Evaluating Defect Prediction Models for a Large Evolving Software System

CONFERENCE PAPER *in* PROCEEDINGS OF THE EUROMICRO CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, CSMR · JANUARY 2009

DOI: 10.1109/CSMR.2009.55 · Source: DBLP

Thilo Mende
Universität Bremen
13 PUBLICATIONS **174** CITATIONS
SEE PROFILE

Rainer Koschke
Universität Bremen
157 PUBLICATIONS **3,471** CITATIONS
SEE PROFILE

Evaluating the Practical Usability of a Defect Prediction Model for a Large Evolving Software System

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

Marek Leszak
Alcatel-Lucent AG, Nürnberg/Germany
mleszak@alcatel-lucent.com
`http://www.alcatel-lucent.com`

Abstract

A plethora of defect prediction models has been proposed and empirically evaluated, often using standard classification performance measures. Recently, flaws of these measures in this context have been highlighted, and more specific performance metrics were proposed.

In this paper, we describe our experiences creating a defect prediction model for a large software system from the telecommunications domain. The system have been evolved since 2001, resulting in more than 3 million lines of C/C++ code. A history of roughly 3 years is analyzed to extract process and static code metrics that are used to build a prediction model with Random Forests.

The performance of the resulting model is comparable to previously published work. Additionally, an evaluation using the performance criteria that were recently proposed is carried out. Furthermore, we develop a new evaluation measure based on the comparison to the optimal model.

1 Introduction

Product quality plays an increasingly critical role in today's software development. Yet, resources for quality assurance, such as testing or code reviews, are limited. *Defect prediction models* have been developed to focus the available resources on files or subsystems that are likely to be error prone.

A lot of research has been conducted to build such models, for instance, based on code complexity metrics or information about changes of the source code. The techniques employed range from regression models to data mining algorithms.

To assess the quality of such predication model, researchers have proposed various measures such as *accuracy*, *ROC-Curves*, *precision* or *recall*, often casting this problem as a (binary) classification problem. There is still an ongoing debate about the appropriateness of the proposed measures [2, 6].

Contributions. In this paper, we focus on the evaluation of the practical usability of generated prediction models. We develop prediction models for a very large, evolving industrial software system. We give a detailed analysis of these models, along with a detailed analysis of different performance measures for such models. Based on the shortcomings identified in traditional classification metrics, we develop our own performance measures based on the distance from an optimal model and the defect density.

Overview. The remainder of this paper is organized as follows. First, we introduce defect prediction models and related research in Section 2. The system under study and our concrete prediction models are described in Section 3 and evaluated in Section 4 using various classification performance measures. Finally, Section 5 concludes.

2 Background

In building prediction models, several decisions need to be made. In this section, we first outline these decisions and then summarize prior prediction models according to this taxonomy.

2.1 Taxonomy

First, one has to define the *dependent variable*, i.e., the variable that should be predicted by the model. Often, the presence of a (corrective) modification to a file within a certain time frame t is taken as the dependent variable, under the assumption that this is a suitable surrogate for the error proneness of the file. For defect prediction models, the outcome is often a binary classification, that is, *defective* vs. *non-defective* within t. In the following, we consider only this case, for the prediction of defect densities, we refer to Knab et al. [16], for instance.

The dependent variable also determines the unit of analysis – i.e., whether one is looking at subsystems, directories, files, or functions. The lower the granularity, the better the prediction model can pinpoint the exact location of a defect. Yet, this decision is often already made, because data has already been collected that way. In the following, we will use *files* as the unit of granularity because that is the most frequently used unit. Yet, *file* is intended as a placeholder for other types of units as well.

Next, one has to determine *independent variables* that are used by the model to predict the outcome of the dependent variable. Often, product metrics, such as complexity measures, or information about the change history of a file are taken as independent variables.

A *classification technique* has to be chosen that uses the independent variables to predict the outcome of the dependent variables. Alternatives include regression models or data mining algorithms. Beside the choice of algorithms, one has to determine an acceptance criterion. Following Fawcett [9], we distinguish *discrete classifiers* that assign class labels, and *scoring classifiers* that assign a score, typically between 0 and 1, to each observation. In our own study, we use only scoring classifiers, thus we have to define an *acceptance criterion* that is used to turn the scores into binary predictions. We use two approaches: Either we define a *cutoff threshold* and consider observations with a score above this threshold as defective, which we refer to as *direct prediction*, or we perform *ranking prediction*, where we rank all observations using their scores and consider a *fixed percentage* of files as defective.

The evaluation of prediction models usually consists of two phases: first, the classification technique is used to build a model using *training data*. The model is intended to describe the relation of dependent variables and observed independent variables. The *model* is an instance of the classification obtained through training with all classification parameters calibrated including the acceptance criterion. Second, the model is evaluated afterwards using *test data*, which differ from the training data.

