A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction

Raimund Moser
Free University of Bolzano-Bozen
Piazza Domenicani 3
I-39100 Bolzano, Italy
+39 0471016138
Raimund.Moser@unibz.it

Witold Pedrycz
University of Alberta
T6G 2V4 Edmonton
Alberta, Canada
+1 7804923333
pedrycz@ee.ualberta.ca

Giancarlo Succi
Free University of Bolzano-Bozen
Piazza Domenicani 3
I-39100 Bolzano, Italy
+39 0471016130
Giancarlo.Succi@unibz.it

ABSTRACT

In this paper we present a comparative analysis of the predictive power of two different sets of metrics for defect prediction. We choose one set of product related and one set of process related software metrics and use them for classifying Java files of the Eclipse project as defective respective defect-free. Classification models are built using three common machine learners: logistic regression, Naïve Bayes, and decision trees. To allow different costs for prediction errors we perform cost-sensitive classification, which proves to be very successful: >75% percentage of correctly classified files, a recall of >80%, and a false positive rate <30%. Results indicate that for the Eclipse data, process metrics are more efficient defect predictors than code metrics.

Categories and Subject Descriptors

D.2.8 [**Metrics**]: Process metrics and product metrics. D.2.9 [**Management**]: Software quality assurance.

General Terms

Measurement, Experimentation.

Keywords

Defect prediction, software metrics, cost-sensitive classification.

1. INTRODUCTION

Defect prediction is an important issue in software engineering. It can be used in assessing final product quality, estimating if contractual quality standards or those imposed by customer satisfaction are met. It can be utilized for decision management regarding resource allocation for testing or formal verification. Within the software engineering community the research on defect prediction has a long tradition. In general, it aims at answering one or several of the following questions:

- Which metrics that are easy to collect during the early phase of software development are good defect predictors?
- Which models, quantitative, qualitative, hybrid, etc., should be used for defect prediction?
- How accurate are those models?
- How much does it cost a software organization to utilize defect prediction models and what are the benefits?

Previous research focused mainly on two different aspects of defect prediction: the relationship between software defects and code metrics [14],and the impact of the software process on the defectiveness of software [27]. The results obtained so far are not conclusive: some authors claimed that process metrics were more effective defect indicators than code metrics [6] while other found the latter very successful [14]. In addition, most of the defect prediction models proposed so far do not consider the *cost* associated with prediction errors (one of the few exceptions is for example [13]). To emphasize the problem of *cost-insensitive* prediction we consider binary classification: if we classify a code unit either as defect free or defective, our model can make two – in terms of software costs - very different mistakes: either it classifies a unit, which *is* defect free, as defective or one, which *is* defective, as defect free. The first type of error implies that we will inspect or test a "clean" code unit, which is clearly a waste of resources for testing respective inspection. However, the second type of error implies that we do not test and eventually fix a code unit, which is defective. It is clear that this might have more serious consequences for software costs as in general fixing a defect in later phases of development or maintenance requires considerable effort.

In this research, we investigate some of the open questions arising in the area of defect prediction using a particular large and up-to-date data set for the Eclipse project (www.eclipse.org). The original data set has been provided and published by Zimmermann *et al.* [31]. While Zimmermann *et al.* used the data for predicting defects for the Eclipse project using logistic regression and complexity metrics our research objectives go far beyond their work as we:

- Annotate the original data with a set of *change metrics* extracted from the source code repository of the Eclipse project (the Eclipse CVS repository, http://dev.eclipse.org).
- Perform a thorough *comparative analysis* between two distinct sets of defect predictors, namely *code metrics* and *change metrics*.

ICSE'08, May 10–18, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-079-1/08/05...$5.00

- Use *cost-sensitive* classification and analyze its impact on prediction models.

Framed in terms of research hypotheses we aim at rejecting H_0:

H_0: *Code metrics* have the same prediction accuracy as *change metrics* for (cost-sensitive) defect prediction.

From a practical point of view this research aims at answering the following questions a software organization might ask when considering the usage of quantitative models for defect prediction:

(a) Are *change metrics* more useful in detecting defective source code than *code metrics*?

(b) *Which change metrics* are good defect predictors?

(c) *How accurate* are such prediction models?

(d) How can *cost-sensitive* analysis be used to minimize effectively misclassification costs and perform a cost-benefit analysis for prediction models?

The paper is structured as follows. Section 2 presents some related work. In Section 3, we report standard accuracy indicators for binary classification and present the idea of cost-sensitive classification in the context of software defect prediction. Section 4 deals with the experimental set-up, in particular the *change metrics* proposed in this research. In Section 5 we present the results of our experiments. Section 6 identifies some limitations of this research. Finally, in Section 7 we draw the conclusions.

2. RELATED WORK

Despite the significant effort spent for defect prediction in research and practice, the relationship between software defects and the various artifacts produced during the software development process are still unknown. Software organizations require models that are both useful and easy to use. Useful means that the prediction accuracy should be high enough to satisfy practical needs. In particular, it should be competitive with manual inspections (which means a percentage of correctly identified defects to be at least around 60% [25]). Easy to use requires that both the collection of predictor variables and the training and application of models should not be too time-consuming and possibly automated. As regards the collection of predictor variables the trend goes towards the automatic mining of the wealth of information contained in various kinds of knowledge repositories used during software development such as code repositories, faults databases, feature request/requirements databases [24], and other. As regards the modeling process many algorithms for both regression and classification have been proposed: due to the wide availability of machine learning tools and their interesting learning capabilities in a domain with very little knowledge nowadays, a common approach is to consider a set of data miners, train them on a training data set or using the hold-out method, and select the one that minimizes a given error function [14]. This seems to be a reasonable approach as the many models proposed and applied so far are neither universally applicable (i.e., they do not work across different application domains) nor consistent, as for almost any study advertising a particular method as the "best" we can find a counter study which claims the opposite.

