

A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models

Erik Arisholm^{1,2}, Lionel C. Briand^{1,2} and Eivind B. Johannessen^{1,2}

¹Simula Research Laboratory, PO Box 134, NO-1325 Lysaker, Norway

²Dept. of Informatics, University of Oslo, Norway

erika@simula.no; briand@simula.no, eivindjo@ifi.uio.no

Abstract.

This paper describes a study performed in an industrial setting that attempts to build predictive models to identify parts of a Java system with a high fault probability. The system under consideration is constantly evolving as several releases a year are shipped to customers. Developers usually have limited resources for their testing and would like to devote extra resources to faulty system parts. The main research focus of this paper is to systematically assess three aspects on how to build and evaluate fault-proneness models in the context of this large Java legacy system development project: (1) compare many data mining and machine learning techniques to build fault-proneness models, (2) assess the impact of using different metric sets entailing different data collection costs, such as source code structural measures and historic change/fault (process) measures, and (3) compare several alternative ways of assessing the performance of the models, in terms of (i) confusion matrix criteria such as accuracy and precision/recall, (ii) ranking ability, using the receiver operating characteristic area (ROC), and (iii) our proposed cost-effectiveness measure (CE).

The results of the study indicate that the choice of fault-proneness modeling technique has limited impact on the resulting classification accuracy or cost-effectiveness. There is however large differences between the individual metric sets in terms of cost-effectiveness, and although the process measures are among the most expensive ones to collect, including them as candidate measures significantly improves the prediction models compared with models that only include structural measures and/or their deltas across releases – both in terms of ROC area and cost-effectiveness. Further, we observe that what is considered the best model is highly dependent on the criteria that are used to evaluate and compare the models. The regular confusion matrix criteria, although popular, are not clearly related to what we consider to be a crucial aspect, namely the cost-effectiveness of using fault-proneness prediction models to focus verification effort where it is the most needed.

1 Introduction

A significant research effort has been dedicated to defining specific quality measures and building quality models based on those measures [1]. Such models can then be used to help decision-making during development of software systems. Fault-proneness or the number of defects detected in a software component (usually a module, class, or file) are the most frequently investigated dependent variables [1]. In this case, we may want to predict the fault-proneness of components in order to focus validation and verification effort, thus potentially finding more defects for the same amount of effort. For example, assuming a class is predicted as very likely to be faulty, one would take corrective action by investing additional effort to inspect

and test this class. Given that software development companies might spend between 50 to 80 percent of their software development effort on testing [2], research on fault-proneness prediction models can be motivated by its high cost-saving potential.

As a part of this study, we have reviewed a selection of relevant publications within the field of fault-proneness prediction models (details are provided in Section 2). The review revealed that a vast number of modeling techniques have been used to build such prediction models. However, there has been no comprehensive and systematic effort on assessing the impact of selecting a particular modeling technique.

To construct fault-proneness prediction models, most studies use structural measures such as coupling and cohesion as independent variables. Although some studies have investigated the possible benefits of including other measures such the number of changes performed on components and their fault history in previous releases, none of the studies assess in a systematic way the impact of using various sets of measures, entailing different data collection costs, on the cost-effectiveness of the prediction models.

A large number of evaluation criteria have been used to evaluate and compare fault-proneness prediction models. Among the most popular evaluation criteria are the ones that can be derived from the confusion matrix such as accuracy, precision, and recall. There is little consistency across the reviewed studies with respect to the criteria and methods that are used to evaluate the models, making it hard to draw general conclusions on what modeling technique or sets of independent variables seems the most appropriate. In addition, the popular confusion matrix criteria are somewhat abstract as they do not clearly and directly relate to the cost-effectiveness of using fault-proneness prediction models to focus verification and validation activities such as testing. Because there exists very little evidence of the economic viability of fault-proneness prediction models [1], there is a need for evaluating and comparing fault-proneness prediction models not only by considering their prediction accuracy, but also by assessing the potential cost-effectiveness of applying such models.

To compare the potential cost-effectiveness of alternative prediction models, we need to consider (surrogate) measures of additional verification cost for the selected, faulty classes. For many verification activities, such as structural coverage testing or even simple code inspections, the cost of verification is likely to be roughly proportional to the size of the class.¹ What we want are models that capture other fault factors in addition to size, so that the model would select a subset of classes with high fault density.

To build fault-proneness prediction models there are a large number of modeling techniques to choose from, including standard statistical techniques such as logistic regression, and data mining techniques such as decision trees [3]. The data mining techniques are especially useful since we have little theory to work with and we want to explore many potential factors (and their interactions) and compare many alternative models so as to optimize cost-effectiveness.

Although there are a large number of publications that have built and evaluated methods for building fault-proneness prediction models, it is not easy to draw practical guidelines from them in terms of what modeling techniques to use, what data to collect, and what practical gains to expect. This paper investigates in a systematic way three practical aspects of the building and evaluation of fault-proneness prediction models; (i) choice of modeling techniques, (ii) choice of

¹ Depending on the specific verification undertaken on classes predicted as fault prone, one may want to use a different size measure that would be proportional to the cost of verification.

independent variables (sets of measures), and (iii) choice of evaluation criteria. This assessment is performed by building a range of fault-proneness prediction models using a selection of relevant modeling techniques. The models are built using different sets of independent variables entailing different data collection costs. This allows us to assess the possible benefits of collecting certain sets of measures. The resulting models are then systematically compared and evaluated using a number of the most popular evaluation criteria such as accuracy, precision and recall. To assess the potential cost-effectiveness in applying the models to focus verification activities, we also compare the models according to a proposed measure of cost-effectiveness within this particular industrial context.

The remainder of this paper is organized as follows: Section 2 provides a comprehensive overview of related works, whereas Section 3 presents our study design. In Section 4 we report our results, comparing several modeling techniques and sets of measures using a number of different evaluation criteria. Section 5 discusses what we consider the most important threats to validity, whereas Section 6 concludes and outlines directions for future research.

2 Fault-proneness Prediction Models

In this section, we first elaborate on the concept of fault-proneness; how it is defined, and possible ways of measuring it. Then, we describe factors that may have an impact on fault-proneness, and thus are candidate predictor variables. We continue by giving a brief summary on how various statistical methods and data-mining techniques have been used in existing fault-proneness studies. Furthermore, we discuss how fault-proneness prediction models have been evaluated. Throughout the following subsections we summarize existing work according to the abovementioned dimensions and discuss implications for our work.

The discussions in the following sections are frequently referring to Appendix A, which gives a summary of this field of research in recent years. Each study is categorized in terms of dependent variable, unit of analysis, selection of measures, modeling techniques and evaluation criteria used, validation method and type of system (see Appendix A). Many of the findings prior to 2002 are summarized in [4], and we therefore focus on empirical research reported since 2001. To obtain this set of papers, we proceeded as follows:

We searched ISI Web of Knowledge and Inspec for papers that matched the following logical expression:

```
((software OR object-oriented) AND (metrics) AND  
(prediction) AND (defect OR fault OR error))
```

First, ISI returned 40 hits. Out of these, 12 papers were included after reading the title and abstract to determine whether they were indeed related to the topic of fault-proneness prediction models. Second, using Inspec, we performed the same search but limited to journal papers only. This search resulted in 32 hits, of which 8 additional papers were included on the basis of reading the title and abstract.

Given that this search was probably not complete, we furthermore checked the included papers for references to additional work on the topic of fault-proneness prediction models. As a result, an additional 13 papers were included leading to a total of 33 papers.

A more comprehensive systematic literature review is certainly possible but we still believe, because of the systematic nature of our search, that the selected papers reflect the current state of the art in a reasonably unbiased way.

2.1 Fault-proneness

Fault-proneness is a difficult concept to define in precise terms and can be measured in many ways. In pragmatic terms, fault-proneness is the probability that a component, e.g., a class, contains a fault. A fault is a (possibly undetected) incorrect program step, process, or data definition in a computer program [5]. In many situations, a more practical definition of fault-proneness that is commonly used is the probability of detecting one or more faults in a component. A fault may be detected as a result of any form of verification and validation activities at different stages of development and maintenance. Some faults remain undetected while others are detected as field failures. A field failure is a systems inability to perform its required functions during operation. Faults that manifest themselves through field failures may be different from those found before the system is deployed. Thus, one may distinguish between pre-release and post-release faults, the latter possibly resulting in field failures. Furthermore, some faults are more severe than others, and thus one may classify faults according to their severity level to distinguish fault-proneness with critical implications. Column 2 in Appendix A gives an overview of the kinds of faults that have been considered in the reviewed studies.

A common conception is that some components are intrinsically more fault-prone than others due to some (possibly unknown) property. For example, components that are fault-prone during system test may continue to be fault-prone during future operation. Thus, the distribution of faults found during pre-release testing may reflect the future distribution of post-release faults. However, a study by Fenton *et al.* suggests that the number of pre-release faults is inversely correlated to the number of post-release faults, i.e., components that are among the most fault-prone during pre-release testing are among the most reliable during field operation [6]. This is, to some degree, further supported by Ostrand *et al.*[7]. It is important to note that these findings do not imply a causal relationship; the fact that post-release fault-proneness is inversely correlated to pre-release fault-proneness might be attributed to the distribution of effort spent during pre-release testing across various components.

When measuring and predicting faults in object-oriented systems, the unit of analysis may be the individual changes done on a particular component, a class, a file, a package or module, executable component or subsystem. Some studies investigating fault-proneness models in the context of object-oriented systems use a class as their unit of analysis, e.g., [8-14]. However, because most revision control systems operate at the file-level, many studies use files as the unit of analysis, e.g., [15-18]. Others aggregate data to a higher level and use collections of related files (modules) as the unit of analysis, e.g., [19], while others analyze on a more detailed level such as methods or procedures, e.g., [20, 21]. Some studies, such as [22], have used the change itself, i.e., each commit to the source code repository, as the unit of analysis.

In addition to the choice of the unit of analysis, there are also different options for constructing the dependent variable to be predicted: binary measures of whether the unit contains one or more faults, counts of faults and fault density. The choice of dependent variable varies across studies, as shown in Column 2 in Appendix A. Nearly half of the studies reviewed in this thesis use the number of faults as the dependent variable [11, 13-18, 23-27]. However, in many cases, the number of faults in a component is small, making it more practical (from a data

analysis perspective) to use a dichotomous variable to indicate the absence or presence of faults rather than fault counts. Half of the studies reviewed in this thesis use a binary dependent variable [9, 25]. Although this recoding allows the use of classification techniques and facilitates analysis, it is a more coarse-grained measure, thus potentially limiting the discriminatory power of the prediction model.

Some studies divide the number of faults by some size measure, e.g., lines of code, and thus obtain a measure of fault density [28]. However, the use of fault density might be problematic as the denominator of the dependent variable is a size measure while certain explanatory variables are also strongly correlated with size. Rosenberg [29] showed that such situations may lead to spurious relationships which are pure mathematical artifacts. It also results in models that are difficult to interpret. Finally, some studies account for the severity of faults. For example, Zhou *et al.* [10] built three prediction models; one to predict the probability of high severity faults, one to predict the probability of low severity faults, and finally a model where the severity of faults was not accounted for.

The choice of dependent variable also depends on how the resulting prediction model is to be used. If the purpose is merely to provide some indicator of quality of each component in a system, then using the number of faults as a dependent variable might be a reasonable choice, assuming that one can find an appropriate modeling technique for the distribution at hand. Conversely, if differentiating components with one fault from components with many faults does not affect decision making (e.g., as in deciding whether or not to spend extra effort to verify that a class does not contain faults), one may be better off to choose a binary dependent variable, in which case the prediction model can provide a ranking of the classes according to fault probabilities.

2.2 Fault-proneness Factors

There are a number of factors that are likely to have an impact on fault-proneness. We divide these factors into three categories:

- Structural measures: They are measures of structural properties derived from the source code. This category includes popular coupling metrics, size metrics and other measures that can be collected from a snapshot of a file (revision).
- Delta measures: These measures capture the amount of *change* – sometimes called *churn* – in a file, e.g., by taking the difference between structural measures between successive releases.
- Process measures: They are not derived from the source code, but are collected from meta data in the revision control system or through human intervention, e.g., by assessing the experience of each developer, the number of developers that have made changes to a file, the number of faults in previous release(s) and simpler measures such as the number of lines added and/or removed.

Our classification of measures into three categories is motivated by practical considerations. Collecting structural measures requires no revision control system or historical data. They are simply derived from a particular snapshot of the code base. The delta measures, on the other hand, require release management and a revision control system to compute the difference between two successive releases for a particular measure. However, if revision control and release management is in place, such measures are inexpensive to collect because it requires

no additional human intervention. Some of the process measures, on the other hand, require intervention from the developers; they need to record the reason for each change in a coherent manner. In addition, the process measures are somewhat domain and process specific, and their definitions are coupled to the way the development team works: how the system is evolving, how the developers locate and record faults, and how they remove them.

One of the underlying hypotheses in building fault-proneness models is that structural properties, such as coupling between object classes [30] and cyclomatic complexity [31], affect fault-proneness. The assumption is that such properties affects the cognitive complexity of the code, which in turn may affect how prone a programmer is to commit errors when developing or changing the code. There are numerous structural property measures proposed in the literature. Important sources in this field of research are the work by McCabe [31], Chidamber & Kemerer [30], Briand *et al.* [32, 33], and Li & Henry [34]. The metrics given in [30] are among the measures most widely used [4]. Many of these measures are, to various degrees, correlated with the size of the components being measured. This is not necessarily a problem depending on how the prediction model is intended to be used [1].

Studies have shown that not only structural properties are important predictors of fault-proneness, but also the history of an individual component and the experience of the developers should be considered when building fault-proneness prediction models. Graves *et al.* suggested that the mere change of a file itself is associated with fault-proneness [35]. Yu *et al.* showed that a component with a previous history of faultiness will continue to be faulty in the future due to possibly unknown underlying factors [36]. There are studies that include the number of distinct developers that have made changes to a component during its lifetime, assuming that one can expect more faults when developers share responsibility on a particular component with other developers, perhaps because (some of) the developers lack of understanding of the changes made by other developers. Further, it is reasonable to assume that it is easiest to make reliable changes to the code if the developer is familiar with the complete history of a component's functionality and code [18]. However, Graves *et al.* showed that the number of developers that had made changes to a module were not associated with fault-proneness [35].

There are a number of studies investigating if and how the three different categories of measures relate to fault-proneness. From Appendix A we can see that two thirds of the reviewed studies built prediction models using structural measures. The Chidamber and Kemerer metrics [30] are among the measures most often used. Only a few of the studies included process metrics, e.g., [16, 18, 25, 37]. Below, we briefly summarize how the various types of measures have typically been used in the reviewed studies.

Tomaszewski *et al.* [23] selected eight metrics out of 14 through a correlation analysis using Spearman Rho. Among the measures selected were WMC and RFC [30], maximum cyclomatic complexity [31] and some size metrics. In addition, the number of lines added or modified since the previous release was used. In fact, this change metric was the best individual predictor of fault density and number of faults.

In [38], the authors used fault and code measures data from the NASA Metrics Data Program (MDP). There were 21 measures available as candidate predictors. Four different data sets were used, and the most important metrics in each data set were selected using correlation-based feature selection (CFS) [39]. Depending on the data set used, the number of variables was reduced from 21 to three to seven. Among the variables selected were McCabe's cyclomatic complexity and Halstead's intelligent count and difficulty metrics [40]. Also included were several line count metrics: the number of lines including comments and number of blank lines.

Vandercruys *et al.* [41] also used data from the NASA MDP. By using a χ^2 -based filter, they selected only a subset of the metrics available – reducing the number of metrics to around 12 depending on the data set that was filtered. Among the metrics selected were Halstead volume and error estimate [40], cyclomatic complexity [31], as well as several size-related metrics such as the total lines of code and lines of comments.

Data available from the NASA MDP was also used in studies by Pai *et al.* [13] and Gondra [42]. Pai *et al.* used the subset of the metrics which are associated with the work of Chidamber and Kemerer: WMC, DIT, RFC, NOC, CBO, LCOM [30]. Their result showed that four metrics were significant in predicting fault-proneness: WMC, CBO, RFC and lines of code. DIT, NOC and to some degree LCOM, were not found to be significant. In [42], the system under study was a system written in C. Thus, we consider the metrics investigated in this study of less importance as our focus in this study is mainly on object-oriented systems. Gondra focused on the Halstead metrics suite and a selection of size metrics and the prediction models yielded an accuracy ranging from 0.73 to 0.87. Elish *et al.* [38] used the metrics available through NASA MDP to compare several data mining techniques. The models yielded an accuracy ranging from 0.83 to 0.93, and nearly all of the precision and recall measures were above 0.9.

Briand *et al.* [43] investigated the impact of a large number of metrics on fault-proneness; 28 coupling measures, 10 cohesion measures, 11 inheritance-related measures and 6 size measures. Each measure's impact on fault-proneness was evaluated through univariate logistic regression. Three multivariate models were built; one using size metrics alone, one including object-oriented measures like cohesion, coupling and inheritance, and one including both the size measures and the object-oriented metrics as candidate predictors. The best model in terms of correctness and completeness were the model based on object-oriented metrics alone, i.e., without the size metrics. This model obtained 92% completeness and 78% correctness using 10-fold cross validation, as apposed to 94% completeness and 81% correctness when assessing goodness-of-fit. The cross-validated accuracy of the model was 80%. Among the findings from the univariate analyses were that coupling measures related to the number of method invocations on a class X initiated from a class C, i.e., import coupling, have a significant impact on fault-proneness for class C. That is, measures like RFC [30] and the ICP measures defined in [44] seem to be related to fault-proneness. However, the fact that a class C is used by many other classes, i.e., high export coupling, seems to have little effect on C's fault-proneness. Both of these findings are also supported in [45] and, to some extent, in [46] and [47]. Contradicting evidence were found in [48], where export coupling measures were significantly associated with fault-proneness. Further findings in [43] were that some of the cohesion measures were significant with respect to fault-proneness ($\alpha=0.05$). However, there is some disagreement on what constitute a proper cohesion measure and the mathematical properties with which a cohesion measures should comply [49] [50] [51]. All the inheritance measures were significant predictors of fault-proneness ($\alpha =0.05$); that is, the more ancestors a class inherits from, or the deeper the class is in the inheritance hierarchy, the higher its fault-proneness. Further, as a class overrides more methods or adds new methods, its fault-proneness also increases [43].

