Replication of Defect Prediction Studies: Problems, Pitfalls and Recommendations

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

ABSTRACT

Background: The main goal of the PROMISE repository is to enable reproducible, and thus verifiable or refutable research. Over time, plenty of data sets became available, especially for defect prediction problems.
Aims: In this study, we investigate possible problems and pitfalls that occur during replication. This information can be used for future replication studies, and serve as a guideline for researchers reporting novel results.
Method: We replicate two recent defect prediction studies comparing different data sets and learning algorithms, and report missing information and problems.
Results: Even with access to the original data sets, replicating previous studies may not lead to the exact same results. The choice of evaluation procedures, performance measures and presentation has a large influence on the reproducibility. Additionally, we show that trivial and random models can be used to identify overly optimistic evaluation measures.
Conclusions: The best way to conduct easily reproducible studies is to share all associated artifacts, e.g. scripts and programs used. When this is not an option, our results can be used to simplify the replication task for other researchers.

Categories and Subject Descriptors

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

Keywords

Defect Prediction Model, Replication

1. INTRODUCTION

The goal of defect prediction models (DPM) is to identify modules in a software system that are likely to contain errors, and thus guide additional quality assurance activities, such as code reviews or testing. Due to their potential cost savings, such models have been extensively researched for more than a decade.[17, 2]

In the early days, e.g. by Ohlson et al.[19], most prediction models were based on proprietary data, thus preventing independent replication. With the rise of the PROMISE repository[1], this situation has changed. This repository collects publicly available data sets, the majority of them for the task of defect prediction. Currently, there are more than 100 such data sets inside the PROMISE repository, and many more are made available elsewhere.

This trend is very beneficial, as it enables researchers to independently verify or refute previous results. Drummond argues that replication — the repetition of an experiment without any changes — is not worthwhile. He favors reproducing experiments with changes, since only this adds new insights.[4] While we agree that the pure replication of experiments on the same data sets should not lead to new results, we argue that replicability is nevertheless important: When applying previously published procedures to new data sets, or new procedures to well-known data sets, researchers should be able to validate their implementations using the originally published results.

Access to the data is only one, albeit important, step towards the replicability of experiments. First of all, building defect prediction models is not a deterministic process: the partitioning into test and training set has to be random, and many statistical learning techniques are inherently nondeterministic. The replication of an experiment will thus not lead to identical results, even when the same software as the original study is used. This problem has to be addressed by the evaluation procedure, as it should make it possible to get at least similar results. However, when independently replicating an experiment, many small decisions have to be made, and these can influence validity, consistency, and easy of replication.

Contributions. In this paper, we replicate two recent studies on defect prediction models in order to show which information is necessary to actually be able to get consistent results. Additionally, we evaluate the second case study with regards to the performance of trivial and random models. The performance of such models is surprisingly good, at least according to the evaluation scheme used in the original study.

Overview. The remainder of this paper is organized as follows: First, we discuss related work in Section 2. Afterwards, in Section 3 and Section 4, we describe the replication of two recent studies on defect prediction models. We then

[1] http://promisedata.org

PROMISE2010, Sep 12-13, 2010. Timisoara, Romania

summarize and discuss our experiences and results in Section 5. Finally, Section 6 concludes.

2. RELATED WORK

Defect prediction models have been researched for more than a decade, and an overview is out of scope in our context. Pointers to earlier work can be found in a recent article by Menzies et al.[17], and in an overview provided by Arisholm et al.[2]

Two different areas are especially important in the following: Studies on the evaluation of defect prediction models, and work focusing on repeatable results. The former is, among others, addressed by Lessmann et al.[12], and Jiang et al.[9] Both propose a standard procedure to statistically compare defect prediction models over multiple data sets. Jiang et al. evaluate a variety of different performance measures, each suitable for different scenarios. However, all of these measures assume equal treatment effort for each file. We have shown that this may be exploited by trivial classifiers ordering files just by their size, which are surprisingly good according to these measures.[13] We revisit this topic in Section 4.

Kitchenham and Mendes illustrate problems during the replication of effort-prediction studies, and how these may render such comparative studies invalid.[10] The problems in the domain of effort prediction are slightly different, for example regarding performance measures and average size of the data sets, thus their results are not directly transferable to defect prediction studies.

Gay et al. [7] argue that it is not enough to share data, but we should share whole experiments instead, including all code to reproduce results. To facilitate that, they present a scripting environment, OURMINE, that enables such a sharing of experiments. We think that this would be beneficial, but there are plenty of studies already published where the code is not publicly available. We thus see our study as complementary to their proposal.

3. CASE STUDY 1

In this section, we replicate a study by Tosun et al. The study investigates the relationship between network metrics and defects, as earlier proposed by Zimmerman et al.[25] This section provides a summary of their results and a partial replication, details can be found in the original paper.[23]

3.1 Summary

Tosun et al. use data sets extracted from five systems in their study, three from small embedded systems (AR3, AR4 and AR5), and two versions of the Eclipse IDE. They compare the predictive performance of different metric sets, namely traditional code complexity metrics and network complexity metrics. Only the code complexity metrics are publicly available, thus we limit our replication to them. A summary of the data sets used in this study can be found in Figure 1.

The study uses binary classification at the module level[2], i.e. a prediction model is used to predict whether a module is defective or not. Traditional classification performance metrics derived from a confusion matrix are used for evaluation:

System	# Metrics	LoC	# Modules	% Defective
AR3	29	5,624	63	0.13 (0.12)
AR4	29	9,196	107	0.19 (0.18)
AR5	29	2,732	36	0.22 (0.20)
v2_0	198	796,941	6,729	0.14 (0.013)
v2_1	198	987,603	7,888	0.11 (0.004)

Figure 1: Summary of systems used by Tosun et al. v2_0 and v2_1 denote Eclipse version 2.0 and 2.1. Values from the original paper are, in case they differ, given in parentheses.