Regarding predictor variables we can identify three different approaches for defect prediction: product-centric, process-centric, and a combination of both. The most studied and traditional approach for defect prediction is to relate software defects to the product itself: this includes measures of the static or dynamic structure of source code or measures extracted from design documents or requirements. Along these lines there have been various studies. Ohlsson and Alberg [18] reported on a study at Ericsson where metrics derived automatically from design documents were used to predict especially fault-prone modules prior to testing. In the work of Basili *et al.* [1] the Chidamber and Kemerer suite of object-oriented design metrics appeared to be useful in predicting class fault-proneness during the early phases of the software life cycle. Subramanyam and Krishnan [26] confirmed these results. More recently Schröter *et al.* [23] investigated the usage relationships between software components with software defects and found it effective for predicting the most defect prone components for the Eclipse project. Zimmermann *et al.* [31] mapped defects from the bug database of Eclipse to source code locations for three releases of the Eclipse project and additionally annotated such data with a vast amount of size and complexity metrics extracted from source code. They found a significant correlation between complexity metrics and pre- and post-release defects. Moreover, they used logistic regression models for predicting successfully defects at a package level. Menzies *et al.* [14] proposed the Naïve Bayes learner and a feature selection method based on information theory and obtained very accurate results for defect prediction on the MDP repository for NASA projects. They concluded that there exists no *best* set of code metrics for defect prediction but such set rather depends on the characteristics of single data sets, feature selection methods, and machine learners. Nagappan *et al.* [17] arrived at similar conclusions as they stated: "However, there is no single set of complexity metrics that could act as a universally best defect predictor". Instead, they suggested a general methodology for selecting relevant complexity metrics for a given data set and creating a defect prediction model using such metrics.

The second branch in defect prediction research aims at relating various process artifacts such as change history of source files, changes in the team structure, testing effort, or technology and other human factors to software defects. Graves *et al.* [6] argued that change data (i.e., number of modifications, the age of a file, size of the modifications, and more) are better defect predictors than source code metrics such as McCabe's cyclomatic complexity. Nagappan and Ball [16] applied successfully statistical regression methods for predicting software defect density using *relative* code churn. They found that *absolute* code churn was a poor predictor of defect density. Hassan and Holt [7] proposed *Hit Rate caching* for locating faults, which is based on a list (cache) that contains the most (recently) modified subsystems of a software system. They obtained useful results by even using simple heuristics for updating and populating the cache. Weyuker *et al.* [27] found that developer information helped to improve their prediction model based on file size and change data. Bell *et al.* [2] applied successfully negative binomial regression models to identify the most fault prone files (20% of files, which contain on average 75% of the total number of faults) in an industrial software system. They used lines of code, various metrics for the age of a file and its change history, and type of programming language as predictor variables and found that change data significantly improved (almost twice as much) prediction accuracy with respect to a model, which uses only lines of code as predictor. Based on code size, change data, and fault history

Ostrand *et al.* [20] obtained highly accurate results for predicting the most fault prone files in a large software system: 20% of the files with the highest predicted number of faults contained on average 83% of the faults that were actually detected. In contrast to the study of Bell *et al.* file size was found to be an important fault predictor.

Finally, since neither the product- nor the process-centric approach seems to be superior to the respective other, researchers tend to combine process and product related measures in order to build more accurate prediction models. However, using a large set of predictor variables comes with two risks: data collection becomes time consuming and costly; and models are overly complicated and inefficient. Knab *et al.* [12] used static code attributes, in particular software size, together with a set of metrics derived from the change history of the Mozilla project in order to build classification trees and obtained promising results (up to 59% of correctly classified instances). Ratzinger *et al.* [22] used 63 predictors including various size metrics, measures of the change history, and other process related metrics (difficulty of the problem, team structure, etc.) for short-term defect prediction. Among other they concluded that "...not size and complexity measures dominate defect-proneness, but many people-related issues are important."

Among the (algorithmic) defect prediction models, which produce a numerical value as output, in general we can distinguish between two different kinds of approaches: regression and classification. The former aims at predicting the *number* of defects present in a software unit whereas the latter usually aims at inferring whether a software unit is defect free or not and hence makes a *binary classification* (sometimes more than two classes are considered, for example by using the defect severity as additional classification index [29], but the general idea is the same). In order to decide which approach to use we have to consider the granularity of the code units for which we want to predict defects: if they are fine-grained, as for example single classes or files in a software system, the software engineer is more interested in whether those units are defect free or not. On the other hand if we consider packages or subsystems than it is more reasonable to ask for *how many* defects are present in a given subsystem. In this research we are interested in predicting defects for single source files; hence, we look for models that are concerned with binary classification.

In contrast to most of the previous work our primary goal is not to analyze prediction accuracy for a single set of predictor variables; instead, we focus on a thorough comparative analysis between the product-, process-, and combined approach for defect prediction. More specifically, we consider binary (at a file level) *cost-sensitive* classification, which – surprisingly - is used only by few authors [10][13].

3. CLASSIFICATION ACCURACY AND COST-SENSITIVE CLASSIFICATION

The selection of the "best" model from a set of several prediction models depends on the indicators used to measure prediction performance. Therefore, in the following we review briefly the standard indicators used for binary classification and their meaning in the context of this study. This will lead us to the question about different types of classification errors and their respective costs, which are discussed in the subsequent Section on *cost-sensitive* classification.

3.1 Assessing Classification Accuracy

A common strategy for training reliable and stable classification (or regression) models when only one data set for both model training and testing is available is as follows: we repeat several times 10-fold cross-validation for computing various error measures [28]. For each 10-fold cross-validation we calculate the following accuracy indicators: percentage of correctly classified instances, *PC*, the true positive rate, *TP*, and the false positive rate, *FP*. We explain the meaning of those indexes with a simple example: let us assume that for a particular data set, containing 105 instances (files in our case), a model provides the following predictions: 43 files are defect free and 62 defective. However, in reality 32 files are defect free and 73 have at least one defect. Knowing the true class distribution (i.e., if a file is defect free or not) we can summarize our prediction results using the so-called confusion matrix (see Table 1).

Table 1. Confusion matrix.