In [9], Olague *et al.* evaluated three metric suites; 1) the metrics proposed by Chidamber and Kemerer [30], 2) the metrics proposed by Bansiya *et al.* [52], and 3) the metrics suite given by Brito e Abreau *et al.* [53]. Of the three metric sets, the Chidamber and Kemerer metrics resulted in the best models in terms of accuracy. Further, the only measures that were significantly associated with faults across 6 successive releases of the Rhino system [54] were RFC, CBO and WMC. The findings in [9] runs counter to [43]. In the former, the inheritance

measures were not significantly associated with fault-proneness, while significant results were found in the latter. However, the study by Briand *et al.* was performed in an academic setting at an undergraduate/graduate level. Lack of experience might have influenced the understanding and use of inheritance by the experiment subjects. The fact that inheritance measures DIT and NOC are not a significantly associated with fault-proneness is further supported in [46]. In a study by El Emam *et al.* [48] DIT was significantly associated with fault-proneness, while NOC was not.

The regression analysis done by Subramanyam *et al.* [24] suggested that the interaction between CBO and DIT has a significant impact on fault-proneness. A somewhat interesting result was the impact that CBO had at different depths in the inheritance hierarchy (DIT). In the C++ based system under study, the fault-proneness of classes with higher CBO values was significantly larger for classes deeper down in the inheritance hierarchy.

Zhou *et al.* [10] distinguished between low and high severity faults. The results showed that design metrics like CBO, DIT, WMC, RFC and LCOM were highly effective in predicting low severity faults. However, none of the metrics led to suitable models to predict high severity faults.

Although most of the research done in recent years focused on the impact of structural properties on fault-proneness, a number of studies investigated other types of fault-proneness factors. For example, Nagappan *et al.* [25, 28] used *code churn* together with dependency metrics to predict fault-prone modules. Code churn is a measure of the amount of code change within a component over time. Graves *et al.* [35] counted the number of changes done in a module as well as the average age of the code. Referring to Graves *et al.*, Weyuker *et al.* constructed a fault-count prediction model using a number of process measures in addition to structural measures. Weyuker *et al.* accounted for the number of developers who modified a file during the prior release, and the number of *new* developers involved on a particular file. In addition, they counted the cumulative number of distinct developers who have modified a file during its lifetime. The model using these process measures showed only slight improvements compared with a model using only structural measures.

Khoshgoftaar *et al.* [37] considered 14 process metrics, such as a variable counting the number of updates done by designers who had 10 or less total updates in their entire company career, the number of different designers making changes to a particular module, and the net increase in lines of code (LOC) for each module. Khoshgoftaar *et al.* did not study the impact of the individual measures on fault-proneness, but their prediction models achieved Type I and Type II misclassification rates ranging within 25-30%. In [22], Kim *et al.* used deltas from 61 complexity metrics and a selection of process metrics, and achieved an accuracy ranging from 64% to 92% on twelve open source applications.

Most of the studies reviewed here considered structural measures, and there is considerable evidence that coupling measures (such as CBO [30]) have an impact on fault-proneness. Further, there is conflicting evidence on how inheritance measures affect fault-proneness although the overall trend indicates that inheritance measures (such as DIT and NOC [30]) alone are not strongly associated with fault-proneness. However, as there might be interaction effects between inheritance measures and other metrics, this should be investigated further. As for cohesion metrics, there is some empirical evidence suggesting that low cohesion is associated with fault-proneness, but the results is not nearly as clear and strong as for coupling. Some of the studies use process measures and deltas to assess fault-proneness. Most of the studies combine these

measures with structural measures. Thus, based on the results of the reviewed studies, it is difficult to assess the impact that process measures and deltas alone have on fault-proneness.

Although there is some empirical evidence regarding what factors drive fault-proneness, building prediction models will remain an exploratory process as we have to expect wide variations across datasets.

2.3 Data Modeling Techniques

Building fault-proneness prediction models has been a field of research for decades, but there is still a need for an exploratory process as the number of variables often is large and their inter-relationship and impact on fault-proneness is currently unknown. The field of data mining and knowledge discovery facilitates the exploratory nature of building fault-proneness prediction models.

There exists a large number of data analysis and mining techniques to build a fault-proneness model, such as classification models determining whether classes or files are faulty. A classical statistical technique used in many existing papers is logistic regression [55]. But many techniques are also available from the fields of data mining, machine learning, and neural networks [3]. One important category of machine learning techniques focuses on building decision trees, which recursively partition a data set, and the most well-known algorithm is probably C4.5 [56]. In our context, each leaf of a decision tree would then correspond to a subset of the data set available (e.g., characterized by components' source code characteristics and their fault/change history, as described in Section 3.4) and this leaf's fault frequency distribution can be used for prediction when all the conditions leading to that leaf are met. Another similar category involves coverage algorithms that generate independent rules where a number of conditions are associated with a probability for a component to contain a fault based on the instances each rule covers in the data set. As opposed to the divide-and-conquer strategy of decision trees, these algorithms iteratively identify attribute-value pairs that maximize the probability of the desired classification and, after each rule is generated, remove the instances that it covers before identifying the next optimal rule.

Both decision tree or coverage rule algorithms generate models that are easy to interpret (logical rules associated with probabilities) and therefore tend to be easier to adopt in practice as practitioners can then understand why they get a specific prediction. Furthermore they are easy to build (many freely available tools exist) and apply as they only involve checking the truth of certain conditions. Another advantage is that, instead of relying on model-level accuracy (e.g., like for Logistic Regression), each rule or leaf has a specific expected accuracy. The level of expected accuracy associated with a prediction therefore varies across predictions depending on which rule or leaf is applied.

Other common techniques include Neural networks, for example the classical back-propagation algorithm [57], which can also be used for classification purposes. A more recent technique that has received increased attention in recent years across various scientific fields [58-60] is the Support Vector Machine classifier (SVM) [3], which attempts to identify optimal hyperplanes with nonlinear boundaries in the variable space in order to minimize misclassification.

Machine learning techniques, such as classification trees, can be improved in terms of accuracy by using metalearners. For example, decision trees are inherently unstable due to the way their learning algorithms work: a few instances can dramatically change variable selection

and the structure of the tree. The Boosting [3] method combines multiple trees, implicitly seeking trees that complement one another in terms of the data domain where they work best. Then it uses voting based on the classifications yielded by all trees to decide about the final classification of an instance. How the trees are generated differ depending on the specific algorithm, and one of the well-know algorithm is AdaBoost [61], which is designed specifically for classification algorithms. It iteratively builds models by encouraging successive models to handle instances that were incorrectly handled in previous models. It does so by re-weighting instances after building each new model and builds the next model on the new set of weighted instances. Another metalearner worth mentioning is named Decorate [3]. This recent technique is claimed [62] to consistently improve not only the base model but also outperform other techniques such as Bagging and Random forest. It is also supposed to outperform boosting on small training sets and rival it on larger ones [3].

Another way to improve classifier models is to use techniques to pre-select variables or features, to eliminate most of the irrelevant variables before the learning process starts. When building models to predict fault-prone components, we often do not have a strong theory to rely on and the process is rather exploratory. As a result, we often consider a large number of possible predictors, many of which turn out not to be useful or strongly correlated. Though in theory the more information one uses to build a model, the better the chances to build an accurate model, studies have shown that adding random information tends to deteriorate the performance of C4.5 classifiers [3]. This happens because as the tree gets built, the algorithm works with a decreasing amount of data, which may increasingly lead to chance selection of irrelevant variables. The number of training instances needed for instance-based learning increases exponentially with the number of irrelevant variables present in the data set. Strong inter-correlations among variables also affect variable selection heuristics in regression analysis [55]. A recent paper [63] compared various variable selection schemes. The authors concluded by recommending a number of techniques which vary in terms of their computational complexity. Among them, two efficient techniques were reported to do well: CFS [39] and ReliefF [64].

Because it is a standard and well-established approach, multivariate logistic regression seems to be one of the most popular techniques for building fault-proneness models, e.g., [65], [43], [48], [23], [25], [9] and [45]. In these studies, the dependent variable is dichotomous: it reflects whether or not a component contains a fault that was uncovered either during system test or operation. Such models output the probability that a given component contains one or more faults. Other studies count the number of faults that has previously occurred in a component, and use this count as a dependent variable, e.g., [15, 17] and [11]. Ostrand *et al.* [15, 17] applied negative binomial regression, which is a suitable regression technique when dealing with skewed right-tail count distributions with low averages. The output of a negative binomial regression model is a conditional probability that a component contains n faults, e.g., “given that a component has a coupling equal to 3 and a cyclomatic complexity equal to 8, what is the probability that the component contains 2 faults?”. Because in most cases, a majority of classes do not contain faults and many fault count distributions show a median close to zero, zero inflated regression models might be more appropriate [11]. Janes *et al.* compared regular Poisson regression with negative binomial regression, with and without the zero-inflated version [11]. The zero-inflated approach yielded the best results in terms of what percentage of classes needed to be inspected to find 80% of the faults.

As discussed in Section 2.2, object-oriented structural measures are among the most frequently used predictors of fault-proneness. A possible problem using these measures in the

context of regression techniques is that they often are correlated [4]. When this correlation is extreme, the estimation of coefficients in logistic regression becomes difficult and inaccurate, a problem referred to as multicollinearity [55]. One way of dealing with multicollinearity is to apply *principal component analysis* (PCA) [66]. Principal component analysis creates a number of orthogonal (uncorrelated) principal components (PCs) that are linear combinations of the original independent variables. These PCs may be applied directly as new independent variables in a regression model. Alternatively, PCA can be used to select a subset of the variables, e.g., by selecting the variable with the highest loading within each PC, and use these variables as independent variables. Also, PCA can be used simply to analyze the dimensions captured by a set of measures and help understand what these measures really capture [46].

Another way of dealing with multicollinearity is to examine the *variance inflation factor* (VIF). For each coefficient, VIF measures how much of the variance is inflated due to collinearity compared to what the variance would have been if there was no multicollinearity. Although one should be careful to use specific thresholds as a rule of thumb [67], VIF values greater than 10 may indicate multicollinearity problems and these variables should be investigated further [68].

Though extreme multicollinearity among independent variables may lead to unstable coefficients, misleading statistical tests, and unexpected coefficient signs [6], in the context of prediction models the main purpose is not to interpret the coefficients to explain why a class has a certain fault-proneness. Thus, multicollinearity is not a major problem if it remains at moderate levels. Nonetheless, one should be aware that testing the significance of the independent variables in a multivariate model is unreliable when multicollinearity is present.

Some of the studies applying multivariate regression, e.g., [25, 26], used PCA to alleviate multicollinearity issues. Others, such as [9, 48], applied univariate analysis on each measure, and built a prediction model using those measures that are significant with respect to fault-proneness. Others again, used either forward or backward stepwise regression to select significant variables [46].

Lately, there has been an increasing interest in alternatives to logistic regression. Briand *et al.* discussed the downsides on using traditional regression techniques, and suggested using multivariate adaptive regression splines (MARS) [69], because MARS suites the exploratory nature of building prediction models [46]. The MARS model performed slightly better in terms of accuracy, completeness and correctness, compared to logistic regression. Also, the authors did a cost/benefit analysis similar to those of an Alberg-diagram [70], which suggested the MARS model outperformed the model built using logistic regression in terms of cost-effectiveness.

Khoshgoftaar *et al.* [19] compared seven models that were built using a variety of tools. The models were built using different regression and classification trees including C4.5, CHAID, Sprint-Sliq and different versions of CART. Also included in the study were logistic regression and case-based reasoning. The techniques were evaluated against each other by comparing a measure of expected cost of misclassification. The differences between the techniques were at best moderate.

Vandecruys *et al.* compared Ant Colony Optimization against well-known techniques like C4.5, support vector machine (SVM), logistic regression, K-nearest neighbour, RIPPER and majority vote [41]. In terms of accuracy, C4.5 was the best technique. However, the differences between the techniques in terms of accuracy, sensitivity and specificity were moderate.

Kanmani *et al.* [12] compared two variants of artificial neural networks against logistic regression and discriminant analysis. Neural network outperformed the traditional statistical

regression techniques in terms of correctness and completeness. The possible benefits of neural networks was also explored by Gondra [42]. In addition, Gondra studied the usefulness of support vector machines (SVMs) to perform simple classification. When considering fault-proneness as a binary classification problem (i.e. faulty vs. non-faulty) using a threshold of 0.5, the accuracy was 87,4% when using SVM compared to 72,61% when using neural networks – suggesting that SVM is a promising technique for classification within the domain of fault-proneness prediction. Success in using SVMs is also reported in [38], where SVM was evaluated against eight other data mining techniques; logistic regression, neural network, radial basis function, Bayesian belief network, naïve Bayes, Random Forest and the C4.5 decision tree algorithm. There were some statistically significant differences between the techniques, but the differences were quite small from a practical standpoint.

Guo *et al.* [21] compared Random Forest [71] with 26 other modeling techniques including logistic regression and 20 techniques available through the WEKA tool. The study compared the techniques using five different datasets from the NASA MDP program, and although the results showed that Random Forests perform better than many other classification techniques in terms of accuracy and specificity, the results were not significant in four of the five data sets. In Elish [38], the authors compared SVM against eight other modeling techniques, among them Random Forest. The modeling techniques were evaluated in terms of accuracy, precision, recall and the F-measure using four data sets from the NASA MDP program. All techniques achieved an accuracy ranging from approximately 0.83 to 0.94. As with the other studies reviewed here, there were some differences, but no single modeling technique was significantly better than the others across data sets.

In this section, we have elaborated on the model building techniques that typically have been used to build fault proneness prediction models. Seven of the studies reviewed compared several modeling techniques [10, 19, 21, 38, 41, 72, 73]. The overall trend seem to be that there are some differences between techniques, but there are wide variations across datasets and studies in terms of which technique yield the best models. In addition, there is little consistency on how the models are evaluated, a topic elaborated in the next section. Thus, it is difficult to compare the results and draw conclusions from these studies.

2.4 Evaluation Criteria and Methods

In this section we describe how fault-proneness prediction models should be evaluated, in terms of evaluation criteria and evaluation methods. First, in terms of evaluation criteria, there are three main aspects of the “quality” of prediction models that we may want to assess:

- *Goodness-of-fit* tells us how well the model explains the data that were used to build the model. Among the most popular measures for models with a continuous dependent variable is the coefficient of determination, R^2 , which is the amount of variability in the dependent variable that is explained by the model.
- *Predictive power* is an assessment of how the model performs when predicting based on data that was not used to build the model and that may represent more recent observations.
- *Cost/benefits assessments* tells us what are the costs accompanied with applying a particular prediction model, e.g., the costs of data collection and

model building, and what benefits can be drawn from using this model, e.g., less latent faults and improved quality using less resources. Measures of costs and benefits tend to be context-dependent.

In many cases, fault-proneness is measured on a continuous scale, for example as a fault count or fault density. However, many of the model building techniques described in the previous subsection are classifiers. That is, they classify software components as faulty or non-faulty. Or rather than a mere classification, most of the classifiers output a probability for a component to be faulty. To distinguish, based on such probabilities, faulty classes from non-faulty-ones, one is required to predefine a certain threshold, or cut-off probability value. By default, this cut-off is 0.5, i.e., components having a probability $p > 0.5$ are classified as faulty whereas the remaining classes are classified as non-faulty. Since we cannot expect a classifier to be 100% accurate, some instances will not be correctly classified. These instances fall into two categories: Type I errors and Type II errors, or false positives and false negatives, respectively. A false positive is a non-faulty class erroneously classified as fault-prone, while a false negative is a faulty class that is misclassified as non-faulty. By varying the cut-off value, one can to some degree control the ratio of false positives versus false negatives.

In the context of software development and testing, the later you discover a fault the more expensive it is to fix. Hence, if the scope of a fault-proneness prediction model is to focus testing activities, the cost of missing a faulty class (i.e., a false negative) will in most cases outweigh the cost of testing a non-faulty class (i.e., a false positive). A confusion matrix, shown in Figure 1, can be used to show to relative frequency or number of false positives and false negatives compared to the ratio or number of correctly classified instances, i.e., true positives and true negatives. Many of the measures that are used to evaluate classifiers can be derived from the confusion matrix. A selection of these measures are explained below. However, although they give an indication as to how well a particular prediction model performs in terms of classification accuracy, they are not directly linked to the possible cost-effectiveness of using such models. Towards the end of this section we elaborate on some criteria that can be used to assess the cost-effectiveness of prediction models.

		Actual	
		Positive	Negative
Predicted by model	Positive	True positive (TP)	False positive (FP)
	Negative	False negative (FN)	True negative (TN)

Figure 1: The confusion matrix

One of the popular measures in the literature that can be derived from the confusion matrix is *accuracy*. Accuracy is the ratio of correctly classified instances.

$$\text{Accuracy} = \frac{TP+TN}{TP + FP + TN + FN}$$

However, the accuracy measure is somewhat ambiguous; although an accuracy of exactly 1 indicates that all instances are correctly classified, an accuracy of 0.8 reflect that 80% of the instances are correctly classified; it does not state whether the remaining 20% are mainly false positives or false negatives. Thus, if we want to determine an appropriate trade-off between type I and type II errors, accuracy is not a suitable measure.

Sensitivity and *specificity* are fine-grained measures that enable us to assess the trade-off between type I and type II errors. The former measure is the percentage of actual positives that are correctly classified, i.e., in our context, the percentage of faulty components classified as such. Sensitivity serves as a measure of how many faulty components we are likely to find (or conversely miss) if we use the prediction model.

$$\begin{aligned}\text{Sensitivity} &= \text{TP} / (\text{TP} + \text{FN}) \\ \text{Specificity} &= \text{TN} / (\text{FP} + \text{TN})\end{aligned}$$

Specificity is the number of non-faulty components correctly classified. Sensitivity is also referred to as *recall*, and is not to be confused with the term *completeness*, which in our context would be defined as the number of faults found in the components classified as fault-prone divided by the total number of faults in the system [4].

Precision is often used in conjunction with recall (sensitivity). It is the number of instances correctly classified as fault-prone (true positives) divided by the total number of instances classified as fault-prone.

