CS-TR-3606
UMIACS-TR-96-14
ISERN-96-03

January 1996

# Evaluating Predictive Quality Models Derived from Software Measures: Lessons Learned

*Filippo Lanubile*
Computer Science Department
University of Maryland
Institute for Advanced Computer Studies
College Park, Maryland 20742
lanubile@cs.umd.edu

*Giuseppe Visaggio*
Dipartimento di Informatica
University of Bari
Via Orabona 4, 70126 Bari, Italy
visaggio@seldi.uniba.it

## ABSTRACT[1]

This paper describes an empirical comparison of several modeling techniques for predicting the quality of software components early in the software life cycle. Using software product measures, we built models that classify components as high-risk, i.e., likely to contain faults, or low-risk, i.e., likely to be free of faults.

The modeling techniques evaluated in this study include principal component analysis, discriminant analysis, logistic regression, logical classification models, layered neural networks, and holographic networks. These techniques provide a good coverage of the main problem-solving paradigms: statistical analysis, machine learning, and neural networks.

Using the results of independent testing, we determined the absolute worth of the predictive models and compare their performance in terms of misclassification errors, achieved quality, and verification cost. Data came from 27 software systems, developed and tested during three years of project-intensive academic courses. A surprising result is that no model was able to effectively discriminate between components with faults and components without faults.

[1] This work was partially supported by NASA under grant 01-5-26775 and the Italian MURST under the 40% project "V&V in software engineering".

## 1. Introduction

The construction of predictive systems is one of the main purposes of software measurement. Predictive systems have been built from product metrics by applying different kinds of modeling techniques. Multiple linear regression analysis has been used to predict the number of corrective changes [13, 14]. Discriminant analysis has been applied to detect fault-prone modules [16, 19]. Logistic regression has been used for modeling to identify high-risk components [3, 4]. Principal component analysis has often been used to improve the accuracy of discriminant models [15, 19] or regression models [3, 4, 14]. Logical classification models have been used extensively to identify high-risk modules [3, 4, 20, 21, 27] and reusable software components [8]. Layered neural networks have already been applied to building reliability growth models [11, 12], to predicting the gross change [16], and the degree of reuse [2]. Holographic networks, a non-connectionist type of neural network, have been proposed for predicting software quality [18]. Empirical investigations have not yet been performed in the software engineering field but have in other areas such as financing [28] and manufacturing [10].

Many of the past studies have focused on predicting the presence of faults early in the software life cycle. Being able to know just after the coding phase, or even design, which parts are more subject to fail, allows software managers to focus their resources on inspecting or testing those error-prone components. The expected benefit is to achieve a more reliable product at a lower cost. However, all the studies have applied a very few candidate techniques (usually two or three). Furthermore, we cannot directly compare the results across the studies because of the lack of common evaluation criteria.

Theories and models become accepted by the scientific communities when different researchers obtain the same results running independent empirical studies. Thus, we began this study with the goal of externally replicating these past studies, and thus to understand which modeling technique, if any, is better in predicting the fault-proneness of software components. Our replication is characterized by the following features:

- Use of product measures as indirect metrics of software quality

Software product metrics are very popular as indirect metrics of quality. Most studies measure both design and code attributes but there is not a unique set of product metrics that all the studies use. Our indirect metrics of software quality are essentially the same used by Munson and Khoshgoftaar [19] to construct their predictive models.

- Reduction of the prediction problem to a classification problem.

A major problem in predicting software quality using the number of component faults as a direct metric is the highly skewed distribution of faults, because the majority of components have no faults or very few faults. Instead of estimating the number of potential faults in a software component, we determine whether a component is likely to be fault-prone or not. In this case, the direct metric of software quality is the class to which the software component belongs (high-risk or low-risk), and the prediction model is reduced to a classification model.

- Broader coverage of the modeling techniques already used in practice for classification.

Classification problems have traditionally been solved by various methods, which originate from different problem-solving paradigms: statistical analysis, machine learning, and neural networks. Statistical methods usually try to find an explicit numerical formula, which determines a classification completely. Machine learning methods try to deduce exact if-then-else rules that can be used in the classification process. The neural network paradigm, instead of producing formulas or rules, trains a neural network to reproduce a given set of correct classification examples. Our study compares the following modeling techniques which cover all three classification paradigms: discriminant analysis, logistic regression, logical classification models, layered neural networks, and holographic networks. Principal component analysis has also been included as an optional preprocessing step before applying discriminant analysis and logistic regression.

The next section characterizes the software environment and the data used in the empirical study. The third section describes how the modeling techniques where used to build the predictive models. The fourth section shows the criteria that we used to validate and compare the models. The fifth and sixth sections report, respectively, the results of testing our predictive models against the evaluation criteria, and the results from other similar studies. Finally, the last section summarizes the lessons we learned from this study.