		Predicted class		
		0	**1**	
True class	**0**	20 (n_{11})	12 (n_{12})	32 ($n_{1.}$)
	1	23 (n_{21})	50 (n_{22})	73 ($n_{2.}$)
		43 ($n_{.1}$)	62 ($n_{.2}$)	105 (n)

1 means that a file has one or more defects while **0** means that it is defect free. The meaning of the entries of the confusion matrix is self-explanatory: the value in the gray shadowed cell for example tells us that our model predicts 23 files as defect free, which in reality have at least one defect! This is for sure not what we want as detecting and fixing errors in later phases of development or after the software has been deployed at a customer's site can be very costly. We rather prefer that our model predicts that a file has a defect, but in reality doesn't (our model predicts 12 such files). In this case we waste resources for inspecting the file but in general inspection costs are cheaper than the later fixing of an undetected defect. However, obviously we would not like to inspect *all* files, as this would be expensive too, and in general we have to find the right balance between the two types of errors (the off-diagonal elements of the confusion matrix). We will return to this issue when describing cost-sensitive classification. Referring to Table 1 the three indicators we compute for assessing a models performance are defined as follows:

PC (percentage of correctly predicted files) = $(n_{11}+n_{22})/n*100\%$, where $n = n_{11}+n_{12}+n_{21}+n_{22}$

TP (true positive rate) = $n_{22}/(n_{22}+n_{21})*100\%$, in information retrieval this is known as the *recall*

FP (false positive rate) = $n_{12}/(n_{12}+n_{11})*100\%$

A good prediction model should yield a high value for *PC* (close to 100%), a high value for *TP* (close to 100%), and a low value for *FP* (close to 0%). If *PC* is high, but the recall (*TP*) low, this means that our model predicts for a large number of files that they are defect free, but in fact are not. Thus, those files pass the software inspection or testing phase and possibly introduce serious problems in later development or the final product – in the worst case resulting in software failures. On the other hand, if *FP* is high we have to inspect a lot of files, which is also not very helpful for planning effective testing or inspections.

3.2 Cost-Sensitive Classification

The idea of *cost-sensitive* classification is to take into account the costs associated with different prediction errors made by a model. Without considering costs of distinctive types of misclassification errors a model predicts the class with the highest associated probability. For example, if it computes for a file a probability of 60% for being defect free and 40% for being defective, it will choose the further as the class value (i.e., it will predict the file as defect free). Instead, *cost-sensitive* classification minimizes a cost function, expressed as a cost matrix, rather than maximizing the probability for a certain class. To illustrate the basic idea and algorithm we provide a simple example for binary classification: We define the following cost matrix *C* (Table 2):

Table 2. Cost matrix.

		Predicted class	
		0	**1**
True class	**0**	0	1
	1	α	0

Furthermore, we have to define a cost function *L*; a common approach for binary classification involves the following combined loss function [3]:

$$L(x)=L(x,\mathbf{1})+L(x,\mathbf{0})$$

$$L(x,\mathbf{1})=Prob(x,c=\mathbf{0})*C(0,1)+Prob(x,c=\mathbf{1})*C(1,1)$$

$$L(x,\mathbf{0})=Prob(x,c=\mathbf{0})*C(0,0)+Prob(x,c=\mathbf{1})*C(1,0)$$

L represents the expected value of a discrete random variable that assigns a cost to each file. This means that the optimal prediction for file *x* is the class $\mathbf{i}\in\{\mathbf{0},\mathbf{1}\}$ that minimizes the value of the loss function $L(x,\mathbf{i})$. $Prob(x,c=\mathbf{i})$ is the probability for class $\mathbf{i}$ given file *x*; and $C(i,j)$ is the number sitting in row *i* and column *j* of the cost matrix *C* (Table 2). Let's make a concrete example: Suppose we have a file *x* for which the predicted probability of being defect free is 60%. Then, using cost matrix *C* in Table 2, $L(x,\mathbf{1})=0.6$ and $L(x,\mathbf{0})=0.4*\alpha$. If we choose $\alpha=1$ then $L(x,\mathbf{1})>L(x,\mathbf{0})$, which means that by minimizing the cost function the model chooses class **0** for prediction. In this special case minimizing the cost function is equal to maximizing the prediction probability, thus the model assigns the class value according to the highest probability. However, if we choose for example $\alpha=5$ then $L(x,\mathbf{0})=2$, which is higher than $L(x,\mathbf{1})$. In this case, by minimizing the cost function *L*, the model predicts class **1**. Although it has a lower probability than class **0**, its cost function has a lower value than the one for class **0**; thus, it is more efficient when considering a cost-benefit analysis with respect to misclassification errors, i.e., when minimizing the costs accounted for by the prediction errors. In our context, specifying values of α larger than 1 means that we account for the fact that false negatives implicate higher costs than false positives; or in other words, that it is more costly to fix an undetected defect during the later life cycle of the software than inspect a file, which is defect-free. There is no optimal value of α. Its value is problem-dependent. A value too high deteriorates prediction accuracy and will produce a large number of false positives. A value too low will result in a model, which is not significantly different from a cost-insensitive model. The only way to determine a useful value for α is by trial and error: run a model with different values for α and choose the one that yields the most benefits (right balance between number of false positives and negatives) for your business domain (considering possibly *real* costs associated with both types of errors).

4. DATA AND EXPERIMENTAL SET-UP

In our experiments, we use a public data set, which includes a large number of static code metrics (198 attributes) and pre- and post-release defects for the Eclipse releases 2.0, 2.1, and 3.0. It is available for download in the PROMISE repository (http://promisedata.org/repository). The authors of the data set, Zimmermann *et al.* [31], extracted the data from the Eclipse code repository and bug database and mapped defects to source code locations (files) using some heuristics based on pattern matching. In addition to the data provided by Zimmermann *et al.* we extracted 18 change metrics from the Eclipse CVS repository and annotated the original data set with them. We skipped a subset of the original files because they did not show a complete CVS history as is needed for our experiments (i.e., they were added/removed during this time period in the CVS repository). In this study we analyze the relationship of change and code metrics with post-release defects only at a file level. Moreover, we complete a binary classification, as we are not interested in predicting the number of defects per file but only whether or not it is defect-free. Table 3 shows a summary of our augmented data set and the class distribution, i.e., the number of defective and defect free files.

We do not use all static code attributes included in the original data set but only a subset of 31 metrics, which were used by Zimmermann *et al.* for defect prediction at a file level. The utilization of all 198 code attributes would involve overly complex models and not yield better performance as most of the measures are highly correlated with each other. The percentage in brackets in column 2 of Table 3 shows the size of our data set with respect to the original one.