There are different approaches to generate test and training data: *Cross-Validation* partitions the data into n random subsets of equal size and builds n models, each with $n-1$ subsets as train data and one subset as test data. The average performance of the n models is then taken as the performance of the classification technique to predict the dependent variable using the independent variables. *Holdout* means that data are partitioned into one test and one training set only. The model is built using the training data and evaluated on the test data. Either the original data are randomly partitioned or the model is trained on data from one release, and test data is acquired from a subsequent release. In the following, we refer to the latter case as *inter-release validation*.

$$\begin{array}{cc|c|c|}
 & & \multicolumn{2}{c}{\text{actual}} \\
 & & + & - \\
\cline{3-4}
\text{predicted} & + & \text{TP} & \text{FP} \\
\cline{3-4}
 & - & \text{FN} & \text{TN} \\
\cline{3-4}
\end{array}$$

Figure 1. Confusion Matrix (+: defective, −: non-defective, TP: true positive, FP: false positive, FN: false negative, TN: true negative.)

The performance of the generated model(s) has to be evaluated. In binary classification, such measures are often based on a *confusion matrix* as depicted in Figure 1. We use *positive* to summarize all defective files, and *negative* to denote non-defective files. Common performance measures that are used for binary classifications can be found in Figure 2. When evaluating binary classifiers, often one class is much less likely than the other class, which is known as *imbalanced data*. The ratio of negative to positive cases, *neg/pos-ratio*, quantifies this imbalance. It is well known that *accuracy* is a poor performance measure in such cases, precision and recall are often used instead [27].

Using scoring classifiers and direct prediction, the cutoff values can be used to favor recall over precision, or vice versa. For example, a model that classifies every file as defective will have a recall of 1, but a very low precision. *Receiver Operating Characteristic (ROC)* curves capture such a trade-off in a graphical way: the recall (often denoted as *tpr* in this context) is shown on the y-axis, while the x-axis shows *fpr*. A scalar metric derived from ROC curves is the *Area Under Curve (AUC)*, i.e., the area enclosed by the ROC-curve and both axes. A perfect classifier has $AUC = 1$, while random guessing results in $AUC = 0.5$. ROC curves, and thus AUC, are independent of changes in the negative/positive ratio [9].

A performance measure that is often used for ranking predictions is the percentage of defects (not defective files, that is *recall*) that is covered by the predicted file set. In the following, we refer to this measure as *defect detection rate* (*ddr*). Measure *ddr* is the sum of defects of all file suggested as defective, devided by the total number of defects.

Name	Definition	Reference / Alternative Name
accuracy (*acc*)	$\frac{TP+TN}{TP+FP+FN+TN}$	overall completeness [8]
recall (*rec*)	$\frac{TP}{TP+FN}$	completeness [6, 11], probability of detection(pd)[21], faulty module completeness[8], true positive rate (tpr)[27]
precision (*prec*)	$\frac{TP}{TP+FP}$	correctness [6, 11] faulty module correctness [8]
F_1-measure (F_1)	$\frac{2 \cdot rec \cdot prec}{rec+prec}$	[27]
false positive rate (*fpr*)	$\frac{FP}{TN+FP}$	[27] Probability of false alarm [21] Type I misclassification rate [14]
false negative rate (*fnr*)	$\frac{FN}{TP+FN}$	[27] Type II misclassification rate [14]
Type I misclassification rate	$\frac{FP}{TP+TN+FP+FN}$	[25]
Type II misclassification rate	$\frac{FN}{TP+TN+FP+FN}$	[25]

Figure 2. Performance measures for binary classifications

2.2 Related Work

Since so many defect prediction models have been proposed, a summary of all of them is not feasible. Instead, we focus on studies where enough data is available to compare our own study to. We use our naming conventions for performance measures, as summarized in Figure 2, which also contains the names used by the original authors.

The dependent variable for all studies discussed below – except those working on publicly available NASA data sets[1] – is the presence of corrective maintenance within a certain time frame, extracted from version control systems, as a surrogate measure of the defect proneness of a file.

Khoshgoftaar et al. [14] applied *discriminant analysis* to build a defect prediction model using control flow complexity metrics and reuse information. Their model had a $fpr = 0.324$ and $fnr = 0.213$. Later, they improved their results by including call graph metrics to $fpr = 0.238$ and $fnr = 0.1375$ [15].

Arisholm et al. compare different data mining algorithms and observe similar performance among them, with AUC values between 0.69 and 0.89 and *recall* between 0.556 and 0.944. However, their *precision* results are very low (0.018 to 0.058) [2]. Additionally, they propose a metric to estimate the cost effectiveness of defect prediction models by quantifying how much less code has to be inspected thanks to a prediction model, compared to random selection of files.

Zimmermann et al. predicted defects for Eclipse based on code complexity measures and achieved $precision = 0.66$, $recall = 0.379$, and $accuracy = 0.711$ at the file level [29].

Menzies et al. use static code measures for publicly available data sets from NASA to generate prediction models with data mining techniques [21]. The best learning algorithm yielded $recall = 0.71$ and $pf = 0.25$, although Zhang points out that precision for these results is only 0.20 [28]. Menzies explains that precision is an unstable measure and models with low precision can still be useful, for instance, in mission-critical systems or when checking false positives is cheap [20].

Guo et al. [10] use Random Forests to build prediction models for the same public NASA data. They achieve accuracy between 0.75 and 0.94 and recall up to 0.87.

