observed that for many of the MDP data sets, fault-free modules tend to be short in terms of lines of code (LoC). They calculated the 90th percentile of LoC for defective and non-defective files. The 90th percentile is the value for LoC so that 90% of the modules are shorter than that value. For KC1, 90% of the fault-free modules are smaller than 42 lines, while for PC5, 90% are smaller than 7 lines. It seems that the trivial model performs better for data sets where the 90th percentile of LoC for fault-free files is lower. This can explain an observation we made in a recent study: We found out that different prediction models consistently performed better when header files where included [20]. Since header files are usually much shorter than



(a) Module-based KC1



(b) Module-based PC5

Figure 2: Module-based Cumulative Lift Charts of three models for two data sets from NASA MDP: KC1 and PC5.

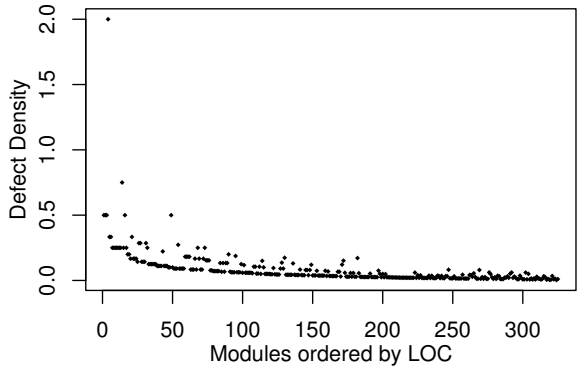


Figure 3: Lines of Code vs. defect density for defective modules in KC1.

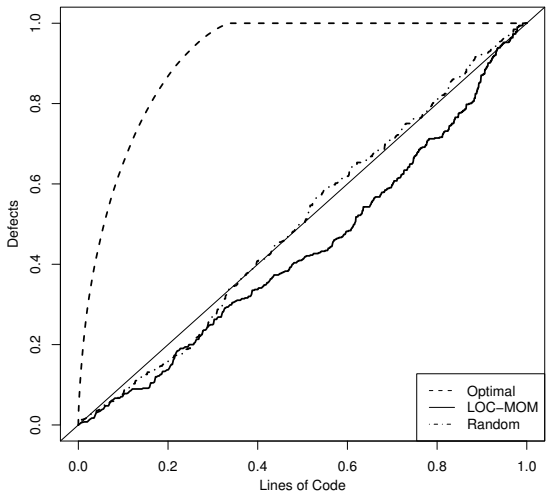
their corresponding implementation files, and at least in our case, header files contained fewer defects, the 90th percentile for fault-free files was lower when we included header files. Thus when selecting the files addressed by a defect prediction model, this effect has to be considered.

The good performance of LoC-MOM on the module level does not necessarily mean that it is usable in practice, or that one should focus validation and verification activities on large files first. Ideally, a defect prediction model identifies modules with a high defect density first. The distribution of defect density in Figure 3 indicates that an ordering of files just by decreasing file size does not create an optimal model. In contrary, many files with high defect density are rather short in terms of LoC.