Table 3. Summary of the Eclipse data used in the study.

Release	# Files	Metrics	Defect free	Defective
2.0	3851 (57%)	31code metrics and 18 change metrics	2665	1186
2.1	5341 (68%)		4087	1254
3.0	5347 (81%)		3622	1725

Zimmermann *et al.* used only code metrics and related them using correlation analysis, logistic regression, and ranking analysis to post-release defects. They obtained promising results for predicting the presence of defects in packages, but only fair results for classifying single files as defect free respective defective. These mixed results were in part the motivation for this research: we suspect that change data contain more information about defect proneness of source files than the structure of source code itself. Therefore, in addition to code metrics, we consider a second set of predictors concerning change history of files. Then, we compare the model used by Zimmermann *et al.* with the one based on change data and a combination of the two. In choosing suitable change metrics we have consulted and in part followed previous work on this topic such as the one presented in the papers of Graves *et al.* [6], Nagappan and Ball [16], and Ostrand *et al.* [20]. We extracted the following metrics, referred to as *change metrics*, from the CVS repository of the Eclipse project (Table 4):

Table 4. List of Change metrics used in the study.

Metric name	Definition
REVISIONS	Number of revisions of a file
REFACTORINGS	Number of times a file has been refactored[1]
BUGFIXES	Number of times a file was involved in bug-fixing[2]
AUTHORS	Number of distinct authors that checked a file into the repository
LOC_ADDED	Sum over all revisions of the lines of code added to a file
MAX_ LOC_ADDED	Maximum number of lines of code added for all revisions
AVE_ LOC_ADDED	Average lines of code added per revision
LOC_DELETED	Sum over all revisions of the lines of code deleted from a file
MAX_ LOC_DELETED	Maximum number of lines of code deleted for all revisions
AVE_ LOC_DELETED	Average lines of code deleted per revision
CODECHURN	Sum of (added lines of code – deleted lines of code) over all revisions
MAX_ CODECHURN	Maximum CODECHURN for all revisions
AVE_ CODECHURN	Average CODECHURN per revision
MAX_CHANGESET	Maximum number of files committed together to the repository
AVE_CHANGESET	Average number of files committed together to the repository
AGE	Age of a file in weeks (counting backwards from a specific release)
WEIGHTED_AGE	See equation (1)

Regarding the definition of change metrics in Table 4 we have to make a few remarks: first, our set of change metrics is obviously only *one* possible proposal for change metrics we can extract from a CVS repository. Other researchers proposed slightly different metrics such as *relative* code churn [16], *weighted damp* measures [6], month with most revisions, average days between revisions, or relative measures for code added/deleted, and other [22]. We introduce a new change metric, which could have an impact on code quality, namely the number of times a file has been refactored. We compute such metric and the BUGFIXES metric of Table 4, i.e., the number of times a file was involved in bug fixing activities, from CVS comments using very simple pattern matching. Obviously, they are only as reliable as the CVS comments made by the developers. However, as a first step our approach is easy to implement and provides approximate estimations for refactoring and bug-fixing activities. It can be refined in future work if it turns out to be successful. As regards the change set metrics, i.e., MAX/AVE_CHANGESET in Table 4, they are defined as follows: the change set of a file *x* is the number of files that have been committed together with file *x* (within a time frame of 2 minutes) to the repository (it is similar to the notion of co-changes [5]). The two metrics compute the maximum respective the average of all change sets for a single file. Finally, we compute the AGE of a file in weeks, starting from the release date and going back to its first appearance in the code repository. The WEIGHTED_AGE is defined as follows:

$$WeightedAge = \frac{\sum_{i=1}^{N} Age(i) \times LOC\_ADDED(i)}{\sum_{i=1}^{N} LOC\_ADDED(i)} \tag{1}$$

In formula (1) *Age(i)* is the number of weeks starting from the release date for revision *i* and *LOC_ADDED(i)* is the number of lines of code added at revision *i*. The WEIGHTED_AGE takes into account that defect proneness not only depends on the size of a file's changes but also when such changes occurred [6]. Intuitively large and recent changes should have a higher impact on defects present in a file than older changes, which probably have already undergone some inspection and testing. Based on our practical experience and the results reported in previous studies, we expect files to be defect prone if they show the following change characteristics: high revision numbers, large code churn, many different authors, involvement in bug fixing activities, and no refactoring.

All predictors used in this research are numerical variables and the dependent variable can be encoded using two values (**1** for defective and **0** for defect free). For analyzing our Null hypothesis potentially we could use all kind of (machine) learners that can do binary classification. However, the goal of this research is not to find the *best* algorithm for doing defect classification on a particular data set, but rather to determine which *set of predictors* offers more information regarding defect proneness of source files. To decide which classifiers to use for this purpose we make the following considerations:

- We want to compare our results with those obtained by Zimmermann *et al.* [31], thus we have to include logistic regression.

[1] Computed by using the following query for determining if the origin of a revision was a refactoring: ... revision comment ILIKE '%refactor%'

[2] Computed by using the following query for determining if the origin of a revision was a bug fix: ... revision comment ILIKE '%Fix%' AND comment NOT ILIKE '% prefix %' AND comment NOT ILIKE '% postfix %'

- We follow the principle of *Occam's razor*, which means that if a simple model serves our purpose there is no need to use more complex models.

Naïve Bayes is a very simple classification architecture, which nevertheless has demonstrated to perform surprisingly well in practice; Menzies *et al.* [14] used it successfully for defect classification using static code attributes and concluded that it is superior to decision trees and the OneR algorithm. On the other hand decision trees have been used for a long time for defect classification and are easy to interpret [12]. Therefore, we envisage logistic regression, Naïve Bayes, and decision trees as the classifiers of choice for tackling our research problem. If they provide consistent results we may safely assume that other, more complex machine learners do so, too. We use the Weka tool [28] and the "Weka Experiment Environment" for running all experiments. A proper introduction and explanation of the classifiers are out of scope of this research and can be found in many texts on pattern recognition and data mining (for instance [28][3]). While logistic regression and Naïve Bayes are fairly simple algorithms, which do not offer too many parameters for model tuning, decision trees are a bit trickier as they come in very different flavors. We use the J48 algorithm of Weka, which is a Java implementation of Quinlan's C4.5 (version 8) algorithm [21].