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$$

It is possible to increase recall by lowering the cut-off value described earlier. In practice, however, this often results lower precision; as we are lowering the threshold, more classes are erroneously predicted as faulty (false positives) and precision drops.

Khoshgoftaar *et al.* suggested that fault-proneness models should be evaluated using two additional evaluation criteria allowing one to assess the inaccuracy of prediction models [74]. The authors defined *Type I* and *Type II misclassification rates* as the ratio of Type I and Type II errors respectively.

$$\begin{aligned}\text{Type I misclassification rate} &= \text{FP} / N \\ \text{Type II misclassification rate} &= \text{FN} / N\end{aligned}$$

A major part of the studies reviewed use a number of the measures derived from the confusion matrix to evaluate their models. Although most studies use the standard cut-off of 0.5 to distinguish the fault-prone components from the less faulty ones, some studies vary the cut-off to find an optimal trade-off between Type I and Type II errors, e.g., [37, 72]. Because the Type II errors are considered the most expensive of the two types of errors, the importance of considering Type II misclassification rates is emphasized by Ostrand *et al.* [16].

All the measures described up to this point are evaluation criteria for classifiers. That is, in the context of fault-proneness models, these measures assess the accuracy of a particular model with regards to component fault-proneness classification. These measures require that one predefines a cut-off probability value, and although these measures are useful, the intent of fault-proneness prediction models is not only to classify instances. For example, if the prediction model was to be used to focus testing on fault-prone components, we would be more interested in the *ranking* of the component, and use the ranking to prioritize their testing. Consequently, it would be preferable to be able to assess how well a particular model is at ranking instances in a correct manner. Further, it would be preferable to evaluate the performance of a prediction model

without first having to choose a specific cut-off value. This is the objective of the receiver operating characteristic (ROC) curve. The ROC curve is a plot of sensitivity versus 1-specificity. Thus, the ROC curve depicts the benefits of using the model (true positives) versus the costs of using the model (false positives) at different thresholds. The ROC curve allows one to assess performance of a prediction model in general – regardless of any particular cut-off value. The area under the ROC curve can be used as a descriptive statistic, and is the estimated probability that a randomly selected positive instance will be assigned a higher predicted p by the prediction model than another randomly selected negative instance [75]. Hence, this statistic quantifies a model’s ability to correctly rank instances. Not many studies use ROC curves to assess the performance of their models. Khoshgoftaar *et al.* used the ROC curve to optimize the models by selecting an appropriate probability cut-off used to distinguish between faulty and non-faulty components [48]. Arisholm *et al.* used the area under the ROC curve to compare prediction models [73].

In addition to the regular confusion matrix criteria, some studies assess the usefulness of their prediction models using measures of a more practical nature. One example is the *expected cost of misclassification* [76]. However, as stated in [19], one should be careful about using this measure for model selection purposes. Another evaluation method is the *Alberg-diagram* [70], in which the components first are sorted in descending order according to their (predicted) fault probability. The x-axis shows the cumulative number of components, whereas the y-axis shows the cumulative number of faults. This curve can then be compared with an *optimal* curve, in which the components are sorted according to the actual number of faults instead of the fault probability. Three studies [11, 13, 65] evaluate their prediction models using Alberg-diagrams in addition to some of the confusion matrix criteria and R^2 . Ostrand *et al.* propose another measure to assess the usefulness of their models; the percentage of faults included in the 5% to 20% of the most fault-prone components as predicted by the model [15, 16, 18]. Tomaszewski *et al.* proposed a measure of the presumed cost reduction in terms of percentage of faults found compared to not using any model at all, and a model based on size, i.e., where the fault-prone components are selected according to their size as larger components (are assumed) to be more fault-prone [23]. Further, Tomaszewski *et al.* compared their prediction models against an optimal model. Ostrand *et al.* assess their models by investigating whether the percentage of LOC in 20% of the files predicted as most fault-prone is smaller than the percentage of faults [17].

Though relevant, the problem with most existing evaluation criteria is that they do not clearly and directly relate to the *cost effectiveness* of using fault-proneness prediction models. For example, assuming a class is predicted as very likely to be faulty, one would take corrective action by investing additional effort to inspect and test the class. Furthermore, if we assume that the *cost* of such activities might be roughly proportional to some property of that class, e.g., its size or complexity, such properties can be used as surrogate measures of the verification cost. The choice of surrogate measure will depend on the specific verification activities undertaken. In our previous work we proposed such a cost-effectiveness measure that not only considers the accuracy of the predictions, but also accounts for the assumed cost of using the model to focus verification and validation activities [72, 73]. Further details will be elaborated in Section 3.7.

The above discussions focused on the evaluation criteria that can be used. How the prediction model is applied for the purpose of model assessment is also of crucial importance. Many studies evaluate their fault-proneness models by applying it to the same data set that was used to build the model, e.g., [11, 16, 20, 23, 24]. These studies are merely doing a goodness-of-

fit analysis, that is, they assess how well the model is at explaining the data that were used to build the model. If the intent is to use these fault-proneness models to predict fault-proneness on new and unknown data, this procedure is not suitable. Rather, a prediction model should be evaluated on new data to obtain more sensible measures of the predictive power of a particular model. Mainly, there are two ways of validating a prediction model; either by (1) dividing the data set in two parts; one training set and one test set (hold-out validation), or (2) by doing what is called k -fold cross-validation. In hold-out validation, typically 2/3 of the data forms the training set, while 1/3 is used as a test set to validate the model. This procedure is suitable when the data set is large enough to allow a proper training set to be formed. If the data set is small, one may resort to k -fold cross validation, where the data set is divided into k parts. Then, k models are built where each of the k subsets is successively used as a test set, while the other $k-1$ subsets form the training set. As k models have to be built, k -fold cross-validation is more computationally intensive than a simple hold-out cross-validation. Ultimately, in the context of fault-proneness prediction models, the training set should consist of one or several releases of a software system, while the training set should consist of later releases of the same system or even releases of another system. Nearly a third of the studies reviewed use later releases as separate test sets [8, 9, 15, 17-19, 27, 73]. Two of these studies apply the prediction models on another system [8, 17].

A major issue in the studies reviewed, is the fact that the models are evaluated using criteria that are not directly linked to the possible costs and benefits of using the prediction models in different contexts, e.g., focusing verification and validation efforts. Further, many studies only considers goodness-of-fit, and do not assess the predictive power of their models on new data, and thus they run the risk of having models that are overfitted, giving optimistic estimates of predictive power. Further, as each study use different evaluation criteria, comparisons of results across studies are difficult.

2.5 Types of System

There are many different types of system that have been investigated in recent studies, ranging from large industry systems consisting of hundreds of thousands lines of code, to small systems developed by students in an academic setting. All of the studies reviewed are case studies. So far, there exists no experiment in a controlled environment, although this presumably could be beneficial to obtain a more in-depth understanding of the causal relationships between candidate predictors and fault-proneness, as illustrated by the somewhat inconsistent results reported in Section 2.2.

A large part of the studies reviewed here collected data from large commercial or legacy software projects [8, 11, 14-20, 23-27, 37, 72, 73]. Many studies [10, 13, 21, 38, 41, 42] used the data sets available through NASA MDP, making it possible to compare results across studies. With the increasing popularity and availability of open source software projects, some studies [9, 22, 65] rely on open source system repositories. However, a study by Chen *et al.* [77] uncovered large deficiencies in such repositories. Thus, researchers need to take great care in using open source software as research subjects.

Two thirds of the studies reviewed investigate object-oriented systems, while a minor part of the studies investigate systems written in a procedural language. Others again, investigate systems developed using both paradigms; mainly those studies that use the data available through NASA MDP.

2.6 Summary

In summary, few studies have compared a comprehensive set of data mining techniques for building fault-proneness prediction models to assess which techniques are more likely to be accurate in various contexts. Most models were evaluated through different confusion matrix criteria and, as a result, it is difficult to provide general conclusions. However, results suggest that the differences between modeling techniques might be relatively small. Most existing studies have used structural measures as candidate predictors whereas only a subset have also included other measures, usually more expensive to collect, such as code churn and process measures. However, no studies have so far attempted to evaluate the benefits of including such measures in comparison with models that contain only structural code measures. In this paper, we assess, in a systematic way, how both the choice of modeling technique and the selection of different categories of candidate measures affect the accuracy and cost-effectiveness of the resulting prediction models based on a complete set of evaluation criteria. We furthermore assess how the choice of evaluation criteria affects what is deemed to be the “best” prediction model.

3 Design of Study

When building fault-proneness prediction models, many decisions have to be made regarding the choice of dependent and independent variables, modeling technique, evaluation method and evaluation criteria. As discussed in the previous section, no systematic study has been performed to assess the impact of such decisions on the resulting prediction models. This paper compares alternative fault-proneness prediction models where we systematically vary three important dimensions of the modeling process: modeling technique (e.g., C4.5, neural networks, logistic regression), categories of independent variables (e.g., process measures, object-oriented code structural measures, code churn measures) and evaluation criteria (e.g., accuracy, ROC, and cost-effectiveness). We assess (i) to what extent different data mining techniques affect prediction accuracy and cost effectiveness, (ii) the effects of using different sets of measurements (with different data collection costs) on the accuracy and cost-effectiveness of the fault-proneness predictions models, and (iii) how our decisions in terms of selecting the “best” model would be affected by using the different evaluation criteria. This section describes the development project, study variables, data collection, and model building and evaluation procedures.

3.1 The Development Project

The legacy system studied is a Java middleware system called COS, serving the mobile division in a large telecom company. COS provides more than 40 client systems with a consistent view across multiple back-end systems, and has evolved through 22 major releases during the past eight years. At any point in time, 30 to 60 software engineers were involved in the project. The core system currently consists of more than 2600 Java classes amounting to about 148 KSLOC. In addition to this, the system consists of a large number of test classes, library classes, and about 1000 KSLOC of generated code, but this code is not considered in our study. As the system expanded in size and complexity, QA engineers felt they needed more sophisticated techniques to focus verification activities on fault-prone parts of the system. We used 13 recent releases of this system for model building and evaluation. As a first step, the focus was on unit testing in order to eliminate as many faults as possible early on in the verification process by applying more stringent test strategies to code predicted as fault-prone.

3.2 Data Collection Procedures

Perl scripts were developed to collect file-level change data for the studied COS releases through the configuration management system (MKS). In our context, files correspond to Java public classes. The data model is shown in Figure 2. Each change is represented as a change request (CR). The *CR* is related to a given *releaseId* and has a given *changeType*, defining whether the change is a critical or non-critical fault correction, a small, intermediate, or large requirement change, or a refactoring change. An individual developer can work on a given CR through a logical work unit called a change package (CP), for which the developer can check in and out files in relation to the CR. For a CP, we record the number of CRs that the responsible developer has worked on prior to opening the given CP, and use this information as a surrogate measure of that person's coding experience on the COS system. For each *Class* (file) modified in a CP, we record the number of lines added and deleted, as modeled by the association class *CP_Class*. Data about each file in the COS system is collected for each release, and is identified using a unique *MKSId*, which ensures that the change history of a class can be traced even in cases where it changes location (package) from one release to the next. This traceability turned out to be crucial in our case because we wanted to keep track of historic changes and faults for each class, and there were quite a few refactoring changes in the project that would result in loss of historic data if we did not use the *MKSId* to uniquely identify each class. Finally, for each release, a code parser (JHawk [78]) is executed to collect structural measures for the class, which are combined with the MKS change information. Independent (change, process, and code structure measurements) and dependent variables (Faults in the next release) were computed on the basis of the data model presented in Figure 2.

3.3 Dependent Variable

The dependent variable in our analysis was the occurrences of corrections in classes of a specific release which are due to field error reports. Since our main current objective was to facilitate unit testing and inspections, the class was a logical unit of analysis. Given that our aim was to capture the fault-proneness of a class in a specific release n , and that typically a fault correction involved several classes, we decided to count the number of distinct fault corrections that was required in each class for developing release $n+1$. Furthermore, in this project, only a very small portion of classes contained more than one fault for a given release, so class fault-proneness in release n is therefore treated as a classification problem and is estimated as the probability that a given class will undergo one or more fault corrections in release $n+1$.

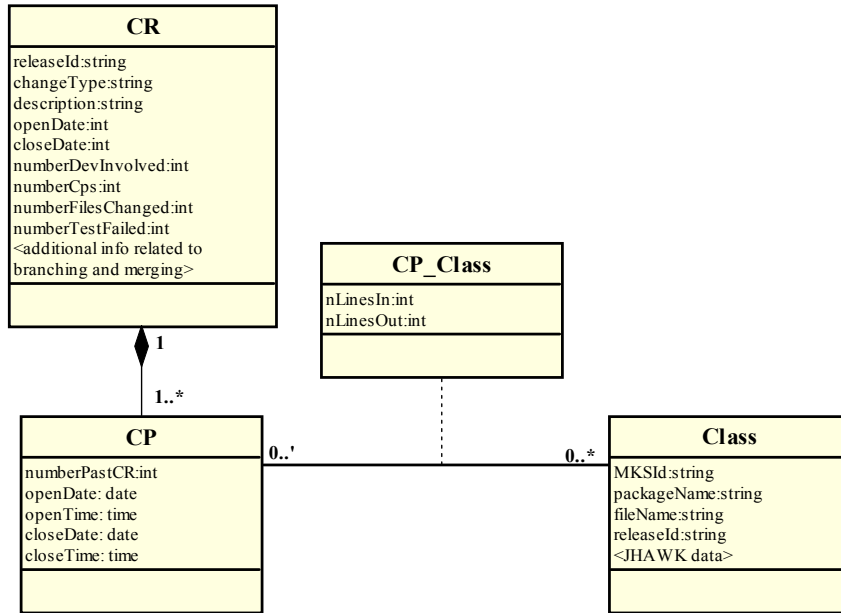


Figure 2: Data Model

3.4 Explanatory Variables

Though many studies on predicting fault-prone classes on the basis of the structural properties of object-oriented systems have been reported (Section 2), a specificity of the study presented here is the fact that we needed to predict fault-proneness for a changing legacy system. Thus, in addition to structural measures, similar to other studies [16-18, 22, 25, 28, 35, 37, 73, 79, 80] we also use data on changes and fault corrections for specific releases and their impact on the code. In our context, past change and fault data could be useful to help predicting fault-proneness by identifying what subset of classes have shown to be inherently fault and change prone in the past. Our explanatory variables can be classified into three categories:

- Object-oriented (OO) code measures, i.e., measures of structural properties derived from the source code. In this study, the JHawk tool was used to collect such measures, as shown in Table 1.
- Delta measures: These measures capture the amount of change – sometimes called churn – in a file between two successive releases. In this study, the delta measures were computed from the JHawk measures given in Table 1.
- Process measures: In this study, the process measures were collected from the configuration management system (MKS), and included a surrogate measure of the experience of each developer performing each change, the number of developers that have made changes to a file, the number of faults in previous release(s) and simpler measures such as the accumulated number of lines added and/or removed in a given release.

The fundamental hypothesis underlying our work is that the fault-proneness of classes in a legacy, object-oriented system can be affected by these measures. Furthermore, it is also likely

that these factors interact in the way they affect fault-proneness. For example, changes may be more fault-prone on larger, more complex classes. The data mining techniques used to build the models will account for such interactions.

The three categories of measures (*OO*, *Delta* and *Process*) incur different costs in terms of data collection effort and process instrumentation requirements. *OO* measures can be collected from simple code snapshots, *Deltas* require that different versions of the system be available, whereas *Process* measures require that developers record detailed information about their work (e.g., changes and fault corrections, developer info, time of changes, whether a change passed certain test procedures) in a systematic and consistent way in configuration management or change management systems. To assess the relative importance of the individual categories of explanatory variables (*OO*, *Delta* and *Process*), they were combined to construct seven different candidate metric sets (*OO*, *Delta*, *Process*, *OO + Delta*, *Process + OO*, *Process + Delta*, *Total*). In Section 4.2 we will show how the many different measures of accuracy and cost effectiveness of the fault-proneness prediction models are affected by the choice of metric set. In this way, we will not only be able to compare individual categories of measures (e.g., *Process* vs. *OO*) but also assess the potential impact of *combining* measures (e.g., *Process + OO*) with regards to a comprehensive set of evaluation criteria (Section 3.7). Based on such analyses, we will be in a better position to determine whether the added cost of collecting, for example, process measures will result in payoffs in terms of better fault-proneness prediction models.

3.5 Model Building Techniques

A detailed description of many of the most popular techniques for building fault-proneness prediction models was provided in Section 2. In this study we compared one classification tree algorithm (C4.5) as it is the most studied in its category, the most recent coverage rule algorithm (PART) which has shown to outperform older algorithms such as Ripper [3], Logistic Regression as a standard statistical technique for classification, Back-propagation neural networks as it is a widely used technique in many fields, and SVM.

For C4.5, we also applied the AdaBoost and Decorate metalearners [3], because decision trees are inherently unstable due to the way their learning algorithms work, and thus we wanted to assess the impact of using metalearners on C4.5. We included Decorate in addition to Adaboost because it is supposed to outperform boosting on small training sets and rivals it on larger ones.

Furthermore, as the outputs of leaves and rules are directly comparable, we combined C4.5 and PART predictions by selecting, for each class instance to predict, the rule or leaf that yields a fault probability distribution with the lowest entropy (i.e., the fault probability the furthest from 0.5, in either direction). This allows us to use whatever technique works best for each prediction instance.

For each metric set, we also used CFS (Correlation-based Feature Selection) [39] to pre-select variables, as further described in Section 2, to assess the effect of such variable pre-selection on the prediction model performance.

All of the above techniques were applied using the WEKA tool and are described in [3]. An attempt was made to optimize the parameters of various techniques, but in most cases the impact of varying these parameters was small and we resorted to using the WEKA default parameters.