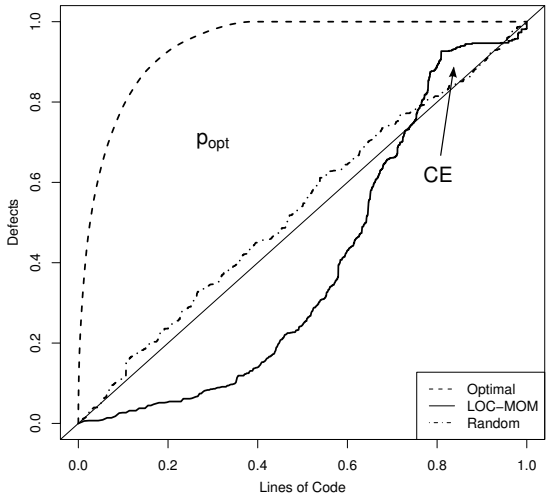
3.2 LoC-based Evaluation of a LoC-MOM

As we have seen in the previous section, our LoC-MOM performs surprisingly well when evaluated on the module level. However, a useful defect prediction model has to outperform the random inspection of source code, as pointed out by Arisholm et al. [1]. They calculate a cost effectiveness estimation based on the assumption that there is a relationship between the effort to inspect or test a file and its size, and that, on average, a random selection of $n\%$ of the source lines contains $n\%$ of the defects. In that case, a defect prediction model is cost-effective when the files predicted as defective contain a larger percentage of defects than their percentage of lines of code. Their performance measure CE can be calculated in a LoC-based cumulative lift chart by calculating the area under the prediction model's curve which lies above a line of slope one. The latter represents the average performance of a random selection of source lines. This area is marked with CE in the LoC-based cumulative lift chart in Figure 4(b).

CE was defined to assess the cost effectiveness of prediction models and not to compare the performance of different classification algorithms, so that negative values for the cost effectiveness are not defined. Additionally, CE ignores the actual defect distribution inside a system. It reports how much better than a random model a predictor is, but it does not tell anything about how close the predictor comes to an optimal model.



(a) LoC-based KC1



(b) LoC-based PC5

Figure 4: LoC-based Cumulative Lift Charts of three models for two data sets from NASA MDP: KC1 and PC5.

Therefore we extended the idea of CE and proposed a new performance evaluation measure p_{opt} by comparing a prediction model with an optimal model [20]. An optimal model would be created as follows: We order all modules by decreasing defect density (and increasing lines of code, in case of ties), and predict the modules with highest defect densities first. When we use our prediction model to order modules according to their score (and use lines of code as a tie breaker again), and plot the result in the same way, we can investigate how well our model performs compared to the optimal model. When we do this for our trivial model, using lines of code as a surrogate for treatment effort, as depicted in Figure 4(a) and Figure 4(b), we can see that the trivial model performs similar to or worse than a random selection of modules.

Similar to the AUC metric for ROC curves, we define Δ_{opt} as the area between the optimal model and the predicted model, which results in a scalar value, where a higher value means a greater difference between the optimal and our predicted model. To get a performance measure that ranks better classifier with higher numerical values than worse classifiers, we define $p_{opt} = 1 - \Delta_{opt}$. The corresponding area is marked p_{opt} in Figure 4(b). This measure has two desirable properties: It takes costs associated with testing or reviewing a module into account, and it considers the actual distribution of faults by benchmarking against a theoretically possible optimal model.

Ostrand et al. state that lines of code may be valid to estimate effort for unit tests, but doubt that it reflects the effort required to test files during integration or system test, and thus conclude that using LoC to assess the cost effectiveness of defect prediction models is not valid [25]. Yet, quite likely cost effectiveness for integration and system test is not totally independent from lines of code. Additionally, it is an important factor for code inspections.

At any rate, we can proceed with the assumption that cost effectiveness is not just a function of the pure number of modules, since testing effort is likely not the same for all modules. Our evaluation measure p_{opt} can be adjusted accordingly by using a more appropriate surrogate measure for integration testing effort that future research comes up with. In the mean time, we argue that LoC is at least superior to a module-based evaluation, as the overly optimistic results for LoC-MOM in Section 3 show.

The difference between CE and p_{opt} is the baseline a prediction model is compared to: For CE, it is a random ordering, while for p_{opt}, it is an optimal ordering. In Section 4.4, we evaluate whether the two metrics actually measure the same, or whether there are differences.

4. COMPARING THREE PERFORMANCE MEASURES

In the last section, we have seen that our model LoC-MOM performs surprisingly well, at least on two MDP data sets. In this section, we evaluate how LoC-MOM performs compared to popular classification algorithms, where the performance is assessed using AUC, p_{opt}, and CE. The goal is to compare the alternative means to evaluate predictors based on their judgement on a trivial predictor versus advanced classifiers.

Name	Modules	% Faulty
KC1	2107	15.42
KC2	522	20.50
KC3	458	9.39
KC4	125	48.80
JM1	10878	19.32
PC1	1107	6.87
PC2	5589	0.41
PC3	1563	10.24
PC4	1458	12.21
PC5	17186	3.00
CM1	505	9.50
MC2	161	32.30
MW1	403	7.69

Figure 5: The thirteen data sets from NASA MDP used for evaluation.

