Deriving Models of Software Fault-Proneness

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

Sandro Morasca
Università degli Studi dell'Insubria
Dipartimento di Scienze Matematiche Fisiche e Naturali
Via Valleggio, 11
I-22100 Como (Italy)
Sandro.Morasca@uninsubria.it

Mauro Pezzè
Università degli Studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Via Bicocca degli Arciboldi, 8
I-20126 Milano (Italy)
pezze@disco.unimib.it

ABSTRACT

The effectiveness of the software testing process is a key issue for meeting the increasing demand of quality without augmenting the overall costs of software development. The estimation of software fault-proneness is important for assessing costs and quality and thus better planning and tuning the testing process. Unfortunately, no general techniques are available for estimating software fault-proneness and the distribution of faults to identify the correct level of test for the required quality. Although software complexity and testing thoroughness are intuitively related to the costs of quality assurance and the quality of the final product, single software metrics and coverage criteria provide limited help in planning the testing process and assuring the required quality.

By using logistic regression, this paper shows how models can be built that relate software measures and software fault-proneness for classes of homogeneous software products. It also proposes the use of cross-validation for selecting valid models even for small data sets.

The early results show that it is possible to build statistical models based on historical data for estimating fault-proneness of software modules before testing, and thus better planning and monitoring the testing activities.

Keywords

Software testing process, software metrics, software faultiness, fault-proneness models, logistic regression, cross-validation

1. INTRODUCTION

Demand for quality in software applications has undergone a rapid growth during the last few years, and testing related issues have become more and more crucial for software producers. An important step towards the improvement of the testing process is the ability of estimating software fault-proneness, i.e., estimating to what extent a software module is expected to be faulty before and after testing. A realistic estimation of software fault-proneness before testing allows software organizations to better focus the testing activities and can improve cost estimation. Estimation of software fault-proneness after a testing session can provide feedback on testing and help in defining delivery and maintenance procedures.

Software fault-proneness, i.e., the probability of presence of faults in the software, cannot be directly measured on software [4]. However, fault-proneness can be estimated based on directly measurable software attributes, if relations are found between these attributes and fault-proneness. Research in this field has been directed in two main directions: definition of metrics to capture software complexity and testing thoroughness, and identification and experimentation of models that relate software metrics to fault-proneness.

Several software metrics have been proposed to characterize software both statically and dynamically. Static metrics measure characteristics of the code structure. Well known static metrics are number of lines of code, cyclomatic complexity [17], number of operators and operands [8], number of knots [30]. Dynamic metrics measure testing thoroughness. Common dynamic metrics are based on structural and data-flow coverage [6]. The existence of correlation between software metrics and fault-proneness, as well as many other non-directly measurable software attributes, has been empirically proved by many authors (see, for instance, [7, 1, 10, 16, 12, 26, 15, 5]).

Unfortunately, the attempts to define models for computing fault-proneness indicators based on software metrics have not succeeded in producing convincing general models. Instead, encouraging results have been achieved by building prediction models based on sets of metrics tailored to specific application domains or processes. Rather than finding a general solution to the problem, this work aims at investigating how to find specific solutions based on the available domain knowledge. To this end, many methods have been explored, based on machine learning principles such as decision trees [27, 24] and neural networks [13], probabilistic approaches such as Bayesian Belief Networks [4], statistical techniques such as discriminant analysis [21] and regression [11], or mixed techniques such as optimized set reduction [2] and the combined use of rough set analysis and logistic regression [20].

Some of the proposed methods give only a discrete characterization of potentially faulty modules, i.e., classify modules as fault-prone or non-fault-prone, while others, and logistic regression in particular, produce a continuous fault-proneness indicator that allows modules to be ordered according to their fault-proneness. We argue that a continuous fault-proneness indicator can be more valuable for planning test and analysis activities, since it provides a better support for defining testing and analysis strategies. Previous work on logistic regression (e.g., [11, 20]) provides a preliminary set of data for evaluating the approach. This work selects valid models using methods that require a large set of experimental data. In particular, they use classic data splitting techniques that partition the available data into two sets used for building and evaluating the model, respectively. This method is sound and can be used to prove the validity of the approach. However, in many practical cases, partitioning the amount of available data may hinder the statistical significance of the results.