Predicted	Actual	
	Defective	Non-Defective
Defective	TP	FP
Non-Defective	FN	TN

- Recall (Re): $\frac{TP}{TP+FN}$
- Precision (Pr): $\frac{TP}{TP+FP}$
- False-Positive-Rate (FP): $\frac{FP}{FP+TN}$
- Balance (Bal):$1 - \frac{\sqrt{(0-FP)^2+(1-Re)^2}}{\sqrt{2}}$

Two statistical learning algorithms are used, each with a different experimental setup: logistic regression and Naive Bayes (NB). The input metrics for both are various code complexity metrics and described in the original study; the number of different metrics for each system can be found in Figure 1.

For the logistic regression models, the input metrics are transformed into principal components using PCA. The prediction is based only on the components cumulatively accounting for (at least) 95% of the variance. Afterwards, cross-validation is used to assess the predictive performance of the logistic regression models as follows: the data is split into two parts, two thirds is used for training, the remaining third for the assessment. This procedure is repeated 50 times, the final performance values are averaged over these iterations.

For Naive Bayes, input metrics are log-transformed, as proposed by Menzies et al.[16].[3] For Naive Bayes, cross-validation uses a 90/10 split to build training and test set, this is again repeated 50 times, and the final performance values are averages of the iterations.

3.2 Replication

In the following, we describe our approach to replicate the results for logistic regression and Naive Bayes. We use the data available in the PROMISE repository[4]. For AR3, AR4

[2]Module correspond to functions in AR3-5 and files for Eclipse.

[3]To avoid numerical errors, metric values below 0.0000001 are replaced by 0.0000001, as done by Menzies et al. [16]

[4]`http://promisedata.org`

and AR5, it is already available in a suitable format, i.e. with a binary label for each module denoting whether it is defective or not.

For both Eclipse versions, each module has associated information about the number of pre- and post-release defects. This information is combined in the original paper, and transformed into a binary variable. However, this transformation is not described in detail. We thus use the simplest way and label a module as defective if it contains at least one pre- or post-release defect. However, when comparing our rate of defective modules for each data set with the ones used by Tosun et al., as denoted in parentheses in Figure 1, we can see that the difference for the Eclipse versions are large. Zimmermann et al.[26] first published the Eclipse data sets, and built prediction models only for post-release defects. However, using only pre- or post-release defects does not yield the same ratio of defective modules as reported by Tosun et al., so we keep our original strategy.

We use the R[21] implementation of logistic regression in the package `glm`, with `family=binominal("logit")` as the only parameter. For Naive Bayes, we use the implementation in the package e1071[3], with the parameters `na.action=na.omit`, `type="raw"`. The output values of both techniques are probabilities for each module, which have to be converted into a binary classification using a threshold. Since no such cutoff value is provided in the original paper, we label a module as defective when the predicted probability is greater than 0.5. This value is also used by Zimmermann et al.[25], which is replicated by Tosun et al.[23].

The calculation of the performance measures is straightforward, except for partitions where no positives (i.e. defective modules) are present. Since data sets AR3, AR4, and AR5 are small, and the majority of modules is not defective, this actually occurs, at least for some partitions. This is especially true the evaluation scheme used for Naive Bayes: when we perform 100 repetitions of this procedure for AR3 — i.e., 50 random splits, each selecting 10% of the modules as the test set — there are on average 20.95 out of 50 test partitions with no defective modules. However, the original authors did not mention how they handle this problem, and we thus ignore it by removing the resulting values when averaging.

The results of our models, and the difference to the values reported by Tosun et al., can be found in Figure 2. Overall, the results are mixed: for precision, the differences are large in most experiments, presumably due to the instability of precision explained by Menzies at al.[15]. For the other performance measures, some results are very similar or identical, while some have large deviations. One reason may be a different cutoff-value used in the original study. We thus use the R package *ROCR*[22] to generate ROC curves depicting the relationship of Re and FP over the whole range of possible cutoff-values. For datasets AR3 and AR5, ROCR is not able to compute this curve, since there are partitions containing no defective modules. For both Eclipse versions, we are able to determine cutoffs yielding similar performance in terms of Pr and FP as described in the original study. However, since this process is not specified in the original study, we are not able to replicate this in a consistent way.

3.3 Discussion

As we have already summarized in the last section, the differences of our results to those reported in the original paper

		Re	Pr	FP	Bal
AR3	glm	0.48	0.28	0.20	0.56
		(0.11)	(−0.02)	(0.10)	(0.01)
	NB	0.81	0.17	0.42	0.61
		(−0.05)	(−0.50)	(0.00)	(−0.08)
AR4	glm	0.41	0.31	0.22	0.54
		(−0.20)	(−0.15)	(0.10)	(−0.17)
	NB	0.72	0.35	0.28	0.67
		(0.22)	(−0.39)	(0.10)	(0.05)
AR5	glm	0.51	0.28	0.35	0.52
		(0.07)	(−0.20)	(0.20)	(−0.07)
	NB	0.80	0.44	0.25	0.72
		(−0.20)	(−0.34)	(−0.04)	(−0.07)
v2_0	glm	0.58	0.69	0.20	0.67
		(−0.11)	(0.13)	(0.01)	(−0.07)
	NB	0.76	0.57	0.43	0.65
		(0.06)	(−0.09)	(0.07)	(−0.02)
v2_1	glm	0.47	0.68	0.10	0.62
		(−0.21)	(0.24)	(0.01)	(−0.14)
	NB	0.81	0.43	0.48	0.63
		(0.00)	(−0.20)	(0.00)	(0.00)

Figure 2: Results for logistic regression and Naive Bayes models of the data sets of Tosun et al. Values in parentheses are the differences to the original results.