The experimental setup is described in Section 4.1. The evaluation results for AUC and p_{opt} are presented in Section 4.2 and Section 4.3, respectively. In Section 4.4, we compare p_{opt} with Arisholm et al.'s performance measure CE assessing the cost effectiveness [1]. Threats to validity are discussed in Section 4.5.

4.1 Experimental Setup

The experimental setup closely follows Lessmann et al. [16] and Jiang et al. [11] in the selection of data sets, algorithms, and evaluation methodology.

Data Sets: We use thirteen data sets from NASA MDP shown in Figure 5. These represent the union of data sets used by Lessmann et al. and Jiang et al. A detailed description of these systems can be found elsewhere [11, 16]. For all data sets, we calculate our dependent variable as a binary classification: *defective* for files where $ERROR\_COUNT > 0$, and *non-defective* otherwise. We use all input attributes available for each data set as independent variables, except the module identifier and metrics related to error count and density. These attributes include static code metrics, such as Halstead's[10] or McCabe's [19] complexity measures.

Algorithms: Our selection of data mining algorithms closely resembles the choice by Jiang et al. and can be found in Figure 6. All algorithms were implemented in version 2.8.1 of R[27]. Jiang et al. additionally used a nearest neighbor clustering algorithm, but the R implementation *knn* was not able to make predictions for all data sets because of too many ties, so we excluded it. All algorithms are used with their default parameters, so further tuning might improve their results. Additionally, we use the trivial classifier LoC-MOM described in Section 3.1.

Evaluation: We use ten times ten-fold cross-validation to train and test all algorithms on all data sets. All algorithms use the same partitioning. As performance measures, we use averaged AUC in Section 4.2, averaged p_{opt} in Section 4.3, and averaged CE in Section 4.4.

Comparing classifiers by just comparing scalar performance measures may be misleading due to inherent variance, so

Name	Description
NB	A naive Bayes classifier from R package e1071[6], naively assuming independent input variables to calculate conditional class probabilities using Bayes Rule.
Logistic	A logistic regression model build using the R function `glm` with parameter `family=binomial("logit")`.
rpart	An implementation of the CART decision tree learner [4] in R package rpart.
Bag	An implementation of the ensemble algorithm *bagging* (Bootstrap aggregating) [2] using bootstrap samples of the training set to build multiple models and average their response. We use the implementation in package *ipred*[26] with rpart as its base classifier.
RF	An implementation of the random forest algorithm [3] using a majority voting of 500 decision trees in package *randomForest*[17].

Figure 6: The six classification algorithms used for evaluation.

statistical hypothesis tests are necessary. One approach, described by Demšar[5], 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 [9] 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[5], 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:

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

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 $k = 6$ and $\alpha = 0.05$, $q_{0.05} = 2.85$.

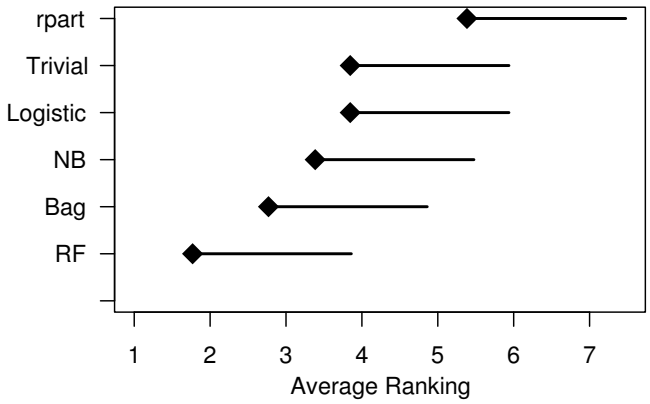


Figure 8: Nemenyi's Critical-Difference Diagram for the evaluation using AUC.

We use Lessmann et al.'s [16] 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.

Implementation: All our analyses are implemented in version 2.8.1 of the statistical package R[27]. We use the R package *ROCR*[28] to calculate AUC values.

4.2 AUC Results

The AUC values for all classifiers and data sets can be found in Figure 7. The ordering of classifiers by average rank is the same as those of Jiang et al. [11], and the values are similar to many values given by Lessmann et al. [16].