This paper uses logistic regression [9] for building models that correlate software metrics and module fault-proneness. In particular, our study discusses and evaluates a method that is not commonly used in logistic regression for computing the quality of prediction of the models; it discusses test strategies that can take advantage of the produced models for tuning the testing processes; and it provides additional empirical data to support the validity of logistic regression for this purpose. We carried out our study in an explorative fashion, comparing several automatically derived models. We propose cross-validation [29] as the technique for evaluating the quality of prediction provided by the models. Cross-validation allows the use of all available data for building the model, while evaluating its quality of prediction. Cross-validation can thus provide statistically relevant results even with a relatively small amount of data. We conducted our empirical study on an industrial-size application that has been used as benchmark in other scientific work [23, 5, 14]. In addition to predicting a module as either faulty or non-faulty, we can also rank modules according to their estimated probability of being faulty. We use Alberg diagrams [22] to show the advantages of tuning the testing process based on the produced models.

The paper is organized as follows. Section 2 gives the technical background necessary to understand the presentation. Section 3 defines the hypotheses and settings of our empirical study. Section 4 reports on the achieved results. Section 5 illustrates a possible use of the proposed fault-proneness models for improving the software process. Section 6 discusses similarities and differences with related work. Finally, Section 7 summarizes the main results presented in this paper and draws some conclusions.

2. 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 [9] and [29].

A logistic regression model estimates the probability that an object belongs to a specific class, based on the values of some 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. In our empirical study, we instantiate logistic regression as follows:

- the objects to be classified are software modules;
- the classes that classify modules are: faulty and non-faulty;
- the dependent variable of the models is a variable Y which assumes value 1 for faulty modules and 0 for non-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 $[Y = 1]$ with 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 does not assume any strict functional form to link the explanatory variables and the probability function. Instead, this functional correspondence has a flexible S-shape that can adjust to several different situations. The coefficients $C_0, C_1, C_2, \ldots, C_n$, are estimated by maximizing the likelihood function on the set of available observations, which are assumed to be independent by hypothesis. In our experiments, the observations are vectors that contain the metrics and the value of Y for software modules. For each traced module, the value of Y is 1 if at least one fault is known and 0 otherwise. The metrics were 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 except the known ones [23].

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

p_i , the statistical significance of coefficient C_i, which provides the probability of obtaining the estimated value of C_i or a more extreme one. 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 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. 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 a threshold is fixed, we can classify as faulty (resp. non-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 faulty and non-faulty modules in the test set. This evaluation method is known as *data splitting* because it assumes that some of the observations are 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 amounts of data. Given a set of explanatory variables, the cross-validation of the correspondent logistic regression model consists of the following steps:

- partition the observations into n approximately equal-size sets;
- compute n logistic regression models for the given 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;
- cumulate the results achieved for the n test sets in order to measure the success rate in classifying faulty and non-faulty modules of the whole initial data set;
- iterate a number of times the procedure using different partitions and average the obtained results.

In common practice, the 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 is used in this work for evaluating the quality of prediction provided by logistic regression models.

3. THE EMPIRICAL STUDY

We empirically evaluated the possibility of building statistically relevant fault-proneness models. The empirical study documented in this paper aims at:

1. Confirming the possibility of building suitable fault-proneness models. Although several papers propose fault-proneness models, there is still a limited amount of experimental data available in literature. The first, although not the main, goal of our empirical study is to provide additional data to increase the confidence in the existence of such models.
2. Validating the suitability of cross-validation for predicting the relevance of fault-proneness models built with logistic regression. We have already commented on the inadequacy of data splitting in presence of limited amount of data, as it often happens in many industrially relevant cases. Since the fault-proneness models that we can derive do not have general validity, but need to be recomputed for different software products and software environments, the possibility of building statistically relevant models starting from limited amount of data can be extremely important. The main goal of the empirical study documented in this paper is to assess cross-validation for evaluating the relevance of fault-proneness models.