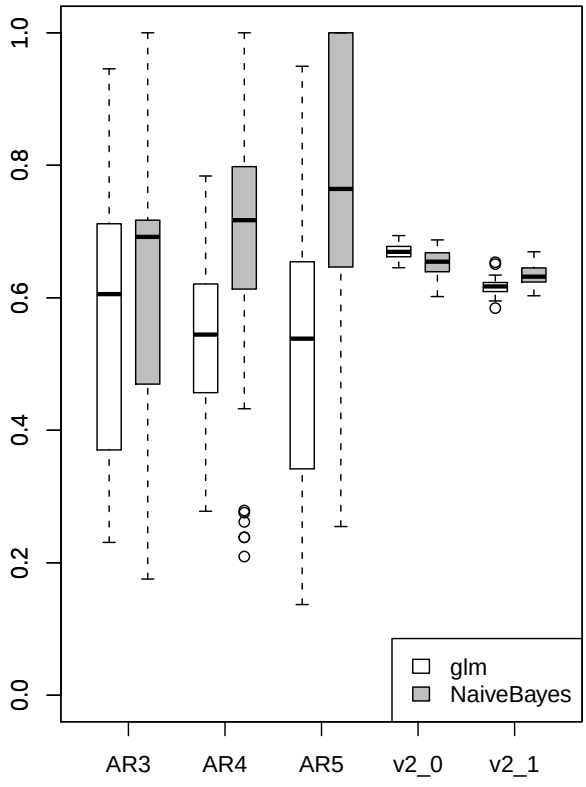

Figure 3: Non-averaged performance measured in Balance for glm and Naive Bayes.

System	LoC	#Classes	% Defective
Eclipse	224,055	997	0.21
Mylyn	156,102	1,862	0.13
Equinox	39,534	324	0.40
PDE	146,952	1,497	0.14
Lucene	73,184	691	0.09

Figure 4: Summary of systems used by D'Ambros et al.

are mixed. One problem, at least for the AR data sets, is the size, combined with a small ratio of defective modules. Many training sets contain only few defective modules: For AR3, the test set (for GLM) contains on average 2.52 defective modules, so that only few different values for each performance measure occur. Thus small differences in the prediction create huge differences in the performance values, leading to unstable results. This can be seen in Figure 3, which depicts the raw performance in terms of Balance for each of the 50 iterations. For AR3-5, the spread is large, so that repetitions of the evaluation procedure may lead to different results. This is not an issue for the Eclipse data sets, since these are large enough.

However, we are also not able to successfully replicate all results for Eclipse. We assume two reasons: different cutoff values, and a different procedure to turn defect count into binary class labels, as discussed in the previous section. Tuning of the cutoff-value improves our results, and thus make the difference smaller. However, from a replication perspective, this is not desirable without further information about the original study.

4. CASE STUDY 2

In a recent study, D'Ambros et al. publish an extensive set of metrics and defect information for five open source systems, and investigate the performance of different bug prediction approaches. In this section, we replicate their study, and perform additional investigations on the predictive performance. In the next section, we provide a brief summary of the experiment itself, details can be found in the original paper.[5]

4.1 Summary

D'Ambros et al. analyze five medium-sized open-source Java systems over a period of roughly five years. They collect code and change metrics for bi-weekly snapshots, and the number and location of post-release defects. All data is collected for top-level classes — due to the file-based nature of the version control systems inner classes are handled together with their outer class. A summary of the data sets can be found in Figure 4.

Using this data, they evaluate the predictive power of different metrics using generalized linear models. The metrics can be categorized based on their source of information:

- Change: Information extracted from the version control system or issue tracker, such as the number of changes per file, the number of bug fixes mentioned in commits, or the number of defects.
- Source code: Traditional code metrics, such as lines of code or object-oriented metrics.

Type	Names	# Input
Change	MOSER	15
	NFIXONLY	1
	NR	1
	NFIX_NR	2
Previous Defects	BF	1
	BUGCAT	5
Source Code	CK	6
	OO	11
	CK_OO	17
	LOC	1
Entropy of Changes	HCM	1
	WHCM	1
	EDHCM	1
	LDHCM	1
	LGDHCM	1
Churn of Metric Changes	CHU	17
	WCHU	17
	LDCHU	17
	EDCHU	17
	LGDCHU	17
Entropy of Code Metrics	HH	17
	HWH	17
	LDHH	17
	EDHH	17
	LGDHH	17
Combined	BF_CK_OO	18
	BF_WCHU	18
	BF_LDHH	18
	BF_CK_OO_WCHU	18
	BF_CK_OO_LDHH	35
	BF_CK_OO_WCHU_LDHH	35

Figure 5: Categorization of the 31 experiments used, and the number of input variables for each experiment.

- Entropy of changes: Complexity of code changes, as proposed by Hassan at al. [8].
- Churn of code metrics: Information about changes of code metrics over time.
- Entropy of code metrics: Complexity of code changes, based on static code metrics.

Details about the metric sets can be found in the original paper[5], the list of metrics, categorized by their information source, can be found in Figure 4. Additionally, this table contains information about the number of metrics used for each experiment. Overall, there are 25 experiments with different metric sets, and 6 additional experiments based on combinations of metric sets.