The Friedman test can be calculated using the formulas described above and yields $F_F = 8.66$. The critical value for the F-Distribution and $\alpha = 0.05$ with 5 and 60 degrees of freedom is 2.368, so the null hypothesis that all classifiers perform equally well can be rejected. Nemenyi's critical difference can be calculated as $CD = 2.09$. The pairwise comparison using Nemenyi's *CD* can be found in Figure 8. Random Forests is the best algorithm, although according to Nemenyi's test, only the difference to rpart is significant.

What is most interesting, however, is the good performance of our trivial classifier: For most of the data sets, its AUC value is well above 0.7 and reaches up to 0.93. Furthermore, it ranks, on average, equal to the logistic regression classifier and quite closely to Naive Bayes, and is among the best classifiers for three data sets. This indicates that an evaluation of defect predictors based on a module-based measure such as AUC alone may produce misleading results. In case AUC is used anyway, one should at least report results for LoC-MOM, since it is easy to obtain and may serve as a lower bound.

4.3 p_{opt} Results

When we perform the same evaluation and use p_{opt} as the underlying performance measure, our results are quite different. The detailed p_{opt} values and the average ranking per classifier can be found in Figure 9. The Friedman test yields

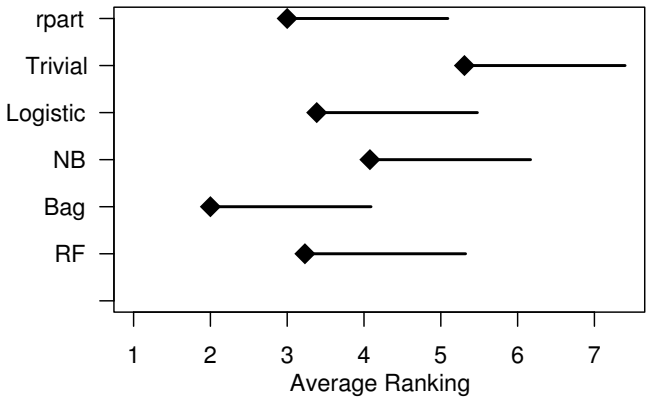


Figure 10: Nemenyi's Critical-Difference Diagram for the evaluation using p_{opt}.

$F_F = 6.56$, again our critical value is 2.368, so we can reject the null hypothesis that there is no significant performance difference between our classifiers.

The results of the pairwise post-hoc test can be found in Figure 10. Using this metric, our trivial classifier performs, as expected, worst among all classifiers. But the ordering of the other algorithms changes as well: Random Forest is no longer the best, and the ranking of rpart is much better than in Section 4.2. A similar effect was observed by Arisholm et al. [1] where a tree-based algorithm performed favorable when lines of code where taken into account.

A visualization of the performance of three different classifiers compared to the optimal model can be found in Figure 11. For PC1, rpart and Bagging perform quite similarly, while Random Forest performs even worse than a random selection of source lines. However, p_{opt} is rather low for all three classifiers, leaving much room for improvement. For data set PC4, the situation is different: all three classifiers achieve quite good performance, which is reflected in a higher p_{opt} value. We have not yet investigated which data set characteristics influence this difference.

4.4 Cost Effectiveness Results

Arisholm et al. [1] have proposed to evaluate defect prediction models based on a measure *cost effectiveness* (CE), which inspired the creation of p_{opt}. The two measures are very similar, however, aim at different goals: Our goal is to be able to compare classifiers in the absolute sense, so we take the optimal model for each data set into account. Their goal for CE is to assess whether a model can be applied cost effectively. They assume that, on average, a random inspection of n% lines of code should be able to find n% of the defects. In a LoC based cumulative lift chart, this random selection yields approximately a line of slope one, and they calculate CE by computing the area above this line that is enclosed by the curve of a prediction model. This area is labeled CE in Figure 4(b).