Gyimóthy et al. validate the OO-Metrics proposed by Chidamber and Kemerer [7] for defect prediction in Mozilla [11]. Using a multivariate regression model and 0.5 as cutoff value, they achieved $acc = 0.6961$, $prec = 0.7257$, and $ddr = 0.6524$[2].

Nagappan et al. [22] build a prediction model with recall up to 0.774 at 0.714 precision based on dependency and churn metrics, which they define as "a measure of the amount of code change taking place within a software unit over time" [22].

Ostrand et al. develop fault prediction models based on the modification and fault history of files [24]. They use a negative binomial regression model to predict defective files based on information about the file type, file status (new or changed etc.), and lines of code. They use the results of their regression model to perform ranking classification, that is, they order all files according to the predicted value, select the

[1]Available at `http://mdp.ivv.nasa.gov/`

[2]The recall can be calculated using their data: $rec = 0.446$.

upper $n\%$ (often n=20) and consider them as defective. They evaluate their results using defect detection rate, and get $ddr = 0.839$. In a subsequent study, they investigate other performance measures, such as recall and precision [25]. They report $acc = 0.82$, $rec = 0.77$, $prec = 0.15$, Type-I=0.17, Type-II=0.01 and $ddr = 0.84$ when they consider the upper 20% of the files as defective. In their conclusion, they consider ddr and Type-II misclassification rate as the most valuable performance measures.

Another way to evaluate models, *Arberg diagrams*, are introduced by Ohlsson and Arberg [23]. These diagrams relate the percentage of source files with the percentage of defects. In Section 4.1, we use a similar way to develop a new performance measure.

3 Experimental Setup

The goal of this research is to build a defect prediction model for a large software system from the telecommunications domain, and to evaluate which performance measures are useful in practice. In this section, we describe the system under study, the data collection process and the experimental setup following our taxonomy in Section 2.1.

3.1 The System under Study

This paper studies the software of Alcatel-Lucent's product LambdaUnite^{TM}MSS, an evolutionary hardware/software system for the metropolitan and wide-area transmission and switching market.

The first commercial version was released in 2001 and enhanced by subsequent feature releases, approximately every 6 months. A detailed description of the development process and an analysis of several process metrics including defect and change data has been previously published by one of the authors [17, 18].

The aspects relevant for this study can be summarized as follows:

- all changes to software files are guarded by identified and managed Modification Requests (MRs) in the central change management database, adapted from IBM Rational ClearDDTS^{TM}toolset.
- all software files are version controlled in IBM Rational ClearCase^{TM}
- links between MRs and touched files are stored automatically during check-out and check-in
- various descriptive attributes per file and per MR are kept and integrity controlled by a dedicated change management team, the change control board

	Nr. of Files	Nr. of Files without headers	SLOC	SLOC without headers
r1	2805	1935	1865565	1698699
r2	3377	2427	2242384	2022429
r3	3420	2458	2109707	1883401
r4	3366	2429	1780620	1551374
r5	3702	2472	1729088	1487520
r6	3778	2531	1772962	1526543

Figure 3. Distribution of Nr. of files and SLOC per Release (with and without header files)

- all these data are consistently available, in one environment, for all product releases

Ayari et al. [3] describe their difficulties when extracting defect information from version control systems and bug trackers for open-source systems, since there is often no clear distinction between corrective maintenance and feature enhancements. This is not an issue in our study, since there is a clear distinction between corrective maintenance and feature enhancements, as each MR is labeled as either *enhancement* or *defect*. The change control board ensures that the MR data, especially such important MR attributes, are set correctly, leading to high data quality.

LambdaUnite now comprises over 3 millions lines of code, mostly written in C++ and C. The number of files and lines of code analyzed in this study can be found in Figure 3. For reasons explained in Section 3.3, we differentiate between all source files and all source files excluding header files. The lines of code for all files in Figure 3 are calculated as the sum of lines of code of all functions.

3.2 Data Collection

We collect MR information and product metrics for six subsequent releases, covering a timespan of roughly 3.5 years.

Overall, we analyzed 4,293 MRs. The distribution of defects and enhancements can be found in Figure 4. Each resolved MR references one or more files, and we collect the following data for each MR and file combination: MR id, file path, target release, MR type, file type, number of changed, added and deleted lines of code, and whether a file is generated or implements tests. The distribution of these data per release can be found in Figure 5.

As one can see in both figures, release 2 could be considered an outlier, since it was an exceptionally large (measured in number of new features) release with an