The experimental setup is based on earlier work by Nagappan et al.[18] Generalized linear models are used to predict — for each combination of system and set of input metrics — the number of post-release bugs occurring in a certain time frame. Due to multicollinearity, the input metrics are transformed into principal components using PCA. The prediction is based only on the components cumulatively accounting for (at least) 95% of the variance.

Cross-validation is used to evaluate the performance of each metric set: The data is split into two parts, 90 % is used as the training set, and 10 % as the test sets. The performance of each experiment, that is, each combination of system and metric set, is evaluated by two measures:

- The *goodness of fit* of a model on the training set is evaluated using R^2. This measure denotes the variability of the data explained by the input variables. D'Ambros use the *adjusted* R^2, which also accounts for the number of input variables.
- The predictive performance on the test set is assessed using Spearman's correlation coefficient ρ between the predicted and the actual number of bugs in each class.

This process is repeated for each combination of data set and input metrics, and the final performance is determined as the average adjusted R^2 and ρ of 50 iterations.

To summarize the results for each metric set over all data sets, D'Ambros et al. use a scoring system. The score determines the relative performance for each combination of metric and data set, compared to the performance of all metric sets on that particular data set. The overall score for one metric set is the sum of its individual scores.

4.2 Replication

We replicate the study using the R system [21], in roughly 320 lines of code. This includes declarative code specifying the location and structure of data files, as well as the evaluation and reporting procedures used on this paper.[5] The data sets are available as comma-separated text files, and the mapping of experiments described in the paper to columns in these files is straight-forward. There in no further information about the modeling algorithm, except that it uses generalized linear models. We thus used the R function `glm` with default parameters.

The results in terms of adjusted R^2 and ρ can be found in Figure 6. The values in parentheses are the differences between our results and the original ones by D'Ambros et al. Overall, the difference between the results is remarkable low: The mean difference for adjusted R^2 between our results is 0.00621, and 0.0030 for Spearman's ρ. The differences for all datasets and metric combinations are also visualized in Figure 7. As we can see, most of the differences are around 0.0, i.e. the results are almost identical. For adjusted R^2, we have two outliers with a difference of 0.57. We have not yet found an explanation for that.

[5]The code is available at `http://informatik.uni-bremen.de/~tmende/promise10`.

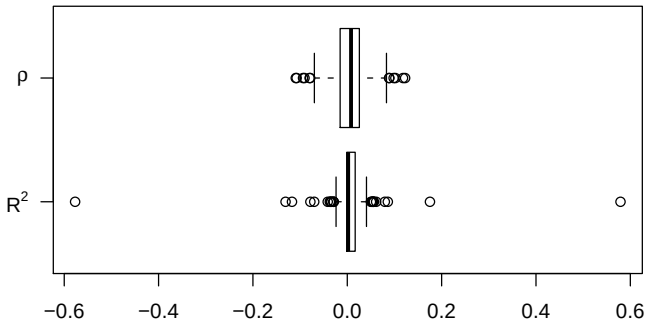

Figure 7: Difference between our results and D'Ambros'.

A replication of the scores on the results reported in the original paper revealed one inconsistency due to rounding errors. When comparing the original scores with our results, 24 out of 31 scores are equal. Two different scores are the result of the outliers described previously.

4.3 Additional Experiments

After our successful replication, we performed further investigations of the data regarding other evaluation procedures and "trivial" models.

4.3.1 Comparison with 10-fold Cross-Validation

D'Ambros et al. use averaging over 50 iterations of random 90% vs. 10% split of the data into training and test set to determine the performance of each experiment. Another common scheme in statistical learning is to use 10-fold cross-validation, i.e. to split the data set into 10 partitions, and perform 10 iterations where each partition serves as the test set once, and the remaining 9 partitions are used for training.

We compare both techniques for each model, however, the differences were negligible. The mean difference in our case is 0.0023 for adjusted R^2 and 0.0028 for ρ. Since 10-fold CV requires less computation, it might be a preferable choice when the learning phase is time-consuming.

4.3.2 Performance of random data

One could assume that the amount of information used for learning has a beneficial influence on the performance: the more input features available, the better. While this is not true for all learning techniques, as redundant or irrelevant features may lead to overfitting, it is supported by a correlation analysis: Spearman's ρ between the number of input features and the resulting score reveals a strong correlation: $\rho = 0.890$ for the goodness of fit in terms of adjusted R^2 and $\rho = 0.729$ for the predictive performance in ρ.