In this section, we calculate CE for our six classifiers and thirteen data sets for two reasons: On the one hand, we want to investigate which algorithms are able to build cost effective prediction models for which MDP data sets. On

	KC1	KC2	KC3	KC4	JM1	PC1	PC2	PC3	PC4	PC5	CM1	MC2	MW1	AR
NB	0.79	**0.84**	**0.81**	0.75	0.69	0.71	0.86	0.76	0.85	0.94	**0.77**	0.72	**0.80**	3.38
Logistic	0.81	0.82	0.69	0.77	0.71	0.84	0.83	0.82	0.91	0.95	0.75	0.68	0.66	3.85
rpart	0.69	0.76	0.65	**0.84**	0.52	0.68	0.51	0.72	0.89	0.82	0.70	0.62	0.69	5.38
Bag	0.82	0.83	0.71	0.82	0.74	0.77	0.73	0.82	0.93	0.96	**0.77**	0.73	0.75	2.77
RF	**0.84**	**0.84**	0.75	0.83	**0.75**	**0.86**	**0.90**	**0.86**	**0.95**	**0.97**	0.75	**0.75**	0.71	1.77
Trivial	0.79	**0.84**	**0.81**	0.47	0.72	0.71	0.84	0.75	0.75	0.93	**0.77**	0.66	0.79	3.85

Figure 7: Performance of all classifiers on thirteen data sets measured using AUC and average rank (AR) per classifier.

	KC1	KC2	KC3	KC4	JM1	PC1	PC2	PC3	PC4	PC5	CM1	MC2	MW1	AR
NB	0.55	0.60	0.58	0.75	0.62	0.56	0.54	0.67	0.77	0.54	0.53	0.54	0.70	4.08
Logistic	0.63	0.63	0.66	0.78	0.57	0.53	0.42	0.61	0.89	0.64	**0.68**	**0.71**	0.58	3.38
rpart	0.57	**0.65**	0.66	**0.86**	0.34	0.64	**0.79**	0.67	0.83	0.64	0.63	0.56	0.63	3.00
Bag	**0.67**	0.62	**0.67**	0.82	0.62	**0.71**	0.72	**0.73**	0.89	**0.73**	0.65	0.66	**0.72**	2.00
RF	**0.67**	0.60	0.63	0.83	**0.63**	0.54	0.55	0.60	**0.90**	0.71	0.61	0.70	0.56	3.23
Trivial	0.53	0.57	0.56	0.41	0.54	0.54	0.56	0.50	0.59	0.47	0.51	0.49	0.66	5.31

Figure 9: Performance of all classifiers on thirteen data sets measured using p_{opt} and average rank (AR) per classifiers.

the other hand, we want to compare p_{opt} and CE and check whether these are actually different metrics.

The CE values are provided in Figure 12. As we can see, most of them are rather low, indicating only a small benefit provided by the prediction models. For four data sets, no classifier achieves $CE > 0.10$. This means that for these data sets, a random selection of source lines performs not much worse than any of our defect prediction model.

To compare CE and p_{opt}, we calculate Spearman's correlation coefficient ρ between 78 models (six classifiers $\times$ thirteen data sets) once evaluated by CE, once by p_{opt}. The relatively high value of $\rho = 0.863$ indicates a strong similarity between both metrics. This can also be observed from Figure 9 and Figure 12 by comparing which classifier performs best on each data set: except for one data set (JM1), the best classifier according to p_{opt} is also the best according to CE.

Nevertheless, p_{opt} and CE measure slightly different things, and their difference depends on the distribution of faults inside a data set: When the area under the optimal model is close to 1, and the predictor is always performing better than a random selection, p_{opt} and CE differ just by 0.5. When defects are distributed across the system, or the predictor performs sometimes better, sometimes worse than the random model, the two measures become different. We conclude that both metrics are best-suited for their purpose: p_{opt} provides a fair evaluation of a predictors performance, while CE offers an insight into the cost effectiveness of a model on one data set.

4.5 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 one specific source, namely, NASA MDP. We cannot necessarily generalize to other data sets from the current study, since the characteristics of these data sets may not be representative.

Nevertheless, our study shows that a module-based evaluation leads to surprising results, and thus measures taking the amount of source code predicted as defective into account, as our measure p_{opt} does, are much more desirable. This is definitely true for the data sets analyzed here. Additionally, our study shows that the performance of a classifier cannot be assessed by looking just at AUC values: For some of the NASA data sets, these are high even for our trivial model.