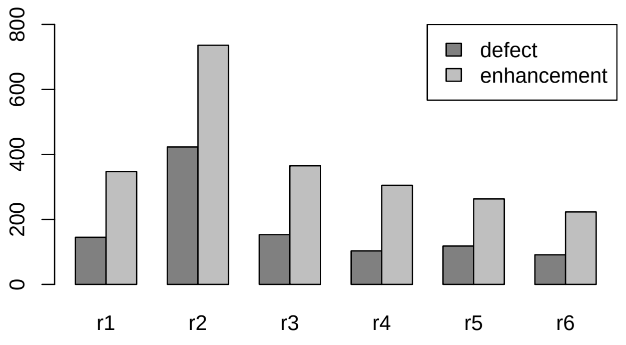

Figure 4. Distribution of defect and enhancement MRs per release

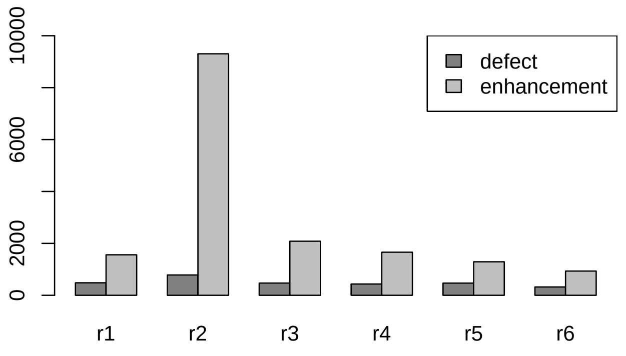

Figure 5. Distribution of files affected by defect and enhancement MRs per release

accordingly long release cycle. We include it in our analysis because we are especially interested in comparing different performance measures, thus we also need realistic data including outliers.

The code metrics summarized in Figure 6 are collected at the function/method level using the Bauhaus Suite[3]. Similar metrics were used successfully by other researchers to build defect prediction models [21, 10, 14]. Even though a more elaborate set of metrics might lead to better prediction models, this is not the main focus of this study, since we are mainly interested in different evaluations of prediction models.

The metrics are aggregated to the file level as total *sum*, except for *NI*, *MaxNest*, and *NumParams* where the maximum value inside a file is used. The file level is used because MRs are tracked at file granularity, and the mapping to functions would have been non-trivial.

[3]`http://www.axivion.com`

Acronym	Description
McCabe	McCabe's original Cyclomatic Complexity [19]
ExtMcCabe	Extended version of Cyclomatic Complexity, short-circuit operators are taken into account
LOC	Last source line of routine - first source line of routine
NI	Number of invocations of other methods
HalLen	Halstead's length [12]
HalVoc	Halstead's Vocabulary
HalVol	Halstead's Volume
HalDif	Halstead's Difficulty
MaxNest	Maximum syntactic nesting level inside a function
NumParams	Number of parameters

Figure 6. Product Metrics

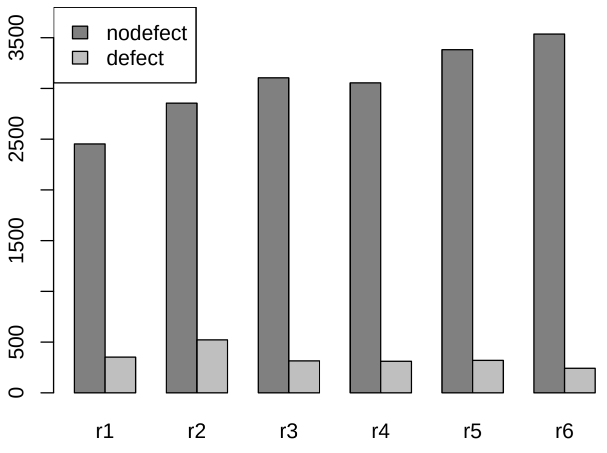

Figure 7. Distribution of non-defective / defective files per release

3.3 Our Defect Prediction Model

In this section, we describe how we build a prediction model for release n following our taxonomy in Section 2.1.

Some performance metrics depend, to a lesser or greater extent, on the negative vs. positive ratio, that is, on the number of non-defective vs. defective files. We therefore use two data sets to investigate different performance metrics: One of them contains all C and C++ source files, except generated files and test files. The second data set additionally includes hand-

	pos/neg	pos/neg without headers
r1	0.143	0.222
r2	0.183	**0.274**
r3	0.101	0.147
r4	0.102	0.147
r5	0.095	0.149
r6	**0.068**	0.106

Figure 8. Ratio of non-defective vs. defective files with and without header files (pos: defective, neg: non-defective)

written header files. We use $h-$ to refer to the former set and $h+$ for the latter. The distinction is relevant because header files mostly contain declarative code, which hardly contains errors. Considering headers, hence, increases the number of files with no or few defects. The influence on the ratio can be found in Figure 8.

Dependent variable: We use *defective?* as dependent variable, and consider a file as *defective* in release n iff there is a defect MR in release n referencing that file, otherwise it is *non-defective*. The distribution of defective vs. non-defective files can be found in Figure 7.

Independent variables: We use two different sets of independent variables: first, we evaluate product metrics collected in release n alone. Afterwards, we combine them with a *churn metric*, calculated for a file f as the total sum of lines that were changed in or added to f by enhancement MRs between release *n-1* and n. We use $c-$ when referring to the product metrics only, and $c+$ for both product and churn metrics.

Classification technique: Random Forest is a classification algorithm that combines the classification of n (often n=500) decision trees into a common score or prediction [5]. This algorithm performs well compared to other advanced data mining algorithms [27] and was successfully used to build software defect prediction models[10, 13].

The main motivation to use a data mining algorithm like Random Forest is that it simplifies the modeling and makes it easy to explore different sets of independent variables.

We use the R[4] implementation of Random Forest with default parameters to build our prediction model.

