Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software

Erik Arisholm
Simula Research Laboratory
Martin Linges v 17, Fornebu
P.O.Box 134, 1325 Lysaker, Norway
erika@simula.no

Lionel C. Briand
Simula Research Laboratory
Martin Linges v 17, Fornebu
P.O.Box 134, 1325 Lysaker, Norway
briand@simula.no

Magnus J. Fuglerud
Simula Research Laboratory
Martin Linges v 17, Fornebu
P.O.Box 134, 1325 Lysaker, Norway
magnujf@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 probability of fault. 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 inspections and would like to be able to devote extra resources to faulty system parts. The main research focus of this paper is two-fold: (1) use and compare many data mining and machine learning techniques to build fault-proneness models based mostly on source code measures and change/fault history data, and (2) demonstrate that the usual classification evaluation criteria based on confusion matrices may not be fully appropriate to compare and evaluate models.

Categories and Subject Descriptors

D.2.8 [**Metrics**]: Complexity measures, Performance measures, Process metrics, Product metrics, Software science

General Terms

Management, Measurement, Economics, Reliability, Experimentation, Verification.

Keywords

Class fault-proneness, prediction, testing

1. INTRODUCTION

A significant research effort has been dedicated to defining specific quality measures and building quality models based on those measures. Such models can then be used to help decision-making during development. Fault-proneness or the number of defects detected in a software module (e.g., class) is the most frequently investigated dependent variable [2]. In this case, we may want to predict the fault-proneness of classes in order to focus validation and verification effort, thus potentially finding more defects for the same amount of effort. 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.

Given that software development companies might spend between 50 to 80 percent of their software development effort on testing [3], research on fault-proneness prediction models can be motivated by its high cost-saving potential. There exists however very little evidence of the economic viability of such models [2]. The most common way to evaluate the prediction models is by assessing their classification accuracy by means of the usual confusion matrix criteria (e.g., precision, recall). However, as the results presented in this paper will show, such criteria do not clearly and directly relate to the cost effectiveness of using class fault-proneness prediction models when applied to focus verification and validation activities. To compare the potential cost-effectiveness of alternative prediction models, we need to consider (surrogate) measures of verification cost for the classes selected for verification. For many verification activities, e.g., 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 where we are likely to find faults but not simply larger classes.

To build such 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 [4]. 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.

This paper presents the results from building a variety of alternative fault-proneness prediction models for a large, evolving Java system in the Telecom domain. We will do so using measures of structural aspects of classes (e.g., their coupling), change characteristics (e.g., their type, diffusion, size and duration, experience of the developers performing them), as well

[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.

as the history of changes and faults (e.g., whether a class had fault corrections in a previous release). We will compare not only their accuracy but also their potential cost-effectiveness to focus verification.

The remainder of this paper is organized as follows. Section 2 discusses a number of relevant data analysis and modeling techniques used in the remainder of the paper. Section 3 presents our case study design. Section 4 relates our approach to existing research. Section 5 compares classification accuracy and cost-effectiveness of the prediction models and discusses implications for model selection. Section 6 concludes and outlines directions for future research.

2. FAULT-PRONENESS MODELING

There exists a large number of modeling techniques to build a fault-proneness model, such as a classification model determining whether classes or files are faulty. A classical statistical technique used in many existing papers is Logistic Regression [5]. But many techniques are also available from the fields of data mining, machine learning, and neural networks [4]. 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 [6]. In our context, each leaf of a decision tree would then correspond to a subset of the data set available (characterized by class source code characteristics and their fault/change history, as described in Section 3.3) and its probability 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 class to contain a fault based on the instances each rule covers. As opposed to the divide-and-conquer strategy of decision trees, these algorithms iteratively identify attribute-value pairs that maximize the probably 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 that 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 providing 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 [7], which can also be used for classification purposes. A more recent technique that has received increased attention in recent years across various scientific fields [8-10] is the Support Vector Machine classifier (SVM), which attempts to identify optimal hyperplanes with nonlinear boundaries in the variable space in order to minimize misclassification.

Based on the above discussion, we will compare here one classification tree algorithm, namely 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 [4], 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. Furthermore, as the outputs of leaves and rules are directly comparable, we will combine 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.

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 [4] 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 algorithm and one of the well-know algorithm we use here is AdaBoost [11] that 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. This recent technique is claimed [12] to consistently improve not only the base model but also outperform other techniques such as Bagging and Random forest [4], which we will not include here for that reason. Since it is also supposed to outperform boosting on small training sets and rivals it on larger ones, it is also considered in our study.

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 components or files, 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, which often turn out not to be useful or are 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 [4]. This happens because as the tree gets built, the algorithm works with a decreasing amount of data, which may 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 [5]. A recent paper [13] has 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 (Correlation-based Feature Selection [14], ReliefF [15]. In our case these two techniques yielded the same selection of variables. Though we used CFS for all modeling techniques, we will only report it for C4.5 as the results were not significantly different for others (i.e., the impact of CFS was at best small) and as, a discussed below, C4.5 will be the technique we ultimately retain to focus testing.