Table 1: Summary of the explanatory variables

	Variable	Description	Source	
OO	No_Methods NOQ NOC	Number of [implemented query command] methods in the class	JHawk	
	LCOM	Lack of cohesion of methods	JHawk	
	TCC MAXCC AVCC	[Total Max Avg] cyclomatic complexity in the class	JHawk	
	NOS UWCS	Class size in [number of Java statements number of attributes + number of methods]	JHawk	
	HEFF	Halstead effort for this class	JHawk	
	EXT/LOC	Number of [external local] methods called by this class	JHawk	
	HIER	Number of methods called that are in the class hierarchy for this class	JHawk	
	INST	Number of instance variables	JHawk	
	MOD	Number of modifiers for this class declaration	JHawk	
	INTR	Number of interfaces implemented	JHawk	
	PACK	Number of packages imported	JHawk	
	RFC	Total response for the class	JHawk	
	MPC	Message passing coupling	JHawk	
	FIN	The sum of the number of unique methods that call the methods in the class	JHawk	
	FOUT	Number of distinct non-inheritance related classes on which the class depends	JHawk	
	R-R S-R	[Reuse Specialization] Ratio for this class	JHawk	
	NSUP NSUB	Number of [super sub] classes	JHawk	
	MI MINC	Maintainability Index for this class [including not including] comments	JHawk	
	Delta		<i>For each OO measure X above:</i>	
		delta_<X>	The difference in each OO measure X between two successive releases.	Calculated
Process	[nm1 nm2 nm3]_CLL_CR	The number of large requirement changes for this class in release [n-1 n-2 n-3]	MKS	
	[nm1 nm2 nm3]_CFL_CR	The number of medium requirement changes for this class in release [n-1 n-2 n-3]	MKS	
	[nm1 nm2 nm3]_CKL_CR	The number of small requirement changes for this class in release [n-1 n-2 n-3]	MKS	
	[nm1 nm2 nm3]_M_CR	The number of refactoring changes for this class in release [n-1 n-2 n-3]	MKS	
	[nm1 nm2 nm3]_CE_CR	The number of critical fault corrections for this class in release [n-1 n-2 n-3]	MKS	
	[nm1 nm2 nm3]_E_CR	The number of noncritical fault corrections for this class in release [n-1 n-2 n-3]	MKS	
	numberCRs	Number of CRs in which this class was changed	MKS	
	numberCps	Total number of CPs in all CRs in which this class was changed	MKS	
	numberCpsForClass	Number of CPs that changed the class	MKS	
	numberFilesChanged	Number of classes changed across all CRs in which this class was changed	MKS	
	numberDevInvolved	Number of developers involved across all CRs in which this class was changed	MKS	
	numberTestFailed	Total number of system test failures across all CRs in which this class was changed	MKS	
	numberPastCr	Total developer experience given by the accumulated number of prior changes	MKS	
	nLinesIn	Lines of code added to this class (across all CPs that changed the class)	MKS	
	nLinesOut	Lines of code deleted from this class (across all CPs that changed the class)	MKS	
		<i>For CRs of type Y={CLL, CFL, CKL, M, CE, E}:</i>		
	<Y>_CR	Same def as <i>numberCRs</i> but only including the subset of CR's of type Y	MKS	
	<Y>_Cps	Same def as <i>numberCpsForClass</i> but only including the subset of CR's of type Y	MKS	
	<Y>numberCps	Same def as <i>numberCps</i> but only including the subset of CR's of type Y	MKS	
	<Y>numberFilesChanged	Same def as <i>numberFilesChanged</i> but only including the subset of CR's of type Y	MKS	
	<Y>numberDevInvolved	Same def as <i>numberDevInvolved</i> but only including the subset of CR's of type Y	MKS	
	<Y>numberTestFailed	Same def as <i>numberTestFailed</i> but only including the subset of CR's of type Y	MKS	
<Y>numberPastCr	Same def as <i>numberPastCr</i> but only including the subset of CR's of type Y	MKS		
<Y>nLinesIn	Same def as <i>nLinesIn</i> but only including the subset of CR's of type Y	MKS		
<Y>nLinesOut	Same def as <i>nLinesOut</i> but only including the subset of CR's of type Y	MKS		

3.6 Training and Evaluation Datasets

To build and evaluate the prediction models, class-level structural and change/fault data from 13 recent releases of COS were used. The data was divided into four separate subsets, as follows. The data from the 11 first releases was used to form two datasets, respectively a training set to build the model and a test set to evaluate the predictions versus actual class faults. More specifically, following the default setting of most tools, two thirds of the data (16004 instances) were randomly selected as the *Training* dataset, whereas the remaining one third (8002 instances) formed the *Excluded* test dataset. Our data set was large enough to follow this procedure to build and evaluate the model without resorting to cross-validation, which is much more computationally intensive. Also, the random selection of the training set across 11 releases reduced the chances for the prediction model to be overly influenced by peculiarities of any given release. Note that in the training set, there were only 303 instances representing faulty classes (that is, the class had at least one fault correction in the next release). This is due to the fact that, in a typical release, a small percentage of classes turn out to be faulty. Thus, to facilitate the construction of unbiased models, we created a balanced subset (606 rows) from the complete training set, consisting of the 303 faulty classes and a random selection of 303 rows representing non-faulty classes. The proportions of faulty and correct classes were therefore exactly 50% in the training set and the probability decision threshold for classification into faulty and correct classes for the test sets can therefore be set to 0.5. Nearly all the techniques we used performed better (sometimes very significantly) when run on this balanced dataset. Consequently, the models reported in this paper were built using this subset of 606 instances.

Finally, the two most recent of the 13 selected releases formed the third and fourth distinct datasets, hereafter referred to as the *COS 20* and *COS 21* datasets, which we also used as test sets. The *Excluded* test set allows us to estimate the accuracy of the model on the current (release 11) and past releases whereas the *COS 20* and *COS 21* test sets indicate accuracy on future releases. This will give us insights on any decrease in accuracy, if any, when predicting the future. The results given in Section 4 were obtained using only the test set (*Excluded*) and the two evaluation sets (*COS 20* and *COS 21*), i.e., the training set was not included. By not including the training set, the results can be interpreted as what one could expect when applying the models on a new set of classes or a new system version.

3.7 Model Evaluation Criteria

Having described our model evaluation procedure, we now need to explain what model accuracy criteria we used. The alternative prediction models were assessed on the basis of all of the following criteria in order to 1) provide a comprehensive comparison of the models and 2) to assess how the choice of criteria affects the ranking of models.

First, we used several confusion matrix criteria [3], including *accuracy*, *precision* and *recall* and *Type I/II misclassification rates*. For example, in our context, precision is the percentage of classes classified as faulty that are actually faulty and is a measure of how effective we are at identifying where faults are located. Recall is the percentage of faulty classes that are predicted as faulty and is a measure of how many faulty classes we are likely to miss if we use the prediction model. We also used the Receiver Operating Characteristic (ROC) area [3]. The larger the area under the ROC curve (the ROC area), the better the model. A perfect prediction model, that classifies all instances correct, would have a ROC area of 100%. See Section 2.4 for further details.

As discussed in Section 2.4, the problem with the general confusion matrix criteria and ROC is that they are designed to apply to all classification problems and they do not clearly and directly relate to the cost effectiveness of using class fault-proneness prediction models in our or any other given application context. Assuming a class is predicted as very likely to be faulty, one would take corrective action by investing additional effort to inspect and test the class. In our context, we consider the cost of such activities to be roughly proportional to the size of the class. For example, regarding control flow testing, many studies show that cyclomatic complexity (number of independent control flow paths) is strongly correlated with code size [23]. Though this remains to be empirically investigated, this suggests that control flow testing over a large number of classes should be roughly proportional to the size of those classes.

Given the above assumption, if we are in a situation where the only thing a prediction model does is to model the fact that the number of faults is proportional to the size of the class, we are not likely to gain much from such a model. What we want are models that capture other fault factors in addition to size. Therefore, to assess cost effectiveness, we compare two curves as exemplified in Figure 3. Classes are first ordered from high to low fault probabilities. When a model predicts the same probability for two classes, we order them further according to size so that larger classes are selected last. The solid curve represents the actual percentage of faults given a percentage of lines of code of the classes selected to focus verification according to the abovementioned ranking procedure (referred to as the model cost effectiveness (CE) curve). The dotted line represents a line of slope 1 where the percentage of faults would be identical to the percentage of lines of code (% NOS) included in classes selected to focus verification. This line is what one would obtain, on average, if randomly ranking classes and is therefore a baseline of comparison (referred to as the *baseline*). Based on these definitions and the assumptions above, the overall cost-effectiveness of fault predictive models would be proportional to the surface area between the CE curve and the baseline. This is practical as such a surface area is a unique score according to which we can compare models in terms of cost-effectiveness regardless of a specific, possibly unknown, NOS percentage to be verified. If the model yields a percentage of faults roughly identical to the percentage of lines of code, then no gain is to be expected from using such a fault-proneness model when compared to chance alone. The exact surface area to consider may depend on a realistic, maximum percentage of lines of code that is expected to be

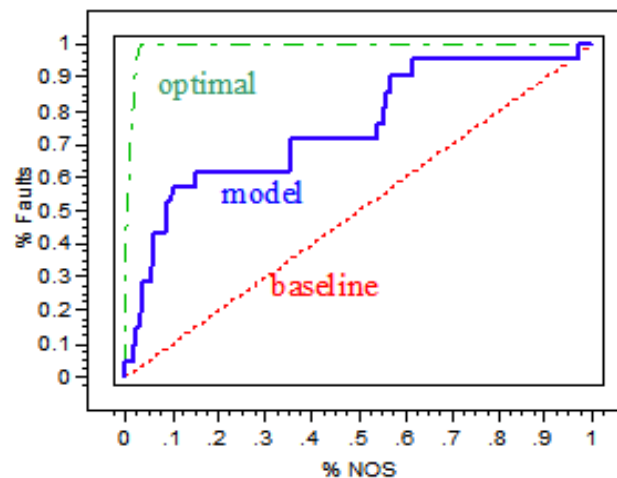


Figure 3: Surrogate Measure of Cost Effectiveness

covered by the extra verification activities. For example, if only 5% of the source code is the maximum target considered feasible for extra testing, only the surface area below the 5% threshold should be considered.

For a given release, it is impossible to determine beforehand what would be the surface area of an *optimal* model. For each release, we compute it by ordering classes as follows: (1) we place all faulty classes first and then order them so that larger classes are tested last, and (2) we place fault-free classes afterwards also in increasing order of size. This procedure is a way to maximize the surface area for a given release and set of faulty classes, assuming the future can be perfectly predicted. Once computed, we can compare, for a specific NOS percentage, the maximum percentage of faults that could be obtained with an optimal model and use this as an upper bound to further assess a model, as shown by the dashed line in Figure 3.

To compare CE areas we need to account for the fact that the optimal model might differ across test sets. Thus, we compute a normalized cost-effectiveness measure as

$$CE_{\pi} = (CE_{\pi}(\text{model}) - CE_{\pi}(\text{baseline})) / (CE_{\pi}(\text{optimal}) - CE_{\pi}(\text{baseline}))$$

where $CE_{\pi}(x)$ is the area under the curve x (*baseline*, *model*, or *optimal*) for a given π percentage of NOS. This measure can be interpreted as a proportion of the optimal cost-effectiveness, a measure which is comparable across evaluation datasets. Depending on the amount of resources available for testing, the percentage of NOS to be tested will vary, so we compute CE for respectively 1%, 5% and 20% of the NOS ($CE_{0.01}$, $CE_{0.05}$, $CE_{0.20}$). Computing a CE area is also a way to compare models without any specific percentage of classes in mind and based on a unique score. This is why we also choose to include the cost-effectiveness at 100% NOS ($CE_{1.00}$). Admittedly such CE values may not be easy to interpret but their purpose is to facilitate the comparison among models based on a measure that should be directly proportional to cost-effectiveness in the context of focusing verification and validation efforts.

3.8 Model Assessment Procedure

We built a total of 112 different fault-proneness models on the basis of our training dataset, i.e., individual prediction models for each of the seven metric sets presented in Section 3.4 (*OO*, *Delta*, *Process*, *OO + Delta*, *Process + OO*, *Process + Delta*, *Total*) with and without CFS, using each of the eight candidate mining techniques presented in Section 3.5 (*Neural network*, *C4.5*, *Decorate C4.5*, *Boost C4.5*, *SVM*, *Logistic regression*, *PART*, *C4.5 + PART*). Each of the 112 models was evaluated on the three distinct evaluation datasets presented in Section 3.6 (*Excluded*, *COS 20*, *COS 21*) and using the evaluation criteria presented in Section 3.7 (*Accuracy*, *Precision*, *Recall*, *Type I/II misclassification rate*, *ROC*, $CE_{0.01}$, $CE_{0.05}$, $CE_{0.20}$, $CE_{1.00}$)

To assess the magnitude of the differences between the model building techniques and the metric sets, we report a number of statistics including the mean, the minimum, and maximum of each criterion. As it is difficult to make any assumptions about the underlying distribution for many of the evaluation criteria we use non-parametric tests to assess the significance of the differences. More specifically, for each evaluation criterion, we report p-values from a matched pair Wilcoxon's signed rank test for

- all pairs of techniques aggregated across metric sets, and
- all pairs of metric sets aggregated across techniques.

Given the large number of tests being performed, we set the level of significance to $\alpha=0.001$. In practice it is useful to not only know the p-values, but also the *size* of the effect. Thus, in addition

to the Wilcoxon p-value on the difference between respectively all pairs of techniques and all pairs of metric sets, we also report effect sizes on these differences using Cohen's d [81].

4 Results

This section reports the results from the assessment procedure that was summarized in Section 3.8. As mentioned in Section 3.4 and 3.5, a number of different models were built; both using a complete set of independent variables and using a CFS-reduced version of the same metric sets. Surprisingly, the performance of the models that were built using the reduced set of metrics were consistently but marginally poorer than the complete set of metrics across most of the evaluation criteria considered. Consequently, to simplify the already quite complex analyses, and since the results would anyway be very similar, we do not provide separate results for respectively the CFS-reduced models and the non-reduced models, but instead combine the two in one analysis.

First, we give an evaluation of the metric sets and modeling techniques using ROC and CE as we consider these criteria the most appropriate to evaluate prediction models in our context. Then, we show the results when considering a selection of the most popular confusion matrix criteria: accuracy, precision and recall, and Type I- and Type II-misclassification rates. At the end of this section we summarize and discuss the results.

The detailed results are reported in tables that form the basis for our discussion in the following subsections. The tables compare metric sets and modeling techniques against one another in terms of the different evaluation criteria. In the tables we report the mean, standard deviation, minimum and maximum value for each metric set and technique. These descriptive statistics are shown in the leftmost columns of the tables – next to the name of the metric set or modeling technique. In the right part of the tables we report the difference between each combination of metric set/modeling technique in terms of effect size and the Wilcoxon test. The latter appears in the upper right side of the diagonal, while the effect size appears in the lower left side of the diagonal. The effect size is shown in bold face if the corresponding Wilcoxon test is significant at $\alpha=0.001$. The results for metric sets and techniques are sorted according to their mean values in each table; either descending or ascending depending on whether higher or lower values are better. Finally, the technique and metric set with the highest average rank when considering the ROC area and the four CE measures in combination are included as the “best technique” and “best metric set”, respectively. The average results for the best technique are included in the tables that compare the metric sets, whereas the average results for the best metric set is included in the tables that compare the techniques.

4.1 Evaluation of Modeling Techniques using ROC and CE

Table 2 shows that the differences among techniques in terms of mean ROC area are in most cases very small, or at least too small to be of practical significance. If we were to use the median as a ranking criterion instead, the ranking of the techniques would be similar. The average ROC area ranges from 0.70 for C4.5 to above 0.75 using Decorate C4.5 and Neural network. That is, the probability that a faulty class will be assigned a higher fault probability than a non-faulty one is on average above 0.7, for all modeling techniques. Decorate C4.5 is the data mining technique which has the lowest standard deviation, and thus yields the most stable results regardless of metric set; the minimum is right below 0.6 while the maximum is 0.9, and the standard deviation is 0.08. C4.5 and PART and the combination of the two are perhaps the techniques that yield the models that are the easiest to interpret. At the same time, C4.5 and

PART are also the ones that yield the smallest ROC area among the techniques assessed in this study; the mean ROC area for C4.5 and PART is significantly smaller than the mean ROC area of the two best techniques. Although C4.5 has the lowest average ROC area overall, the ROC area when using C4.5 in combination with the Process metrics is similar to the mean ROC area using Neural network when not considering any particular metric set, suggesting that C4.5 is in fact a technique that may give fairly good results given that the optimal set of metrics (Process) is used. Considering the ease of interpretation of decision trees, one might choose this technique if the goal is not only to predict fault-proneness, but also to interpret the model and explain it to practitioners. If the results from using C4.5 are not sufficient, Adaboost can be applied to further improve the model, as the combination of C4.5 and boosting is the technique that yields the best overall ranking across all evaluation criteria.

In Table 3 through Table 6 the data mining techniques are compared using the surrogate measure of cost-effectiveness described in Section 3.7. The difference in average cost-effectiveness between the most and least cost-effective techniques ranges from 0.04 to 0.08 percentage points depending on which threshold π is used. Although there is to some degree a significant difference between the extremes, the differences are negligible considering the uncertainty in the data. Using the optimal set of measures (*Process*), all techniques yield a cost-effectiveness of approximately 30% of the optimal model at $\pi = 0.20$ NOS. Although there is still room for improvement, this is more than three times as cost-effective compared to a model based on random selection.