Validation: We perform both 10-fold cross-validation, denoted *cv* (i.e., test and training data are generated from release n), and inter-release validation *ir* (i.e., n is used as training data, and $n+1$ is used for testing). Although *ir* is the more realistic evaluation, since this is how a model would be applied in practice, we include *cv* so that we have more models to compare performance metrics.

[4]`http://www.r-project.org`

Evaluation: Since the goal of our study is to investigate different performance metrics for defect prediction models, all of the performance measures described in Section 2.1 are calculated. Most of the calculations and evaluations are based on *ROCR* [26], an *R* package to evaluate and visualize classifier performance.

4 Evaluation

In this section, we evaluate our defect prediction models using various performance measures to get an understanding which measures are useful in practice.

We evaluate along the following axes: $h-/h+$ (without and with header files), $c-/c+$ (product metrics without and with churn metrics), and *cv/ir* (cross-validation and inter-release validation). That is, for the cross-validation *cv* (release r1-r6), we generate $2 \times 2 \times 6 = 24$ and for the inter-release validation (release r2-r6) $2 \times 2 \times 5 = 20$ models.

We look at both ways to use Random Forests as a binary classifier: Direct prediction via cutoff thresholds in the next section, and ranking prediction in Section 4.2. Afterwards, in Section 4.3, we compare different scalar performance measures using Spearman's Correlation Coefficient.

4.1 Direct Prediction

One way to evaluate the performance of a binary classifier is to inspect its accuracy, that is, the ratio of correctly classified instances. However, as noted in Section 2.1, accuracy is not suited for imbalanced data sets: Imagine a classifier that labels all files as non-defective (i.e., $FP = 0, TP = 0$). In that case, the accuracy of such a classifier can be calculated as $acc = \frac{TN}{FN+TN}$. Using the data from Figure 3 and Figure 8, we get an accuracy of 0.785 and 0.932 for the releases with the largest (r2 without headers) and smallest (r6 with headers) neg/pos ratio.

For imbalanced data sets, recall and precision are often used instead of accuracy, in our case quantifying the percentage of total defective files that are predicted as defective, and the percentage of false positives in the predicted file set.

Figure 9 depicts, for five revisions and the model c+/h+/cv, precision and recall on the y axis, depending on the cutoff value on the x axis. We can see that we get precision and recall values comparable with most

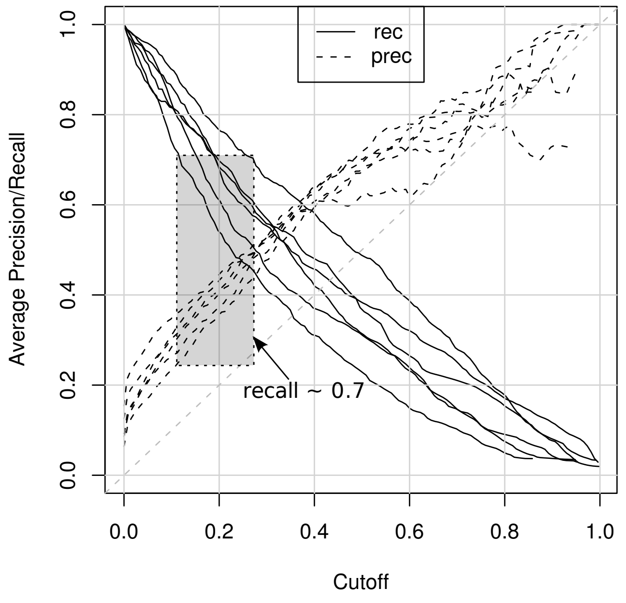

Figure 9. Averaged precision and recall depending on cutoff value for c+/h+/cv and r1-r6