Our empirical study was conducted on a program derived from an antenna configuration system developed by professional programmers. This software system, which contains over $10,000$ lines of code, provides a language-oriented user interface for configuration of antenna arrays. Users enter high level descriptions and the program computes the corresponding antenna orientations. The availability of detailed data on faults revealed during testing makes the program extremely useful for studies on fault-proneness. This program has been already used in experiments on testing ([23, 5, 14]) and is a good benchmark for experiments. The dataset contains 136 modules, of which 109 do not contain faults and 27 contain at least one fault. The term *modules* is used to refer to subprograms as in [5, 27, 24]. The total number of documented faults is 39.

For each module in the dataset we considered 33 different software metrics computed with commercial and prototype tools. The considered metrics are reported in Appendix A. In the data analysis, we built models that classify modules as either faulty or non-faulty, based on the values obtained for the static metrics.

We associated a variable Y with each module in the dataset. The value of Y is 1 if the module contains at least 1 fault, 0 otherwise. Considering Y as dependent variable, we ran logistic regression with different sets of software metrics as explanatory variables. In particular, we built models for all possible sets of metrics, with cardinality between 1 and 6, selectable among the 33 available software metrics. This yielded more than 1,300,000 models. Since models with too many explanatory variables lead to less credible increments in the model goodness of fit[1], the threshold of 6 metrics was a good compromise between the number of models to be computed and the potential significance of the models.

Once all the models were automatically computed, only the ones with good statistical significance of the explanatory variables (all p_i's< 0.05) and goodness of fit ($R^2 > 0.3$) have been considered for further analysis. For such models, we evaluated the quality of prediction by means of cross-validation. The fault-proneness threshold for classifying modules as faulty or non-faulty was fixed to 0.5 (50% probability). In particular the following three parameters have been considered to evaluate the success rate of a model:

Overall completeness:
the proportion of modules that have been classified correctly. This parameter measures the ability of a model to correctly classify the target software modules, regardless of the category in which the modules have been classified.

Faulty module completeness:
the proportion of faulty modules that have been correctly classified as faulty. This parameter measures how well a model identifies the most fault-prone part of the modeled software system.

Faulty module correctness:
the proportion of modules identified as faulty that were

[1]When the number of explanatory variables reaches the number of observations, the R^2 of a logistic regression model becomes 1 by construction

Number of expl. variables	Number of models	Best R^2
1	33	0.1735
2	528	0.2692
3	5,456	0.3241
4	40,920	0.4314
5	237,336	0.4884
6	1,107,568	0.5270

Table 1: Summary of the computed fault-proneness models

faulty indeed. 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.

The second and third of such parameters focus on evaluating the performance of models with respect to identification of faulty modules. As already noticed, faulty modules can be considered more relevant than non-faulty ones as far as verification issues are concerned and in software engineering in general ([20]). The early identification of faulty modules is likely to augment the efficacy of the testing process and to improve the overall quality of the delivered software that are the aims of this study.

4. RESULTS

Table 1 contains a summary of the fault-proneness models that have been automatically produced for the examined software. Column 2 indicates the number of models that contain a given number of explanatory variables (column 1). Columns 3 reports the best values of goodness of fit (R^2) for the models in each set. Good values of R^2 (> 0.3) are achieved for models based on more than two explanatory variables.

Since several models with good significance indices exist, a strategy is needed for selecting the most suitable ones, i.e., the candidates for being used on future projects. We investigated selection strategies based on goodness of fit, i.e., the highest R^2, and quality of prediction, i.e., either highest overall completeness, or highest module completeness, or highest faulty module correctness. During the selection, R^2 was used as discriminator whenever the applied strategy selected more than one model. The four selected models are given in Appendix B.

Figure 1 plots the parameters of the four selected models against the selection strategy that was applied. The figure compares the values of R^2, overall completeness, faulty module completeness, and faulty module correctness for the models selected by the four strategies.

