An Empirical Evaluation of Fault-Proneness Models

Giovanni Denaro
Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32
I-20133 Milano (Italy)
denaro@elet.polimi.it

Mauro Pezzè
Università degli Studi di Milano - Bicocca
DISCo
Via Bicocca degli Arcimboldi, 8
I-20126 Milano (Italy)
pezze@disco.unimib.it

ABSTRACT

Planning and allocating resources for testing is difficult and it is usually done on empirical basis, often leading to unsatisfactory results. The possibility of early estimating the potential faultiness of software could be of great help for planning and executing testing activities. Most research concentrates on the study of different techniques for computing multivariate models and evaluating their statistical validity, but we still lack experimental data about the validity of such models across different software applications.

This paper reports an empirical study of the validity of multivariate models for predicting software fault-proneness across different applications. It shows that suitably selected multivariate models can predict fault-proneness of modules of different software packages.

Keywords

Software process, testing process, software metrics, software faultiness, fault-proneness models, logistic regression, cross-validation, principal component analysis.

1. INTRODUCTION

Software testing and analysis are complex and expensive activities. Planning and allocating resources for testing and analysis is difficult and it is usually done on empirical basis, often leading to unsatisfactory results. Delay in testing and delivery of products of poor quality are common experience in software production.

The possibility of early identifying the presence of faults in software modules could provide an important support for planning and executing testing activities. Researchers have been trying to identify metrics to measure software complexity since the early seventies [15]. Software complexity metrics should indicate the difficulties in designing software and thus the probability of erring. Unfortunately, none of the metrics defined so far can fully capture all aspects of software complexity and fault distribution. Since the early eighties, researchers recognized that sets of metrics can capture software complexity better than any metrics alone [1]. Unfortunately the great variety of software makes it difficult to identify a general model and a general set of metrics optimal for all kinds of software applications. In the nineties, researchers focused on specialized multivariate models, i.e., models based on sets of metrics selected for specific application areas and particular development environments [20, 3, 11, 17]. Specialized multivariate analysis elaborates historical data for identifying sets of metrics and relative models that are expected to be valid only within classes of software with given characteristics.

The definition of specialized multivariate models is articulated in three main phases: the choice of techniques for building models, the identification of methods for estimating the quality of prediction of the models, and the empirical evaluation of the effectiveness of the chosen techniques and methods [22]. Many techniques for building multivariate models have been studied: decision trees, neural networks, discriminant analysis, optimized set reduction, rough set analysis, and linear and logistic regression. These techniques produce models, whose validity may vary greatly. Several approaches for evaluating the quality of prediction of different models are available. For example, in the case of logistic regression, the validity of the models can be assessed by computing the statistical significance of coefficients and the goodness of fit. These values approximate the capability of the models to fit historical data. Other techniques, such as cross-validation and data splitting, can give information on the quality of prediction of the models.

The evaluation of the quality of prediction is important but not sufficient. In fact, the estimated quality of prediction cannot be considered valid in general. On the contrary, it is important to estimate the range of validity of a model, i.e., to identify the set of software packages for which the model is valid. Otherwise, we risk to use the wrong models thus obtaining poor results.

The hypothesis over the range of validity of a model can be validated with empirical studies. Such studies require the availability of detailed data about faultiness of groups of software applications that we expect to share similar fault-proneness models. Unfortunately enough data are seldom available and rarely accessible. This is why the many studies of techniques for building multivariate models and the variety of methods for computing the quality of prediction do not correspond to enough empirical studies [5].

This paper reports an empirical study of the validity of multivariate models built with logistic regression [7] and as-

ICSE'02, May 19-25, 2002, Orlando, Florida, USA.

sessed with cross-validation [22]. It thus contributes to the research in this area with important empirical data. In particular, this paper reports an experiment with the Apache Web server [16]. Multivariate models has been studied by first building models from data about known faults in the Apache Web server 1.3 (hereafter Apache 1.3), and then using the best models for estimating the faults of the Apache Web server 2.0 (hereafter Apache 2.0). The effectiveness of the models has been experimentally evaluated by comparing the modules of Apache 2.0 that are indicated as fault-prone with the modules of Apache 2.0 that contain actual faults. Apache 1.3 and Apache 2.0 are used for the same application area, have been developed with the same development process, and differ for more that 50% of the code. Thus, they well represent classes of homogeneous industrial applications where reuse of existing code can represent a fairly high percentage of the code.

The experiment described in the paper suggests that high quality multivariate models can predict the presence of faults in software applications that belong to the same class, i.e., that share the application domain, the development process, and the development teams.

Section 2 describes the setting of the experiment: it illustrates alternatives, motivates choices, and defines the requirements for the experiment. Section 3 details the software used in the experiment, defines the data needed for the experiment, and describes the procedure used for collecting the required data. Section 4 presents the experiment and discusses the obtained results. Section 5 overviews related work. Section 6 discusses the results of our research, indicates their limits, and outlines our current research on this topic. Appendix A gives the minimal technical information about logistic regression and cross-validation needed to understand the experiment presented in this paper. Appendix B lists the software metrics collected for the experiment. Appendix C presents the models built during the experiment.

2. EXPERIMENTAL SETTING

Many models of software complexity can give useful information. However, the data on presence of faults in software modules provided by the models proposed so far do not have general validity. That is, the same model can provide a good estimation of the presence of faults for a specific software package and a bad estimation for a different package. The lack of general models does not preclude the existence of valid fault-proneness models for specific classes of software applications, i.e., models that allow to estimate the probability of software to be faulty within classes of homogeneous software applications.

2.1 Goal of the Experiment

The general goal of our work is to investigate the existence of classes of software for which there exist useful fault-proneness models. Fault-proneness models are models that are built from information about the code and its faults, and that relate code to faults. The existence of such classes of software would allow to derive fault-proneness models from historical data, i.e., data available for old applications of a given class, and then use such models for predicting fault-proneness of new software applications of the same class. Such models could be useful during both planning and executing testing activities. Planning testing activities could take advantage of information about software fault-proneness for anticipating costs and allocating activities, while test execution can use this information to evaluate the quality of the results.