3. DESIGN OF STUDY

A case study was performed to build and evaluate the alternative fault-proneness prediction models described in Section 2. This section describes the development project, study variables, data collection, and model building and evaluation procedures.

3.1 Telenor COS Development Environment

A large Java legacy system (COS) is being maintained at Telenor, Oslo, Norway, and there is a constant shortage of resources and time for testing and inspections. The quality assurance engineers wanted to investigate means to focus verification on parts of the system where faults were more likely to be detected. 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. Though many studies on predicting fault-prone classes on the basis of the structural properties of object-oriented systems have been reported [2], one specificity of the study presented here is the fact that we need to predict fault-proneness for a changing legacy system. We therefore not only need to account for the structural properties of classes across the system, but also for changes and fault corrections on specific releases and their impact on the code, among a number of factors potentially impacting fault-proneness. Another interesting issue to be investigated is related to the fact that past change and fault data are typically available in legacy systems and such data could be useful to help predicting fault-proneness, e.g., by identifying what subset of classes have shown to be inherently fault prone in the past.

The legacy system studied is a middleware system serving the mobile division in a large telecom company. It 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. We used 12 recent releases of this system for model building and evaluation. At any time, somewhere between 30 to 60 software engineers have been involved in the project. The core system currently consists of more than 2600 Java classes amounting to about 148K SLOC[2]. 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. As further discussed below, the prediction model described below is currently being applied to focus verification in COS release 22.

3.2 Dependent Variable

The dependent variable in our analysis is the occurrences of corrections in classes of a specific release which are due to field error reports. Since our main current objective is to facilitate unit testing and inspections, the class was a logical unit of analysis. However, it is typical for a fault correction to involve several classes and we therefore count the number of distinct fault corrections that was required in that class for developing the *next* release n+1. This aims at capturing the fault-proneness of a class in the *current* release n. 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.

3.3 Explanatory Variables

The fundamental hypothesis underlying our work is that the fault-proneness of classes in a legacy, object-oriented system can be affected by the following factors:

- the structural characteristics of classes (e.g., their coupling)
- the amount of change (requirements or fault corrections) undertaken by the class to obtain the current release
- the experience of the individual performing the changes
- other, unknown factors that are captured by the change history (requirements or fault corrections) of classes in previous releases

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. Explanatory variables are defined in Table 1.

3.4 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 1. 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-crititical fault correction, 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* modified in a *CP*, we record the number of lines added and deleted, as modeled by

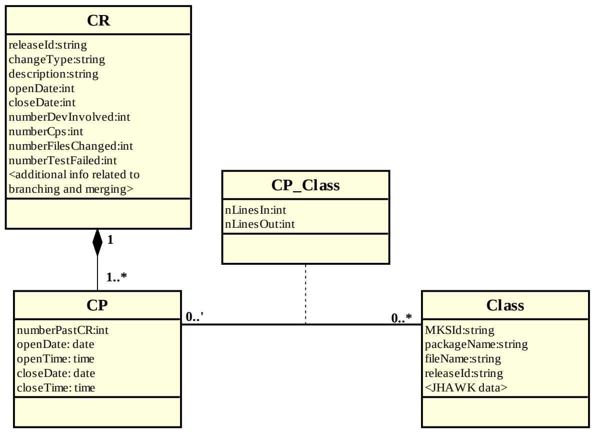

Figure 1 COS Data model

[2] In addition to this, the system consists of a large number of test classes, library classes, and about 1000K SLOC of generated code, but this code is not considered in our study.

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. Finally, for each release, a code parser (JHawk [16]) is executed to collect structural measures for the class, which are combined with the MKS change information. Independent and dependent variables (Faults in release n+1) were computed on the basis of the data model presented in Figure 1.

3.5 Model Building and Assessment

To build and evaluate the prediction models, class-level structural and change data from 12 recent releases of COS were used. The data was divided into three separate subsets, as follows. The data from the 11 first releases was used two form two datasets; a training set to build the model and a test set to evaluate the

Table 1 Summary of the explanatory variables in the study

Variable	Description	Source
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
[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
	FOR CRs of type X={CLL, CFL, CKL, M, CE, E}:	
<X>_CR	Same def as *numberCRs* but only including the subset of CR's of type X.	MKS
<X>_CPs	Same def as *numberCpsForClass* but only including the subset of CR's of type X	MKS
<X>numberCps	Same def as *numberCps* but only including the subset of CR's of type X	MKS
<X>numberFilesChanged	Same def as *numberFilesChanged* but only including the subset of CR's of type X	MKS
<X>numberDevInvolved	Same def as *numberDevInvolved* but only including the subset of CR's of type X	MKS
<X>numberTestFailed	Same def as *numberTestFailed* but only including the subset of CR's of type X	MKS
<X>numberPastCr	Same def as *numberPastCr* but only including the subset of CR's of type X	MKS
<X>nLinesIn	Same def as *nLinesIn* but only including the subset of CR's of type X	MKS
<X>nLinesOut	Same def as *nLinesOut* but only including the subset of CR's of type X	MKS