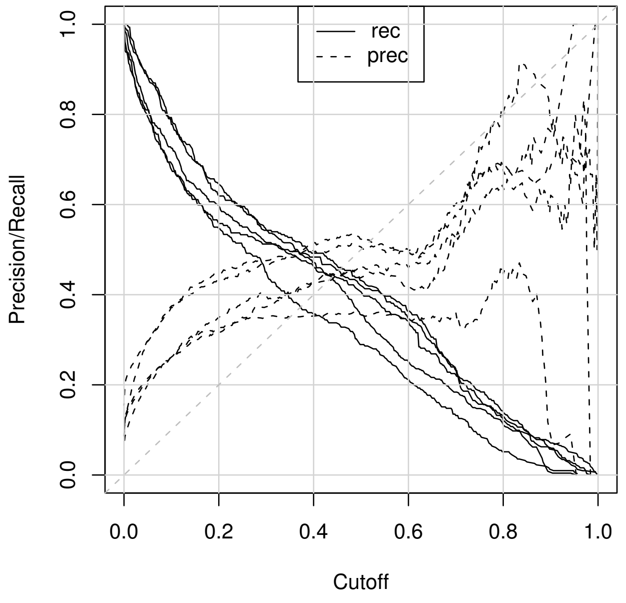

Figure 10. Precision and recall depending on cutoff value for c+/h+/ir and r2-r6

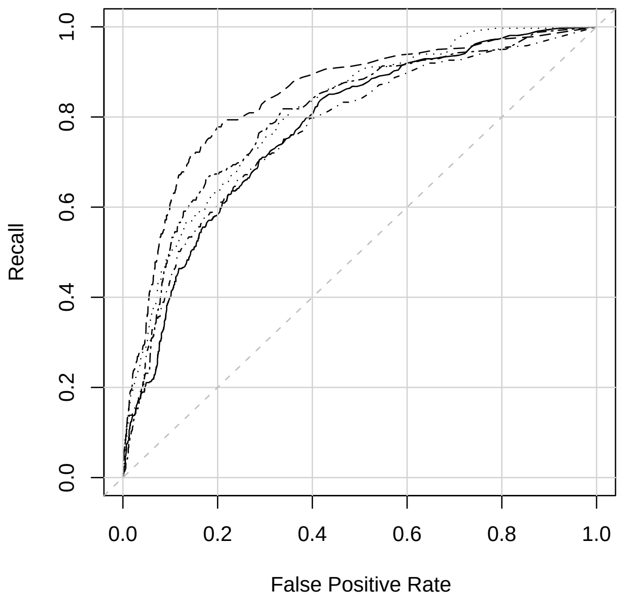

Figure 11. ROC-Curves for c+/h+/ir and r2-r6

work discussed in Section 2.2, e.g., for $recall = 0.7$, we get precision roughly between 0.20 and 0.40.

The low values for precision may limit the practical usability of such a model, as pointed out by Zhang [28]. But as noted by Fawcett [9], any performance metric that uses values from both columns in our confusion matrix in Figure 1 is sensitive to changes in the neg/pos ratio. This is true for precision, as also noted by Menzies et al. [20], and we can see the result in Figure 10: for our five models: the variance for recall is much lower than for precision.[5]

A third way to evaluate the performance, one that is insensitive to the neg/pos ratio, are ROC curves, and the area under curve (AUC) as scalar metric. The ROC curve in Figure 11 shows the performance for one specific model, namely c+/h+/ir. As we can see, at $fpr = 0.25$, we achieve recall between 0.65 and 0.80, comparable e.g., with Menzies et al. [21].

The AUC values for all our models can be found in Figure 12. As we can see, including information about the change history using our churn metric has a positive influence for most models. This confirms prior research, e.g., by Arisholm et al. [1].

As Arisholm et al. [2] point out, a useful defect prediction model has to outperform the random inspection of source code. They calculate a cost estimation based

[5]We do not see this effect in Figure 9, since both precision and recall are averaged over all our 10 folds in the cross-validation model.

	r1	r2	r3	r4	r5	r6
c-/h-/cv	0.807	0.797	0.822	0.818	0.868	0.824
c-/h-/ir		*0.751*	*0.828*	*0.743*	*0.869*	*0.796*
c+/h-/cv	0.823	0.835	0.828	0.862	0.879	0.834
c+/h-/ir		*0.774*	*0.808*	*0.770*	*0.845*	*0.800*
c-/h+/cv	0.867	0.858	0.875	0.862	0.888	0.874
c-/h+/ir		*0.818*	*0.876*	*0.800*	***0.904***	*0.851*
c+/h+/cv	0.876	0.894	0.884	0.886	0.899	0.888
c+/h+/ir		*0.842*	*0.866*	*0.825*	*0.887*	*0.849*

Figure 12. (Average) AUC

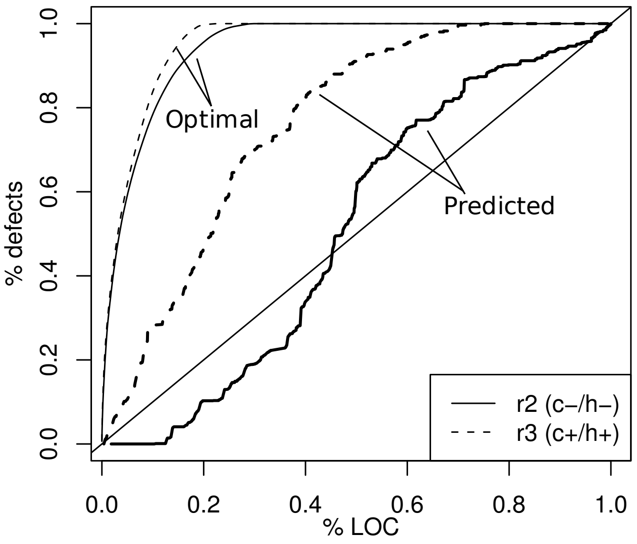

Figure 13. Actual and predicted distribution of faults in lines of code

on the assumption that there is a relationship between the effort to inspect or test a file and its size, and that faults are evenly distributed inside files. 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.