In our work we are interested in classes of software systems that can be identified as follows:

- they belong to the same application domain, i.e., they solve similar problems;
- they are developed with similar methods, techniques, processes, and implementation environments;
- they are developed by homogeneous teams, i.e., teams of people with similar technical background and working with similar teamwork procedures.

Such classes correspond to many industrially relevant cases, where software is developed for the same application domains by teams working with fairly stable development environments. Such classes are expected to share faultiness characteristics, in fact it is common practice to plan test and analysis activities referring to the experience in the development groups and the application domains. Common instances of such classes are families of products, i.e., software products developed over time by the same company. In this case, the products solve a similar if not the same problem; they are usually developed with similar methods, techniques, process, and development environment; they are built by people who although different are usually hired with similar criteria and requirements.

Aim of the experiment reported in this paper is to prove that there exist classes of products with these characteristics that share common fault-proneness models. Once identified a specific class of software, this result can be achieved starting from two software products that belong to the identified class, and for which we have detailed faultiness data, namely a significant set of known faults with information about their location. One of the products and the relative faultiness information is used to built a set of good fault-proneness models; the other product is used to validate the capability of the constructed fault-proneness models for predicting faults. Comparing the estimated fault-proneness of the second product with the actual fault distribution allows to evaluate the effectiveness of the constructed models. A single experiment cannot be generalized, but it can indicate that the hypothesis of existence of classes of software products that share common fault-proneness models, is valid indeed.

2.2 Target Application

We conducted our experiments on Apache 1.3 and Apache 2.0 ([16]) since, on one hand, they belong to one of the classes of software applications we are interested in, and on the other hand we have been able to access the information about code and faults, as required to perform the experiments. Apache 1.3 and Apache 2.0 well approximate all properties that characterize the identified classes of software:

- they belong to the same application domain, being two versions of the same software product.
- both are developed with the open-source development process described in Reference [16], that fixes development language, development rule, and synchronization

procedures. Developers are free to choose their preferred CASE tools, but most of them rely on similar free software.

- both are managed by the same core of people who work with a large number of volunteers following precise procedures [16].

The Apache Web servers are of size and complexity comparable with industrial applications, and Apache 1.3 and Apache 2.0 differ each other for more that 50% of the code, i.e., from the experiment viewpoint, they should not be considered a variation of the same application, but two different products developed by an homogeneous team[1]. The amount of code reuse in industrial software developed by the same company for the same application domain is often larger than 50% of the code. Thus Apache 1.3 and Apache 2.0 represent common industrial cases of reuse of software within applications of the same class.

In the case of Apache, the versions system repository records all changes, while the problem reporting database records detailed information about problem reports. The information required for our experiments can be calculated by comparing the versions system repository and the problem reporting database.

2.3 Fault-Proneness Models

Fault-proneness models can be built using many different methods that mainly belong to few main classes: machine learning principles, probabilistic approaches, statistical techniques, and mixed techniques. Machine learning techniques have been investigated by Porter and Selby, who studied the use of decision trees [21, 20], and by Khoshgoftaar, Lanning, and Pandya, who applied neural networks [11]. Probabilistic approaches have been exploited by Fenton and Neil, who propose the use of Bayesian Belief Networks [4]. Statistical techniques have been investigated by Khoshgoftaar, who applied discriminant analysis with Munson [18] and logistic regression with Allen, Halstead, Trio, and Flass [9]. Mixed techniques have been suggested by Briand, Basili, and Thomas, who applied optimized set reduction [3], and by Morasca and Ruhe, who worked by combining rough set analysis and logistic regression [17].

In the experiment described in this paper, we used logistic regression [7][2]. Most of the models give only a discrete approximation of potentially faulty models, i.e., they classify models as either fault-prone or not. Logistic regression produces a continuous indicator, i.e., it associates to each module a fault-proneness index that indicates the probability of the module to be faulty. Continuous indicators can be more useful than discrete indicators for planning test and analysis activities, because they can be adapted depending on the specific needs. For example, if we can increase testing effort only for a given percentage X of modules, continuous indicators allow to straightforwardly identify the X% of most fault-prone modules, while discrete models do not.

[1]Since the core of developers of Apache 1.3 and Apache 2.0 was actually almost the same, the development of Apache 2.0 may have benefited from the learning during the Apache 1.3 project. Our experiment does not take into account this factor.

[2]Readers not familiar with logistic regression can find an essential overview of the few key concepts required to understand this paper in Appendix A.

The constructed logistic regression models estimate the probability that software modules are highly-faulty or non-highly-faulty based on the value of given subsets of metrics computed on the modules. Given a set of software modules with data about faults and metrics, logistic regression can produce many models, with different prediction capabilities. We need criteria to select a subset of relevant models. A first index of quality of the models is given by statistics that measure the significance of the coefficients occurring in the models and the percentage of uncertainty that can be attributed to the models. These indexes indicate the quality of the models for the available data, but do not provide information about the quality of the models for predicting fault-proneness of new software products.

The quality of the models can be further evaluated by comparing the efficiency of the models on subsets of the available data by means of cross-validation (see Appendix A or [22] for details). Cross-validation measures the quality of models referring to the data available for the same software product. The technique uses all data for both computing the models and evaluating their quality, thus allowing for evaluating the quality of prediction even if we have only a limited amount of data.

In a preliminary experiment, we constructed fault-proneness models using the faultiness data available for an antenna configuration system; we selected the models with acceptable percentage of uncertainty and high significance of coefficients; and we chose the models with best quality. We then used the highest quality models to evaluate fault-proneness of Apache 1.3, and we compared the produced fault-proneness indicators with the known faults. The results are very bad: the computed fault-proneness does not give more information that a random index, despite the quality of the used models. This preliminary experiment demonstrates that the quality of prediction of models is not valid in general; However, we cannot deduce that models are not valid for any pair of software products.

To investigate if good quality models maintain a good ability of prediction when used within the same class of software systems, we proceeded as follows. We first used quality measures for selecting a subset of statistically relevant models based on data of a software product. We then assumed that such models can provide useful information about fault-proneness of other software products of the same class. In particular, we used the statistically relevant fault-proneness models built from the faultiness data of Apache 1.3 to estimate the faultiness of Apache 2.0. We finally compared the results with the actual faults that have been identified in Apache 2.0. This experiment allows to measure the quality of fault-proneness models across two applications of the same class.