predictions. More specifically, 66.7 percent of the data (16311 instances) were randomly selected as the *Training* dataset, whereas the remaining 33.3 percent (8143 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 307 instances representing faulty classes (that is, the class had at least one fault correction in release n+1). This is due to the fact that, in a typical release, a small percentage of classes turn out to be faulty. For reasons further discussed in Section 5.1, to facilitate the construction of unbiased models, we created a balanced subset (614 rows) from the complete training set, consisting of the 307 faulty classes and a random selection of 307 rows representing non-faulty classes. Finally, the most recent of the 12 selected releases formed the third distinct dataset, hereafter referred to as the *COS 20* test dataset, which we will also used as a test set. 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* test set indicates accuracy on a future release. This will give us insight on the level of decrease in accuracy to be expected, if any, when predicting the future.

Having described our model evaluation procedure, we now need to explain what model accuracy criteria we use. First, we consider the standard confusion matrix criteria [4]: *precision* and *recall*. 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.

Another common measure is the ROC[3] area [4]. A ROC curve is built by plotting on the vertical axis the number of faults contained in a percentage of classes on the horizontal axis. Classes are ordered by decreasing order of fault probability as estimated by a given prediction model. The larger the area under the ROC curve (the ROC area), the better the model. A perfect ROC curve would have a ROC area of 100%.

Though relevant, the problem with the general confusion matrix criteria 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 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. Such activities are likely to be roughly proportional to the size of the class. For example, that would be the case for structural coverage testing or even simple code inspections. So, 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 the cost effectiveness, we compare two curves as exemplified in Figure 2. Classes are first ordered from high to low fault

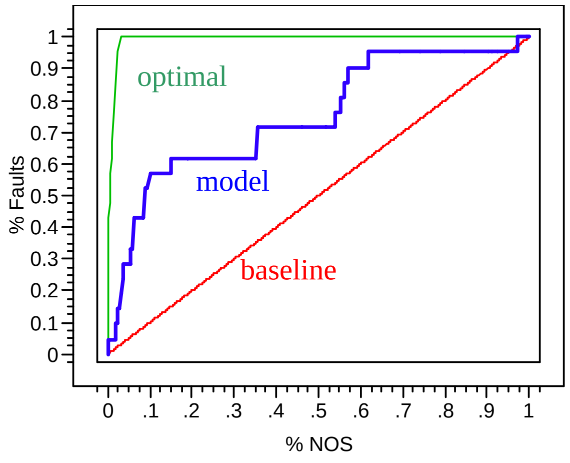

Figure 2 Computing a Surrogate Measure of Cost Effectiveness

probability. 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, our working assumption is that 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 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, (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 2.

[3] Receiver Operating Characteristic

4. RELATED WORK

There is a lot of research on fault-proneness models in OO systems. A survey is provided in [2]. In summary, structural properties of software, such as cyclomatic complexity or coupling, do seem to have an impact on fault proneness. Due to space constraints, the remainder of this section focuses on providing an in-depth discussion of research that has attempted to build fault-proneness prediction models in the context of evolving software systems for which change and/or fault history data were available.

Instead of using product measures such as lines of code or cyclomatic complexity, one study took a slightly different approach, and attempted to build change risk prediction models on the basis of simple change data [17]. The results suggest that the *risk* of performing a given change, in the sense that the change will cause a future software failure, can be successfully modeled on the basis of simple change measures such as the size, duration, diffusion and type of change.

A study on fault proneness prediction in a large, evolving system was reported in [18]. The dependent variable was the module-level cumulative number of faults (from the initial build to the last build). The independent variables were based on 12 different size and control-flow measures, but the measures were not used directly. Instead, principal component regression [5] was performed. More specifically, the 12 initial measures were transformed into three new variables on the basis of three identified principal components. Finally, for each of the three variables, the sum (across all builds) of the absolute values of the differences between all pairs of successive builds was computed. These sums of differences, denoted as *code churn* in [18], formed the independent variables. The first independent variable (mainly representing size changes in the modules across the builds) was found to be a significant predictor of the cumulative number of faults in a module leading to an adjusted R-Sq [5] of 0.61. No attempts were made to perform a cross-validation or evaluate the cost-benefits of the model on new releases, so no further direct comparisons with our study is possible.

A study of fault-proneness in a very large and long-lived software system was reported in [19]. The dependent variable was the number of fault incidences within a two-year period. The independent variables consisted of various product and process measures collected from repositories just before that two-year period. The product measures included module-level size and complexity measures, e.g., lines of code and cyclomatic complexity. The process measures included the number of *past* faults in a module, the number of changes or deltas to a module over its entire history, and the *age* of the code. Thus, the variables used in [19] are similar in type to the ones used in our study. However, the goals were different as the main focus in [19] was to identify the *reasons* for faults whereas the main goal in this paper is to build an optimal prediction model. As a result, this work did not attempt to evaluate the accuracy or cost-effectiveness of the obtained models by applying them to predict *future* faults, though this was vital in our case.