Following these lines, we propose a new metric Δ_{opt} to evaluate defect prediction models. Consider the ratio of faults compared to the ratio of lines of code, as depicted in Figure 13. An optimal model would be created as follows: we order all files by decreasing defect density (and increasing lines of code, in case of ties), and predict the files with highest defect densities first. The optimal models for two releases are labeled *optimal* in Figure 13. When we use our prediction model to order files according to their score (and use lines of code of a file 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.

	r2	r3	r4	r5	r6	mean
c-/h-	**0.433**	0.217	0.315	0.250	0.292	0.301
c+/h-	0.377	0.212	0.243	0.250	0.280	0.272
c-/h+	0.401	0.200	0.285	0.224	0.265	0.275
c+/h+	0.343	**0.192**	0.218	0.229	0.254	0.247
mean	0.389	0.205	0.265	0.238	0.272	0.274

Figure 14. Δ_{opt} for all ir models and all releases

This evaluation resembles Arberg diagrams [23] (see Section 2.2, with the main difference that we use percentage of lines of code instead of percentage of modules for the x-axis. As described by Arisholm et al. [2], we assume that the number of modules does not necessarily produce a cost-effective model, since the costs for inspection and testing are associated with a module's size.

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 the best classifier as 1 and the worst as 0, we define $p_{opt} = 1 - \Delta_{opt}$. This measure has two desirable properties: it compares classifiers to the best possible model, and it takes the cost-effectiveness of a classifier into account.

In Figure 14, one can find the Δ_{opt} values for all *ir* models. When we compare a ranking of models according to p_{opt} with a ranking according to AUC, we see that e.g., the best model in Figure 14 (c+/h+/r3) is not the best model in Figure 12 (although this model is also quite good according to AUC). We further investigate the relationships between different scalar measures in Section 4.3.

4.2 Ranking Prediction

In this section, we describe the results when we use Random Forests to perform ranking prediction, i.e., we order all files by their predicted score, and consider a fixed percentage of files as defective.

Among others, Ostrand et al. [24] use their model in this way, and provide a detailed analysis using various performance measures [25]. The same evaluation has been done for our models; the results can be found in Figure 15. As we can see, our models are comparable to the numbers reported in Section 2.2, at least for *c+* models.

Ostrand et al. [25] conclude their evaluation of different performance measures suggesting *ddr* and Type-

	ddr	recall	% LOC	Type-I	Type-II
c-/h-/ir					
r2	0.649	0.429	0.523	0.108	0.123
r3	0.765	0.610	0.348	0.122	0.050
r4	0.720	0.492	0.435	0.138	0.065
r5	0.802	0.709	0.489	0.109	0.038
r6	0.738	0.628	0.455	0.141	0.036
c+/h-/ir					
r2	0.709	0.466	0.541	0.101	0.115
r3	0.717	0.571	0.352	0.127	0.055
r4	0.787	0.543	0.437	0.131	0.058
r5	0.778	0.681	0.436	0.112	0.041
r6	0.737	0.624	0.414	0.141	0.036
c-/h+/ir					
r2	0.744	0.567	0.565	0.113	0.067
r3	0.857	0.768	0.401	0.130	0.021
r4	0.806	0.582	0.478	0.147	0.039
r5	0.860	0.809	0.531	0.130	0.016
r6	0.809	0.719	0.530	0.154	0.018
c+/h+/ir					
r2	0.799	0.590	0.549	0.109	0.063
r3	0.807	0.673	0.383	0.138	0.030
r4	0.842	0.630	0.446	0.142	0.034
r5	0.850	0.769	0.509	0.134	0.020
r6	0.831	0.744	0.490	0.153	0.016

Figure 15. Performance measures when selecting 20% of files

II misclassification rate as the most important metrics. For *ddr*, we agree that it is a useful measure in practice, however, it has to be combined with the associated ratio of predicted source code. Otherwise, it is too fragile regarding large amounts of non-defective, short files, such as header files. We see this effect in Figure 15, where the models including header files constantly perform better than the ones without header files.

For Type-II misclassification rate, we remark that it depends far too heavily upon the neg/pos ratio, at least following the definition of Ostrand et al. [25]. The maximum possible value, i.e., all defective files are classified as non-defective, depends only on the ratio of defective to all files – the maximum possible Type-II error is between 0.064 and 0.129 for all our models. We prefer the definition of Khoshgoftaar et al. [14], known as *fnr* in this paper.

	auc	acc	ddr	p_{opt}	%def.
auc	-				
acc	0.655	-			
ddr	0.808	0.805	-		
p_{opt}	0.583	0.526	0.556	-	
%def.	0.531	0.952	0.764	0.507	-

Figure 16. Correlation Coefficent ρ for scalar performance metrics and %defect

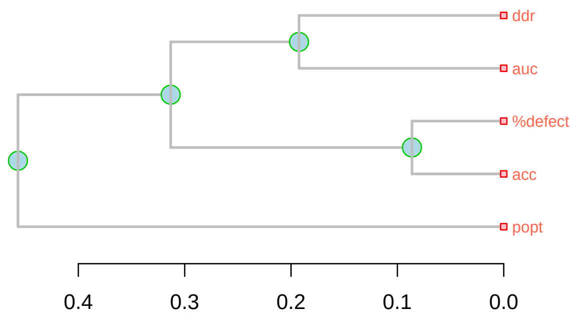

Figure 17. Scalar performance metrics clustered by $1 - \rho$

4.3 Correlation between scalar performance measures

As we have seen, there are plenty of ways to compare the performance of defect prediction models. The most useful metric would be a scalar one, since this enables a direct comparison of different models.