3. DATA COLLECTION

The Apache Web servers are developed as open-source projects. Open-source projects are developed by a large number of volunteers, spread all over the world, coordinated by a small group of development managers who are responsible for the official releases. Developers and testers rarely if ever meet face to face; Communications take place mainly through the Internet, and most process data and activities are recorded in electronic form. The large amount of electronically available data includes the data needed for our experiments.

Data for the experiments were extracted mainly from two sources: the Apache CVS repository and the problem reporting database.

All files of the Apache project are stored in a CVS repository [12]. The repository tracks date and time of each change, the identifier of the developer responsible for the change, the number of lines of code added or deleted, and a description of the change. The content of a CVS repository can be logged, so that the whole record history of the contained files can be accessed. The revisions of the files in a CVS repository can be extracted. When the application is frozen in a release, all the file revisions included in the release are marked with a release tag. This allows us to extract all files of a given release, to know whether or not a given file has been modified between two releases, and to inspect the changes to the files between two releases.

Each time a new bug is signaled by some users of the Apache Web server, a new problem report (PR for short) is opened and recorded in the problem reporting database. The information about each PR stored in the database includes: a PR identifier, a description of the problem, the status (opened, suspended, analyzed, feedback, closed) of the bug, and the associated level of criticality (non-critical, serious, critical).

The file revisions in the Apache CVS repository that correspond to problem reportings are linked to the corresponding record in the problem reporting database. According to the Apache managers, this information has been tracked in at least the 90% of the cases [16]. We used such information for automatically discriminating between corrective and non-corrective changes. We considered as corrective changes the changes linked to a PR, and we approximate the number of faults known for a given file with the number of corrective changes recorded in the history record of the file.

To perform our experiments, we collected data from Apache 1.3 and Apache 2.0. For both programs we identified the implementation modules, i.e., the files containing executable statements. For each module we extracted from the project repositories the known faults, and we computed a set of software metrics.

We identified as implementation modules the files with a *dot c* extension contained in releases 1.3.10 and 2.0.19 of Apache 1.3 and Apache 2.0, respectively. We identified 128 modules in Apache 1.3 and 143 modules in Apache 2.0.

To identify known faults we compared two baselines: release 1.3.10 with release 1.3.0 for Apache 1.3, and release 2.0.19 with release 2.0.alpha for Apache 2.0, respectively. The number of known faults for each module was computed as the number of corrective changes recorded in the repository between the two baselines, considering for each module the oldest available revision. In this way we identified 443 faults for Apache 1.3 and 383 faults for Apache 2.0. The Apache development process does not prescribe formal testing sessions. Software verification is demanded to essential checks of developers and field reporting. Thus, for Apache 1.3 and Apache 2.0 we cannot distinguish between pre- and post-release faults. In our experiment, we considered all faults reported from the field as a realistic approximation of the faults present in the programs. The distribution of faults and the fault density in Apache 1.3 and Apache 2.0 are illustrated in Figure 1. The irregular distribution of faults and fault density for modules of increasing size indicates that size alone may not be a reliable predictor of faultiness.

We computed 38 different software metrics for both versions using RSM ([14]) and TestBed ([13]). The collected metrics are reported in Appendix B. The collected data are recorded in two datasets containing the implementation modules, the known faults and the metrics: dataset $D1$ for Apache 1.3 and dataset $D2$ for Apache 2.0.

4. THE EMPIRICAL STUDY

The performed experiment consists of three main steps: (1) construction of fault-proneness models using faultiness data from Apache 1.3, (2) selection of the most accurate models according to statistical measures, (3) evaluation of the effectiveness of the selected models to predict faultiness of Apache 2.0.

4.1 Building Models

In general, we are not interested in estimating the number of faults in each module, but rather in identifying highly faulty modules, i.e., modules responsible for a large amount of faults in the system. According to the 80:20 empirical rule, a small amount of code (often quantified as 20% of the code) is responsible for the majority of software faults (often quantified as 80% of the faults) [2]. To identify highly faulty modules, we selected the smallest set of modules responsible for 80% of the known faults in the system. In dataset $D1$, this corresponds to modules that contain more than 4 faults. According to this taxonomy, dataset $D1$ contains 29 highly faulty modules and 99 non-highly faulty modules.

Logistic regression produces models that correlate independent variables with discrete dependent ones. In our case, software metrics are the independent variables and the two-value variable (highly-faulty/non-highly-faulty) is the target dependent variable. We used two-class logistic regression on the dataset $D1$ for deriving models that correlate software metrics to the probability of modules to be highly-faulty. The computed models yield as result the probability of modules to be highly-faulty. We define this probability as fault-proneness of the software modules.

To produce a model with logistic regression, we need to indicate the metrics that we would like to use as explanatory variables. Different sets of metrics lead to distinct models with different quality of prediction. Thus, we run logistic regression using several sets of metrics as independent variables. We adopted two distinct approaches:

- we built models based on subsets of the available metrics. We call such models *direct models*;
- we built models based on derived variables, called *orthogonal variables*, computed on the space of the available metrics. We call such models *orthogonal models*.

In both cases, we were not able to identify a-priori subsets of variables that could perform better as independent variables. Thus, we built many logistic regression models, and we selected the optimal ones in a second stage.

Since models with a small set of independent variables better characterize the target system, we built all models with at most 5 independent variables out of the 38 available metrics. This yielded 577,734 direct models.