In [21], a case study on modeling fault-proneness over a sequence of four releases was presented. The system was composed of 800 KLOC of C code. The data was based on change reports as no design or code was available to the researchers. The independent variables thus only included measures such as the number of times a component was changed together with other components, number of files fixed in a given component and the number of lines of code added and deleted in a component, for a given release. A component was defined as a collection of files in the same directory, and it was assumed that the directory structure reflected the functional architecture. The components in the system were classified as fault-prone if the number of faults exceeded a given threshold value. One objective of the paper was to compare different statistical techniques (classification trees and discriminant analysis with or without the use of PCA to determine the set of candidate variables) for building fault-proneness models over a sequence of releases. Amongst others, they built fault-proneness classification trees (for each release) and evaluated the stability of the classification trees over successive releases. However, these results are not reported at a sufficient level of detail to enable a comparison with our results.

Preliminary results from our attempts to build fault prediction models for the COS system was reported in [22]. In this previous paper, we used only a very small subset of releases (three instead of 12) and used only logistic regression to build the models as our focus was not on applying data mining techniques. Furthermore, we used a partly different set of independent variables since at this point we did not have complete access to the underlying configuration management database, but instead had to rely on change data collected through an existing monitoring system used in the project (XRadar). Despite the small number of releases considered, one of the main conclusions from that study was the importance of including historic data about faults and changes in order to improve cost-effectiveness.

One of the main pieces of related work has been reported by Ostrand, Weyuker, and Bell in [1] and we therefore perform a detailed comparison of their study with ours. Their goal was also to predict fault-proneness in evolving systems mostly composed of Java code. One difference is that they predict fault proneness in files instead of classes, as their systems were not only coded in Java. This should, however, be similar for most files as Java files normally contain one public class and possibly their inner classes. Another important difference was that their main focus was to support *system* testing whereas in Telenor COS the main goal was to support the focus of extra unit testing to prevent as many faults as possible to reach subsequent testing phases and deployment.

They looked at two systems. For the first one (“Inventory”), all life cycle faults were considered whereas for the second one (“Provisioning”) only post unit testing faults were accounted for. They studied 17 Inventory releases and 9 Provisioning releases, but due to the small number of faults reported in the latter, they merged releases into three “pseudo” releases. The number of releases we considered in COS (12) is similar but we only accounted for post-release faults.

Descriptive statistics of the systems reported in [1] and in this paper (COS) are provided in Table 2. Inventory peaks at 1950 files and 538 KLOC in its last release whereas Provisioning is slightly above 2000 files and 437 KLOCS. Because it only accounts for post unit testing faults, the latter only has between 6 to 85 faults per release whereas the former has between 127 and 988 faults per release. The COS system is smaller with 148 KLOC of Java code. The number of faults across COS releases is expectedly more comparable to the provisioning system: 1 to 117.

Table 2 System information for [1] and this paper

	Number Releases	# KLOC range	# Faults range	# Faulty files range
Inventory	17	145–538	127–988	584–1950
Provisioning	9	381–437	6–85	6–64
COS	12	128–148	1–117	7–83

A fault was defined as a change made to a file because of a Modification Request (MR), which seems identical to our definition. (If n MRs change a file, this is counted as n faults.) One difference with our COS data collection though was that Ostrand *et al.* had no reliable data regarding whether a change was due to a fault. As a result, they used and validated a heuristic where changes involving less than three files were considered faults.

Ostrand *et al.* used negative binomial regression [5], which is a natural technique to use when predicting small counts. Since on average, a faulty file contained 2-3 faults in their systems, their modeling approach is perfectly justified. In our case, most faulty classes contained one fault and we therefore resorted to Logistic Regression to classify a class as faulty or not. In addition to this statistical approach, because this is one important focus of this paper, we tried out and compared many of the data mining techniques available to us for the sake of comparing prediction results. Some of those techniques have practical advantages discussed in Section 3.5, the main one being that they produce interpretable models, something we noticed was important for the practitioners using these models. There are, however, very few studies performing comprehensive comparisons of modeling approaches.

Based on their negative binomial regression model, for both systems, Ostrand et al. reported that the 20% most fault prone files contain an average of 83% of the faults across releases. These files represent an average of 59% of the source code statements. They used 20% as this was the "knee of the fault curve" where the number of faults contained in faulty files started to plateau. Comparisons with our results are provided in Section 5.4.