The diagrams show that (1) none of the models selected with the different strategies outstand positively or negatively with respect to any other parameters, and (2) all four selected models present good values for all parameters. While high statistical significance (R^2) is obtained by construction, since we discriminated models according to R^2, the uniformity and quality of the other parameters demonstrate the validity of cross-validation for selecting models. In fact cross-validation produced models that present stable highly valid prediction indexes.

Trying to interpret the selected models, we notice that all four selected models refer to explanatory variables that are representative of the following dimensions of software complexity:

Size: the cumulative impact of size related metrics is positive in all models, confirming the intuitive idea that the likelihood of having a fault increases with the size of the software code.

Comments: all models indicate that the number of comment lines in a module influences the likelihood of having a fault. The higher the number of comment lines the lower the likelihood of having a fault. We can interpret this variable as an indicator of the pressure during development: low number of comments can denote high pressure on the programmers, and thus higher probability of introducing errors.

Interface complexity: all models indicate that highly complex interfaces (number of parameters and number of exit points) increase fault-proneness. This confirms the intuition that modules designed for complex interaction patterns, and thus with complex interfaces, tend to be more fault-prone.

Internal complexity: all models indicate that the complexity in the code structure, e.g., logic flow complexity or number of input and output nodes, positively impacts on the presence of faults.

The obtained results provide further evidence of the possibility of building suitable fault-proneness models using logistic regression and indicate the validity of cross-validation as a means for predicting the relevance of fault-proneness models.

5. FAULT-PRONENESS MODELS AND THE SOFTWARE PROCESS

Fault-proneness models have many possible uses within software processes. Here we would like to briefly illustrate an important use of models built with logistic regression, which not only discriminates between potentially faulty and non-faulty modules, as many other techniques, but provide also an index of faultiness.

The impact of the possibility of ordering software modules according to fault-proneness on the process is illustrated by the Alberg diagram of Figure 2 ([22]), which plots the cumulative percentage of faults of the modules of the software under test, ordered according to specific criteria. The solid line represents the cumulative percentage of faults for the program used in our empirical study if modules are ordered according to the known faults. Knowing this order would facilitate the definition of an optimal testing strategy. Unfortunately such order is usually known after testing. The dashed and thin lines represent the cumulative percentage of faults for the program used in our empirical study if modules are ordered according to the fault-proneness as provided by two of the selected models, namely the ones corresponding to the best R^2 and the best overall completeness, respectively. It can be noticed that the distance between the optimal order and the order provided by the two models is very small, and thus the models constructed using logistic regression may be successfully used instead of the optimal order to define effective testing strategies.

6. RELATED WORK

Several studies address the relationships between software

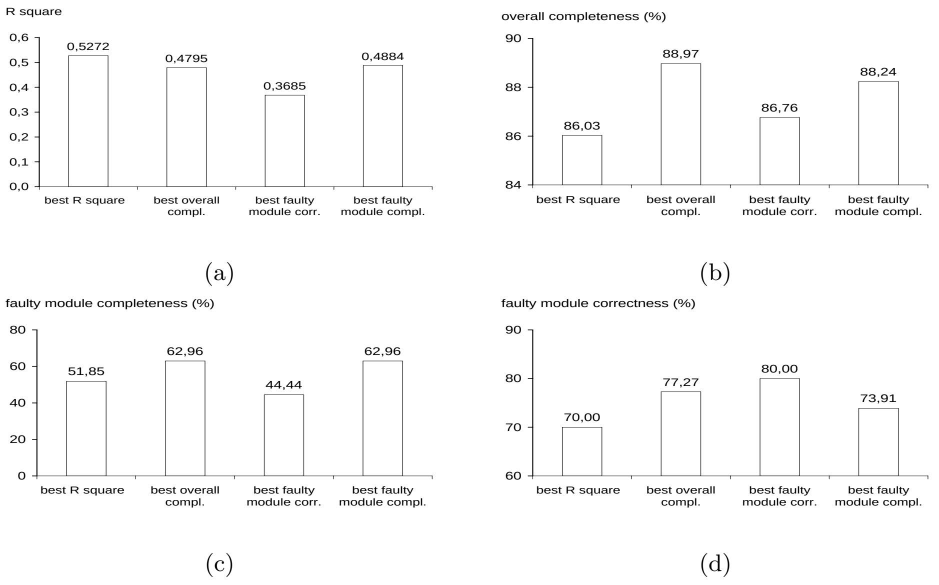

Figure 1: Parameters of the selected models. The four diagrams report the four selected models on the X axis and the value of one of the four parameters on the Y axis. In particular, diagram (a) reports the value of R^2, while diagrams (b), (c), and (d) report the overall completeness, the faulty module completeness, and the faulty module correctness.