The biggest threat to p_{opt} is that lines of code may not be an appropriate surrogate measure for treatment costs. However, as described in Section 3.2, it can be easily adopted to more appropriate measures. As long as no other more appropriate measure is known and available, we recommend to use measures taking the size of modules into account, because otherwise performance of classifiers is too dependent on the fault distribution. Additionally, the results in Section 3.1 and Section 4.2 show that module-based evaluations are too optimistic.

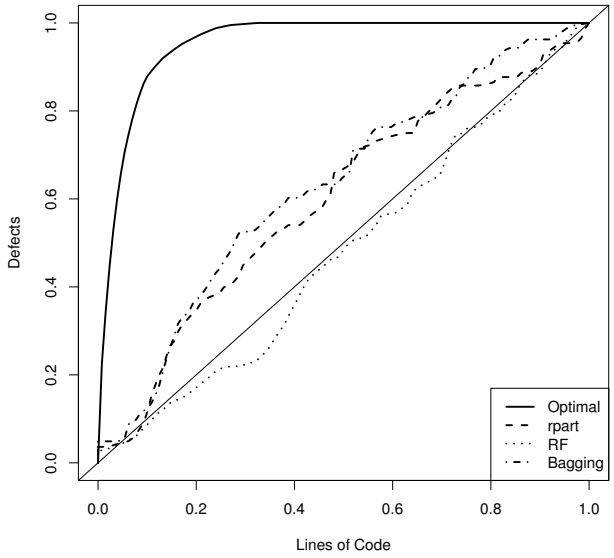
5. CONCLUSION

In this paper, we evaluated a trivial defect prediction model based only on the size of modules measured in LoC on thirteen data sets from the NASA MDP. This model performs surprisingly well when evaluated using AUC to assess predictive performance, and we were not able to show statistical significant differences to some advanced data mining algorithms.

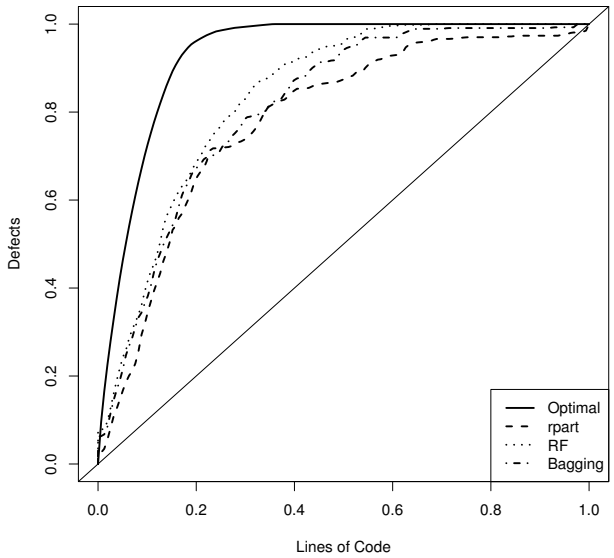
When the same model is evaluated using our proposed performance measure p_{opt} that takes the size of the modules into account, it becomes apparent that the trivial model is in fact not better or even worse than a random selection of source files. Additionally, it reveals that on many data sets, all algorithms considered here are far from an optimal prediction. This result indicates the need for further research to improve existing prediction models, not only by more sophisticated classification algorithms, but also by searching for better independent variables.

	KC1	KC2	KC3	KC4	JM1	PC1	PC2	PC3	PC4	PC5	CM1	MC2	MW1	AR
NB	0.007	0.013	0.059	0.128	**0.054**	0.055	0.097	0.129	0.195	0.014	0.038	0.025	0.153	4.000
Logistic	0.044	0.025	0.090	0.153	0.001	0.044	0.041	0.076	0.318	0.077	**0.133**	**0.095**	0.062	3.308
rpart	0.014	**0.041**	0.118	**0.221**	—	0.107	**0.287**	0.118	0.259	0.082	0.103	0.026	0.122	2.885
Bag	0.078	0.023	**0.122**	0.180	0.025	**0.168**	0.217	**0.182**	0.316	**0.168**	0.106	0.067	**0.161**	2.000
RF	**0.082**	0.016	0.072	0.198	0.030	0.041	0.051	0.075	**0.325**	0.152	0.085	0.087	0.059	3.308
Trivial	0.004	0.007	0.050	0.001	—	0.039	0.117	0.013	0.039	0.001	0.034	0.013	0.129	5.500

Figure 12: Cost Effectiveness according to Arisholm et al. and average rank (AR) per classifier on thirteen data sets.