We then performed *principal component analysis* [8] on the 38 available software metrics to identify the orthogonal dimensions in the space of the independent variables. The

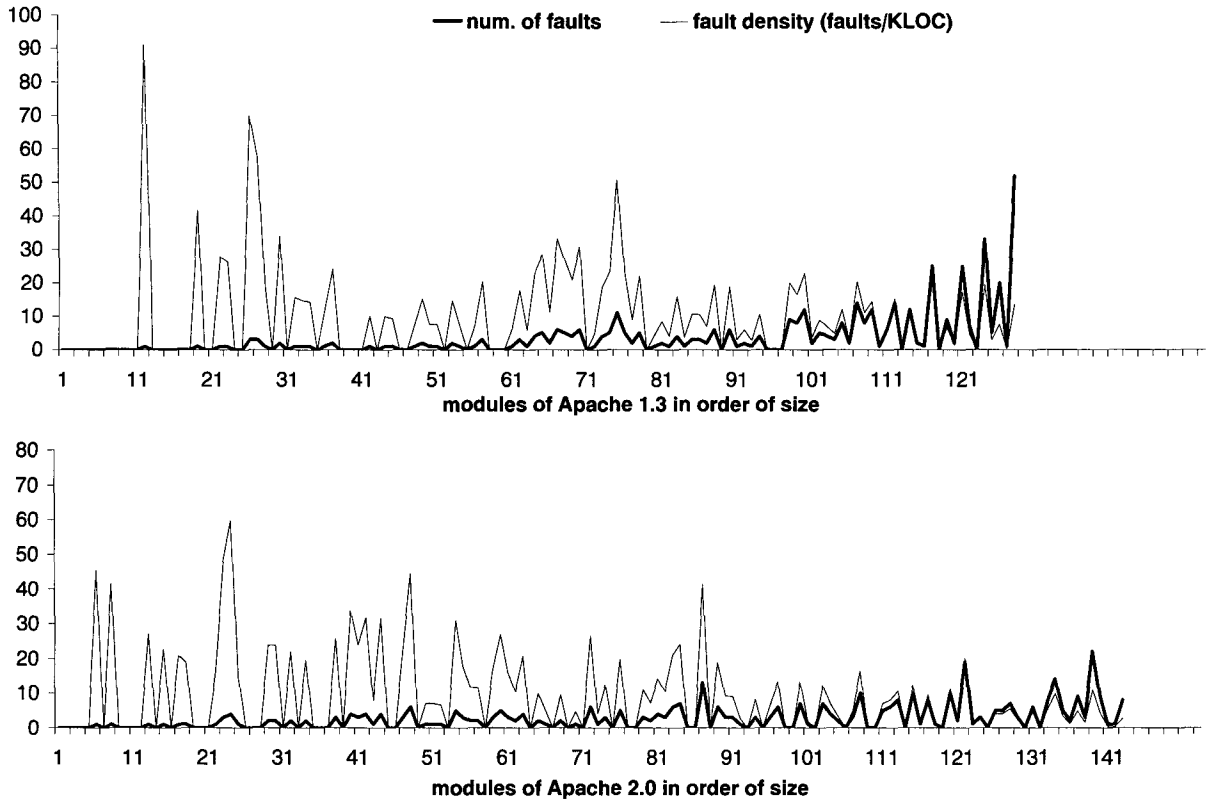

Figure 1: Distributions of faults and fault density in Apache 1.3 and Apache 2.0.

orthogonal variables, also called principal components, identified with this technique are linear combinations of the initial metrics. They are chosen such to explain the maximum possible amount of variance in the space of the independent variables. We computed models based on principal components to better understand the validity of models built directly on software metrics by comparing the efficiency of the two sets of models.

Principal component analysis produces a large number of principal components. Each principal component is characterized by a value that represents the weight of the component in determining the total data variance. We selected the nominal subset of principal components that accounts for at least 95% of the total data variance. The 9 selected principal components are shown in Table 1. We built the 511 orthogonal models using all possible subsets of such new components as explanatory variables.

Not all computed models have the same statistical significance. The statistical significance of logistic regression models can be assessed based on both indexes that measure the statistical significance of the explanatory variables, and the goodness of fit, i.e., the percentage of uncertainty that is attributed to the model. The statistical significance of the i-th explanatory variable is indicated with p_i, while the goodness of fit is indicated with R^2. Appendix A gives more details about p_i and R^2. For our experiments we used only models with acceptable statistical significance, i.e., with both all $p_i < 0.05$ and $R^2 > 0.3$. Only 2,271 models out of the 577,734 direct models and 29 models out of 511 orthogonal models survived.

4.2 Selecting Models

We cannot imagine a scenario where we use 2,300 models for computing fault-proneness of an industrial application, but we need to identify the champion, i.e., the model that we expect to perform best among the acceptable ones. Different criteria may lead to different champions. For our experiments, we selected four champions from each of the sets of direct and orthogonal models, using different criteria. In this way we have been able to evaluate the differences among these criteria.

Goodness of fit provides a first criterion to discriminate among the acceptable models. We thus selected the direct and the orthogonal model with best the goodness of fit (best R^2). Cross-validation indicates other criteria to select models. Cross-validation allows to evaluate the quality of prediction of a model using the same set of data used for building the model. Roughly speaking, we partition the data available for building models in n subsets; and we iteratively select one of the n subsets to validate the logistic regression models build using the data of the remaining $n-1$ subsets. A model is validated by selecting a threshold for classifying modules as faulty or non-faulty. In our experiment we fix this threshold to 0.5. The parameters to evaluate models with cross-validation are the number of modules predicted faulty, respectively non-faulty, versus the number of mod-

Principal component	Eigenvalue	Proportion of explained variance
PC1	25.063	65.96%
PC2	4.012	10.56%
PC3	1.998	5.26%
PC4	1.581	4.16%
PC5	1.111	2.92%
PC6	0.906	2.38%
PC7	0.723	1.90%
PC8	0.622	1.64%
PC9	0.565	1.49%
total variance explained		96.27%

The table reports on the results of the principal component analysis. Principal component analysis involves a mathematical procedure that transforms a number of (possibly) correlated variables into a (smaller) number of uncorrelated variables called principal components. Principal component analysis is performed on the covariance matrix of the initial set of variables and yields as results a matrix of eigenvectors and a diagonal matrix of eigenvalues. The eigenvectors represent the coefficients of the linear combination of the initial variables that yields each principal component, while the eigenvalues represent the importance of each principal component in explaining the variance in the set of initial variables. In principal component analysis the eigenvalues are extracted in decreasing order. The most important eigenvalues in the case of the analyzed set of software metrics are reported in the second column of the table and the correspondent percentage of explained variance is reported in the third column. The process of extracting the principal components is controlled by a stopping rule. A common stopping rule consists in extracting the most important principal component that cumulatively account at least for the 95% of the variance in the initial set of variables. In our case this led to 9 principal components that account for the 96.27% of the variance in the dataset of the available software metrics.