5. EXPERIMENTS

The first step of our experiment consists in building three models, one using only our proposed change metrics, referred to as *change model*, one using only the static code metrics used in [31], referred to as *code model*, and one using both types of metrics, referred to as *combined model*, for predicting the presence or absence of defects in files.

5.1 Standard Defect Prediction

A comparison of the performance, i.e., percentage of correctly classified instances (*PC*), true positive rate *TP* (recall), and false positive rate *FP*, of the 3 models and for the 3 machine learners applied is displayed in Table 5. Three releases of the Eclipse project, 2.0, 2.1, and 3.0 have been taken into consideration for which the data have been analyzed. Table 5 has to be interpreted as follows: each row represents the average and standard deviation of the accuracy indexes for repeated 10-fold cross-validation. The indexes themselves are average values for each single 10-fold cross-validation. We repeated 10-fold cross-validation *10 times* to make sure that results are not biased by the data distribution of one specific 10-fold cross-validation. The shading of the cells of Table 5 has the following meaning: a dark solid cell means that its value is significantly better (either higher or lower depending on the index) than the values of all other cells in the same row and for the same accuracy indicator. Gray shaded cells indicate that their values are significantly better than the value of the remaining clear cell, but not significantly different from each other – again for the same index and within the same row. Significance is computed using a Kruskal-Wallis test with the value of α set up to 0.05 [8].

Table 5. Prediction results for repeated (10 times) 10-fold cross-validation.

	Change metrics			**Code metrics**			**Change + code metrics**		
2.0	**PC**	**TP**	**FP**	**PC**	**TP**	**FP**	**PC**	**TP**	**FP**
NB	73	25	4	66	54	28	73	44	13
LR	76	38	6	74	34	7	78	47	8
J48	82	69	11	72	44	15	81	69	12
2.1	**PC**	**TP**	**FP**	**PC**	**TP**	**FP**	**PC**	**TP**	**FP**
NB	77	24	5	74	40	15	76	36	10
LR	79	28	5	79	27	4	80	38	6
J48	83	60	10	77	38	10	81	58	11
3.0	**PC**	**TP**	**FP**	**PC**	**TP**	**FP**	**PC**	**TP**	**FP**
NB	74	30	4	71	28	7	73	30	6
LR	78	47	6	73	38	9	79	51	7
J48	80	65	13	71	46	17	78	64	14

NB – Naïve Bayes, LR – Logistic Regression, J48 – decision tree. All values are in percentages.

To give an example: for release 2.1 the *change model* that uses a decision tree has a significantly higher percentage of correctly classified instances than the other two models using the same classifier. However, as regards the recall (*TP*) it performs still significantly better than the *code model* but similar to the *combined model* (both cells are gray shaded). Regarding the number of false positives (*FP*) it has similar performance to the *code model* and both models are significantly better than the *combined model*.

As expected the performance of the three models depends on both the machine learner and the data set (Eclipse release), see Table 5. However, we can make some general observations, which seem to support our hypothesis that change metrics are *better* defect predictors than code metrics. In particular, those are:

For Eclipse release 2.0:

- Regarding the percentage of correctly classified instances the *change model* performs significantly better than the *code model*. The same is true with respect to the other two indicators (*TP* and *FP*): The *code model* produces only for the Naïve Bayes a clearly higher recall.
- The *combined model* shows only similar performance to the *change model*, thus it seems not to be worth to collect code metrics in addition to change metrics.
- As regards the performance of the different learners decision trees are the most successful.

For Eclipse release 2.1:

- We obtain similar results that confirm the findings of release 2.0. The only remarkable difference is that for the logistic regression learner the *combined model* performs significantly

better than the remaining two, which show comparable accuracy.
- Decision trees again perform best.

For Eclipse release 3.0:

- For Naïve Bayes and J48 the *change model* outperforms the *code model* with respect to all 3 accuracy indicators. The logistic model produces similar results to the one obtained for release 2.1.
- Decision trees produce highly accurate (*PC=80%, TP=65.3%, FP=13.2%*) results.

Overall, we find that the *change model* clearly outperforms the *code model* for three types of machine learners and with respect to the percentage of correctly classified instances, the recall, and the false positive rate. The *combined model* performs similar to the *change model*, but not significantly better. Thus, it does not pay off the extra effort to collect such larger set of predictor variables. Moreover, the results suggest that code and change metrics are not orthogonal predictors; it seems rather that code metrics are a weaker subset of change metrics with regard to the ability of detecting defects. These results are consistent for the Naïve Bayes and decision tree learners, and only the logistic regression deviates to some extent by favoring the *combined model*. They also do not change if we select a subset of features by applying a principal components analysis before building the models.

By analyzing the decision trees a small set of predictors emerges as the most important ones: MAX/AVE_CHANGESET, REVISIONS, REFACTORINGS, and BUGFIXES. The splitting thresholds for each attribute change from release to release. However, the general conclusions remain the same: files with a large MAX_CHANGESET or low revision numbers tend to be defect free. Those with a smaller MAX_CHANGESET, and a low revision number, and those that have been refactored several times are likely to be defect free. Finally, files with higher values for BUGFIXES are defective, which could indicate that fault elimination activities tend to induce new defects. Such insights are valuable for understanding the underlying model of defect generation and constructing causal models for defect prediction.

5.2 Cost-Sensitive Defect Prediction

For cost-sensitive analysis we report only results for the decision tree learner. As with cost-insensitive classification it turns out that prediction results for the other two learners in general are not as good as the ones for the J48 decision tree.

For the previous cost-insensitive models the percentage of correctly identified classes is sufficiently high (for the *change model* using decision trees for example higher than 80%, see Table 5) and the false positive rate tolerable (less than 30%). However, in general the recall (the true positive rate) is relatively low (for example less than 65% for release 3.0). A low false positive rate means that in practice we would not waste too many resources for inspecting or testing defect free files, which is good. A low recall means that we do not at all inspect a quite large number of files, which in fact are defective! This is for sure undesirable as costs for fixing defects in later phases of development increase dramatically.