We thus want to investigate whether the performance in-

	Adjusted R^2					Spearman's ρ				
	Eclipse	Mylyn	Equinox	PDE	Lucene	Eclipse	Mylyn	Equinox	PDE	Lucene
MOSER	.449	.209	.606	.543	.604	.441	.328	.505	.234	.263
	(-.005)	(.003)	(.010)	(.026)	(.034)	(.118)	(.044)	(-.029)	(.069)	(.025)
NFIXONLY	.141	.044	.419	.149	.398	.316	.147	.466	.123	.214
	(-.002)	(.001)	(-.002)	(.011)	(-.000)	(.028)	(-.001)	(.037)	(.010)	(-.070)
NR	.375	.128	.518	.379	.489	.403	.119	.555	.274	.239
	(-.005)	(-.000)	(-.002)	(.014)	(.002)	(.039)	(.020)	(.007)	(.029)	(-.057)
NFIX_NR	.378	.129	.519	.379	.489	.402	.130	.554	.275	.235
	(-.005)	(-.000)	(-.002)	(.014)	(.030)	(.021)	(.039)	(-.013)	(.020)	(-.042)
BF	.450	.131	.468	.557	.558	.448	.141	.538	.301	.332
	(-.037)	(-.030)	(-.035)	(.018)	(-.001)	(.038)	(-.018)	(.046)	(.022)	(-.045)
BUGCAT	.483	.161	.507	.558	.564	.419	.254	.500	.292	.332
	(.028)	(.030)	(.038)	(.019)	(.005)	(-.015)	(.123)	(-.013)	(.008)	(-.021)
CK	.378	.117	.556	.057	.371	.405	.278	.555	.274	.187
	(-.041)	(-.078)	(-.117)	(-.577)	(-.008)	(.015)	(-.021)	(.102)	(-.010)	(-.027)
OO	.402	.171	.618	.637	.237	.415	.324	.562	.293	.156
	(.020)	(.056)	(.061)	(.579)	(-.131)	(.038)	(.098)	(.078)	(.037)	(-.060)
CK_OO	.418	.197	.672	.653	.384	.428	.310	.527	.292	.173
	(.012)	(.027)	(.053)	(.035)	(.175)	(.033)	(.013)	(.037)	(.029)	(-.041)
LOC	.341	.043	.408	.039	.081	.406	.245	.529	.266	.145
	(-.007)	(.004)	(.000)	(-.001)	(.004)	(.026)	(.023)	(.054)	(.016)	(-.027)
HCM	.364	.024	.495	.132	.311	.456	.030	.545	.272	.253
	(-.002)	(.000)	(.000)	(.002)	(.003)	(.040)	(.031)	(.019)	(.028)	(-.055)
WHCM	.371	.038	.338	.167	.486	.441	.090	.550	.294	.250
	(-.002)	(-.000)	(-.002)	(.002)	(-.004)	(.040)	(.014)	(.017)	(.021)	(-.038)
EDHCM	.206	.026	.346	.256	.222	.384	.089	.557	.278	.291
	(-.003)	(.000)	(.001)	(.003)	(.002)	(.013)	(.019)	(.062)	(.020)	(-.015)
LDHCM	.159	.011	.464	.272	.218	.390	.081	.567	.299	.281
	(-.002)	(-.000)	(.001)	(.005)	(.002)	(.013)	(.017)	(-.014)	(.019)	(.006)
LGDHCM	.053	.000	.508	.219	.143	.400	.061	.561	.289	.282
	(-.001)	(.000)	(.000)	(.010)	(.002)	(.036)	(.031)	(-.001)	(.026)	(-.048)
CHU	.448	.170	.645	.646	.482	.387	.224	.457	.253	.185
	(.003)	(.001)	(.000)	(.018)	(.026)	(.016)	(-.002)	(-.053)	(.002)	(-.107)
WCHU	.510	.192	.649	.631	.515	.435	.271	.566	.276	.240
	(-.002)	(.001)	(.004)	(.023)	(.037)	(.016)	(-.008)	(.006)	(-.002)	(-.045)
LDCHU	.556	.219	.590	.642	.512	.406	.255	.538	.317	.268
	(-.001)	(.005)	(.009)	(.026)	(.054)	(.011)	(-.020)	(-.025)	(.010)	(-.025)
EDCHU	.510	.232	.533	.624	.523	.378	.256	.493	.307	.268
	(.001)	(.005)	(.008)	(.026)	(.056)	(.016)	(-.003)	(.029)	(.013)	(-.012)
LGDCHU	.471	.107	.646	.536	.514	.404	.214	.563	.272	.250
	(-.002)	(.012)	(.004)	(.050)	(.021)	(-.038)	(.026)	(-.003)	(.083)	(-.040)
HH	.483	.200	.668	.545	.485	.436	.288	.573	.252	.209
	(-.001)	(.001)	(.001)	(.031)	(.052)	(.031)	(.011)	(.089)	(-.014)	(-.109)
HWH	.474	.147	.622	.658	.512	.415	.192	.465	.259	.204
	(.001)	(.001)	(.001)	(.017)	(.028)	(-.010)	(-.020)	(-.015)	(-.007)	(-.059)
LDHH	.530	.213	.599	.563	.429	.406	.260	.577	.287	.254
	(-.001)	(.004)	(.003)	(.041)	(.086)	(-.002)	(-.012)	(.047)	(-.009)	(-.079)
EDHH	.485	.231	.472	.550	.439	.373	.267	.536	.308	.257
	(-.000)	(.005)	(.003)	(.035)	(.080)	(.007)	(-.006)	(-.050)	(.004)	(-.080)
LGDHH	.476	.136	.662	.502	.453	.423	.201	.581	.254	.256
	(-.003)	(.006)	(.002)	(.055)	(.034)	(.002)	(.016)	(.089)	(.018)	(-.091)
BF_CK_OO	.489	.215	.705	.667	.586	.448	.303	.557	.305	.306
	(-.003)	(.002)	(-.002)	(.018)	(.000)	(.009)	(.026)	(.010)	(.023)	(-.056)
BF_WCHU	.533	.195	.650	.647	.599	.444	.259	.559	.273	.304
	(-.003)	(.002)	(.005)	(.020)	(.005)	(-.004)	(-.006)	(.026)	(-.009)	(-.006)
BF_LDHH	.560	.222	.617	.630	.595	.427	.234	.563	.267	.300
	(-.001)	(.005)	(.002)	(.029)	(.003)	(.005)	(.013)	(.030)	(-.038)	(-.052)
BF_CK_OO_WCHU	.489	.215	.705	.667	.586	.448	.303	.557	.305	.306
	(-.070)	(-.035)	(-.029)	(.006)	(-.024)	(.023)	(-.003)	(.033)	(-.005)	(.008)
BF_CK_OO_LDHH	.586	.264	.730	.697	.621	.425	.294	.523	.299	.283
	(-.001)	(.002)	(.000)	(.017)	(.003)	(-.015)	(.003)	(-.048)	(-.013)	(-.094)
BF_CK_OO-	.586	.264	.730	.697	.621	.425	.294	.523	.299	.283
_WCHU_LDHH	(-.034)	(-.013)	(-.024)	(.006)	(-.029)	(.017)	(-.032)	(-.069)	(.010)	(-.058)

Figure 6: Comparison between D'Ambros Results and ours. The difference between our results and the original ones are in parentheses.