Table 1: The principal components selected for Apache 1.3.

ules actually faulty or non-faulty. Different ratios of this cross comparisons discriminate among models. For our experiment, we selected three additional direct and three additional orthogonal models, according to the following criteria:

best overall completeness. Overall completeness is defined as the proportion of modules that are classified correctly:

$$\text{overall completeness} = \frac{\text{CPFM} + \text{CPNFM}}{\text{M}}$$

where, $CPFM$ is the number of correctly predicted faulty modules, $CPNFM$ is the number of correctly predicted non-faulty modules and M is the total number of modules.

This parameter measures the ability of models to correctly classify the target software modules, regardless of the category in which the modules have been classified. We selected the direct and the orthogonal models that present best overall completeness in their category.

best faulty module completeness. Faulty module completeness is defined as the proportion of faulty modules that are correctly classified as faulty by the model:

$$\text{faulty module completeness} = \frac{\text{CPFM}}{\text{FM}}$$

where, $CPFM$ is the number of correctly predicted faulty modules and FM is the total number of actual faulty modules.

This parameter measures the ability of the models to identify the most fault-prone part of the target software. We selected the direct and the orthogonal models that present best faulty module completeness in their category.

best faulty module correctness. Faulty module correctness is defined as the proportion of faulty modules within the set of modules that are classified as faulty by the model.

$$\text{faulty module correctness} = \frac{\text{CPFM}}{\text{PFM}}$$

where, $CPFM$ is the number of correctly predicted faulty modules and PFM is the total number of predicted faulty modules.

This parameter measures the efficiency of a model, in terms of the percentage of actually faulty modules among the ones that are candidates for further verification. We selected the direct and the orthogonal models that present best faulty module correctness in their category.

We then selected a total of 4 direct plus 4 orthogonal models. Table 2 presents a summary of the fault-proneness models computed for Apache 1.3, using dataset $D1$. The first row indicates the number of direct and orthogonal models computed during the experiment. The second row indicates the number of statistically acceptable direct and orthogonal models among the computed ones (all $p_i < 0.5$, and $R^2 > 0.3$). The following rows indicate the values of the indexes used for selecting the champions for the different criteria: the best value for R^2, the highest percentage of correct guesses, the highest percentage of correctly predicted

	Direct models	Orthogonal models
Number of computed models	577,734	511
Number of valid models	2,271	29
Best R^2	0.5380	0.3802
Best overall completeness	90.6%	86.72%
Best faulty completeness	68.2%	50.0%
Best faulty correctness	85.7 %	77.78%

Table 2: Summary of fault-proneness models

faulty modules, the highest percentage of modules that were predicted faulty and are faulty indeed, respectively.

The indexes of the selected orthogonal models are lower than the indexes of the corresponding direct models. Moreover, direct models can be easily interpreted, while orthogonal models are hard to interpret. In fact, direct models refer to subsets of software metrics, and thus we have a direct indication of the factors that influence fault-proneness, while orthogonal models are combinations of software metrics, and thus need two stages of interpretation. The argument in favor of orthogonal models is time. Selecting the champion for orthogonal models requires the computation of only 511 models, i.e., less than 0.1% of the models computed for selecting the champion of direct models. The selected direct and orthogonal models are given in Appendix C.

4.3 Evaluating Models

The 8 selected models were applied on the dataset $D2$ in order to investigate their usability in the context of future projects. Apache 1.3 as been developed, used, and maintained between 1996 and 1999, while Apache 2.0 is being developed since 2000. We thus have complete information about faults in Apache 1.3, and incremental data about faults in Apache 2.0. The faultiness data of Apache 2.0 are still incomplete. Thus we attempted to evaluate if the selected models were able to predict the current defect trend, rather than the global faultiness. To this end, we proceeded as follows:

1. we estimated the fault-proneness (probability to be fault-prone) of the modules in the dataset $D2$ using the 8 selected models;
2. we ordered the modules in the dataset according to the estimated fault-proneness;
3. we computed the cumulative number of currently known faults for the modules according to the obtained orders for the 8 different models for different percentages of modules;
4. we compared the obtained values with the values obtained ordering the modules according to the number of known faults.

Table 3 presents a first summary of the results. We first ordered the modules of the system according to the fault-proneness given by each model. We then computed the percentage of modules in the different orders that accounts for the 25%, 50%, 75% and 90% of the known faults, respectively. The results are given in the columns of the table. Each row corresponds to a champion model. The last row gives the average values among the 8 models. For example, the first row says that, if we consider the direct model with best R^2 and we cumulatively consider the faults in the modules ordered by decreasing fault-proneness as predicted by the model, the first 12.6% of the modules are responsible for 25% of the known fault, the first 32.2% of the modules are responsible for 50% of the faults, the first 53.8% of the modules are responsible for 75% of the faults, and the first 69.9% of the modules are responsible for 90% of the faults.

Since faults are not equally distributed in software modules, the usefulness of a model is higher if the modules indicated as more fault-prone are responsible for a high percentage of faults. All values in the table are largely less than the average percentage of modules if selected randomly. Thus all selected models perform well. Moreover, we do not notice large differences among the models. Thus, in this case the mechanisms used for selecting the champion models do not impact on the results.

Figure 2 shows an Alberg diagram ([19]) that compares the performances of the fault-proneness models with the actual faultiness. In particular, it compares the cumulative percentage of faults in the modules ordered by:

- number of known faults,
- faultiness estimated with the best R^2 orthogonal model,
- faultiness estimated with the best faulty correctness direct model,
- average random selection.