Table 2: Area under ROC curve for the modeling techniques

	Mean	Std. Dev.	Min	Max	Best metric set (Process)	Neural network	Decorate C4.5	SVM	Logistic regression	Boost C4.5	PART	C4.5 + PART	C4.5	Wilcoxon ($\alpha = 0,001$)	
Neural network	0,756	0,091	0,543	0,935	0,826	-	0,902	0,811	0,045	0,036	0,001	0,000	0,000		
Decorate C4.5	0,752	0,077	0,598	0,899	0,779	0,048	-	0,515	0,109	0,006	0,000	0,000	0,000		
SVM	0,749	0,112	0,453	0,942	0,724	0,072	0,034	-	0,556	0,164	0,011	0,004	0,001		
Logistic regression	0,737	0,097	0,454	0,919	0,722	0,205	0,174	0,114	-	0,551	0,026	0,013	0,009		
Boost C4.5	0,732	0,085	0,510	0,856	0,806	0,279	0,252	0,173	0,057	-	0,006	0,000	0,005		
PART	0,708	0,086	0,468	0,861	0,776	0,548	0,543	0,412	0,317	0,280	-	0,661	0,467		
C4.5 + PART	0,703	0,087	0,468	0,862	0,778	0,599	0,599	0,459	0,370	0,336	0,059	-	0,579		
C4.5	0,699	0,091	0,470	0,873	0,762	0,629	0,630	0,489	0,403	0,372	0,099	0,041	-		
Effect size															

Table 3: Cost-effectiveness for modeling techniques at $\pi = 0.01$ NOS

	Mean	Std. Dev.	Min	Max	Best metric set (Process)	Logistic regression	Neural network	Decorate C4.5	Boost C4.5	C4.5 + PART	PART	C4.5	SVM	Wilcoxon ($\alpha = 0,001$)	
Logistic regression	0,137	0,161	-0,043	0,665	0,185	-	0,012	0,024	0,052	0,074	0,025	0,012	0,007		
Neural network	0,104	0,197	-0,043	0,807	0,142	0,186	-	0,421	0,724	0,848	0,700	0,661	0,292		
Decorate C4.5	0,101	0,230	-0,043	0,870	0,339	0,179	0,010	-	0,445	0,347	0,130	0,833	0,427		
Boost C4.5	0,099	0,161	-0,043	0,556	0,254	0,236	0,026	0,013	-	0,742	0,821	0,715	0,361		
C4.5 + PART	0,090	0,129	-0,043	0,371	0,160	0,319	0,079	0,059	0,058	-	0,853	0,505	0,510		
PART	0,087	0,120	-0,043	0,371	0,139	0,353	0,103	0,080	0,085	0,030	-	0,618	0,349		
C4.5	0,080	0,135	-0,043	0,371	0,152	0,383	0,139	0,114	0,126	0,079	0,052	-	0,873		
SVM	0,079	0,162	-0,043	0,689	0,153	0,358	0,135	0,112	0,122	0,077	0,053	0,006	-		
Effect size															

Table 4: Cost-effectiveness for modeling techniques at $\pi = 0.05$ NOS

	Mean	Std. Dev.	Min	Max	Best metric set (Process)	Logistic regression	Boost C4.5	PART	Neural network	C4.5 + PART	Decorate C4.5	C4.5	SVM	Wilcoxon ($\alpha = 0,001$)	
Logistic regression	0,099	0,082	-0,029	0,255	0,160	-	0,055	0,095	0,003	0,130	0,029	0,001	0,000		
Boost C4.5	0,076	0,088	-0,037	0,301	0,143	0,272	-	0,878	0,763	0,954	0,584	0,230	0,134		
PART	0,074	0,070	-0,037	0,202	0,096	0,333	0,027	-	0,688	0,855	0,274	0,124	0,113		
Neural network	0,073	0,085	-0,035	0,263	0,134	0,309	0,029	0,004	-	0,897	0,456	0,225	0,027		
C4.5 + PART	0,070	0,085	-0,037	0,239	0,127	0,347	0,066	0,045	0,038	-	0,449	0,138	0,208		
Decorate C4.5	0,062	0,085	-0,037	0,294	0,174	0,443	0,160	0,150	0,134	0,097	-	0,518	0,230		
C4.5	0,052	0,072	-0,037	0,184	0,097	0,607	0,293	0,302	0,270	0,228	0,302	-	0,924		
SVM	0,051	0,083	-0,035	0,220	0,113	0,583	0,291	0,296	0,268	0,229	0,131	0,017	-		
Effect size															

Table 5: Cost-effectiveness for modeling techniques at $\pi = 0.20$ NOS

	Mean	Std. Dev.	Min	Max	Best metric set (Process)	Boost C4.5	PART	Decorate C4.5	Logistic regression	C4.5 + PART	Neural network	C4.5	SVM	Wilcoxon ($\alpha = 0,001$)	
Boost C4.5	0,168	0,132	-0,061	0,389	0,289	-	0,956	0,576	0,137	0,717	0,010	0,046	0,017		
PART	0,162	0,140	-0,078	0,382	0,302	0,051	-	0,494	0,326	0,463	0,093	0,031	0,068		
Decorate C4.5	0,156	0,119	-0,052	0,377	0,300	0,096	0,040	-	0,936	0,897	0,072	0,186	0,021		
Logistic regression	0,155	0,154	-0,111	0,458	0,274	0,090	0,041	0,007	-	0,763	0,002	0,464	0,064		
C4.5 + PART	0,152	0,148	-0,079	0,423	0,326	0,119	0,068	0,035	0,025	-	0,199	0,063	0,213		
Neural network	0,130	0,150	-0,097	0,524	0,286	0,274	0,219	0,196	0,169	0,148	-	0,735	0,518		
C4.5	0,129	0,139	-0,092	0,398	0,273	0,294	0,237	0,215	0,183	0,162	0,009	-	0,745		
SVM	0,123	0,152	-0,090	0,511	0,230	0,316	0,261	0,241	0,209	0,189	0,042	0,035	-		
Effect size															

Table 6: Cost-effectiveness for modeling techniques at $\pi = 1.0$ NOS

	Mean	Std. Dev.	Min	Max	Best metric set (Process)	Boost C4.5	Decorate C4.5	Neural network	Logistic regression	PART	SVM	C4.5	C4.5 + PART	Wilcoxon ($\alpha = 0,001$)
Boost C4.5	0,272	0,208	-0,259	0,607	0,536	-	0,320	0,049	0,037	0,037	0,033	0,000	0,001	
Decorate C4.5	0,259	0,236	-0,294	0,650	0,526	0,062	-	0,083	0,051	0,098	0,005	0,007	0,019	
Neural network	0,227	0,235	-0,249	0,720	0,535	0,205	0,135	-	0,441	0,839	0,487	0,985	0,584	
Logistic regression	0,217	0,247	-0,262	0,674	0,362	0,241	0,171	0,040	-	0,849	0,130	0,907	0,735	
PART	0,213	0,243	-0,216	0,656	0,499	0,262	0,191	0,058	0,017	-	0,681	0,441	0,208	
SVM	0,200	0,281	-0,331	0,742	0,342	0,292	0,225	0,103	0,064	0,048	-	0,745	0,839	
C4.5	0,196	0,252	-0,202	0,636	0,515	0,333	0,259	0,129	0,087	0,071	0,018	-	0,811	
C4.5 + PART	0,192	0,237	-0,214	0,654	0,510	0,359	0,281	0,147	0,103	0,086	0,031	0,013	-	
Effect size														

4.2 Evaluation of Metric Sets using ROC and CE

As shown in Table 7, the differences in average ROC area between the metric sets (across techniques) are moderate. The average ROC area ranges from 0.65 for Deltas up to 0.77 when using the Process metric set. The Delta metric set is significantly worse than the other combinations of metrics. The ROC area for all but the Delta set is above 0.7.

Table 7: Area under ROC curve for the metric sets

	Mean	Std Dev	Min	Max	Best technique (Boost C4.5)	Process	Process + OO	Total	Process + Delta	OO + Delta	OO	Delta	Wilcoxon ($\alpha = 0,001$)
Process	0,772	0,097	0,453	0,942	0,806	-	0,968	0,852	0,034	0,041	0,004	0,000	
Process + OO	0,768	0,072	0,608	0,915	0,763	0,041	-	0,438	0,004	0,000	0,000	0,000	
Total	0,759	0,089	0,546	0,884	0,761	0,132	0,108	-	0,011	0,000	0,000	0,000	
Process + Delta	0,736	0,086	0,510	0,929	0,703	0,387	0,402	0,264	-	0,880	0,103	0,000	
OO + Delta	0,720	0,080	0,562	0,840	0,736	0,578	0,627	0,460	0,192	-	0,003	0,000	
OO	0,702	0,085	0,532	0,849	0,690	0,761	0,834	0,654	0,398	0,220	-	0,001	
Delta	0,648	0,079	0,468	0,821	0,665	1,397	1,584	1,317	1,069	0,910	0,659	-	
Effect size													

Though the smallest ROC area (0.45) is obtained when using the Process metrics², this set of metrics is at the same time best in terms of mean and maximum ROC area. Compared to the Process metrics alone, there seems to be no immediate gain by combining them with the OO metrics. However, as can be seen from Table 7, by adding the OO metrics, the minimum ROC area is lifted above 0.6, and the standard deviation is lower.

² It is worth noting that all ROC areas below 0.5 were obtained using the CFS-reduced data sets.

Table 8: Cost-effectiveness for the metric sets at $\pi = 0.01$ NOS

	Mean	Std Dev	Min	Max	Best technique (Boost C4.5)	Process	Process + Delta	Process + OO	Total	Delta	OO	OO + Delta	Wilcoxon ($\alpha = 0,001$)
Process	0,190	0,209	-0,043	0,775	0,254	-	0,402	0,030	0,078	0,000	0,000	0,000	
Process + Delta	0,175	0,212	-0,043	0,870	0,157	0,072	-	0,279	0,279	0,002	0,000	0,000	
Process + OO	0,123	0,151	-0,043	0,511	0,129	0,367	0,281	-	0,630	0,004	0,000	0,000	
Total	0,116	0,176	-0,043	0,689	0,109	0,388	0,307	0,048	-	0,068	0,000	0,000	
Delta	0,049	0,088	-0,043	0,360	0,008	0,879	0,774	0,597	0,475	-	0,263	0,000	
OO	0,025	0,071	-0,043	0,208	0,009	1,061	0,950	0,832	0,676	0,306	-	0,017	
OO + Delta	0,001	0,070	-0,043	0,362	0,025	1,215	1,102	1,036	0,856	0,605	0,335	-	

Table 9: Cost-effectiveness for the metric sets at $\pi = 0.05$ NOS

	Mean	Std Dev	Min	Max	Best technique (Boost C4.5)	Process	Process + Delta	Total	Delta	Process + OO	OO + Delta	OO	Wilcoxon ($\alpha = 0,001$)
Process	0,130	0,075	-0,029	0,301	0,143	-	0,227	0,001	0,000	0,000	0,000	0,000	
Process + Delta	0,116	0,079	-0,027	0,289	0,117	0,185	-	0,019	0,001	0,004	0,000	0,000	
Total	0,083	0,085	-0,037	0,255	0,105	0,590	0,404	-	0,479	0,177	0,000	0,000	
Delta	0,071	0,070	-0,026	0,201	0,027	0,817	0,606	0,156	-	1,000	0,000	0,000	
Process + OO	0,071	0,081	-0,037	0,258	0,102	0,761	0,566	0,147	0,001	-	0,000	0,000	
OO + Delta	0,009	0,044	-0,037	0,163	0,035	1,957	1,669	1,087	1,048	0,938	-	0,939	
OO	0,006	0,035	-0,037	0,101	0,000	2,121	1,810	1,192	1,177	1,042	0,096	-	
Effect size													

Table 10: Cost-effectiveness for the metric sets at $\pi = 0.20$ NOS

	Mean	Std Dev	Min	Max	Best technique (Boost C4.5)	Process	Process + Delta	Delta	Total	Process + OO	OO + Delta	OO	Wilcoxon ($\alpha = 0,001$)
Process	0,285	0,088	0,102	0,524	0,289	-	0,000	0,000	0,000	0,000	0,000	0,000	
Process + Delta	0,233	0,092	0,041	0,389	0,276	0,574	-	0,005	0,000	0,000	0,000	0,000	
Delta	0,183	0,140	-0,030	0,458	0,147	0,874	0,426	-	0,936	0,141	0,000	0,000	
Total	0,170	0,112	-0,071	0,387	0,207	1,143	0,619	0,103	-	0,023	0,000	0,000	
Process + OO	0,129	0,121	-0,076	0,331	0,192	1,476	0,971	0,410	0,348	-	0,000	0,000	
OO + Delta	0,022	0,091	-0,106	0,261	0,057	2,933	2,303	1,356	1,440	0,997	-	0,320	
OO	0,007	0,078	-0,111	0,222	0,011	3,343	2,652	1,550	1,683	1,202	0,184	-	
Effect size													

Table 11: Cost-effectiveness for the metric sets at $\pi = 1.0$ NOS

	Mean	Std Dev	Min	Max	Best technique (Boost C4.5)	Process	Process + Delta	Delta	Total	Process + OO	OO + Delta	OO
Process	0,478	0,165	-0,122	0,742	0,536	-	0,000	0,000	0,000	0,000	0,000	0,000
Process + Delta	0,394	0,107	0,119	0,669	0,372	0,604	-	0,000	0,000	0,000	0,000	0,000
Delta	0,236	0,195	-0,216	0,674	0,236	1,343	1,008	-	0,400	0,701	0,000	0,000
Total	0,224	0,219	-0,213	0,531	0,318	1,308	0,984	0,054	-	0,479	0,000	0,000
Process + OO	0,199	0,194	-0,223	0,470	0,273	1,552	1,247	0,190	0,125	-	0,000	0,000
OO + Delta	0,037	0,185	-0,306	0,357	0,137	2,512	2,358	1,045	0,925	0,853	-	0,004
OO	-0,013	0,178	-0,331	0,294	0,036	2,863	2,774	1,334	1,191	1,139	0,275	-
Effect size												

Wilcoxon ($\alpha = 0,001$)

If we turn to cost-effectiveness, the results for the metric sets are quite different. In

Table 8 through Table 11 we compare the metric sets in terms of cost-effectiveness. Looking back at Table 7, we can see that the OO metrics are on par with the Process metrics when considering the ROC area. However from Table 8 through Table 11, we observe that in terms of cost-effectiveness the difference between these two sets of metrics is much larger. At $\pi = 0.20$ NOS (Table 10), the cost-effectiveness using OO metrics are not even 1% of the optimal model, while the cost-effectiveness by using the Process metrics alone are one third of the optimal model, and over three times as cost-effective than the baseline (random model).

As explained in Section 3.8, a number of models were built by using different data mining techniques. Because three separate test sets were applied to the each of these prediction models, we obtained a fairly large number of observations for each metric set. These samples form distributions which we can compare. Figure 4 depicts the distribution in cost-effectiveness for the prediction models built and evaluated using the Process metrics and the OO metrics, respectively. The plot shows the median cost-effectiveness for each group of prediction models. In addition to the median shown as a solid line, the area between the 25 and 75 percentiles is shaded. This visualization can be interpreted as simplified boxplots of the cost-effectiveness when using the two metric sets at discrete levels of NOS. As can be seen from the figure, the distribution in cost-effectiveness using the process metrics is far from the baseline, and nearly not overlapping with the corresponding distribution obtained from using the OO metrics. Looking at the plot for the process metrics, we observe that the 25-percentile for the process metrics are close to 50% *Total faults* at $CE_{0.20}$. This shows that among the models using the process metrics alone, a majority of them (3/4) located more than 50% of the faults in 20% of the most fault-prone classes as predicted by the model. Further, the 75-percentile at $CE_{0.20}$ for the process metrics is at 70% *Total faults*, indicating that 25% of the most cost-effective models in fact identified over 70% of the faults in the 20% most fault-prone classes. This is comparable to the results obtained by Ostrand *et al.* [15-17].

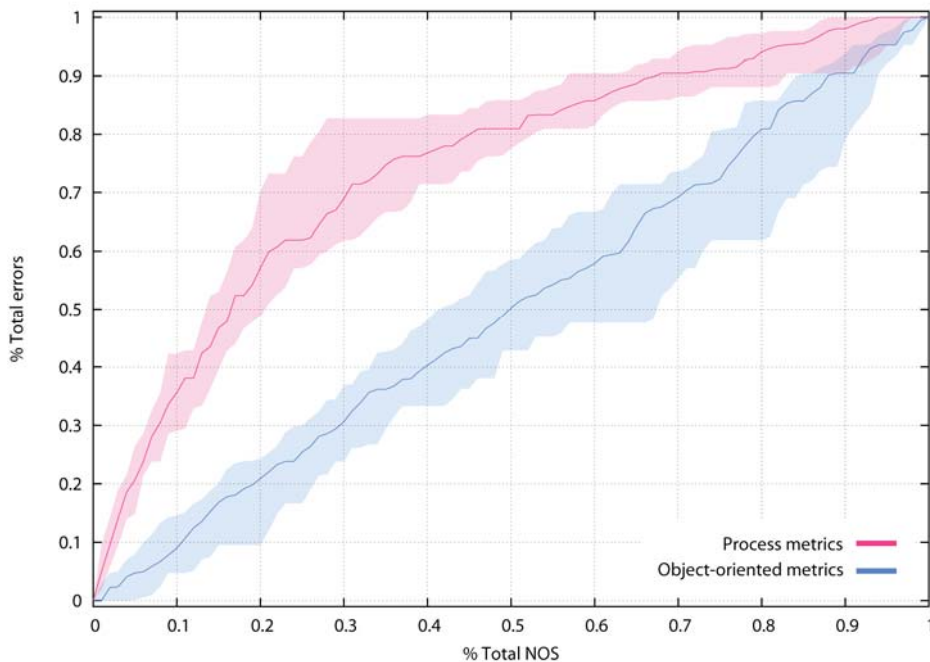


Figure 4: Median and 25-/75-percentiles for process metrics and object-oriented metrics

Figure 4 supports the results in the tables comparing metric sets, showing that the cost-effectiveness obtained by using the OO metrics is close to zero. It is worth noting that there are in fact a large number of models using the OO metrics that have negative cost-effectiveness: the median of the OO metrics is close to the baseline with slope 1, indicating that 50% of the observations are below this baseline, and thus these models are not more cost-effective than a completely random model. It is interesting that the average cost-effectiveness for OO metrics is close to zero across all thresholds. Turning back to Table 8 through Table 11, note also that the cost-effectiveness of the models built using other metric sets decreases when the OO metrics are added. For example, this is visible when comparing the cost-effectiveness of the process metrics with that of the process metrics in combination with the OO metrics (Process+OO): The process metrics are consistently more cost-effective, but when adding the OO metrics, this combination is consistently ranked among the least cost-effective. That is, adding the OO metrics consistently degrades the cost-effectiveness of a model. Further, we observe that although the deltas have the smallest average ROC area, these metrics are consistently more cost-effective than the OO metrics. The low cost-effectiveness of the OO metrics may be due to their correlation with size measures, which has been reported in many other papers [82].