Table 3 Precision, Recall, and ROC Area for all Techniques

	Prec. Excl.	Prec. COS 20	Rec. Excl.	Rec. COS 20	ROC Excl	ROC COS 20
C4.5	4.7	1.8	71.1	66.7	79.0	85.2
PART	4.6	1.5	78.5	55.6	81.7	77.1
SVM	4.7	2.4	74.5	72.2	80.7	83.7
LogisticReg.	5.4	2.6	75.8	72.2	82.0	79.4
DecorateC4.5	5.5	2.2	76.5	66.7	83.6	82.2
BoostC4.5	4.7	1.4	75.2	55.6	79.4	69.2
CFSC4.5	4.8	2.3	77.9	66.7	79.6	79.1
C4.5+PART	5.1	3.1	77.9	94.4	81.0	89.1
Neural Net	5.8	2.2	73.2	66.7	82.6	82.7

The analysis in [1] used much fewer explanatory variables: number of LOCs per file, whether the file was changed from the previous release, the age of file in terms of number of releases, the number of faults in previous release, and the programming language. It was also reported that other variables were used but turned out not to be significant additional predictors: number of changes to file in previous release, whether a file was changed prior to the previous release, and cyclomatic complexity.

5. PREDICTION MODELS

In this section, we compare the predictions of the various modeling techniques selected in Section 2. We first compare them in terms of the usual confusion matrix criteria (Recall, Precision, ROC area) and then in terms of cost-effectiveness as defined in Section 3.5. We then compare our results to the ones published in [1] in order to determine commonalities and differences. Other differences in terms of objectives and methodology were already discussed in Section 4.

5.1 Precision, Recall, and ROC Area

First it is important to note that all results presented in this section are based on a balanced training set. As discussed in Section 4, there is a small percentage of faulty classes in COS, as it is usually the case in most systems. Nearly all the techniques we used performed better (sometimes very significantly) when run on a balanced dataset formed of all faulty classes plus a random sample of the same size of correct 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 0.5 to achieve balanced precision and recall.[4]

Table 3 provides confusion matrix statistics for faulty class predictions and we can see that differences among techniques in terms of precision, recall, and ROC area are in most cases very small, or at least too small to be of practical significance. The results are less consistent across techniques on the COS 20 test set but this is to be expected as this release had a small number of faults and it is therefore subject to more random variation. Therefore neither the metalearners (Boosting, Decorate) nor the variable/feature selection techniques (CFS, RELIEF) seem to make a clear, practically significant difference. One exception is that the combination of C4.5 and PART seems to bring notable improvement in terms of recall for COS 20. ROC areas are overall rather high and mostly above 80%, but the question remains about how to interpret such a result to assess the applicability of models. Despite small differences in classification accuracy, as discussed in Section 2, it is important to recall that certain techniques are easier to use and more intuitive than others. This is the case of classification trees such as the ones produced by C4.5 or coverage rule algorithms such as PART. Furthermore, the feature selection techniques tend to simplify the models. For example, the decision tree generated by C4.5 after using CFS went from 24 leaves to 17 leaves and was built based on 29 variables instead of the original 112 variables considered. This may be of practical importance when applying the models.

[4] As you change the threshold, recall increases and precision decreases, or vice-versa

The very small precision numbers are worth an explanation. This is due to the very imbalanced test sets, which are realistic, but which result nonetheless in low precision values. Even if there is a small misclassification probability of correct classes into faulty classes, when the proportion of correct classes is very large, this results into low precision. For example, based on a balanced test set with randomly selected correct classes and all faulty classes, we obtained precision numbers of 0.857 and 0.726, for the COS 20 and Extended test sets, respectively. So we can see that for imbalanced test sets, it is not easy to interpret precision values.

5.2 Cost-Effectiveness

We now turn our attention to Table 4 where cost-effectiveness values are reported for all techniques and for selected percentages of classes. Though we also report the results for 100% of the classes (entire CE area), one would in practice focus on small percentages as such prediction models would typically be used to select a small part of the system. But recall that computing a CE area is just a way to compare models without any specific percentage of classes in mind and based on a unique score. Note that in Table 4, the symbol "-" stands for negative CE values, which we do not need to report. The reason why CE values may look small, though they have been multiplied by 100, is that the vertical and horizontal axes are percentages, and therefore values below 1. Admittedly such CE values are not easy to interpret but their purpose is only to facilitate the comparison among models based on a measure that should be directly proportional to cost effectiveness in our context.

We can see that, as opposed to confusion matrix criteria, there are wide variations in CE across techniques and class percentages. For example, for 5% NOS in the Excluded test set, CE ranges from 0.014 (PART) to 0.406 (Boosting+C4.5). What we can also observed is that C4.5, though never the best, is never far from it for all class percentages and for both the COS 20 and Excluded test sets. On the other hand we see that some techniques provide unstable results which vary a great deal depending on the test set and the selected NOS percentage. For example, PART varies from reasonably good CE values to negative CE values. Based on these results, we selected C4.5 to apply within the COS project as it is both simple (very interpretable, easy to build and apply) and stable in terms of cost effectiveness.