## 2. Data Description

The data for this study was collected from projects performed by 27 teams of three students, during three years of a software engineering course at the University of Bari, Italy. Each team developed a business application, based on the same requirements specification, but independently designed and coded over a period of 4-10 months. The resulting software systems range in size from 1100 to 9400 lines of Pascal source code.

From each system, we randomly selected a group of 4-5 components, ranging in size from 60 to 530 lines of code, for a total of 118 components. Here, the term software component refers to a functional abstraction of code such as a procedure, function or main program. Each group of 4-5 components was tested by a different student team from another software engineering course. Faults found during testing were attributed to individual components.

The distribution of faults discovered during the independent unit testing, shown in Figure 1, was heavily skewed in favor of components with no faults or only one fault. To build unbiased classification models, we decided to have an approximately equal number of components in the classes of reliability. Thus, we defined as high-risk any software component where faults were detected during testing, and low-risk any component with no faults discovered.

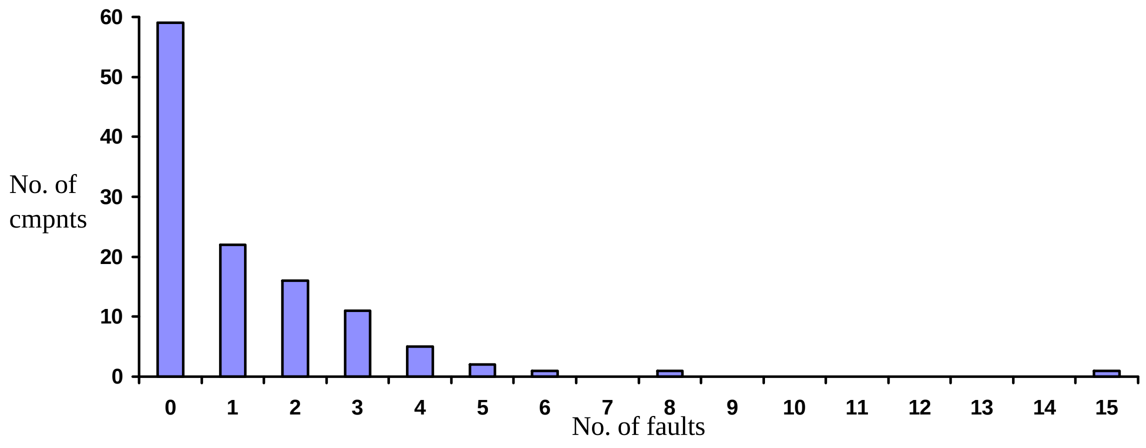

Figure 1. Distribution of faults per software component

Besides the fault data, 11 software product metrics were used as predictor variables to construct the predictive models. Table 1 shows the product metrics we used in this study. The metrics have been selected to measure both the implementation and the design attributes of the components, such as size, control flow structure, data structure, and coupling; one documentation metric is also measured.

| **Symbol** | **Name/Description** |
|---|---|
| *Size* | |
| LOC | Number of lines of code |
| NCLOC | Number of non-comment lines of code |
| N | Halstead program length, where N = N1+ N2 and N1is the total number of operators |
| V | Halstead volume, where $V = N * \log_2 n$ and n = n1 = n2 is the program vocabulary |
| *Control Flow Structure* | |
| VG | McCabe cyclomatic complexity, where VG = $e - n + 2$ for a flowchart with *e* edges and *n* nodes |
| *Data Structure* | |
| n2 | Halstead number of unique operands |
| N2 | Halstead total number of operands |
| *Coupling* | |
| fanin | Henry&Kafura fan-in, where the fan-in of a module M is the number of local flows that terminate at M, plus the number of data structures from which information is retrieved by M |
| fanout | Henry&Kafura fan-out, where the fan-out of a module M is the number of local flows that emanate from M, plus the number of data structures that are updated by M |
| IF | Henry&Kafura information flow, where IF = $(\text{fanin} * \text{fanout})^2$ |
| *Documentation* | |
| DC | Density of comments, where DC = CLOC / LOC and CLOC is the number of comment lines of program text |

Table 1. Predictor variables

## 3. Building the Predictive Models

For each of the 118 components, we had 11 product metrics and the risk class resulting from testing. We divided the data set into two groups. Two thirds of the components (79 observations) were randomly selected to create and tune the predictive models. The remaining third of the components (39 observations) provided the data to test the models. From now on, the first group of observations will be called the training set, and the second one the testing set.

There are many ways to build a predictive model using a given modeling technique. We describe our implementation choices to make possible the replication of the experiment in other environments as well as improvement in the application of the techniques.