If we were to use the prediction models to focus verification and validation efforts by, say, inspecting the 20% most fault-prone classes – the gain from using the process metrics (finding 60% of the faults on average) compared to the average of what would be obtained with random orders (finding 20% of the faults) is substantial. Of course, this is a somewhat simplified view for both scenarios, as we probably cannot expect to find *all* faults by applying a particular fault-proneness model to focus verification and validation³. Still, the gain from using a prediction

³ A suitable cost-benefit model that accounts for the percentage of faults that are not discovered during verification efforts is given in [4] L. C. Briand and J. Wust, "Empirical studies of quality models in object-oriented systems," *Advances in Computers*, Vol 56, vol. 56, pp. 97-166, 2002..

model based on process metrics is substantial compared with the baseline model. On the other hand, we also see that there is much room for improvement when compared to an optimal ranking of the classes: the best model is approximately 50% of the optimal model in terms of cost-effectiveness.

The results show that the OO metrics are good predictors of faulty classes (i.e., large ROC area), but these metrics do not result in cost-effective prediction models. Many OO metrics have been shown to be associated with size [82], and this fact might explain the low cost-effectiveness of the OO metrics, because the surrogate measure for cost-effectiveness penalize models which mostly capture a size effect. Although the process metrics are presumably more expensive to collect, the results show that collecting process metrics is likely to be cost-effective.

4.3 Evaluating Techniques and Metric Sets using other Evaluation Criteria

In the two previous subsections, metric sets and modeling techniques were compared using two evaluation criteria: ROC area and cost-effectiveness (CE). This section presents the results when using some of the more commonly used evaluation criteria. More specifically, we will consider the most popular measures that can be derived from the confusion matrix as explained in Section 3.7. We did not investigate in detail how these classification accuracy measures are affected by different probability cut-off values. Still, the results given in this section are comparable to most studies, which in most cases do not vary the threshold, but rather use the default value of 0.5, as shown in Section 2. We first consider *accuracy* as it is the most prominent measure in the studies reviewed. Then, we show our results for *precision*, *recall* and *Type I-* and *Type II-misclassification rates* as these evaluation criteria are also widely used.

One of the conclusions in the two previous subsections was that the Process metrics set seems to be the overall best metric set and Boost C4.5 the best modeling technique in terms of average ROC area and cost-effectiveness. Consequently, to facilitate comparisons with the previous subsections, we still show the Process/Boost C4.5 results in a separate column.

Accuracy

Table 12 and Table 13 show the accuracy for modeling techniques and metric sets, respectively. As higher accuracy is considered better than lower accuracy, the tables are sorted in descending order according to the mean values.

Table 12: Accuracy of modeling techniques

	Mean	Std. Dev.	Min	Max	Best metric set (Process)	SVM	Logistic regression	C4.5 +PART	Neural network	Decorate C4.5	C4.5	Boost C4.5	PART	Wilcoxon ($\alpha = 0,001$)	
SVM	0,863	0,061	0,744	0,985	0,869	-	0,000	0,145	0,007	0,000	0,000	0,000	0,000		
Logistic regression	0,845	0,060	0,753	0,983	0,867	0,295	-	0,830	0,265	0,007	0,004	0,000	0,000		
C4.5+PART	0,838	0,105	0,650	0,970	0,934	0,287	0,077	-	0,806	0,000	0,000	0,000	0,000		
Neural network	0,830	0,090	0,681	0,986	0,916	0,432	0,199	0,089	-	0,017	0,014	0,000	0,000		
Decorate C4.5	0,807	0,104	0,634	0,970	0,915	0,652	0,443	0,298	0,230	-	0,059	0,002	0,000		
C4.5	0,793	0,125	0,568	0,969	0,912	0,709	0,527	0,391	0,334	0,122	-	0,494	0,270		
Boost C4.5	0,783	0,107	0,658	0,961	0,903	0,925	0,719	0,528	0,477	0,234	0,092	-	0,452		
PART	0,771	0,125	0,526	0,964	0,901	0,932	0,750	0,582	0,537	0,314	0,177	0,099	-		
Effect size															

The differences in accuracy among modeling techniques are smaller than the differences among metric sets. If one were to select a particular modeling technique based on the average accuracy, one would probably select SVM or logistic regression, although these techniques yield lower accuracy when used in conjunction with the optimal metric set (Process).

Table 13: Accuracy of metric sets

	Mean	Std Dev	Min	Max	Best technique (Boost C4.5)	Delta	Process	Process+Delta	Total	Process+OO	OO+Delta	OO	Wilcoxon ($\alpha = 0,001$)	
Delta	0,908	0,085	0,739	0,986	0,889	-	0,367	0,000	0,000	0,000	0,000	0,000		
Process	0,902	0,050	0,744	0,982	0,903	0,089	-	0,000	0,000	0,000	0,000	0,000		
Process+Delta	0,871	0,070	0,760	0,971	0,868	0,475	0,504	-	0,000	0,000	0,000	0,000		
Total	0,797	0,084	0,612	0,945	0,684	1,319	1,519	0,959	-	0,351	0,000	0,000		
Process+OO	0,776	0,070	0,642	0,899	0,711	1,697	2,065	1,354	0,267	-	0,031	0,000		
OO+Delta	0,744	0,085	0,526	0,896	0,745	1,925	2,252	1,622	0,620	0,408	-	0,037		
OO	0,715	0,074	0,568	0,834	0,680	2,420	2,947	2,156	1,029	0,845	0,364	-		
Effect size														

It is worth pointing that the Delta metric set yields the highest accuracy. Looking at the results for ROC area in Table 7 in Section 4.2, Delta was the metric set giving the smallest average ROC area, and thus one would probably conclude that using these metrics to predict fault-proneness is not optimal, thus running counter to what one would conclude when considering the accuracy measure.

Furthermore, what is considered the best metric set is highly dependent on which cut-off that is used. Here we have used a threshold of 0.5 because it is commonly used in the existing literature, however, it is difficult to give a rule of thumb as to what cut-off to use because there would probably be large variations across studies as these results are highly dependent on properties of the data set. In our case, the most accurate models are obtained when using cut-off values above 0.8. This is due to the highly unbalanced nature of our data sets: only a small percentage of the classes are faulty. Although high accuracy is intuitively a desired property, our results suggest that accuracy is not necessarily an appropriate measure for evaluating how useful fault-proneness prediction models are.

Precision and Recall

Two other evaluation criteria that are widely used are the *precision* and *recall* measures, as explained in Section 3.7. Table 14 and Table 15 show the results for these measures using the different metric sets. The metric sets are sorted in descending order according to their mean precision/recall.

From Table 14 we see that the precision ranges from 3% to approximately 10%. This indicates that when using a cut-off of 0.5 to distinguish faulty classes from non-faulty ones, only a small part of the fault-prone classes identified by the prediction model is in fact faulty – that is, most of the classes predicted as faulty are false positives. Although the maximum for Delta is above 0.4, the precision of our models is much lower than comparable studies who typically achieved precision in the range of 0.7 to 0.95 [13, 38, 65]. The reason we get a relatively low precision is probably because only 0.5% to 2% of the classes in our data sets are in fact faulty.

Thus, even a few false positives have a huge impact on the precision of the prediction models. However, as argued in [83], for such unbalanced data, the prediction models can still be useful despite having low precision.

Table 14: Precision for the metric sets

	Mean	Std Dev	Min	Max	Best technique (Boost C4.5)	Delta	Process	Process + Delta	Total	Process + OO	OO + Delta	OO
Delta	0,104	0,094	0,040	0,429	0,076	-	0,288	0,000	0,000	0,000	0,000	0,000
Process	0,082	0,047	0,020	0,273	0,082	0,294	-	0,000	0,000	0,000	0,000	0,000
Process + Delta	0,067	0,035	0,019	0,160	0,061	0,521	0,362	-	0,000	0,000	0,000	0,000
Total	0,044	0,021	0,014	0,101	0,030	0,871	1,024	0,768	-	0,486	0,000	0,000
Process + OO	0,039	0,020	0,013	0,110	0,031	0,941	1,162	0,942	0,223	-	0,000	0,000
OO + Delta	0,032	0,014	0,013	0,061	0,031	1,063	1,426	1,288	0,671	0,432	-	0,012
OO	0,029	0,013	0,013	0,058	0,025	1,108	1,520	1,411	0,854	0,620	0,227	-
Effect size												

Wilcoxon ($\alpha = 0,001$)

Table 15: Recall (or Sensitivity, TP rate) for the metric sets

	Mean	Std Dev	Min	Max	Best technique (Boost C4.5)	Process + OO	Total	OO	OO + Delta	Process + Delta	Process	Delta
Process + OO	0,623	0,113	0,389	0,889	0,677	-	0,689	0,925	0,252	0,000	0,000	0,000
Total	0,612	0,138	0,278	0,833	0,723	0,087	-	0,752	0,490	0,000	0,000	0,000
OO	0,609	0,117	0,333	0,781	0,597	0,122	0,023	-	0,408	0,001	0,000	0,000
OO + Delta	0,593	0,137	0,361	0,833	0,609	0,235	0,134	0,122	-	0,005	0,000	0,000
Process + Delta	0,518	0,175	0,167	0,755	0,556	0,712	0,595	0,611	0,480	-	0,203	0,000
Process	0,492	0,162	0,139	0,833	0,554	0,936	0,794	0,826	0,674	0,151	-	0,000
Delta	0,362	0,160	0,056	0,616	0,429	1,884	1,671	1,762	1,552	0,929	0,810	-
Effect size												

Wilcoxon ($\alpha = 0,001$)

Table 15 shows the corresponding results for *recall*. We see that the models typically capture somewhere between 36% and 62% of the faulty classes on average using a cut-off equal to 0.5. This is comparable to other studies, e.g., [13, 22, 65], while other studies achieved recall close to 1 [38]. With respect to recall, the Total metric set is best, and looking at the results when using the overall best modeling technique (Boost C4.5) in combination with the total set of metrics, we observe that 72% of the faults are captured on average by these models.

Among the modeling techniques, the differences in average precision are small, typically in the range from 0.05 to 0.07 (Table 16). The rule- and tree-based modeling techniques are techniques that seem to yield low precision, whereas these techniques are at the same time those that yield higher recall than SVM, neural network and logistic regression (Table 17).

Table 16: Precision for each of the modeling techniques

	Mean	Std. Dev.	Min	Max	Best metric set (Process)	SVM	Neural network	C4.5+PART	Logistic regression	Decorate C4.5	C4.5	Boost C4.5	PART	Wilcoxon ($\alpha = 0,001$)								
SVM	0,073	0,081	0,019	0,429	0,069	-	0,009	0,290	0,007	0,005	0,021	0,000	0,000		Wilcoxon ($\alpha = 0,001$)							
Neural network	0,064	0,074	0,016	0,429	0,107	0,125	-	0,471	0,954	0,378	0,132	0,014	0,001			Wilcoxon ($\alpha = 0,001$)						
C4.5+PART	0,059	0,037	0,015	0,175	0,097	0,226	0,078	-	0,526	0,051	0,000	0,000	0,000				Wilcoxon ($\alpha = 0,001$)					
Logistic regression	0,058	0,050	0,020	0,316	0,062	0,233	0,095	0,033	-	0,300	0,267	0,025	0,006					Wilcoxon ($\alpha = 0,001$)				
Decorate C4.5	0,053	0,037	0,014	0,150	0,084	0,316	0,176	0,155	0,097	-	0,138	0,001	0,000						Wilcoxon ($\alpha = 0,001$)			
C4.5	0,052	0,033	0,013	0,143	0,078	0,341	0,201	0,198	0,129	0,035	-	0,171	0,139							Wilcoxon ($\alpha = 0,001$)		
Boost C4.5	0,048	0,032	0,014	0,131	0,082	0,412	0,278	0,326	0,233	0,160	0,131	-	0,363								Wilcoxon ($\alpha = 0,001$)	
PART	0,047	0,036	0,013	0,148	0,075	0,416	0,284	0,329	0,241	0,171	0,143	0,019	-									Wilcoxon ($\alpha = 0,001$)
Effect size																						

Table 17: Recall for each of the modeling techniques

	Mean	Std. Dev.	Min	Max	Best metric set (Process)	Boost C4.5	PART	C4.5	Decorate C4.5	Neural network	Logistic regression	SVM	C4.5 + PART	Wilcoxon ($\alpha = 0,001$)								
Boost C4.5	0,592	0,155	0,222	0,833	0,554	-	0,492	0,268	0,055	0,000	0,000	0,000	0,000		Wilcoxon ($\alpha = 0,001$)							
PART	0,571	0,151	0,278	0,833	0,482	0,139	-	0,706	0,694	0,022	0,015	0,003	0,000			Wilcoxon ($\alpha = 0,001$)						
C4.5	0,570	0,161	0,194	0,795	0,488	0,140	0,005	-	0,776	0,014	0,007	0,001	0,000				Wilcoxon ($\alpha = 0,001$)					
Decorate C4.5	0,567	0,158	0,167	0,781	0,465	0,163	0,027	0,021	-	0,003	0,013	0,003	0,000					Wilcoxon ($\alpha = 0,001$)				
Neural network	0,522	0,194	0,056	0,778	0,467	0,399	0,280	0,269	0,251	-	0,426	0,134	0,918						Wilcoxon ($\alpha = 0,001$)			
Logistic regression	0,519	0,164	0,167	0,778	0,548	0,461	0,331	0,317	0,298	0,020	-	0,379	0,904							Wilcoxon ($\alpha = 0,001$)		
SVM	0,507	0,192	0,111	0,889	0,494	0,487	0,368	0,355	0,338	0,078	0,065	-	0,356								Wilcoxon ($\alpha = 0,001$)	
C4.5 + PART	0,505	0,155	0,194	0,775	0,440	0,561	0,428	0,410	0,392	0,096	0,083	0,010	-									Wilcoxon ($\alpha = 0,001$)
Effect size																						

Type I and Type II Misclassification Rates

Ostrand *et al.* argue that Type II errors are the most expensive, and that prediction models should be selected and evaluated by their *Type II misclassification rate* [16]. This measure is also used by Khoshgoftaar *et al.* [19, 37]. In Table 18 we report the average Type II misclassification rate for each technique using a default cut-off equal to 0.5. As smaller numbers are considered better (less errors in predictions), the table is sorted in ascending order according to the average for each technique.

Table 18: Type II misclassification rates for each modeling technique

	Mean	Std Dev	Min	Max	Best metric set (Process)	Boost C4.5	PART	Decorate C4.5	C4.5	Neural network	Logistic regression	C4.5+PART	SVM	Wilcoxon ($\alpha = 0,001$)	
Boost C4.5	0,005	0,003	0,001	0,011	0,006	-	0,758	0,144	0,132	0,000	0,000	0,000	0,000		
PART	0,006	0,003	0,001	0,010	0,007	0,072	-	0,350	0,272	0,012	0,034	0,000	0,006		
Decorate C4.5	0,006	0,003	0,002	0,011	0,007	0,118	0,045	-	0,598	0,011	0,020	0,000	0,002		
C4.5	0,006	0,003	0,002	0,011	0,007	0,142	0,071	0,027	-	0,034	0,032	0,000	0,001		
Neural network	0,006	0,003	0,002	0,013	0,007	0,311	0,247	0,208	0,181	-	0,831	0,965	0,061		
Logistic regression	0,006	0,003	0,002	0,012	0,006	0,337	0,271	0,231	0,203	0,015	-	0,961	0,062		
C4.5+PART	0,006	0,003	0,002	0,011	0,008	0,411	0,342	0,301	0,269	0,069	0,055	-	0,203		
SVM	0,007	0,003	0,002	0,012	0,007	0,420	0,358	0,320	0,292	0,106	0,094	0,045	-		
Effect size															

The Type II misclassification rate is typically small, suggesting that a large part of the prediction models assigns a predicted fault probability above 0.5 to most of the faulty classes. Our Type II misclassification rates are slightly smaller (better) than those reported in earlier studies, where this rate typically ranged from 0.01 [16] to 0.3 [12]. Although the differences among the modeling techniques presented here are small, if we were to select a particular technique based on the results in this table, we would conclude that the decision trees or rule-based techniques, i.e., C4.5 (with or without boosting) or PART, yield the best prediction models in terms of Type II misclassification rates. This contradicts our conclusion based on the ROC area in Section 4.1.

Table 19: Type I misclassification rates for each modeling technique

	Mean	Std Dev	Min	Max	Best metric set (Process)	SVM	Logistic regression	C4.5+PART	Neural network	Decorate C4.5	C4.5	Boost C4.5	PART	Wilcoxon ($\alpha = 0,001$)	
SVM	0,130	0,061	0,003	0,248	0,125	-	0,000	0,162	0,006	0,000	0,000	0,000	0,000		
Logistic regression	0,149	0,061	0,005	0,242	0,127	0,297	-	0,839	0,276	0,006	0,004	0,000	0,000		
C4.5+PART	0,155	0,106	0,020	0,346	0,058	0,287	0,075	-	0,787	0,000	0,000	0,000	0,000		
Neural network	0,164	0,091	0,002	0,311	0,078	0,433	0,198	0,090	-	0,019	0,016	0,000	0,000		
Decorate C4.5	0,187	0,105	0,020	0,362	0,077	0,657	0,445	0,302	0,233	-	0,060	0,002	0,000		
C4.5	0,201	0,126	0,021	0,426	0,081	0,712	0,528	0,394	0,336	0,120	-	0,452	0,259		
Boost C4.5	0,212	0,108	0,029	0,336	0,091	0,933	0,724	0,534	0,481	0,236	0,095	-	0,483		
PART	0,223	0,126	0,026	0,470	0,092	0,935	0,750	0,585	0,537	0,312	0,177	0,095	-		
Effect size															

Because the Type I and Type II misclassification rates are inversely correlated – that is, in most cases decreasing the number of Type II errors leads to an increase in the number of Type I errors – it is useful to compare the results in Table 18 with the Type I misclassification rates given in Table 19. Table 19 clearly illustrates that modeling techniques that have lower Type II misclassification rates have higher Type I misclassification rates. If we were to select the modeling technique that would yield best results in terms of Type I misclassification rates, we

would probably choose another modeling technique than when considering Type II misclassification rates. That is, considering Type II misclassification rates we concluded that the rule- or decision tree-based techniques were best, while from Table 19 we conclude that these are significantly worse than SVM. In practice, one would probably consider a trade-off between these types of misclassification rates. One option is to investigate the consistency in ranking for each technique across the evaluation criteria. Then, neural networks would perhaps be a good compromise.