However, from a general standpoint, if one can pre-determine what percentage of the code will, for example, undergo additional verification and testing, one may be in a position to choose the specific model optimizing CE for this particular percentage. For example, though NN does not fare particularly well in general, it does well for 1%. We can also assess the gain of using a predictive model for a specific percentage in a way which is more interpretable than CE areas. Figure 2, which was presented earlier, corresponds to actual curves for C4.5 on the COS 20 test set. If we take these results as an example, we see that, for example, 10% of the lines of code lead to nearly 60% of the defects. If we now assume we would have a perfect, optimal model to predict the future, we would obtain 100% of the faults for that NOS percentage (see Figure 2). The gain compared to what the average would obtain with random orders (10% of the faults) is substantial, but we also see that there is much room for improvement when compared to an optimal order. This was not clearly visible when only considering the confusion matrix precision and recall, or the ROC area.

5.3 Variables Selected in the C4.5 Tree

The C4.5 decision tree in Figure 3 has 24 leaves, implying that any one prediction will be assigned one of the possible 24 fault probabilities associated with these leaves. The tree makes use of 21 variables out of the 112 originally considered.

From Figure 3, we can see variables that belong to two distinct, broad categories:

- Nine variables relate to the amount of change undergone by classes in the current release or one of the last three releases. However, in some cases this should be carefully interpreted as it may also capture whether the class is new: for example, when nm1_CLL_CR = -1, this means the class did not exist in release n-1.
- Eleven variables relate to the source code properties of the class, such as inheritance, number of methods, cyclomatic complexity, cohesion, and coupling (fan out).

Of course, it is always difficult to interpret such results as many variables are often inter-correlated, as suggested by the fact that

Table 4 Cost-Effectiveness for all Techniques

	1% Excl.	1%. COS20	5% Excl.	5%. COS20	20% Excl	20% COS20	100% Excl	100% COS20
C4.5	0.026	0.041	0.397	0.695	3.56	6.86	18.03	24.96
PART	0.010	-	0.014	-	1.55	5.27	18.32	33.61
SVM	0.005	0.023	0.117	0.670	2.43	4.68	17.78	21.14
Logistic Reg.	0.028	0.054	0.222	0.71	2.72	4.88	18.22	17.49
Decorate C4.5	0.003	0.016	0.210	0.726	4.18	6.16	19.06	15.16
Boost C4.5	0.02	-	0.406	-	2.73	-	16.04	0.34
CFS C4.5	0.026	0.041	0.32	0.635	3.19	6.62	17.46	20.36
Neural Net	0.031	-	0.193	0.069	1.50	0.70	12.79	12.55
C4.5 + PART	0.026	0.041	0.21	0.172	2.47	6.16	19.12	31.71