We can see that the curves corresponding to the faultiness estimated with the two fault-proneness models are very close. The curves for the other six champion models do not differ significantly, and are not reported in the diagram only for the sake of readability. This confirms the results reported in Table 3. Moreover, the curves corresponding to the faultiness estimated with the two models tend to approximate the curve corresponding to the actual faults and are significantly different from the curve corresponding to random selection of modules. This confirms that the models provide useful results. In Figure 2, we can observe that currently the 20% of the most faulty modules are responsible for the 60% of actual faults[3], while the 20% of most fault-prone modules are responsible for the 40% of the actual faults. As expected, the information provided by the models is not completely accurate, nonetheless is useful. We can observe that the 50% of the modules that are indicated as most fault-prone by our models are responsible for the

[3]These data are coherent with the results of other studies that indicate numerical approximations of the Pareto hypothesis [5]

	Percentage of modules that account for the x% of the known faults			
	x=25	x=50	x=75	x=90
Direct model with best R^2	12.6%	32.2%	53.8%	69.9%
Direct model with best overall completeness	10.5%	28.7%	56.6%	74.8%
Direct model with best faulty completeness	11.2%	28.0%	51.0%	66.4%
Direct model with best faulty correctness	9.1%	25.9%	54.5%	69.9%
Orthogonal model with best R^2	13.3%	24.5%	47.6%	66.4%
Orthogonal model with best overall completeness	12.6%	28.0%	50.3%	72.0%
Orthogonal model with best faulty completeness	12.6%	25.9%	51.0%	68.5%
Orthogonal model with best faulty correctness	11.2%	25.2%	46.2%	69.2%
Average	11%	27%	51%	70%

Table 3: Effectiveness of the selected models in predicting faults on the Apache 2.0 project

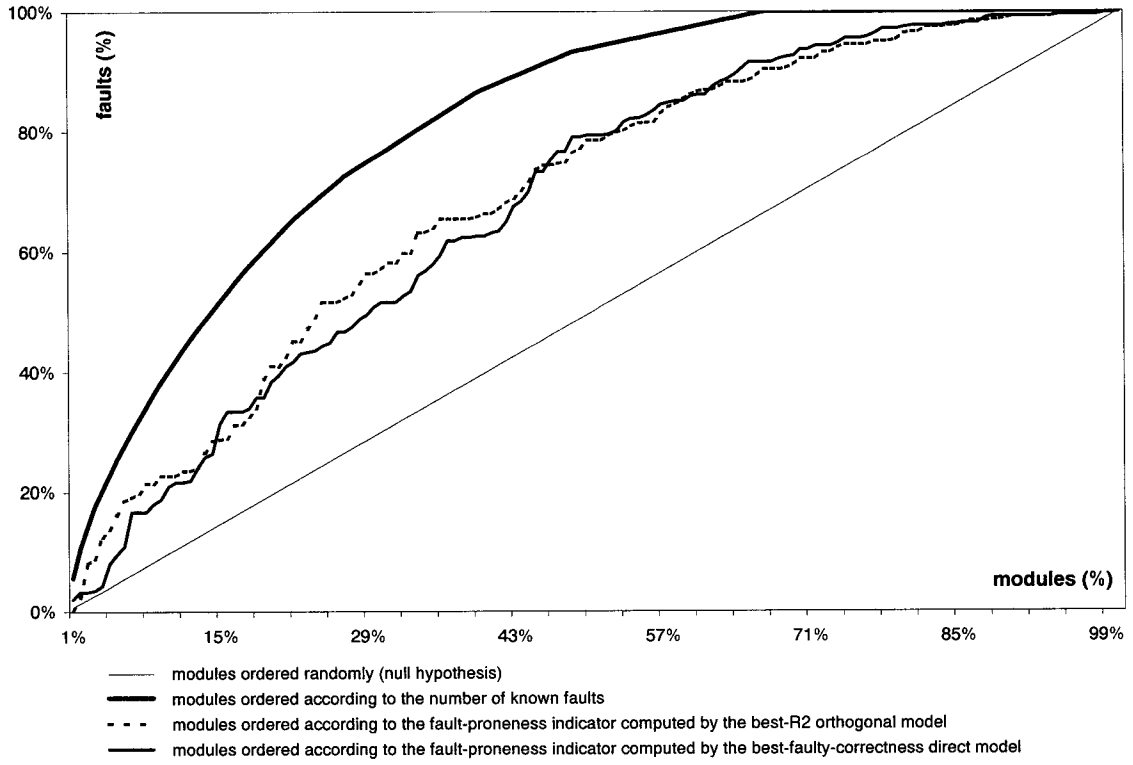

Figure 2: Alberg diagram: cumulative percentage of faults in the Apache 2.0 modules, considered in different orders.

80% of the actual faults. Thus, focusing testing on the 50% of the most fault-prone modules can significantly improve testing results.

4.4 Preliminary evaluation of the costs

Collecting metrics and applying models for estimating fault-proneness are inexpensive activities, both being linear in the size of the software and in the number of metrics. In our experiment we use commercially available tools that perform these activities in negligible time. The cost of the approach depends mainly on the construction and selection of the models. Such activity requires the construction and cross-validation of a number of models combinatorial in the number of available metrics. With the help of suitable tools, we have been able to construct and validate the models required in our experiment in a few hours of machine time of a Pentium II PC. Thus, the overall cost is much less than the cost of testing. Moreover, a careful selection of metrics and the use of principal components can significantly reduce the costs. In our experiment, we built the orthogonal models in few minutes of machine time.

5. RELATED WORKS

Fenton and Ohlsson have recently highlighted the scarce availability of public empirical data supporting common software engineering hypotheses [5]. In their work, they provide

empirical support for accepting or rejecting a number of hypotheses, including the *Pareto* principle, i.e., the hypothesis that a small number of modules contains most of the faults. They verify that single software size metrics are not good predictors for software faults. They also restate the hypothesis that "software systems produced in similar environments have broadly similar fault densities at similar testing and operational phases". They support this hypothesis by comparing total fault densities of two consecutive releases of an industrial project, but they recognize that the provided data are not sufficient to validate the hypothesis, and they call for additional experimental data. Our empirical study provides new data for supporting this hypothesis. The data provided in this paper properly extends the preliminary data presented by Fenton and Ohlsson, since we provide evidence that fault-proneness data computed on a software package are related to faultiness of a different software package with similar characteristics.

Khoshgoftaar and Lanning investigate the stability of principal components of software complexity across different software products [10]. They use principal component analysis for finding orthogonal dimensions in the space of software metrics computed on a a set of programs developed under different conditions. They provide evidence that principal components can vary across software products, but they can be stable across products developed by the same organization. Our research goes further: we provide empirical evidence that the stability of principal components of software complexity allows for using fault-proneness models across different packages of the same class of software products.