As can be seen from the results above, which modeling technique or metric set can be considered “best” is highly dependent on the criteria used for evaluation. The prediction models in this case study yield a recall and accuracy comparable to recent studies. However, the precision of our models is very low due to the unbalanced nature of our data sets, and choosing another cut-off than 0.5 can possibly yield very different results.

4.4 Discussion

In the subsections above we have evaluated and compared several carefully selected modeling techniques and metric sets that entail different data collection costs. Our goal was to assess what measures are necessary to achieve practically useful predictions, what modeling techniques seem to be more helpful, and how our conclusions differ depending on the evaluation criteria used.

We observe that the *Process* measures on average yield the most cost-effective prediction models, whereas the *OO* metrics on average is no better than a model based on random selection of classes. Although the *Delta* measures alone does not yield particularly large ROC areas, these measures still yield more cost-effective prediction models than the *OO* metrics.

Turning to the evaluation criteria, a first observation is that using general confusion matrix criteria raises a number of issue: (i) it is difficult to assess if the default cut-off of 0.5 is appropriate and if not, what other cut-off value should be used; (ii) none of these criteria strongly relate to the main goal in our context, that is *ranking* classes according to their fault-proneness to prioritize and increase the cost-effectiveness of verification; (iii) none of these criteria are clearly related to the possible cost-effectiveness of applying a particular prediction model.

Further, another issue when evaluating prediction models is that what can be considered the best modeling technique or set of measures is highly dependent on the evaluation criteria used for evaluation. Consequently, it is crucial that the criteria used to evaluate fault-proneness prediction models are closely linked to the intended, practical application of the prediction models.

We argue that ROC and CE capture two properties that are of high importance within our context, namely class ranking and cost-effectiveness: The area under the ROC curve reflects the probability that a faulty class is assigned a higher fault probability than a non-faulty one, while the CE measure allows us to compare prediction models according to their cost-effectiveness based on a number of assumptions. As shown in Section 4.2, these two measures capture two different dimensions of model performance: The difference between the *Process* and the *OO* metric sets was not clearly visible when only considering the ROC area, whereas the differences considering CE were relatively large. The results showed that an apparently accurate model is not necessarily cost-effective. Consequently, we emphasize the importance of considering not only measures such as the ones that can be derived from the confusion matrix, but also specific measures that are more closely related to the possible cost-effectiveness of applying fault-proneness prediction models to focus verification efforts.

5 Threats to Validity

The evaluation of techniques and metric sets were done using data from one single environment. The data collected were from 13 major releases over a period of several years. The system has endured a large extent of organizational and personnel change. Thus, it is unlikely that the results are heavily affected by individual developers and their experience, or the traits of certain releases of the system. Still, as with most case studies, one should be careful to generalize the specific results to all systems or environments. However, at a more general level, we believe that many methodological lessons can be learned from this study, including the need for doing systematic and comprehensive evaluations to ensure that the prediction models have the desired properties (e.g., cost effectiveness) for the purpose at hand.

In this study, we have not accounted for the actual cost of making the measures available and collecting them. Consequently, there are some initial costs associated with this process improvement activity that we do not account for. In particular, the Process metric set, being most cost-effective, is at the same time the measures that have the highest cost with respect to data reporting and collection.

The prediction models built in this case study were built using default parameters. That is, we have not systematically investigated how the models are affected by varying the parameters. There are possibly a large number of potential combinations of parameters for each modeling technique and optimizing the parameters with respect to some criteria for each technique would be very computational intensive. Furthermore, optimizing the modeling parameters might also lead to overfitted models that is highly specific to the training set. One way to alleviate this potential threat would be to apply evolutionary programming to optimize the parameters with respect to some property, e.g., cross-validated measures of ROC or CE.

Note also that the use of statistical tests in this study to test the differences between techniques and metric sets are somewhat exploratory in nature. In particular, from a formal standpoint, the notion of p-values is questionable in our context, because we have not taken a random sample from a target population, but rather used all the data we had available and computed p-values on differences between subsets of our data. For this reason we have also reported effect sizes, which are not problematic in this regard.

6 Conclusions and Further Work

Our review of recent studies revealed that many studies do not comprehensively and systematically compare modeling techniques and types of measures to build fault-prediction models. Many works also do not apply suitable evaluation methods and show little consistency in terms of criteria and methods that are used to evaluate the prediction models. Thus, it is hard to draw general conclusions on which measures and modeling techniques to use to build fault-proneness prediction models based on the existing body of studies. Further, most studies evaluate their models using confusion matrix criteria while we have shown that the metric set or technique that is put forward as the best is highly dependent on the specific criteria used.

Except for a few studies, i.e., [21, 38], there has been no systematic and comprehensive effort on comparing modeling techniques to build accurate and useful fault-proneness prediction models. In this paper, we do not only compare a carefully selected set of modeling techniques in a systematic way, but we also compare the impact of using different types of measures as predictors, based on different evaluation criteria. By doing so, we also propose a systematic

process and associated data analysis procedures for the rigorous comparison of models in terms of their cost effectiveness.

More precisely, we have empirically evaluated all combinations of three distinct sets of candidate measures (OO structural measures, code churn measures, and process change and fault measures) and eight, carefully selected modeling techniques, using a number of evaluation criteria. Overall, the findings are that the measures and techniques that are put forward as the “best” is highly dependent on the evaluation criteria applied. Thus, it is important that the evaluation criteria used to evaluate the prediction models are clearly justified in the context in which the models are to be applied.

Within the field of software verification we propose a surrogate measure of cost-effectiveness (CE) that enables us to assess and compare the possible benefits of applying fault-proneness prediction models to focus software verification efforts, e.g., by ranking the classes according to fault-proneness and focusing unit testing on the π % most fault-prone components. Using this CE measure to evaluate the prediction models in our case study revealed that using OO metrics to build fault-proneness prediction models does not necessarily yield cost-effective models – possibly because these metrics show strong correlation with size related measures, and prediction models that merely capture size are not cost-effective under the assumption that verification costs are proportional to size. Further, this case study clearly suggests that one should consider process-related measures, such as measures related to the history of changes and faults, to improve prediction model cost-effectiveness. Regarding the choice of modeling technique, the differences appear to be rather small in terms of cost-effectiveness, although Adaboost combined with C4.5 overall gave the best results. Note however that we have only compared techniques using default parameters, and as future work we will try to optimize the parameters while attempting to avoid overfitting.

The CE measure proposed in this paper is a *surrogate* measure to facilitate comparisons of prediction models using a criterion that is directly linked to the assumed cost-effectiveness of using such models to focus verification efforts. In order to assess the *real* cost-effectiveness and possible return on investment, we have recently performed a pilot study where the C4.5 prediction model was applied in a new release of the COS system. In this pilot study, developers spent an additional week of unit testing on the most fault-prone classes and several serious faults that otherwise would have slipped through to later testing phases or even the production system was discovered and corrected. Preliminary results suggest a return of investment of about 100 percent by preventing these faults from slipping through to later phases where they would have been more expensive to correct [84]. Due to these promising preliminary results, plans are underway to perform large-scale evaluations of the costs and benefits of using the prediction models to focus testing in the COS project.

References

- [1] L. C. Briand and J. Wuest, "Empirical Studies of Quality Models in Object-Oriented Systems," *Advances in Computers*, vol. 59, pp. 97-166, 2002.
- [2] J. S. Collofello and S. N. Woodfield, "Evaluating the effectiveness of reliability-assurance techniques," *Journal of Systems & Software*, vol. 9, pp. 191-195, 1989.
- [3] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, Second ed.: Morgan Kaufman, 2005.
- [4] L. C. Briand and J. Wust, "Empirical studies of quality models in object-oriented systems," *Advances in Computers, Vol 56*, vol. 56, pp. 97-166, 2002.
- [5] "IEEE standard glossary of software engineering terminology," in *IEEE Std 610.12-1990*, 1990.

- [6] N. E. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE Transactions Software Engineering*, vol. 25, pp. 675-689, 1999.
- [7] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *2002 International Symposium on Software Testing and Analysis*, 2002.
- [8] P. Tomaszewski, L. Lundberg, and H. Grahn, "Improving Fault Detection in Modified Code — A Study from the Telecommunication Industry," *Journal of Computer Science and Technology*, vol. 22, pp. 397-409, 2007.
- [9] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes," *IEEE Transactions on Software Engineering*, vol. 33, pp. 402-419, 2007.
- [10] Y. Zhou and H. Leung, "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults," *IEEE Transactions on Software Engineering*, vol. 32, pp. 771-789, 2006.
- [11] A. Janes, M. Scotto, W. Pedrycz, B. Russo, M. Stefanovic, and G. Succi, "Identification of defect-prone classes in telecommunication software systems using design metrics," *Information Sciences*, vol. 176, pp. 3711-3734, 2006.
- [12] S. Kanmani, V. R. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai, "Object-oriented software fault prediction using neural networks," *Information and Software Technology*, vol. 49, pp. 483-492, 2007.
- [13] G. J. Pai and J. B. Dugan, "Empirical Analysis of Software Fault Content and Fault Proneness Using Bayesian Methods," *IEEE Transactions on Software Engineering*, vol. 33, pp. 675-686, 2007.
- [14] M. M. T. Thwin and T.-S. Quah, "Application of neural networks for software quality prediction using object-oriented metrics," *Journal of Systems and Software*, vol. 76, pp. 147-156, 2005.
- [15] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the Location and Number of Faults in Large Software Systems.," *IEEE Transactions on Software Engineering* vol. 31, pp. 340-355, 2005.
- [16] T. J. Ostrand and E. J. Weyuker, "How to measure success of fault prediction models," in *Fourth international workshop on Software quality assurance (SOQUA)* Dubrovnik, Croatia: ACM, 2007.
- [17] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Automating algorithms for the identification of fault-prone files," in *2007 International symposium on Software testing and analysis*, London, United Kingdom, 2007.
- [18] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Using Developer Information as a Factor for Fault Prediction," in *International Workshop on Predictor Models in Software Engineering, 2007.* , 2007.
- [19] T. M. Khoshgoftaar and N. Seliya, "Comparative Assessment of Software Quality Classification Techniques: An Empirical Case Study," *Empirical Software Engineering*, vol. 9, pp. 229-257, 2004.
- [20] A. P. Nikora and J. C. Munson, "Developing fault predictors for evolving software systems," in *Ninth International Software Metrics Symposium (METRICS'03)*, 2003, pp. 338-350.
- [21] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *15th International Symposium on Software Reliability Engineering*, 2004, pp. 417-428.
- [22] S. Kim, J. E. J. Whitehead, and Y. Zhang, "Classifying Software Changes: Clean or Buggy?," *IEEE Transactions on Software Engineering*, vol. 34, pp. 181-196, 2008.
- [23] P. Tomaszewski, L. Lundberg, and H. Grahn, "Increasing the Efficiency of Fault Detection in Modified Code," in *12th Asia-Pacific Software Engineering Conference*, 2005.
- [24] R. Subramanyam and M. S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Transactions on Software Engineering*, vol. 29, pp. 297-310, 2003.
- [25] N. Nagappan and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study," in *First International Symposium on Empirical Software Engineering and Measurement*, 2007, pp. 364-373.
- [26] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *28th international conference on Software engineering*, Shanghai, China, 2006.
- [27] T. M. Khoshgoftaar and E. B. Allen, "Ordering Fault-Prone Software Modules," *Software Quality Journal*, vol. 11, pp. 19-37, 2003.
- [28] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *27th international conference on Software engineering*, St. Louis, MO, USA, 2005.
- [29] J. Rosenberg, "Some misconceptions about lines of code," in *Fourth International Software Metrics Symposium*, 1997, pp. 137-142.
- [30] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476-493, 1994.
- [31] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 308-320, 1976.
- [32] L. C. Briand, J. Daly, and J. Wust, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Engineering*, vol. 3, pp. 65-117, 1998.
- [33] L. C. Briand, J. W. Daly, and J. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 25, pp. 91-121, 1999.
- [34] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *Journal of Systems and Software*, vol. 23, pp. 111-122, 1993.
- [35] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, vol. 26, pp. 653-661, July 2000.

- [36] T. J. Yu, V. Y. Shen, and H. E. Dunsmore, "An analysis of several software defect models," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1261-1270, Sept. 1988.
- [37] T. M. Khoshgoftaar and N. Seliya, "Analogy-Based Practical Classification Rules for Software Quality Estimation," *Empirical Software Engineering*, vol. 8, pp. 325-350, 2003.
- [38] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, pp. 649-660, 2008.
- [39] M. Hall, "Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning," in *Seventeenth Int. Conf. on Machine Learning*, 2000, pp. 359-366.
- [40] H. H. Maurice, *Elements of Software Science (Operating and programming systems series)*: Elsevier Science Inc., 1977.
- [41] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen, "Mining software repositories for comprehensible software fault prediction models," *Journal of Systems and Software*, vol. 81, pp. 823-839, 2008.
- [42] I. Gondra, "Applying machine learning to software fault-proneness prediction," *Journal of Systems and Software*, vol. 81, pp. 186-195, 2008.
- [43] L. C. Briand, J. W. Daly, V. Porter, and J. Wust, "Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems," *Journal of Systems and Software*, vol. 51, pp. 245-273, 2000.
- [44] Y.-S. Lee, B.-S. Liang, S.-F. Wu, and F.-J. Wang, "Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow," in *International Conference on Software Quality*, Maribor, Slovenia, 1995.
- [45] K. K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Investigating the Effect of Coupling Metrics on Fault Proneness in Object-Oriented Systems," *Software Quality Professional*, vol. 8, 2006.
- [46] L. C. Briand, W. L. Melo, and J. Wust, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE Transactions on Software Engineering*, vol. 28, pp. 706-720, 2002.
- [47] L. C. Briand, J. Wust, S. V. Ikonovskii, and H. Lounis, "Investigating Quality Factors In Object-Oriented Designs: an Industrial Case Study," in *21st International Conference of Software Engineering (ICSE'99)*, Los Angeles, CA., 1999, pp. 345-354.
- [48] K. El Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, pp. 63-75, 2001.
- [49] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based Software Engineering Measurement," *IEEE Transactions on Software Engineering*, vol. 22, pp. 68-85, 1996.
- [50] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22, pp. 751-761, 1996.
- [51] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*: Prentice-Hall, Inc., 1996.
- [52] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, pp. 4-17, 2002.
- [53] F. Brito e Abreu and W. Melo, "Evaluating the Impact of Object-Oriented Design on Software Quality," in *Proceedings of the Third International Software Metrics Symposium (METRICS'96)*, Berlin, 1996, pp. 90-99.
- [54] Rhino, "<http://www.mozilla.org/rhino/>."
- [55] R. J. Freund and W. J. Wilson, *Regression Analysis: statistical modeling of a response variable*: Academic Press, 1998.
- [56] R. Quinlan, *C4.5: Programs for Machine Learning*: Morgan Kaufmann, 1993.
- [57] P. Werbos, *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*: Wiley, 1994.
- [58] V. N. Vapnik, *The Nature of Statistical Learning Theory*: Springer, 1995.
- [59] T. Joachims, "Learning to Classify Text Using Support Vector Machines," 2002.
- [60] M. A. Shipp, K. N. Ross, P. Tamayo, A. P. Weng, J. L. Kutok, R. C. Aguiar, M. Gaasenbeek, M. Angelo, M. Reich, G. S. Pinkus, T. S. Ray, M. A. Koval, K. W. Last, A. Norton, T. A. Lister, J. Mesirov, D. S. Neuberg, E. S. Lander, J. C. Aster, and T. R. Golub, "Diffuse large B-cell lymphoma outcome prediction by gene expression profiling and supervised machine learning," *Nat Med*, vol. 8, pp. 68-74, 2002.
- [61] Y. Freund and R. Schapire, "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting," in *European Conference on Computational Learning Theory*, 1995.
- [62] R. Melville and R. Mooney, "Creating Diversity in Ensembles using Artificial data," *Information Fusion*, vol. 6, pp. 99-111, 2005.
- [63] M. A. Hall and G. Holmes, "Benchmarking Attribute Selection Techniques for Discrete Class Data Mining," *IEEE transactions on knowledge and data engineering* vol. 15, pp. 14-37, 2003.
- [64] I. Kononenko, "On Biases in Estimating Multivalued Attributes," in *fourteenth Int. Joint conf. on Artificial Intelligence*, 1995, pp. 495-502.
- [65] G. Denaro and M. Pezze, "An empirical evaluation of fault-proneness models," in *24rd International Conference on Software Engineering*, 2002, pp. 241-251.
- [66] G. Dunteman, *Principal Component Analysis*: SAGE publications, 1989.
- [67] R. M. O'Brien, "A Caution Regarding Rules of Thumb for Variance Inflation Factors," *Quality and Quantity*, vol. 41, pp. 673-690, 2007.
- [68] R. L. Mason, R. F. Gunst, and J. L. Hess, *Statistical Design and Analysis of Experiments (Second Edition)*: Wiley-Interscience, 2003.