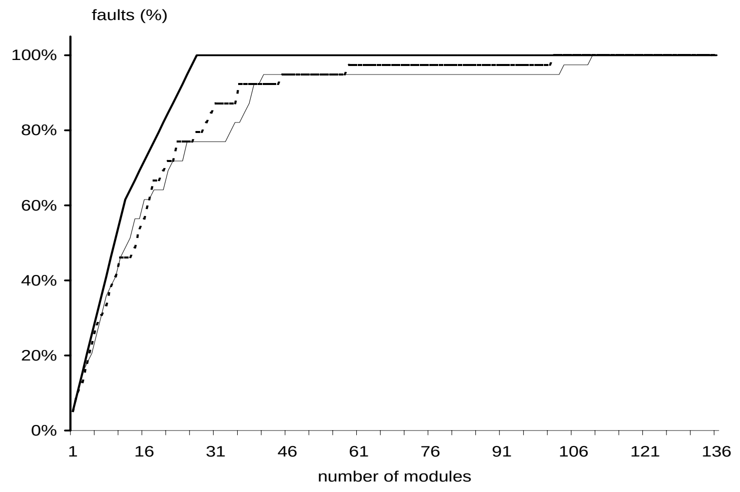

Figure 2: Alberg diagram. The *X* axis reports the number of modules of the software system, while the *Y* axis reports the correspondent cumulative number of faults. In the diagram, the solid line corresponds to the modules ordered according to the known faults; the dashed line corresponds to the modules ordered according to the fault-proneness indicator computed by the best-R^2 model; the thin line corresponds to the modules ordered according to the fault-proneness indicator computed by the best-overall-completeness model.

metrics and software fault-proneness. In the seventies, research tried to identify single metrics that could quantify software complexity ([17, 8]). From the late eighties, more research has tried to relate software complexity to sets of different metrics, deriving so called multivariate models.

Multivariate models can be built based on the analysis of available data of past projects. Besides logistic regression used in this paper, other methods have been explored.

Decision trees [25, 27, 24] are able to identify classes of objects based on a set of local decisions. Each node of a decision tree is associated with an explanatory variable. When a node is reached, the actual value of the associated explanatory variables identifies an output edge that leads to another node. During the classification, the tree is traversed, starting from the root node, until a leaf node is reached. Leaf nodes are associated with classification values. Decision trees can be automatically built from sets of historical observations and are particularly useful when the explanatory variables are discrete, although they can be adapted to handle continuous explanatory variable (in this case intervals are considered).

Neural networks [13] are sets of interconnected neurons, each having a number of inputs, an output, and a transformation function. Neural networks are commonly organized in a sequence of layers (at least three), and the neurons of each layer receive as inputs the weighted sum of the outputs of the previous layer. The first layer receives as inputs the explanatory variables and the last layer outputs the resulting classification value. Neural networks natively handle continuous explanatory variables.

Optimized set reduction [2] attempts to determine which subsets of observations from the historical data set provide the best characterizations of the objects to be assessed. Each of the *optimal* subsets is then characterized by a set of predicates (a pattern), which can be applied for classifying new objects. Optimized set reduction can handle either continuous or discrete explanatory variables and provides the expected value of the dependent variable.

All these methods provide simple classifications, i.e., an estimation of which modules are likely to be faulty and which not. Instead, logistic regression provide a fault-proneness index that can be used for ordering modules.