Ohlsson and Alberg analyze data from a project at Ericsson Telecom AB [19]. Their results support the hypothesis of validity of design and complexity measures for estimating faults. However such study does not report on the use of the derived models on new projects.

6. CONCLUSIONS

Being able to estimate software faultiness before and during testing and analysis activities would greatly help software testing and analysis. Unfortunately, many studies presented in the last years still lack experimental evaluation. In particular, we lack data on the validity of multivariate fault-proneness models within sets of similar projects. Such data are fundamental to understand the validity of the models and evaluate their effectiveness in competitive projects. Experiments in this area are difficult due to the large amount of detailed data required. Such data are seldom available for industrial scale projects and hardly accessible when present in industrial organizations.

This paper provides an experimental evaluation of multivariate fault-proneness models on an industrial scale project. It observes that open-source projects such as Apache keep record of the data required to evaluate fault-proneness models. It also argues that Apache 1.3 and Apache 2.0 represent a significant domain for experimenting with fault-proneness models.

The paper shows that multivariate fault-proneness models build with data from Apache 1.3 provide reliable faultiness estimations for Apache 2.0. It also provides data both to weight different models obtained with logistic regression and to compare distinct criteria to select prediction models among the many models that can be built with logistic regression.

These results represent a first confirmation of the hypothesis underlying the works on fault-proneness models: such models can predict software faultiness across homogeneous applications, i.e., applications that share application domain, development environment, and team characteristics. The results presented in this paper are a first important reference for the many studies on fault-proneness models; they indicate the setting for additional experiments that ought to be performed to provide the additional data needed to generalize the results; and introduce a new research topic in the field. In fact we now need to better identify classes of homogeneous programs within which fault-proneness models maintain their validity, by both refining the hypothesis of homogeneity and identifying limits.

7. ACKNOWLEDGMENTS

We would like to thank Sandro Morasca for the fruitful discussions and suggestions, Andrea Brambilla for contributing to the execution of the experiments, Roy Fielding and Apache.org for giving us access to the data required for our experiments.

8. REFERENCES

[1] B. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[2] B. Boehm and P. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, Oct. 1988.

[3] L. Briand, V. Basili, and W. Thomas. A pattern recognition approach for software engineering data analysis. *IEEE Transactions on Software Engineering*, 18(11):931–942, Nov. 1992.

[4] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, Sept./Oct. 1999.

[5] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, Aug. 2000.

[6] M. H. Halstead. *Elements of Software Science.* Elsevier North-Holland, New York, 1th edition, 1977.

[7] D. Hosmer and S. Lemeshow. *Applied Logistic Regression.* Wiley-Interscience, 1989.

[8] I. Jolliffe. Principal component analysis. In *Principal component analysis.* Springer Verlag, New York, 1986.

[9] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. M. Flass. Using process history to predict software quality. *Computer*, 31(4):66–72, Apr. 1998.

[10] T. M. Khoshgoftaar and D. L. Lanning. Are the principal components of software complexity data stable across software products? In *Proceedings of the Second International Software Metrics Symposium*, pages 61–72, 1994.

[11] T. M. Khoshgoftaar, D. L. Lanning, and A. S. Pandya. A comparative-study of pattern-recognition techniques for quality evaluation of telecommunications software. *IEEE Journal On Selected Areas In Communications*, 12(2):279–291, 1994.

[12] R. Krause. CVS: An introduction. *Linux Journal*, 87:72, 74, 76, July 2001.

[13] Liverpool Data Research Associates Ltd.,

http://www.ldra.com/. *LDRA Testbed Technical Manual, Revision 11*, June 1999.

[14] M Squared technologies, http://msquaredtechnologies.com/. *Resource Standard Metrics for C, C++ and Java, Technical Manual, Version 6.01*, 2001.

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

[16] A. Mokus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: The apache sever. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 263–272, 2000.

[17] S. Morasca and G. Ruhe. A hybrid approach to analyze empirical software engineering data and its application to predict module fault-proneness in maintenance. *The Journal of Systems and Software*, 53(3):225–237, Sept. 2000.

[18] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–33, May 1992.

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

[20] A. A. Porter and R. W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, Mar. 1990.

[21] R. W. Selby and A. A. Porter. Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering*, 14(12):1743–1757, Dec. 1988.

[22] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Tecniques with Java Implementations*. Morgan Kaufmann Publishers, 2000.

[23] M. R. Woodward, M. A. Hennell, and D. Hedley. A measure of control flow complexity in program text. *IEEE Transactions on Software Engineering*, 5(1):45–50, Jan. 1979.

APPENDIX

A. LOGISTIC REGRESSION AND CROSS-VALIDATION

This section briefly presents logistic regression and cross-validation, which we use for computing and evaluating fault-proneness models. Detailed presentations of logistic regression and cross-validation can be found in [7] and [22].

A logistic regression model estimates the probability that an object belongs to a specific class, based on the values of some numerically quantified attributes of the object. The variable describing the classes to be estimated is called dependent variable of the model. The variables that quantify the object attributes are called explanatory (or independent) variables of the model. For example in our experiments, we instantiate logistic regression as follows:

- the objects to be classified are software modules;
- the classes that classify modules are: highly-faulty and non-highly-faulty;
- the dependent variable of the models is a variable Y which assumes value 1 for highly-faulty modules and 0 for non-highly-faulty ones;
- the explanatory variables are software metrics, that quantify the attributes of the software modules.

A logistic regression model having Y as dependent variable estimates the probability of the event $[Y = 1]$ as defined by the following equations:

$$p(Y = 1) = \pi(Linear) = \frac{e^{Linear}}{1 + e^{Linear}}$$

$$Linear = C_0 + C_1X_1 + C_2X_2 + \cdots + C_nX_n$$

where, X_1, X_2, $\ldots$, X_n are n explanatory variables and the coefficients C_0, C_1, $\ldots$, C_n identify a linear combination ($Linear$) of them.