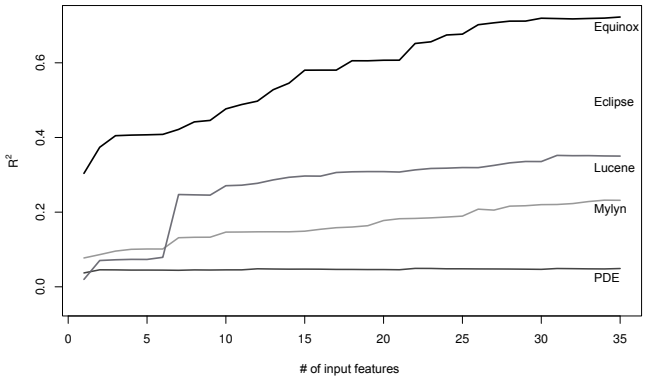

Figure 8: Performance of prediction models for randomly generated data, depending on the number of input features.

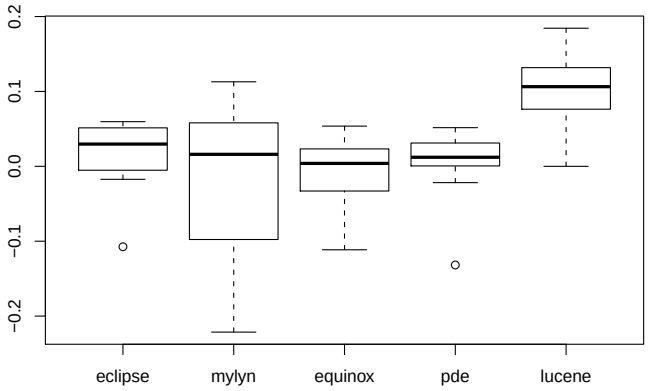

Figure 9: Distribution of difference between each predictor and a trivial model ordering classes by size, measured in $\delta\rho$.

crease for larger input sets is purely an artefact of overfitting, although this should be prevented by the evaluation procedure. We therefore perform two experiments with artificial data.

First, we randomly permute the dependent variable, i.e. the number of post-release defects, and use each set of independent variables for prediction. As expected, the performance is close to 0, measured both in terms of ρ and adjusted R^2. We conclude that the amount of input features alone is not sufficient for good performance, and that the evaluation measures detect this kind of overfitting.

In the past, several researchers have made the observation that the size of a file in terms of lines of code is often a surprisingly good predictor for the number of bugs in each file.[20, 13, 6] We therefore investigate how randomly generated metrics — correlated with size as many software metrics — perform. For each class and its associated defect count and size measured in lines of code, we randomly generate variations of the size, and evaluate the performance for increasing number of input attributes. Each new attribute is created by multiplying the size of a class by a random number drawn from a normal distribution with mean 1 and standard deviation 0.5.

The results can be found in Figure 8. As we can see, the performance in terms of adjusted R^2 increases with each additional (random) input attribute, at least for four out of five systems. For three of the systems (Eclipse, Mylyn and Equinox), the difference between the best original adjusted R^2 and the best adjusted R^2 for our random model is low (0.06, 0.03 and 0.01 respectively). However, for PDE, such a random model seems to not work at all.

Even though a model based on random variations of the size does not work on all data sets, we think it is important to check an evaluation procedure against the simplest possible model. In this case, such a model worked too well for three data sets, thus questioning the value of goodness of fit to determine the performance across different input features.

4.3.3 Performance of a trivial model

Until now, we have considered and evaluated the absolute performance of each experiment per data set. In practice, however, a more realistic comparison is one where a trivial model serves as a benchmark. We have made surprisingly good experience with the performance of models ordering files only by size measured in lines of code[13], and thus use such a model as a benchmark.

In practice, the predictive performance of a model is most important, thus we compare Spearman's ρ for each experiment with the performance of an ordering by size, measured in lines of code. This results, for each combination of metric set and system, in the difference $\delta\rho$, which is positive when the model is better than the trivial ordering. The differences range from -0.221 to 0.184, and depend again on the system, as we can see in Figure 9. We can also observe from that figure that the differences are, on average, rather low, except for Lucene. This means that a model ordering classes by size is performing almost as good as many of the models based on complex metrics which may be expensive to collect. However, as shown by Koru et al.[11], a largest-first strategy may not be cost-effective. We revisit this topic at the end of the next section.

4.3.4 Practical performance improvements

An assessment of the predictive performance in terms of Spearman's ρ shows the theoretical performance of a model. However, in practice, the number of bugs found with the help of a prediction model matters. There is still no consensus which measure is appropriate to quantify the practical usability. [1, 17, 13, 24, 15] One measure that is easy to interpret and a single scalar value, which makes comparisons between models easy, is the *defect detection rate (ddr)*. It denotes the percentage of defects found in a fixed percentage of files. This measure is for example used by Ostrand et al. [20], with a cutoff value of 20%. A ddr of 0.7 thus denotes that a prediction model is able to identify 70% of the defects in 20% of the files.