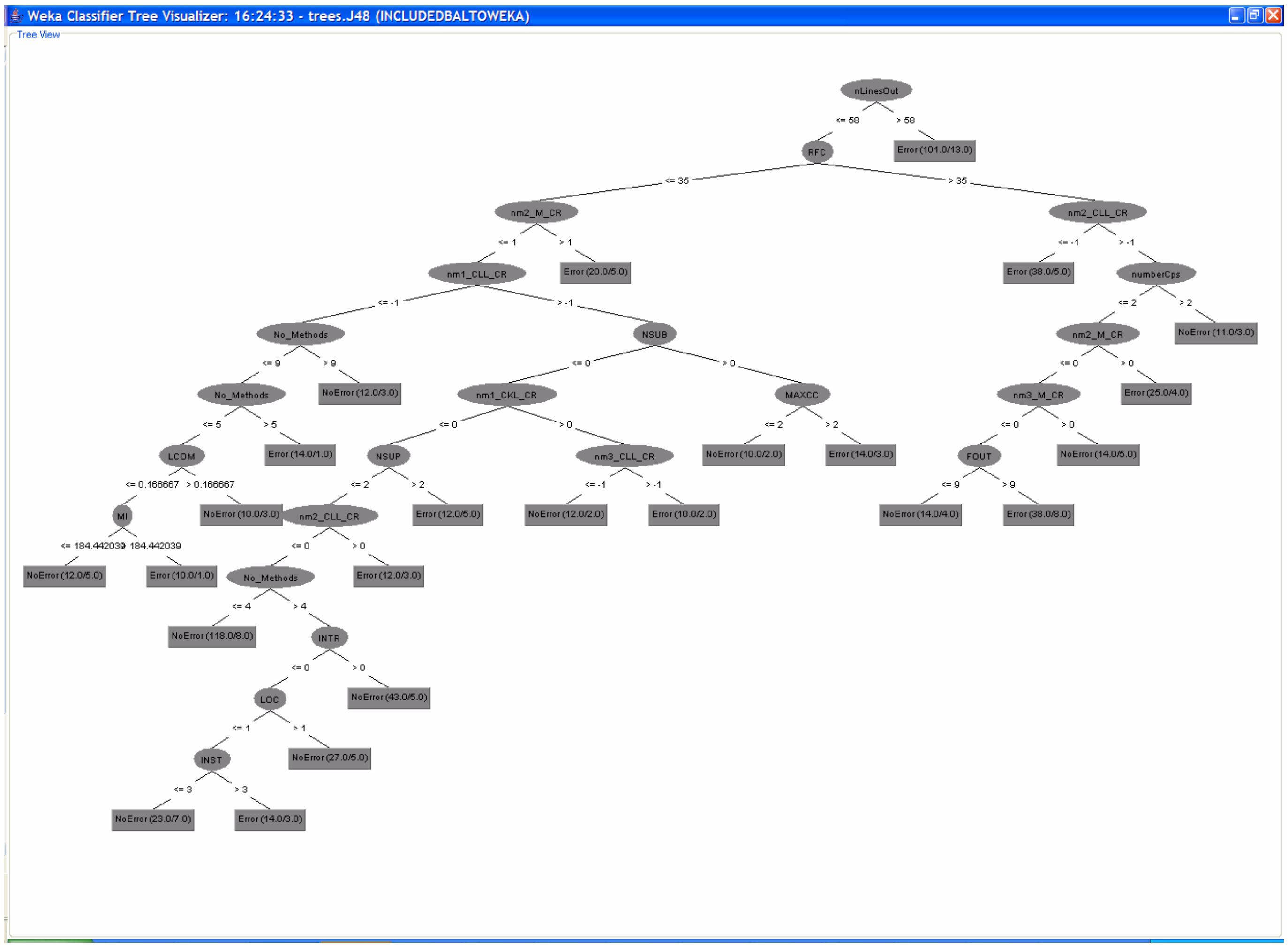

Figure 3 C4.5 Decision Tree

CFS only selected 29 variables out of 112 (Section 3.3). But what this variable selection tells us is that both properties of the class source code and change/fault history are useful and complementary predictors.

If we compare the variables selected with the ones in [1], a study which was discussed in Section 4, their Binomial regression model also used a code metric: file size in lines of code, and whether the file was changed in the previous release. Their variable coding the age of a file in terms of releases, is coded in our case across many variables (nm* measures in Table 1) capturing whether the class was new in one of the last three releases by assigning a -1 value to variables. The number of faults in the previous release is captured by the nm1_E_CR and nm1_CE_CR measures in Table 1, which separate critical from non critical faults. In addition, we also capture changes and fault corrections in the last three releases and distinguish requirements changes according to their size and complexity (as defined in Section 3.3). We also have many additional structural measures for cohesion, coupling, and many other attributes, some of them being selected in the predictive models as discussed above. Variables capturing the number of developers involved and their past experience on the COS system (numberPastCR measures) are also considered, but are not selected in the predictive C4.5 model we ended up choosing to focus verification. On the other hand, we do not have a programming language variable in our data set as all our predictions involve Java classes.

5.4 Fault Distributions in Fault-prone classes

In order to compare our results with the results of Ostrand, Weyuker, and Bell [1], let us look at the percentage of faults in the top 20% classes, a threshold that these authors indicated was the "knee of the curve" where the number of faults start to plateau. Though their unit of analysis was a file, we explained above that it should be comparable for Java classes and we therefore compare our results to theirs using their evaluation criterion.

We see that if we consider our C4.5 model, the 20% most fault prone classes account for 69% and 71% of faults for the Excluded and COS 20 test sets, respectively. This is significantly less than the 83% average reported in [1]. However, if we look at other modeling techniques, some of them such as the one combining PART and C4.5 reach 72% and 90%, respectively. But if we look at the percentage of lines of code, 20% of classes correspond to 59% of the code in their study. If we go back to the C4.5 CE analysis in Figure 2, we can see that around 59% NOS we capture around 90% of the faults. Our results in terms are therefore comparable to what Ostrand et al. obtained, but it shows that one must be careful about using size measures such as number of classes or files. The reason why we obtained a slightly larger percentage might be due to the additional variables we consider, but this is hard to ascertain.

6. CONCLUSIONS

This paper focused on using and comparing proven data mining and machine learning techniques to build fault-proneness models in a Java legacy system. Given the very large number of techniques that can potentially be considered, we had to strike a balance between minimizing the selection by considering existing empirical results and covering a wide diversity of techniques. More precisely the models we build predict the probability for a class to contain at least a fault based on a number of variables characterizing the class source code, amount of change in the last release, change and fault history over past releases, and developers' experience and number. In the short term, the intended application of such models in our context is to help focus class testing on high fault probability classes by investing extra testing resources. This in turn is expected to prevent many faults from slipping to subsequent testing phases and operation where fault correction is much more expensive.