To get a better understanding which scalar metrics perform similarly on the same models, we calculate *Spearman's correlation coefficient* ρ that captures the monotone relationship between the rankings of two samples [4]. We use the following scalar performance metrics to order each of our 44 models: Area under Curve, maximum accuracy, maximum F_1-measure, *ddr* with 20% as of the ranked files, and p_{opt} as defined in Section 4.1. Additionally, we include the ratio of defective files as %defective.

The pairwise calculated ρ values can be found in Figure 16. To visualize the results, we use a hierachical clustering algorithm with $1-\rho$ as the distance measure and average for agglomeration. The resulting dendrogram can be found in Figure 17.

Overall, the correlation between each of the performance measures are mixed. The dendrogram reveals (as expected) the close relationships between *acc* and %defect, and between *auc*, *ddr* and f_1. Measure f_1

seems to be an outlier, since it has the smallest correlation with all other metrics. When we exclude f_1, p_{opt} has the smallest correlation with all other performance metrics, and although the sample size for this analysis is rather small, this indicates that it measures a different concept. Recall that the size of the files predicted influences only p_{opt} and none of the other measures.

5 Conclusion

In this paper, we described the construction of a defect prediction model for a very large system from the telecommunication domain. The generated models perform comparable to previously published work. Additionally, we confirm the observation of other researchers that not all of the performance measures developed in information retrieval are appropriate in our context. We need more specific adaptations taking into account the size of the predicted files, for instance.

During this research, we made some interesting observations that are summarized below.

The selection of data to include, e.g., which files to consider etc., has, via the neg/pos ratio, a strong influence on the performance potentially. Consequently, researchers in this field should always report what types of files they include in their studies.

Despite using rather simple metrics, our models perform surprisingly well, at least when information about the change history (churn metrics) is taken into account, which aligns with other researchers' experience [24, 1].

Random Forests are easy to use in practice and offer sufficient out-of-the-box performance. Their models are not easily interpretable. This may be a problem if programmers want to learn how to avoid defects. For pure prediction in the context of cost-effective testing, it is less a problem.

As we have seen in Section 4, the performance of a classification model is difficult to assess, especially when information about the percentage of predicted source code or the neg/pos ratio is missing. Researchers should always give these data.

We assessed different performance evaluators. We conclude from our study that recall, *ddr* and *fpr* are most useful among the traditional measures. Yet, they do not take into account the predicted code size in terms of lines of code. Chances are high, that any predictor that simply reports the largest files, will outperform all other predictors. Size correlates with test effort, however. What we really want is to identify most defects with least effort, that is, the files with highest defect density. Our newly proposed metric p_{opt} measures this aspect. Our study indicates that p_{opt} may have little correlation with the other measures and, hence, actually makes a difference. This observation should be further studied on larger samples.

References

[1] E. Arisholm and L. C. Briand. Predicting fault-prone components in a java legacy system. In *ISESE*, pages 8–17, New York, NY, USA, 2006. ACM.

[2] E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *ISSRE*, 2007.

[3] K. Ayari, P. Meshkinfam, G. Antoniol, and M. D. Penta. Threats on building models from cvs and bugzilla repositories: the mozilla case study. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 215–228, New York, NY, USA, 2007. ACM.

[4] J. Bortz, G. Lienert, and K. Boehnke. *Verteilungsfreie Methoden in der Biostatistik*. Springer-Verlag, 2000.

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

[6] L. C. Briand, W. L. Melo, and J. Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. *TSE*, 28(7):706–720, 2002.

[7] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[8] G. Denaro and M. Pezze. An empirical evaluation of fault-proneness models. In *International Conference on Software Engineering*, pages 241–251, 2002.

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

[10] L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proneness by random forests. In *Proc. 15th International Symposium on Software Reliability Engineering ISSRE 2004*, pages 417–428, 2004.

[11] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *TSE*, 31:897–910, 2005.

[12] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[13] 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*, 2008.

[14] T. Khoshgoftaar, E. Allen, K. Kalaichelvan, and N. Goel. Early quality prediction: a case study in telecommunications. *IEEE Software*, 13(1):65–71, 1996.

[15] T. M. Khoshgoftaar and E. B. Allen. Classification of fault-prone software modules: Prior probabilities, costs, and model evaluation. *Empirical Software Engineering*, 3(3):275–298, 1998.

[16] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 119–125, New York, NY, USA, 2006. ACM.

[17] M. Leszak. Experiences with process modeling and quality control for development of embedded telecommunication systems. In *5th International Workshop on Software Process Simulation and Modeling (ProSim 2004)*, volume 2004, pages 140–148. IEE, 2004.

[18] M. Leszak. Software defect analysis of a multi-release telecommunications system. In *PROFES*, pages 98–114, 2005.

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

[20] 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'". *TSE*, 33(9):637–640, 2007.

[21] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *TSE*, 33(1):2–13, 2007.

[22] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proc.ESEM 2007*, pages 364–373, 2007.

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

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

[25] T. J. Ostrand and E. J. Weyuker. How to measure success of fault prediction models. In *SOQUA*, pages 25–30, New York, NY, USA, 2007. ACM.

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

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

[28] 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.

[29] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proc. of PROMISE*, May 2007.