- [69] J. H. Friedman, "Multivariate Adaptive Regression Splines," *The Annals of Statistics*, vol. 19, pp. 1-67, 1991.
- [70] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, pp. 886-894, 1996.
- [71] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, pp. 5-32, 2001.
- [72] E. Arisholm and L. C. Briand, "Predicting Fault-prone Components in a Java Legacy System," in *5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE)*, , Rio de Janeiro, Brazil, 2006, pp. 8-17.
- [73] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software," in *The 18th IEEE International Symposium on Software Reliability, 2007. ISSRE '07.*, 2007, pp. 215-224.
- [74] T. M. Khoshgoftaar and E. B. Allen, "Modeling software quality with classification trees," in *Recent Advances in Reliability and Quality Engineering* vol. 2, H. Pham, Ed. Singapore: World Scientific Publishing, 2001, pp. 247-270.
- [75] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (ROC) curve," *Radiology*, vol. 143, pp. 29-36, April 1, 1982 1982.
- [76] R. A. Johnson and D. W. Wichern, *Applied multivariate statistical analysis*. Upper Saddle River, N.J.: Pearson Prentice Hall, 2007.
- [77] K. Chen, S. R. Schach, L. Yu, J. Offutt, and G. Z. Heller, "Open-Source Change Logs," *Empirical Software Engineering*, vol. 9, pp. 197-210, 2004.
- [78] JHawk, "<http://www.virtualmachinery.com/jhawkprod.htm>."
- [79] T. M. Khoshgoftaar and G. Kehan, "Count Models for Software Quality Estimation," *IEEE Transactions on Reliability*, vol. 56, pp. 212-222, 2007.
- [80] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the Location and Number of Faults in Large Software Systems.," *IEEE Transactions on Software Engineering*, vol. 31, pp. 340-355, 2005.
- [81] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2 ed.: Lawrence Erlbaum Associates, 1988.
- [82] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Transactions on Software Engineering*, vol. 27, pp. 630-650, July 2001 2001.
- [83] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'", " *IEEE Transactions on Software Engineering*, vol. 33, pp. 637-640, September 2007 2007.
- [84] M. J. Fuglerud, "Implementing and Evaluating a Fault-proneness Prediction Model to Focus Testing in a Telecom Java Legacy System," in *Dept. of Informatics*. vol. M. Sc.: University of Oslo, 2007.
- [85] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, pp. 897-910, 2005.
- [86] X. Jin, Z. Liu, R. Bie, G. Zhao, and J. Ma, "Support Vector Machines for Regression and Applications to Software Quality Prediction," in *Computational Science – ICCS 2006*, 2006, pp. 781-788.
- [87] T. M. Khoshgoftaar, E. B. Allen, and D. Jianyu, "Using regression trees to classify fault-prone software modules," *IEEE Transactions on Reliability*, vol. 51, pp. 455-462, 2002.
- [88] G. Succi, W. Pedrycz, M. Stefanovic, and J. Miller, "Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics," *Journal of Systems and Software*, vol. 65, pp. 1-12, 2003.

Appendix A: Papers reviewed

Author(s)	Dependent variable	Unit of analysis	Measures	Modeling techniques	Evaluation criteria	Validation method	Type of system
Arisholm et al. [72]	- Absence or presence of faults	- Class	- 14 Structural measures - 6 Delta measures - 5 Process measures - In addition, some measures of code violations, coding style errors etc.	- Univariate logistic regression - Multivariate logistic regression	- False positive rate and false negative rate at cut-off values ranging from 0 to 1.	- Leave-one-out cross-validation	- Large java legacy system consting of 1700 classes and 110KLOC
Arisholm et al. [73]	- Absence or presence of faults	- Class/file	- Structural measures - Process measures such as the amount of change undertaken and number of developers involved	8 Data mining techniques: - C4.5 - PART - SVM - Decorate C4.5 - Boost C4.5 - C4.5+PART - Neural network	- Confusion matrix criteria; precision, recall - Area under ROC curve	- 2/3 forms the training set - 1/3 is used as a test set. In addition, a later release of the same system is used as a separate test set.	- Large java legacy system consting of 2600 classes and 148KLOC
Briand et al. [46]	- Absence or presence of faults	- Class	- 23 structural measures	- Multivariate and univariate logistic regression - MARS	- Correctness - Completeness - Cost-benefit model	Two validation methods: (1) 10-fold cross-validation (2) System i forms training set whereas System ii forms the evaluation set	Two Java applications: - (i) Xpose (144 classes) - (ii) Jwriter (68 classes)

Author(s)	Dependent variable	Unit of analysis	Measures	Modeling techniques	Evaluation criteria	Validation method	Type of system
Denaro et al. [65]	- Highly faulty or not (more than 4 faults)	- Module	- 8 size measures - 30 structural measures including Halstead's difficulty, effort and program volume.	- Logistic regression	- R ² - Alberg-diagram - Confusion matrix criteria; accuracy, precision, recall	- Cross validation	- Apache release 1.3 and 2.0 (C)
Elish and Elish [38]	- Absence or presence of faults	- (i) Function - (ii) Method	- Structural properties	- Logistic regression - K-nearest neighbour - Multi-layer perceptron - Radial basis function - Bayesian belief network - Naïve Bayes - Random forest - Decision tree	- Accuracy - Precision - Recall - F-measure	- 10-fold cross-validation run 100 times using different seed values	- (i) The CM1 and PC1 data sets from NASA MDP (C) - (ii) The KC1 and KC3 data sets from NASA MDP (C++)
Gondra [42]	- Fault-proneness (neural network) - Absence or presence of faults (SVM)	- Function	- Structural properties - Some size metrics	- Neural network - Support vector machines (SVM)	- Mean squared error - Proportion of incorrect classifications (1-accuracy)	- 2/3 forms the training set - 1/3 is used as a test set	- The JM1 data set from NASA MDP, 315 KLOC (C)

Author(s)	Dependent variable	Unit of analysis	Measures	Modeling techniques	Evaluation criteria	Validation method	Type of system
Guo et al. [21]	- Absence or presence of faults	- (i) Function - (ii) Method	- 16 structural measures (included McCabe's and Halstead's) - 5 size measures	- Random forest - Discriminant analysis - Logistic regression - 20 data mining techniques using WEKA - See5/C5 - ROCKY	- Confusion matrix criteria; accuracy, sensitivity, specificity	For random forest: - 2/3 form the training set - 1/3 is used for evaluation/validation For all others: - 10 times 10-fold cross validation	- (i) The CM1, JM1 and PC1 data sets from NASA MDP (C) - (ii) The KC1 and KC2 data sets from NASA MDP (C++)
Gyimóthy et al. [85]	- Number of bugs - Absence or presence of bugs	- Class	- The 6 CK'94 metrics	- Multivariate and univariate linear regression - Multivariate and univariate logistic regression - C4.5 - Neural network	- Accuracy (in the paper called precision) - Recall (in the paper called correctness) - Completeness	None	- Version 1.0 through 1.6 of the Mozilla email and browser suite (C++)
Janes et al. [11]	- Number of defects	- Class	- CK '94 class level metrics - NOS	- Poisson regression - Negative binomial regression - Zero-inflated negative binomial regression (all are univariate)	- Spearman rank correlation - Dispersion - Alberg-diagrams	None	- Five real-time telecommunication systems written in C++ (63400 LOC in total)

Author(s)	Dependent variable	Unit of analysis	Measures	Modeling techniques	Evaluation criteria	Validation method	Type of system
Jin et al. [86]	- Number of changes	- Module	- 5 structural measures - 6 size measures	- Multivariate linear regression - Conjunctive rule - Locally weighted regression - Support vector machine regression	- Mean absolute error - Correlation coefficient	- 10-fold cross-validation	- MIS dataset
Kanmani et al. [12]	- Absence or presence of faults (faults found during testing)	- Class	- 57 Structural OO measures including CK'94, Briand's coupling measures as well as Li and Henry's metrics - 7 Size measures	- Back propagation neural network - Probabilistic neural network - Discriminant analysis - Logistic regression	- Type I and Type II error rates - Correctness - Completeness - Effectiveness - Efficiency	- 2/3 forms the training set - 1/3 is used as a test set	- Object-oriented library management system developed by graduate students (10-15KLOC)
Khoshgoftaar et al. [27]	- (1) Number of faults - (2) Debug code churn	- Module	(1): - Structural properties (e.g., number of unique operands, Halstead cycl. compl.) - Size metrics (2): - Structural properties (e.g. McCabes complexity metrics, number of edges&nodes in control flow graph etc.) - Size metrics	- Multiple linear stepwise regression	- R ² - Average absolute and relative error - The percentage of faults obtained compared to an «optimal» (actual) model at different thresholds (percentage of modules)	- (1) 2/3 form the training set while the remaining 1/3 is used to evaluate/validate the model - (2) Release 1 forms the training set, release 2 is used to evaluate/validate model	Two systems: (1) Military system written in Ada (2) Large legacy telecommunications system

Author(s)	Dependent variable	Unit of analysis	Measures	Modeling techniques	Evaluation criteria	Validation method	Type of system
Khoshgoftaar et al. [79]	- Number of faults - Probability of two faults or more	- Module (Ada package)	- (1) 7 structural measures including some size related measures - (2) five product measures obtained during inspection	- Logistic regression - Poisson regression - Zero-inflated Poisson regression	- Average absolute error - Average relative error - Type I and Type II misclassification rates	- 2/3 form the training set - 1/3 form the test set	Two case studies: - (1) Large military telecom system written in Ada - (2) Two large embedded applications used for config. of wireless telecom products
Khoshgoftaar et al. [37]	- Absence or presence of customer-discovered faults	- Set of related source-code files (modules)	- 24 Structural measures - 14 Process measures - 4 Software execution metrics	- Case Based Reasoning by (i) Majority vote and (ii) Data clustering	- Type I and Type II misclassification rates, where Type II is considered most important	- Train using release 1 - Select model using leave-one-out cross validation - Test using release 2, 3 and 4	- Large legacy telecommunication software, procedural paradigm (1000KLOC)
Khoshgoftaar et al. [19]	- Absence or presence of faults detected during system operation (post-release)	- Set of related files (data collected at file level, and then aggregated)	- Structural measures - Software execution metrics (execution time)	- Logistic regression - Case-based reasoning - CART - Regr. tree using S-PLUS - Sprint-Sliq - C4.5 - Treedisc	- Type I and Type II error rates (model selection) - Expected cost of misclassification (model evaluation)	- Train using release 1 - Select using release 2 - Evaluate using release 2, 3 and 4	- Large-scale legacy telecommunications system, procedural paradigm (PROTEL)

Author(s)	Dependent variable	Unit of analysis	Measures	Modeling techniques	Evaluation criteria	Validation method	Type of system
Khoshgoftaar et al. [87]	- Absence or presence of faults in modules that was changed since the prior release	- Module; one or more functionally related source-code files	- 26 structural measures including size-related measures - 4 metrics capturing the average execution time of a module	- Regression tree using S-Plus	- Type I and Type II misclassification rates - Estimated profit and ROI	- Release 1 was used as training set - Release 2-4 were used as separate test sets	- Embeded real-time system consisting of more than 10.000 KLOC written in a procedural language (PROTEL)
Kim et al. [22]	- Clean or buggy commit	- Change (committed change to source code repository)	- 8 RCS meta measures, e.g. day of week and for commit, cummulative number of changes and bugs - The deltas between the new and old revision for 61 complexity metrics	- Support vector machine	- Confusion matrix criteria; accuracy, precision, recall	- 10 fold cross validation	- 12 open source software projects including Apache, Subversion, Eclipse and PostgreSQL
Nagappan et al. [25]	- Number of post-release failures - Absence or presence of post-release failures	- System binaries	- Change (churn) measures; lines added, deleted or modified. Number of files that churned and number of changes. - Architectural dependencies	- Multivariate linear regression using PCA (count) - Multivariate Binary logistic regression using PCA (failure-proneness)	- F-test (coeff. sign.) - R ² , both adjusted, Nagelkerkes, and Cox & Snell - Spearman rank correlation - Pearson correlation - Precision and recall	- Random split; 2/3 training, 1/3 test. Repeated 5 times.	- Windows 2003 Server

Author(s)	Dependent variable	Unit of analysis	Measures	Modeling techniques	Evaluation criteria	Validation method	Type of system
Nagappan et al. [26]	- Number of post-release failures	- System binaries	- 11 structural measures at function level (aggregated to module level as Total and Maximum) - 4 structural measures at class level (aggregated to module level as Total and Maximum) - 3 structural measures at module level	- Univariate and multivariate (using PCA) linear regression	- R ² and adjusted R ² - Spearman and Pearson rank correlation	- Random split for each subsystem; 2/3 training, 1/3 test. Repeated 5 times. - 5 models; one for each component is applied to the other 4 components.	- 5 object-oriented components in Windows; including Internet Explorer 6 and IIS
Nikora et al. [20]	- Cumulative number of faults across releases	- Function / procedure	- 6 size measures - Some control flow graph measures	- Multiple linear regression using principal components	- R ²	None	- Space shuttle mission software
Olague et al. [9]	- Absence or presence of faults	- Class	- CK '94 class metrics - Abreu's metrics - Bansiya and Davis' metrics	- Univariate binary logistic regression (used for variable selection) - Multivariate binary logistic regression - Also linear regression was used, but were not successful in pred. faults	- Hosmer-Lemeshow test - Percentage correctly classified (accuracy)	For release $x < n < y$, where $x - y = 5$: - Train using set n - Test/evaluate on $n+1$	- Mozilla Rhino, (Open source Java system)

Author(s)	Dependent variable	Unit of analysis	Measures	Modeling techniques	Evaluation criteria	Validation method	Type of system
Ostrand et al. [16]	- Number of faults (Pre- and post-release)	- File	<ul style="list-style-type: none"> - Lines of code (LOC) - Whether file is new or changed/unchanged - Age of file - Number of faults in prev. rel. - Language (java, perl, c, xml etc.) - Number of different developers who have worked one the file 	- Negative binomial regression	- Confusion matrix criteria; accuracy, recall, precision, type I and type II error ratios at different percentages of files selected that are predicted as most fault-prone	None	- Large industrial software systems (doesn't state language, design paradigm etc.)
Ostrand et al. [15]	- Number of faults (Pre- and post-release)	- File	<ul style="list-style-type: none"> - Lines of code (LOC) - Whether file is new or changed/unchanged - Age of file - Number of faults in prev. rel. - Language (java, perl, c, xml etc.) 	- Negative binomial regression	- Percentage of faults included by model in th top 20% most fault-prone files	<ul style="list-style-type: none"> - Training set - Test/evaluate on later releases of the same system 	- Large industrial software systems; one written in Java, and the other mainly in SQL
Ostrand et al. [17]	- Number of faults (Pre- and post-release)	- File	<ul style="list-style-type: none"> - LOC - Age - Number of prior changes and faults - Exposure (the fraction of the release which a new file existed) - Language (C++, SQL, C etc.) 	- Negative binomial regression	<ul style="list-style-type: none"> - The percentage of LOC included in the fault-prone files vs. the percentage of faults included in those files - Whether % LOC in the fault-prone files is smaller than the percentage of faults. 	<ul style="list-style-type: none"> - Model for release N was built using release 2 through N-1 - In addition, two models built from another system were assessed 	- 35 releases of a large maintenance support system (C++, SQL an others)

Author(s)	Dependent variable	Unit of analysis	Measures	Modeling techniques	Evaluation criteria	Validation method	Type of system
Pai et al. [13]	- Number of faults - Absence or presence of faults	- Class	- 6 CK'94 class level metrics - LOC	- Linear regression - Bayesian networks: * Bayesian linear regression * Bayesian poisson regression * Bayesian logistic regression	- Kolmogorov-Smirnov - Deviance information criterion - Alberg-diagrams - Confusion matrix measures; sensitivity, specificity, precision, Type I and Type II error rates	- 10-fold cross validation	- The KC1 data set from NASA MDP (C++, 43 KLOC, 145 classes)
Subramanyam et al. [24]	- Number of defects	- Class	- Some of the CK '94 class measures (WMC, CBO, DIT) and size (NOS)	- Linear regression using Box-Cox transformation and weighted least squares	- Adjusted R ²	None (built from and applied to one release)	- Commercial object-oriented B2C e-commerce application suite (C++ and Java)
Succi et al. [88]	- Number of faults (defects)	- Class	- LOC - The 6 CK'94 metrics	- Negative binomial regression - Zero-inflated binomial regression - Poisson regression	- Relative standard error - Dispersion - Pareto analysis (using 80% of the faults)	None	- Two commercial applications each of consisting of approx. 150 classes

Author(s)	Dependent variable	Unit of analysis	Measures	Modeling techniques	Evaluation criteria	Validation method	Type of system
Thwin et al. [14]	- Number of faults	- Class	- 8 structural measures including CK metrics	2 neural network techniques: - General regression neural network - Ward neural network	- R squared, mean square error, mean/maximum/minimum absolute error	- 10 fold cross validation	- Three object-oriented subsystems totaling 43KLOC in size, 97 classes. The subsystems are part of a large industrial system consisting of 200 subsystems.
Tomaszeski et al. [8]	- Number of faults and fault density	- Class	- 7 CK'94 class level metrics - Cyclomatic complexity - 5 size measures - Number of new or modified LOC	- Univariate and multivariate linear regression	- R ² - Spearman rank correlation - Presumed cost reduction in terms of percentage faults detected compared to optimal model, and further compared to a simple model based on size and finally a random model.	- Build model from one release of one system, evaluating the model on a later release of the same system and on another system	- Two large object-oriented telecommunication systems (500 KLOC and 600 KLOC)
Tomaszewski et al. [23]	- Number of faults and fault density	- Class	- CK'94 class level metrics - 5 size measures - Number of new or modified LOC	- Stepwise multivariate linear regression	- R ² - F-test - Presumed cost reduction wrt. percentage of faults detected compared to optimal model, a model based on size and finally a random model.	None (built from and applied to one release)	- Large object-oriented telecommunication system (250 KLOC)

Author(s)	Dependent variable	Unit of analysis	Measures	Modeling techniques	Evaluation criteria	Validation method	Type of system
Vandecruys et al. [41]	- Absence or presence of faults	- (i) Function or subroutine - (ii) Method	- Size metrics - Structural measures such as Halstead volume, effort and difficulty, and cucromatic complexity etc.	- AntMiner+ - RIPPER - C4.5 - Logistic regression - k-nearest neighbour - Support vector machine - Majority vote	- Confusion matrix criteria; accuracy, sensitivity, specificity	- 70% training set - 30% test set	- (i) The PC1 and PC4 data sets from NASA MDP (C) - (ii) The KC1 data set from NASA MDP (C++)
Weyuker et al. [18]	- Number of faults (Pre- and post-release)	- File	Same as for the ISSTA'07 paper, but in addition a number of measures meant to capture the number of developers involved in developing a file.	- Negative binomial regression	- Percentage of faults found in the (predicted) 20% most fault-prone files	- Model for release N was built using release 2 through N-1, for N >= 6	- 35 releases of a large maintenance support system (C++, SQL an others)
Zhou et al. [10]	- Absence or presence of (1) high severity faults, (2) low severity faults, and (3) both	- Class	- 7 CK'94 class level metrics; WMC, DIT, RFC, NOC, CBO, LCOM and LOC	- Univariate logistic regression - Multivariate logistic regression - Naive Bayes network - Random forest - Nearest neighbour with generalization	- Confusion matrix criteria; correctness and an awkward definition of precision - Completeness	- Leave-one-out cross validation	- The KC1 data set from NASA MDP