In order to assess the practical prediction performance, we therefore use ddr with a cutoff value of 20%. The results can be in found Figure 10. Additionally, we compare the difference between the ddr of each model with a trivial model ordering classes by their size. The difference range from -0.167 (the trivial model identifies 16% more defects on average) to 0.267 (the model identifies on average 26%, more defects than the trivial model). There are rather large differences across data sets: For Mylyn and Equinox, the majority of prediction models is worse than a trivial one.

	Eclipse	Mylyn	Equinox	PDE	Lucene	Mean
MOSER	0.690 (-0.009)	0.530 (0.027)	0.534 (-0.038)	0.511 (0.004)	0.618 (0.197)	0.577 (0.036)
NFIXONLY	0.542 (-0.157)	0.336 (-0.167)	0.538 (-0.034)	0.366 (-0.142)	0.553 (0.133)	0.467 (-0.073)
NR	0.696 (-0.003)	0.410 (-0.092)	0.577 (0.005)	0.537 (0.030)	0.646 (0.226)	0.573 (0.033)
NFIX_NR	0.699 (0.000)	0.413 (-0.089)	0.582 (0.010)	0.537 (0.030)	0.647 (0.227)	0.576 (0.036)
BF	0.718 (0.019)	0.427 (-0.075)	0.565 (-0.007)	0.571 (0.064)	0.662 (0.242)	0.589 (0.049)
BUGCAT	0.707 (0.008)	0.497 (-0.006)	0.526 (-0.046)	0.560 (0.053)	0.662 (0.242)	0.590 (0.050)
CK	0.715 (0.016)	0.507 (0.005)	0.568 (-0.004)	0.519 (0.012)	0.506 (0.086)	0.563 (0.023)
OO	0.667 (-0.032)	0.535 (0.033)	0.565 (-0.007)	0.564 (0.057)	0.469 (0.049)	0.560 (0.020)
CK_OO	0.678 (-0.021)	0.529 (0.026)	0.562 (-0.010)	0.563 (0.056)	0.510 (0.090)	0.568 (0.028)
LOC	0.699 (0.000)	0.502 (0.000)	0.572 (0.000)	0.507 (0.000)	0.420 (0.000)	0.540 (0.000)
HCM	0.721 (0.022)	0.337 (-0.166)	0.579 (0.007)	0.531 (0.023)	0.661 (0.241)	0.566 (0.026)
WHCM	0.721 (0.022)	0.389 (-0.114)	0.561 (-0.011)	0.596 (0.089)	0.585 (0.165)	0.570 (0.030)
EDHCM	0.607 (-0.092)	0.393 (-0.109)	0.570 (-0.001)	0.579 (0.072)	0.580 (0.160)	0.546 (0.006)
LDHCM	0.617 (-0.082)	0.400 (-0.103)	0.585 (0.013)	0.602 (0.094)	0.510 (0.090)	0.543 (0.003)
LGDHCM	0.629 (-0.069)	0.376 (-0.126)	0.591 (0.020)	0.566 (0.059)	0.643 (0.223)	0.561 (0.021)
CHU	0.694 (-0.005)	0.459 (-0.043)	0.546 (-0.026)	0.542 (0.035)	0.545 (0.125)	0.557 (0.017)
WCHU	0.720 (0.021)	0.484 (-0.018)	0.552 (-0.020)	0.526 (0.019)	0.590 (0.170)	0.574 (0.034)
LDCHU	0.701 (0.003)	0.515 (0.013)	0.553 (-0.018)	0.579 (0.072)	0.687 (0.267)	0.607 (0.067)
EDCHU	0.696 (-0.003)	0.507 (0.004)	0.542 (-0.030)	0.583 (0.075)	0.666 (0.246)	0.599 (0.059)
LGDCHU	0.695 (-0.004)	0.453 (-0.049)	0.560 (-0.012)	0.540 (0.033)	0.637 (0.217)	0.577 (0.037)
HH	0.710 (0.011)	0.487 (-0.015)	0.572 (0.000)	0.536 (0.028)	0.594 (0.174)	0.580 (0.040)
HWH	0.696 (-0.003)	0.443 (-0.059)	0.548 (-0.024)	0.545 (0.038)	0.580 (0.159)	0.562 (0.022)
LDHH	0.711 (0.012)	0.511 (0.009)	0.568 (-0.003)	0.582 (0.074)	0.641 (0.221)	0.603 (0.062)
EDHH	0.687 (-0.012)	0.523 (0.021)	0.532 (-0.039)	0.578 (0.070)	0.625 (0.205)	0.589 (0.049)
LGDHH	0.723 (0.024)	0.430 (-0.072)	0.570 (-0.002)	0.557 (0.049)	0.574 (0.154)	0.571 (0.031)
BF_CK_OO	0.718 (0.019)	0.524 (0.022)	0.562 (-0.010)	0.573 (0.066)	0.640 (0.220)	0.603 (0.063)
BF_WCHU	0.734 (0.035)	0.472 (-0.030)	0.549 (-0.023)	0.538 (0.030)	0.622 (0.201)	0.583 (0.043)
BF_LDHH	0.728 (0.029)	0.509 (0.007)	0.567 (-0.005)	0.568 (0.061)	0.646 (0.226)	0.604 (0.064)
BF_CK_OO_WCHU	0.718 (0.019)	0.524 (0.022)	0.562 (-0.010)	0.573 (0.066)	0.640 (0.220)	0.603 (0.063)
BF_CK_OO_LDHH	0.730 (0.031)	0.541 (0.038)	0.561 (-0.011)	0.548 (0.041)	0.607 (0.187)	0.597 (0.057)
BF_CK_OO_WCHU_LDHH	0.730 (0.031)	0.541 (0.038)	0.561 (-0.011)	0.548 (0.041)	0.607 (0.187)	0.597 (0.057)

Figure 10: Results measured as defect detection rate in 20% of the files, and difference with trivial prediction model ordering files by size only.