Logistic regression has been used in empirical software engineering for a number of goals, including estimation of software fault-proneness. It has been used mostly for correlational studies. Here we report on some studies that also used data splitting techniques to assess the models obtained. Morasca and Ruhe compare and combine logistic regression with rough set analysis, for identifying the most fault-prone part of a software systems [20]. The empirical study shows that logistic regression and rough set analysis have different strengths and weaknesses and that their combined use can improve the quality of the models. Khoshgoftaar et al. use logistic regression for identifying models for software fault-proneness [11]. The empirical study uses stepwise regression for finding logistic regression models. Stepwise regression is a model generation procedure based on adding new explanatory variables to the model while they significantly contribute to improve the model. Although the paper presents interesting results, stepwise regression may lead to suboptimal models [18]. Moreover, Khoshgoftaar et al. use data splitting to estimate the quality of prediction provided by the models, i.e, part of the data is excluded from the model generation procedure. Since data splitting derives models from a subset of available data, namely it excludes data used to validate the constructed models, it may provide less accurate results for limited amount of available data. Briand et al. [3] used logistic regression to correlate software design metrics with the fault-proneness of the final code.

In this paper, we use a combinatorial approach based on building all the models for fixed-size subsets of metrics, and we propose cross-validation to evaluate the constructed models. Cross-validation allows for evaluating the provided quality of prediction without excluding data from the model generation procedure, and thus applies better when limited amount of data are available.

There are very few work on models for residual fault proneness. Related research has been conducted by Voas et al., who introduced the concept of testability for anticipating testing effectiveness ([28, 19]).

For a comprehensive, critical survey of the literature, the reader is referred to [4].

7. CONCLUSIONS AND FUTURE WORK

This paper proposes logistic regression and cross-validation for deriving the correlation between software metrics and software fault-proneness. The paper fulfills several goals. It first provides further empirical evidence of the effectiveness of logistic regression for building models of fault-proneness. It proposes cross-validation to evaluate the produced models in order to obtain accurate results also in presence of a limited amount of data. It thus provides an approach that can be effectively applied to many industrial cases where the amount of available data is limited. It finally suggests that fault-proneness models can be merged into a software process.

The technique proposed in this paper requires some preliminary work on existing data for building the models of software fault-proneness and tuning the testing process. However, it results in a minimal overhead on the processes themselves, being easily implemented with simple elaborations of data that can be automatically collected with commercial tools. The proposed model selection strategy based on cross-validation enables for automating also the data analysis. The main requirement on the processes for the application of this technique is a mature testing documentation, i.e., detailed description of failures and faults, and their relation with the testing process.

We do not aim at providing a general model of fault-proneness (which might even not exists), but rather a methodology for building different models, for classes of homogeneous products, that can be of practical use if included in the testing processes of groups working on specific product lines. Continuous validation of the current models on the data that become available while progressing in the development is required to limit the divergence of the models from the development practice that may vary for many reasons including beneficial feedback from the models themselves.

We are now continuing our experiments on larger data sets aiming at further validating the methodology for building fault-proneness models and at producing models of residual fault-proneness with statistical relevance.

8. ACKNOWLEDGMENTS

We would like to thank Alex Orso, Adam Porter, Sergio Scarpati and Maurilio Zuccalà for their important contribution to the experiments. This research is partially supported by the European Commission within the Best Practice Project Promote.

9. REFERENCES

[1] V. Basili and D. Hutchens. An empirical study of a syntactic complexity family. *IEEE Transactions on Software Engineering*, 9(6):664–672, November 1983. Special Section on Software Metrics.

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

[3] L. Briand, S. Morasca, and V. Basili. Defining and validating measures for object-based high-level design. *IEEE Transactions on Software Engineering*, 25(5):722–743, September/October 1999.

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

[5] P. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. *ACM SIGSOFT Software Engineering Notes*, 23(6):153–162, November 1998. Proceedings of the ACM SIGSOFT Sixth Internatioal Symposium on the Foundations of Software Engineering.

[6] P. Frankl and E. Weyuker. Provable improvements on branch testing. *IEEE Transactions on Software Engineering*, 19(10):962–975, October 1993.

[7] G. Gill and C. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12):1284–1288, December 1991.