To take into account the different costs associated in practice with the two types of errors (low recall and high false positive rate) we perform cost-sensitive analysis as explained in Section 3.2. While in theory simple, in practice it is not easy to apply reasonable cost-sensitive classification as we do not know the real cost associated with a concrete type of misclassification error. Based on the experiences reported in the literature and our own we assume that on average inspecting a set of files costs *less* than fixing defects in *later* phases of development or maintenance, which have not been disclosed by the prediction model. A quantification of such cost, which is reflected by the α value of the cost matrix (Section 3.2), is hardly possible. However, we can use the following strategy: we know that our cost factor α has to be greater than 1 as in general inspection costs are lower than costs caused by overlooking defective files, which in turn have to be fixed in later phases of development. We use different values for α, starting from *2, 3...* and plot false positive rate versus recall for each α value. The resulting curves are similar to the ROC (Receiver Operating Characteristics) curves, which are commonly used for evaluating data mining schemes [28]. The only difference is that instead of plotting the (*TP*, *FP*) pair for different class probabilities we plot it for different values of α. Figure 1 shows such plot for α values ranging from *0.1* to *100* and the *change model* using the J48 learner for release 2.0.

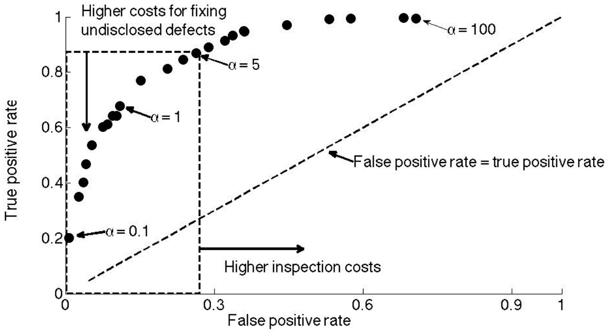

Figure 1. Recall and false positive rate for different cost factors α.

Clearly, for a good predictor the upper left corner is the place to be: there it has a high recall (i.e., most of the defective files are detected) and a low number of false positives. If we do not consider costs, which means $\alpha=1$, then the overall prediction accuracy for the predictor in Figure 1 is fairly high (82%), but the recall is only about 69% with a false positive rate of 12%. A software organization clearly would prefer a model, which has a higher recall and instead take into account a higher false positive rate. By applying cost-sensitive classification using cost factors larger than 1 we shift the recall towards its upper bound (100%) and simultaneously the false positive rate to the right (see Figure 1). It is evident that an increase of the recall implies an increase of the false positive rate as we can always obtain a recall of 100% by simply predicting that *all* files are defective. To determine when to stop increasing the recall we have to use some heuristics. Based on the results of previous work [14] we define that for a good prediction model the false positive rate should not be higher than around 30%. This implies that a cost factor around $\alpha=5$ is a reasonable choice for our data (it turns out that this holds for all three releases). Clearly, such choice is truly experimental as it relies on an inspection of the empirical curve of Figure 1. For different development environments and software projects we cannot expect *a priori* that a cost factor of 5 would yield the best tradeoff between recall and false positive rate, but we have to re-compute it.

Using a cost factor of 5 we repeat our earlier experiment for the decision tree learner J48 and the 3 different models. Table 6, which is similar to Table 5, shows the results of repeated (10 times) 10-fold cross-validation.

Table 6: Prediction results of repeated (10 times) 10-fold cross-validation and a cost factor *α=5*.

	Change metrics			**Code metrics**			**Change + code metrics**		
2.0	**PC**	**TP**	**FP**	**PC**	**TP**	**FP**	**PC**	**TP**	**FP**
J48	77	87	26	63	77	42	79	81	21
2.1	**PC**	**TP**	**FP**	**PC**	**TP**	**FP**	**PC**	**TP**	**FP**
J48	80	80	19	70	65	28	80	74	17
3.0	**PC**	**TP**	**FP**	**PC**	**TP**	**FP**	**PC**	**TP**	**FP**
J48	75	83	29	62	75	43	75	79	26

Cost-sensitive classification produces surprisingly good results: it does not affect too much classification accuracy and false positive rates, but produces excellent results for the recall. The *change model* has for all releases a recall higher than 80% and a false positive rate of less than 30%. Moreover, it outperforms clearly the *code model*, for which we can report results similar to the one found by Menzies *et al.* [14], but still much better than those reported in [31]. The *combined model* shows similar classification accuracy to the *change model*, but a significantly lower recall. Hence, it not only does not pay of the extra effort to accomplish it, but it even seems to deteriorate the recall with respect to a model that uses only change metrics as predictors.

Overall, change metrics are very efficient defect predictors and provide clearly better results than models based on static code measures not only in the context of this study but also with respect to previous work [14][31].

The original data sets used in this work include an additional predictor we did not consider so far, namely the number of pre-release defects. Repeating our experiments using such variable as additional predictor we find that it improves dramatically our results (*CP* around 95%, *TP* around 90% with a *FP* of less than 2%). A closer look reveals that such predictor is highly correlated with post-release defects; hence, it is able to predict the post-release defect distribution. Thus, given the availability of such information for the Eclipse project we would highly recommend to use a simple model based on pre-release defects only rather than collecting change and code metrics. However, for other software projects this seems not to work, as the post-release defect distribution might be completely different from the pre-release defect distribution [19].

Finally, we apply the *change model* for the 3 machine learners and using a cost factor of *α=5* for *actual* defect prediction. It means, that we build the model using data from release *n* (training data) and apply it on release *n+1* (test data). Due to space constraints we do not report the details. However, not surprisingly a model trained on data of release *n-1* does not provide as accurate prediction results for a future release *n* than for the same release. In particular, in some cases the values for the recall and false positive rate are rather unsatisfactory. However, we can observe that the 3 learners provide good results for *single* accuracy indicators. To improve overall prediction accuracy we might consider a combination of several models and use majority voting to decide whether or not a file is defect free. Another proposal for improving prediction accuracy when building models in an iterative way, i.e. for several releases of the same project, could be to consider the past defect distribution as additional input variable. We applied this idea successfully for effort prediction [15]. However, it turns out that for defect prediction such model does not enhance significantly the *change model* considered so far.