Logistic regression is based on maximum likelihood and does not assume any strict functional form to link the explanatory variables and the probability function. Instead, this functional correspondence has a flexible shape that can adjust itself to several different situations. The regression coefficient, C_1, $C_2, \ldots, C_n$, are estimated optimizing the likelihood function on the set of available observations (that have to be independent by hypothesis). In our experiments the observations are vectors that collect metrics and the value of Y for software modules. Observations are based on historical data. For each traced module, the value of Y is 1 if at least 5 faults are historically known and 0 otherwise. The metrics are directly measured by means of commercial and prototype tools. We rely on the approximation that the subject application is stable enough to have no faults, but the historically known ones.

A logistic regression model is preliminarily assessed based on the following statistics:

- p_i, the statistical significance of each estimated coefficient, which provides the probability that the corresponding coefficient C_i is different from zero by chance. In other words, p_i measures the probability that we erroneously believe that the metric X_i has an impact on the estimated probability that a module is highly-faulty. A significance threshold of 0.05 is commonly used to assess whether an explanatory variable is a significant predictor;
- R^2, the goodness of fit, which provides the percentage of uncertainty that is attributed to the model fit. This coefficient is not to be confused with least-square regression R^2: they are built upon very different formulae, even though they both range between 0 and 1. The higher R^2, the higher the effect of the explanatory variables, the more accurate the model. As opposed to the R^2 of least-square regression, high values of R^2 are rare for logistic regression, because of technical reasons. Values of R^2 greater than 0.3 can be considered good in the context of logistic regression.

A logistic regression model can be evaluated by applying the model on a new set of observations (test set) for which we know the actual value of the dependent variable. Once fixed a threshold, we can classify as highly-faulty (resp. non-highly-faulty) the modules with a fault-proneness value higher (resp. lower) than the threshold. The quality of prediction of the model is measured as the success rate in

classifying modules in the test set. This evaluation method, also known as data splitting, assumes the availability of additional observations to be used as test set.

Unlike data splitting, cross-validation assesses quality of prediction referring to the same data used to build the model, and thus produces statistically valid indicators of quality also for relatively small amount of data. Given a set of explanatory variables, cross-validation consists of the following steps:

- partition the observations into n approximately equal-size sets;
- compute n logistic regression models for the given set of explanatory variables. Each model is computed based on the observations belonging to $n-1$ sets of the partition, while the observations belonging to the remaining set are used as test-set to compute quality of prediction for the n models;
- compute the quality of prediction of the models referring to the selected set of explanatory variables by summing the values computed above.

Iterating the experiment for k different partitions allow for averaging the obtained results.

In the common practice, observations are partitioned in 10 sets and the partitions are stratified, i.e., in each partition the distribution of the objects belonging to each class is approximately the same as in the whole dataset (*stratified ten-fold cross-validation.*) Stratified ten-fold cross-validation has been used throughout this work for evaluating the quality of prediction provided by logistic regression models.

B. COLLECTED SOFTWARE METRICS

The metrics collected for the empirical study described in this paper are:

Size	size of the file, measured in bytes.
LOC	lines of code, excluding comment and blank lines.
eLOC	effective lines of code, excluding comments, blank lines, and stand-alone braces or parenthesis.
lLOC	logical lines of code, i.e., lines of code as identified by semi-colon.
Comments	comment lines.
dComm	comments in declarations.
eComm	comments in executable code.
Lines	all-inclusive count of lines of code.
CDENS	comment density, i.e., $Comments/eLOC$.
Functions	number of defined functions.
FP	number of formal parameters of functions.
FR	number of return points of functions.
IC	interface complexity, i.e., $FP + FR$.
Exits	procedure exits, i.e., number of distinct calls or exits made within procedures.
V(g)	cyclomatic complexity [15].
eV(g)	essential complexity [15].
Knots	number of knots [23].
eKnots	essential knots [23].
CO	number of comparison operators.
Blocks	number of basic blocks, i.e., sequences of statements with one entry point, one exit point and no internal branches.
AveBlockL	average length of basic blocks.
NEST	maximum number of nesting levels.

Halstead's software science [6]:

N1	total number of operators;
N2	total number of operands;
n1	number of unique operators;
n2	number of unique operands;
N	program length, i.e., $N1 + N2$;
n	program vocabulary, i.e., $n1 + n2$;
V	program volume, i.e., $N * \log 2n$;
D	difficulty, i.e., $(n1/n2) * (N2/n2)$;
E	effort, i.e., $D * V$.
Loops	total number of loops.
Int1	number of first order intervals.
MOI	maximum order of intervals.
LCSAJ	number of linear code sequence and jumps.
MaxLCSAJdensity	maximum number of LCSAJ passing through a code statement.
BRANCH	number of branching nodes.
EXEC	number of executable statements.

C. FAULT-PRONENESS MODELS

C.1 Direct Models

selection strategy	R^2	metrics	C_i	p_i
best R^2 (0.54)	0.54	Knots	-0.015	.0499
		N	0.054	.0031
		n	-0.009	.0101
		MOI	1.425	.0223
		EXEC	0.012	.0144
		Intercept	-7.64	<.0001
best overall completeness (90.6%)	0.49	FR	-0.055	.0415
		V(g)	0.027	.0384
		n2	0.069	.0026
		n	-0.016	.0113
		EXEC	0.011	.0316
		Intercept	-3.55	<.0001
best faulty module correctness (85.7%)	0.43	FR	-0.073	.0260
		CO	0.022	.0154
		N2	-0.003	.0413
		n2	0.029	.0119
		Intercept	-3.16	<.0001
best faulty module completeness (63.6%)	0.44	lLOC	0.016	.0030
		CO	-0.005	.0373
		MOI	1.248	.0145
		EXEC	-0.004	.0031
		Intercept	-6.44	<.0001

C.2 Orthogonal Models

selection strategy	R^2	pr.comp.	C_i	p_i
best R^2 (0.38)	0.38	PC1	0.0017	.0002
		PC4	-0.0064	.0005
		Intercept	-3.52	<.0001
best overall completeness (86.72%)	0.35	PC2	0.0008	<.0001
		Intercept	-3.50	<.0001
best faulty module correctness (70.00%)	0.31	PC8	-0.0006	<.0001
		Intercept	-3.19	<.0001
best faulty module completeness (40.91%)	0.38	PC5	-0.0060	<.0001
		PC6	-0.0041	<.0001
		Intercept	-3.45	<.0001