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

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

[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In Bruno Fadini, editor, *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, Sorrento, Italy, May 1994. IEEE Computer Society Press.

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

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

[13] T. Khoshgoftaar, D. Lanning, and A. 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.

[14] J. M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *Proceedings of the 22th International Conference on Software Engineering*, pages 126–135, Limerick, Ireland, June 2000.

[15] M. Lehman, D. Perry, and J. Ramil. Implications of evolution metrics on software maintenance. In T. Koshgoftaar and K. Bennett, editors, *Proceedings; International Conference on Software Maintenance*, pages 208–217. IEEE Computer Society Press, 1998.

[16] H. F. Li and W. K. Cheung. An empirical study of software metrics. *IEEE Transactions on Software Engineering*, SE-13(6):697–708, June 1987.

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

[18] P. McCullagh and J. A. Nelder. *Generalized Linear Models*. Chapman and Hall, London, second edition, 1989.

[19] K. Miller, L. Morell, R. Noonan, S. Park, D. Nicol, B. Murrill, and J. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Transactions on Software Engineering*, 18(1):33–43, January 1992.

[20] 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, September 2000.

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

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

[23] A. Pasquini, A. Crespo, and P. Matrella. Sensitivity of reliability-growth models to operational profile errors vs. testing accuracy [software testing]. *IEEE Transaction on Reliability*, 45(4):531–540, December 1996.

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

[25] J. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. QUINLAN86.

[26] R. Selby and V. Basili. Analysing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, 1991.

[27] R. Selby and A. Porter. Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering*, 14(12):1743–1757, December 1988. Special Issue on Artificial Intelligence in Software Applications.

[28] J. Voas and K. Miller. Semantic metrics for software testability. *Journal Of Systems and Software*, 20(3):207–216, 1993.

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

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

January 1979.

APPENDIX

A. COLLECTED METRICS

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

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.

Lines : all-inclusive count of lines of code.

FP : number of formal parameters of functions.

FR : number of return points of functions.

IC : interface complexity, i.e., $FP + FR$.

V(g) : cyclomatic complexity.

CO : number of comparison operators.

LFC : linear flow complexity, i.e., $CO + V(g)$.

OC : operational complexity, i.e., weighted sum of operations for all expressions occurring in a module.

V'(g) : enhanced cyclomatic complexity, i.e., $V(g)/OC$.

eV(g) : essential complexity.

NEST : maximum number of nesting levels.

Halstead's software science :

- **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., $D * V$.

CALLS : number of function calls.

BRANCH : number of branching nodes.

OAC : operation argument complexity, i.e., a weighted sum of the arguments in each operation.

NION : number of input/output nodes.

ANION : adjusted number of input/output nodes.

CONTROL : number of control statements.

EXEC : number of executable statements.

NSTAT : number of statements, i.e., $CONTROL+EXEC$.

CDENS : control density, i.e., $CONTROL/NSTAT$.

B. SELECTED MODELS

selection strategy	R^2	metrics	C_i	p_i
best R^2 (0.5272)	0.5272	eLOC	-0.82	<0.0001
		Comm	-0.44	0.0007
		Lines	0.48	<0.0001
		FP	0.44	0.0164
		LFC	-0.18	0.0141
		EXEC	0.33	0.0104
		Intercept	-4.63	<0.0001
best overall completeness (89%)	0.4795	eLOC	-0.36	0.0001
		Comm	-0.37	0.0007
		Lines	0.34	<0.0001
		IC	0.38	0.0052
		NION	-0.96	0.0001
		Intercept	-3.23	0.0010
best faulty module correctness (80%)	0.3685	eLOC	-0.09	0.0125
		Lines	0.09	0.0006
		FP	0.29	0.0167
		NION	-0.89	<0.0001
		Intercept	-1.22	0.0498
best faulty module completeness (63%)	0.4884	eLOC	-0.37	0.0001
		Comm	-0.39	0.0004
		Lines	0.36	<0.0001
		FR	-1.05	0.0001
		IC	0.43	<0.0030
		Intercept	-4.40	<0.0001