To conclude the results obtained by (cost-sensitive) defect prediction provide strong evidence for rejecting our Null hypothesis. In the context of this study defect predictors based on change data outperform significantly those based on static code attributes. This result holds for the considered machine learners, predictors, and accuracy indicators, and is even more manifest in the context of cost-sensitive classification.

6. LIMITATIONS

Drawing general conclusions from empirical studies in software engineering is difficult because any process depends to a large degree on a potentially large number of relevant context variables. For this reason, we cannot assume *a priori* that the results of a study generalize beyond the specific environment in which it was conducted. However, the results of this study are in line with a number of observations made by other researchers. We hope that our experiment contributes towards strengthening the existing empirical body of knowledge and laying the foundations for a future *theory* of defect prediction.

As regards the internal validity of our study we base our conclusions on only three, although very common, data miners. In theory there could be an algorithm, which for example provides much better results for the *code model*. However, the fact that we use rather simple learners that are proven to work well in many practical situations, gives us some confidence that we should not expect very different results when using more sophisticated techniques. Improvements of more advanced models are likely to produce analytic continuations of the findings of this study rather than abrupt changes.

A possible threat to the conclusion validity is our particular choice for code and change metrics as representatives for the defect information contained in source code respective its change history. Although those metrics are widely used and accepted by other researchers there is no consensus as concerns their universality. We do not yet understand the complex mechanism of why and how defects are generated during the software development process. Thus, in theory there could be other, much more complex metrics hidden in source code, which are very powerful defect indicators but nobody discovered them yet. In this light it would be safer to claim that change data offer more defect information with respect to the common size and complexity metrics used so far, *not* with respect to any kind of code metrics.

Finally, the results of any experiment depend on the reliability of the data. In our case there are several possible sources for erroneous data: the mapping between defects and locations in source code could be flawed; or the extraction of the code or change metrics from the respective repositories; or some of the change metrics that are based on developers' reliability in writing appropriate CVS comments could be incorrect. Since we use a public data set we cannot validate data quality directly. As regards the change metrics we used an in-house developed tool to extract them automatically from the Eclipse CVS repository and are very confident about their accuracy. The only source of ambiguity comes with the fact that we use two change metrics, the number of refactorings and the number of fault fixes applied to a file, which depend on CVS comments whose reliability we are unable to assess. As regards the public Eclipse data set Zimmermann *et al.* [31] described in detail how they mined the Eclipse CVS repository and bug database and created the final data set, which makes us feel very confident about its correctness.

7. DISCUSSION AND CONCLUSIONS

The results of this research strongly endorse building defect predictors using change data of source code, which can be retrieved easily from code repositories such as CVS [30]. A set of 18 change metrics, the J48 decision tree learner, and a cost factor of $\alpha=5$ for *cost-sensitive* classification generated very accurate results for three releases of the Eclipse project: >75% percentage of correctly classified files, a recall of >80%, and a false positive rate <30%. These results are very promising as they clearly outperform predictors based on static code attributes for the Eclipse project and also the state of the art in defect prediction using code metrics as reported in [14].

The findings of this research confirm observations made by other researchers that change data, and more in general process related metrics, contain more discriminatory and meaningful information about the defect distribution in software than the source code itself [6][22]. We offer a simple explanation for this phenomenon: while complexity metrics are related with the *cognitive* effort for understanding the source code they are not necessarily sound indicators for software defects. For example, a source file may be very complex and still defect free, because the developer who coded it was very skilled and did a very prudent job. However, a prediction model based on complexity metrics would classify it as defective. On the other hand, if a file is involved in *many* changes throughout its life cycle there is a high probability that at least one of those changes introduces a defect, regardless of its complexity. Obviously there is also some correlation between code complexity and defects, as otherwise defect prediction models based on code metrics wouldn't work as well as they do [14].

Our high-level conclusions are that overall change data are *effectively better* indicators for the presence or absence of software defects than static code attributes. Therefore, future research on defect prediction could focus to a significant extent on the following issues:

(a) Which information contained in change data and other process related knowledge repositories is relevant for defect prediction (we should aim at defining a causal model)?

(b) How to extract automatically such information from those repositories and turn it into powerful defect predictors?

While most of the past research effort has been invested in code metrics based approaches and only produced mixed results [4] there remains much more to be explored in the area of how the software process impacts the generation of software defects during the software's life cycle.

As a practical result of this research we recommend to software practitioners, who are interested in estimating defects, to use the following guidelines:

- First, the past pre- or post-release defect distribution might give a first clue where most defects are.
- Second, build a simple prediction model using change data that can be easily extracted from a revision management system such as the number of revisions of a file.
- Third, if a simple model does not produce satisfactory results, consider all change metrics proposed in this study for defect prediction.
- Fourth, if the recall is rather low tune the model using cost-sensitive classification and determine a cost matrix, which is reasonable for your business.
- Finally, if you dispose of code metrics you could consider a *combined model*, which in some cases might produce slightly better prediction results.

Our results also comment on the relative merits of certain predictors: it turns out that of all 18 considered change metrics a few of them are very powerful defect indicators. They have a straightforward interpretation: first, files with high revision numbers are by nature defect prone. Second, files that are part of large CVS commits are likely to be defect free. We explain this by the fact that larger CVS commits probably follow from a more time-intensive development session, in which files have been analyzed or modified more carefully. Third, bug-fixing activities are likely to introduce new defects. And finally, refactoring seems to improve software quality as files that have been refactored several times show very few defects. The last two findings can be turned into simple recommendations for software developers: in order to contain defects pay particularly attention when *fixing* defects and be ready to *refactor* those files, which have been often changed or are involved in many bug-fixing activities.

8. REFERENCES

[1] Basili, V. R., Briand, L. C., and Melo, W. L. 1996. A Validation of Object Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering, 22(10): 267-271.

[2] Bell, R. M., Ostrand, T. J., Weyuker, E. J. 2006. Looking For Bugs in All the Right Places. International Symp. on Software Testing and Analysis, (Portland, Maine, USA, July 17-20, 2006), ISSTA'06.