The usual and general way to compare classifier models is to use criteria based on the confusion matrix; precision, recall, ROC area. Our results show that the modeling techniques do not show, in most cases, practically significant differences in terms of these criteria. However, we believe that using such general evaluation criteria can be misleading as they are not direct, surrogate measures of the cost-effectiveness of using such fault-prone models. The problem with cost-effectiveness models is that they tend to be context-specific. In our context, where extra testing is applied to a subset of classes in their decreasing order of predicted fault-proneness, we want to detect as many faults as possible while covering the least code possible with our extra testing. The underlying assumption is that the extra testing will be roughly proportional to the size of the code tested. Based on our proposed cost-effectiveness analysis procedure, we concluded that significant differences were indeed visible across modeling techniques, as opposed to what was concluded based on the confusion matrix. Furthermore, a relatively simple, easy to apply and interpret technique, namely the C4.5 decision trees, happen to perform very well overall (for different percentages of code and test sets), thus suggesting that more complex modeling techniques are not required. Such techniques include metalearners, feature (variable) selection techniques, and higher time complexity algorithms such as neural networks.

Though we use data coming from one large industrial project, the data was gathered across many releases during which significant organizational and personnel change took place. Furthermore, we do not believe that this project environment has any specificity that would somehow make it substantially different from other object-oriented, legacy Telecom systems with frequent releases and high personnel turnover. This is also supported by the fact that we obtain results that are similar to those of other case studies in the Telecom domain. This is encouraging as this suggests such prediction models could be applicable in a variety of environments.

From a more general standpoint, regardless of how it is defined in any specific context, we recommend to use a specific cost-effectiveness model in addition to standard confusion matrix criteria. The current status of this project is that our C4.5 decision tree model is currently applied as we write this paper in the testing of COS release 22. Early feedback from developers is very positive as they were surprised to uncover so many new faults by investing three extra days of unit testing on classes predicted as the most fault prone. Future work includes a thorough evaluation of cost-benefits based on the collection fault and correction data during this extra testing effort.

7. ACKNOWLEDGMENTS

Thanks to the members of the COS project team at Telenor who made significant contributions to this research. Thanks also to Valery Buzungu who assisted with the data extraction and to Hans Christian Benestad who helped us with the JHawk tool. The research was partly financed by the Research Council of Norway through the research project EVISOFT.

8. REFERENCES

[1] 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, no. 4, pp. 340-355, 2005.

[2] L. C. Briand and J. Wuest, "Empirical Studies of Quality Models in Object-Oriented Systems," *Advances in Computers*, vol. 59, pp. 97-166, 2002.

[3] J. S. Collofello and S. N. Woodfield, "Evaluating the effectiveness of reliability-assurance techniques," *Journal of Systems & Software*, vol. 9, no. 3, pp. 191-195, 1989.

[4] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, Second ed: Morgan Kaufman, 2005.

[5] R. J. Freund and W. J. Wilson, *Regression Analysis: statistical modeling of a response variable*: Academic Press, 1998.

[6] R. Quinlan, *C4.5: Programs for Machine Learning*: Morgan Kaufmann, 1993.

[7] P. Werbos, *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*: Wiley, 1994.

[8] V. N. Vapnik, *The Nature of Statistical Learning Theory*: Springer, 1995.

[9] T. Joachims, "Learning to Classify Text Using Support Vector Machines," 2002.

[10] 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, no. 1, pp. 68-74, 2002.

[11] Y. Freund and R. Schapire, "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting," *Proc. European Conference on Computational Learning Theory*, 1995.

[12] R. Melville and R. Mooney, "Creating Diversity in Ensembles using Artificial data," *Information Fusion*, vol. 6, no. 1, pp. 99-111, 2005.

[13] M. A. Hall and G. Holmes, "Benchmarking Attribute Selection Techniques for Discrete Class Data Mining," *IEEE transactions on knowledge and data engineering* vol. 15, no. 6, pp. 14-37, 2003.

[14] M. Hall, "Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning," *Proc. Seventeenth Int. Conf. on Machine Learning*, pp. 359-366, 2000.

[15] I. Kononenko, "On Biases in Estimating Multivalued Attributes," *Proc. fourteenth Int. Joint conf. on Artificial Intelligence*, pp. 495-502, 1995.

[16] JHawk, "http://www.virtualmachinery.com/jhawkprod.htm."

[17] A. Mockus and D. M. Weiss, "Predicting Risk of Software Changes," *Bell Labs Technical Journal*, pp. 169-180, April-June 2000.

[18] A. P. Nikora and J. C. Munson, "Developing fault predictors for evolving software systems," *Proc. Ninth International Software Metrics Symposium (METRICS'03)*, pp. 338-350, 2003.

[19] 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, no. 7, pp. 653-661, 2000.

[20] T. J. Yu, V. Y. Shen, and H. E. Dunsmore, "An analysis of several software defect models," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1261-1270, 1988.

[21] M. C. Ohlson, A. Amschler Andrews, and C. Wohlin, "Modelling fault-proneness statistically over a sequence of releases: a case study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, pp. 167-199, 2001.

[22] E. Arisholm and L. C. Briand, "Predicting Fault-prone Components in a Java Legacy System," *Proc. 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE)*, , Rio de Janeiro, Brazil, pp. 8-17, 2006.