(a) PC1



(b) PC4

Figure 11: LoC-based cumulative lift chart for three classifiers on two data sets.

Interestingly, the worst algorithm according to AUC is quite good according to p_{opt}, namely rpart. We currently do not have an explanation for this, but a similar behavior was observed by Arisholm et al. [1], where rather simple algorithms performed better under a changed performance measure.

We conclude that performance measures should always take into account the percentage of source code predicted as defective, at least for unit testing and code reviews. If better cost metrics for specific quality assurance techniques are known, it is easy to modify p_{opt} accordingly. And even when module-based metrics are preferred, at least the trivial classifier LoC-MOM should be assessed in order to be able to "normalize" the resulting performance values.

In future work, we want to investigate whether cost-sensitive learning algorithms can be used to build better defect prediction models. Additionally, we need more appropriate surrogate measures for treatment costs, since we expect that our approach of using LoC leaves room for improvements.

Acknowledgements

This work was performed as part of the project ArQuE, which is partially funded by the German Ministry of Education and Research (BMBF) under grant number 01 IS F14.

6. REFERENCES

[1] E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *ISSRE '07: Proceedings of the 18th IEEE International Symposium on Software Reliability Engineering*, pages 215–224. IEEE Press, 2007.

[2] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[3] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[4] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

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

[6] E. Dimitriadou, K. Hornik, F. Leisch, D. Meyer, and A. Weingessel. e1071: Misc functions of the department of statistics (e1071), TU Wien, 2009. R package version 1.5-19.

[7] K. E. Emam, S. Benlarbi, N. Goel, and S. N. Rai. Comparing case-based reasoning classifiers for predicting high risk software components. *Journal of Systems Software*, 55(3):301–320, 2001.

[8] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

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

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

[11] Y. Jiang, B. Cukic, and Y. Ma. Techniques for evaluating fault prediction models. *Empirical Software Engineering*, 13(5):561–595, 2008.

[12] Y. Jiang, B. Cukic, and T. Menzies. Can data transformation help in the detection of fault-prone modules? In *DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems*. ACM, 2008.

[13] Y. Jiang, B. Cukic, and T. Menzies. Costs curve evaluation of fault prediction models. In *ISSRE'08: Proceedings of the 19th International Symposium on Software Reliability Engineering*, pages 197–206. IEEE Press, 2008.

[14] Y. Jiang, B. Cukic, T. Menzies, and N. Bartlow. Comparing design and code metrics for software quality prediction. In *PROMISE '08: Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 11–18. ACM, 2008.

[15] T. M. Khoshgoftaar and E. B. Allen. Ordering fault-prone software modules. *Software Quality Journal*, 11(1):19–37, 2003.

[16] 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*, 34(4):485–496, 2008.

[17] A. Liaw and M. Wiener. Classification and regression by randomForest. *R News*, 2(3):18–22, 2002.

[18] Y. Ma and B. Cukic. Adequate and precise evaluation of quality models in software engineering studies. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. IEEE Press, 2007.

[19] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[20] T. Mende, R. Koschke, and M. Leszak. Evaluating defect prediction models for a large, evolving software system. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pages 247–250. IEEE Press, 2009.

[21] 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*, 33(9):637–640, 2007.

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

[23] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.

[24] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[25] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Automating algorithms for the identification of fault-prone files. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 219–227, New York, NY, USA, 2007. ACM.

[26] A. Peters and T. Hothorn. *ipred: Improved Predictors*, 2008. R package version 0.8-6.

[27] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[28] T. Sing, O. Sander, N. Beerenwinkel, and T. Lengauer. ROCR: visualizing classifier performance in R. *Bioinformatics*, 21(20):3940–3941, 2005.

[29] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 1st edition, May 2005.

[30] H. Zhang and X. Zhang. Comments on "data mining static code attributes to learn defect predictors". *IEEE Transactions on Software Engineering*, 33(9):635–637, 2007.