[3] Duda, R. O., Hart, and P. E., Stork, D. G. 2002. Pattern Classification. 2nd edition, Wiley Interscience.

[4] Fenton, N., Neil, M. 1999. A Critique of Software Defect Prediction Models. IEEE Transactions on Software Engineering, 25(5): 675 – 689 (October 1999).

[5] Gall, H., Jazayeri, M., Ratzinger, J. 2003. CVS release history data for detecting logical couplings. Proc. of the International Workshop on Principles of Software Evolution (Lisbon, Portugal), IEEE Computer Society Press, pp.13–23.

[6] Graves, T. L., Karr, A. F., Marron, J. S., Siy, H. 2000. Predicting fault incidence using software change history. IEEE Transactions on Software Engineering, 26(7): 653 – 661 (July 2000).

[7] Hassan A. E., and Holt, R. C. 2005. The Top Ten List: Dynamic Fault Prediction. Proc. 21st IEEE International Conference on Software Maintenance (Budapest, Hungary, September 25 - 30, 2005).

[8] Hollander, M. and Wolfe, D. A. 1973. Nonparametric Statistical Methods. Wiley.

[9] Hall, M. M., and Holmes, G. 2003. Benchmarking Attribute Selection Techniques for Discrete Class Data Mining. IEEE Trans. Knowledge and Data Eng., 15(6): 1437-1447 (June 2003).

[10] Khoshgoftaar, T. M., Bhattacharyya, B. B., Richardson, G. D. 1992. Predicting Software Errors, During Development, Using Nonlinear Regression Models: A Comparative Study. IEEE Transactions on Reliability, 41(3): 390-395 (September 1992).

[11] Khoshgoftaar, T. M., Geleyn, E., Nguyen, L., and Bullard, L. 2002. Cost-Sensitive Boosting In Software Quality Modeling. Proc. of the 7th IEEE international Symposium on High Assurance Systems Engineering (October 23 - 25, 2002), Hase'02.

[12] Knab, P., Pinzger, M., Bernstein, A. 2006. Predicting Defect Densities in Source Code Files with Decision Tree Learners. Proc. International Workshop on Mining Software Repositories (Shanghai, China, May 22–23, 2006), MSR’06.

[13] Lanubile, F., Visaggio, G. 1997. Evaluating Predictive Quality Models Derived from Software Measures: Lessons Learned. Journal Systems Software, 38: 225-234.

[14] Menzies, T., Greenwald, J., Frank, A. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. IEEE Transactions on Software Engineering, 32(11): 1-12 (January 2007).

[15] Moser, R., Pedrycz, W., Succi, G. 2007. Incremental effort prediction models in Agile Development using Radial Basis Functions. Proc. 19th International Conf. on Software Engineering & Knowledge Engineering (Boston, MA, USA, July 9-11, 2007), SEKE'07, pp. 519-522.

[16] Nagappan, N., Ball, T. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. Proc. of 27th International Conference on Software Engineering (St. Louis, MO, USA, May 15–21, 2005), ICSE ’05.

[17] Nagappan, N., Ball, T., Zeller, A. 2006. Mining Metrics to Predict Component Failures. Proc. of 28th International Conference on Software Engineering (Shanghai, China, May 20–28, 2006), ICSE’06.

[18] Ohlsson, N., and Alberg, H. 1996. Predicting Error-Prone Software Modules in Telephone Switches. IEEE Transactions on Software Engineering, 22(12): 886-894.

[19] Ohlsson, N., and Fenton, N. 2000. Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Transactions on Software Engineering, 26(8): 797—814.

[20] Ostrand, T. J., Weyuker, E. J., Bell, R. M. 2005. Predicting the Location and Number of Faults in Large Software Systems. IEEE Transactions on Software Engineering (April 2005), 31(4): 340-355.

[21] Quinlan, R. 1993. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers (San Mateo, CA, 1993).

[22] Ratzinger, J., Pinzger, M., Gall, H. 2007. EQ-Mine: Predicting Short-Term Defects for Software Evolution. Proc. of FASE’07 (Braga, Portugal, 24 March - 1 April, 2007), pp. 12-26.

[23] Schröter, A., Zimmermann, T., Zeller, A. 2006. Predicting Component Failures at Design Time. Proc. of ACM-IEEE 5th International Symposium on Empirical Software Engineering (Rio de Janeiro, Brazil, 2006), ISESE’06.

[24] Schröter, A., Zimmermann, T., Premraj, R., and Zeller, A. 2006. If Your Bug Database Could Talk .... Proc. of ACM-IEEE 5th International Symposium on Empirical Software Engineering, Volume II: Short Papers and Posters (Rio de Janeiro, Brazil, 2006), ISESE’06.

[25] Shull, F., Boehm, V. B., Brown, A., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., and Zelkowitz, M. 2002. What We Have Learned About Fighting Defects. Proc. of 8th Int’l Software Metrics Symp., pp. 249-258.

[26] Subramanyam, R., and Krishnan, M. S. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. 2003. IEEE Transactions on Software Engineering (April 2003), 29(4): 297-310.

[27] Weyuker, E. J., Ostrand, T. J., Bell, R. M. 2007. Using Developer Information as a Factor for Fault Prediction. Proc. 3rd International Workshop on Predictor Models in Software Engineering (Minneapolis, MN, USA, May 20, 2007), PROMISE’07.

[28] Witten, I. H., and Frank, E. 2005. Data Mining: Practical machine learning tools and techniques. 2nd Edition, Morgan Kaufmann (San Francisco, 2005).

[29] Zhou, Y., and Leung, H., Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults. 2006. IEEE Transactions on Software Engineering (October 2006), 32(10): 771—789.

[30] Zimmermann, T., and Weißgerber, P. 2004. Preprocessing CVS Data for Fine-Grained Analysis. Proc. of International Workshop on Mining Software Repositories (Edinburgh, Scotland, UK, May 25, 2004), MSR’04.

[31] Zimmermann, T., Premraj, R., Zeller, A. 2007. Predicting Defects for Eclipse. Proc. 3rd International Workshop on Predictor Models in Software Engineering (Minneapolis, MN, USA, May, 2007), PROMISE’07.