For Eclipse, roughly half (17 out of 31) are better, and for PDE and Lucene most are better in terms of ddr. Additionally, the differences for PDE and especially Lucene are large, compared to the average difference. The conclusion of this part is thus mixed: For some data sets, the practical performance improvements are marginal at best, while for others, combinations of many metrics seems to be beneficial.

Measure ddr as used in this section makes the assumption that the treatment effort — for example for additional code reviews — is independent of the file size. Arisholm et al. [1] argue that this is not realistic for manual reviews, and we have shown that models performing well in terms of ddr are useless when the effort is depending on the size.[13] Whether or not the metric sets may be used cost-effective in practice thus depends on the type of treatment, and has to be evaluated using appropriate performance measures and modelling techniques.[14, 17]

4.4 Discussion

The description of the experimental setup by D'Ambros et al. is well structured and suitable for replications with very similar results. The only inconsistencies or missing information were regarding minor details, such as the handling of ties during the calculation of Spearman's ρ, the handling of invalid values, and when rounding is performed. Nevertheless, we are able to independently reproduce the results of the original study.

However, the evaluation procedure, especially the chosen performance measures, are, at least for some data sets, overly optimistic: Random input values and trivial models are surprisingly well-performing. We argue that such models should always be used for benchmarking.

5. SUMMARY AND DISCUSSION

Our experiment to replicate two defect prediction studies where the data itself is publicly available lead to mixed results. For one study, we are able to reproduce the originally published results with good accuracy. For the other study, we are not able to do so, presumably because of missing information about the chosen cutoff-values, and the size of some of the data sets used.

Although we cannot generalize from only two studies, we can derive the following recommendations to facilitate the reproducibility of defect prediction studies:

- Summaries of the data sets, as given in both studies, are helpful to check whether the chosen data set and input variables are consistent with the original study.
- If cutoff-values are used to derive performance measures, they should be explicitly described.
- Data transformations, such as the conversion to binary class labels for the Eclipse data set in our first case study, have to be explicit. This is also true for the removal of outliers or any other data transformation, although this is not done in our case studies.
- For small data sets, the variance of performance measures is often large. Providing details about the distribution of observed performance values, instead of

scalars only, makes it easier to assess results during replication.

- The partitioning during cross-validation may lead to test sets containing no defective modules, so that performance measures may be not defined. Handling of this cases, should be reported as well, especially for small or very skewed data sets.

6. CONCLUSION

In this paper, we have replicated two studies on defect prediction models, and described our experiences and problems during the replication. We were not able to successfully reproduce the results of the first case study, but did get very close in the second. Additionally, we performed a more in-depth analysis of the second case study using randomly generated data and trivial prediction models.

The replication of studies is only possible when all details of the original study are known (or at least easy to guess). When this is not the case, even access to the original data is not enough. Our discussions for both case studies summarize the most important, albeit not exhaustive, list of information which is necessary for a successful replication. The best way to achieve this is to share whole experiments, as proposed by Gay et al.[7], in the form of scripts. One way to proceed in that direction is to start an "official" set of PROMISE scripts.

Our experiments with random and trivial models have shown that an absolute analysis of performance data is not sufficient, a comparison with trivial benchmarks often reveals interesting results or flaws in the evaluation. Although these benchmarks do not perform well on all datasets, they are easy to perform, and we therefore encourage their use as a sanity check.

Acknowledgements. We would like to thank the donors of the data sets, who made this study possible, and Marco D'Ambros for answering questions regarding his scoring system. We would also like to thank the anonymous reviewers for their valuable comments.

7. 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] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. Technical Report TR 2008-06, Simula Research Laboratory, 2008.

[3] 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.

[4] C. Drummond. Replicability is not reproducibility: Nor is it good science. In *Proceedings of the Twenty-Sixth International Conference on Machine Learning: Workshop on Evaluation Methods for Machine Learning IV*, 2009.

[5] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *7th IEEE Working Conference on Mining Software Repositories*, 2010.

[6] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 2001.

[7] G. Gay, T. Menzies, B. Cukic, and B. Turhan. How to build repeatable experiments. In *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pages 1–9, New York, NY, USA, 2009. ACM.

[8] A. E. Hassan. Predicting faults using the complexity of code changes. In *International Conference on Software Engineering*, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

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

[10] B. Kitchenham and E. Mendes. Why comparative effort prediction studies may be invalid. In *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pages 1–5, New York, NY, USA, 2009. ACM.

[11] A. G. Koru, K. E. Emam, D. Zhang, H. Liu, and D. Mathew. Theory of relative defect proneness. *Empirical Software Engineering*, 13(5):473–498, 2008.

[12] 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.

[13] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pages 1–10, New York, NY, USA, 2009.

[14] T. Mende and R. Koschke. Effort-aware defect prediction models. In *European Conference on Software Maintenance and Reengineering*, pages 109–118, 2010.

[15] 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.

[16] 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.

[17] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 2010.

[18] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *International Conference on Software Engineering*, November 2006.

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

[20] 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.

[21] R Development Core Team. *R: A Language and Environment for Statistical Computing*.

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

[23] A. Tosun, B. Turhan, and A. Bener. Validation of network measures as indicators of defective modules in software systems. In *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pages 1–9, New York, NY, USA, 2009. ACM.

[24] 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.

[25] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *International Conference on Software Engineering*, pages 531–540, New York, NY, USA, 2008. ACM.

[26] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Predictor Models in Software Engineering*, May 2007.