### 3.1 Principal component analysis

Linear modeling applications, such as regression and discriminant analysis, can produce unstable models when the independent variables are strongly related. In this case, principal component analysis is applied to reduce the dimensions of the metric space and obtain a smaller number of orthogonal domain metrics [7].

In our study, the principal component analysis applied on the eleven product metrics produced three distinct complexity domains, having eigenvalues greater than 0.9. In Table 2, each column shows the degree of relationship between the eleven metrics and the three orthogonal domains. Values in bold print indicate which domain dominates a metric. Domain 1 includes the metrics measuring implementation attributes, domain 2 contains those metrics related to design attributes, and domain 3 consists of the only metric that was intended to capture the documentation characteristics. The three principal components account for 85 percent of the variability in the eleven metrics. For each software component of our data set, the values of the three domain metrics were derived and used as input to discriminant analysis and logistic regression.

| **Metric** | **Domain 1** | **Domain 2** | **Domain 3** |
|---|---:|---:|---:|
| $V$ | **0.98338** | 0.07621 | -0.04700 |
| $N$ | **0.98209** | 0.09208 | -0.04874 |
| LOC | **0.97486** | 0.07603 | -0.02989 |
| NCLOC | **0.97448** | 0.06976 | -0.05521 |
| $N_2$ | **0.95392** | 0.11957 | -0.06178 |
| $v(G)$ | **0.87488** | 0.19214 | -0.01642 |
| $\eta_2$ | **0.73870** | 0.01342 | -0.00998 |
| fanout | 0.16845 | **0.88696** | 0.01091 |
| IF | -0.01610 | **0.85161** | 0.02215 |
| fanin | 0.12539 | **0.82472** | -0.12569 |
| DC | -0.07395 | -0.06408 | **0.99215** |
| Eigenvalues | 6.306010 | 2.102089 | 0.980321 |

Table 2. Rotated factor pattern

## 3.2 Discriminant analysis

Discriminant analysis develops a discriminant function or classification criterion to place each observation into one of a set of mutually exclusive groups [7]. It requires that there exists a prior knowledge of the classes, in our case low-risk and high-risk components. To develop the classification criterion, we used a parametric method that uses a measure of generalized square distance and is based on a pooled covariance matrix. Using the generalized square distance, a posteriori probability of an observation belonging to one of the two groups is computed. An observation is classified into the group with the greatest probability of membership. We built two different discriminant models: the first one, applying discriminant analysis directly on the original eleven product metrics, and the second one, using as input the three domain metrics obtained from the principal component analysis.

## 3.3 Logistic regression

Logistic regression refers to an analysis that computes the probability of class membership according to the following equation [1]:

$$\log\left(\frac{p}{1-p}\right) = c_0 + \sum_{i=1}^{n} c_i * x_i$$

where $p$ is the probability that a software component is high-risk, and $x_i$ are the predictor variables. The regression coefficients $c_i$ are computed through a maximum-likelihood estimation.

As for the discriminant analysis, two regression models were built: the first one is based on the eleven product measures, while the second one uses the three domain metrics that have been generated from the principal component analysis.

## 3.4 Logical classification models

Logical classification models are classifiers that can be expressed as decision trees or sets of production rules. They are generated through a recursive algorithm that selects metrics that best discriminate between components within a target class and those outside it. To automatically build the classification model we used the C4.5 system [23], a variation on the ID3 system [22]. The C4.5 system partitions continuous attributes, in our case the indirect metrics of reliability, finding the best threshold among the set of training cases. The recursive partition method continues to subdivide the training set until each subset in the partition contains cases of a single class, or until no test offers any improvement. The decision tree was transformed into a collection of rules, by removing the conditions that were not helpful for discriminating between classes and by excluding rules that did not contribute to the accuracy of the set of rules as a whole.

## 3.5 Layered neural networks

We used a typical feed-forward neural network [24], characterized in our experiment by one input layer of eleven neurons, each connected to a product metric, one output layer of only one neuron that provides the predicted risk, and one layer of fifty hidden neurons. Among the supervised algorithms we chose the most popular one, the back-propagation algorithm, which adjusts network weights by iteration until a user-defined error tolerance is achieved or a maximum number of iterations has been completed. The network weights were initially set to random values between -1.0 and 1.0 using a sigmoid distribution. We trained the network with a value of 0.1 for the error tolerance, 1 for the learning rate, and 0.7 for the momentum rate. We stopped the training after 9000 iterations with 78 recognized observations over 79.

Since the network's input and output are bounded between 0 and 1, we reduced input data using a direct scaling. When testing the network, we increased the error tolerance to 0.5 so that low-risk components correspond to observations with an output value in the first half of [0, 1] and high-risk components to observations with an output value in the second half.

### 3.6 Holographic networks

With holographic networks, information is encoded inside holographic neurons rather than in the connection weights between neurons [29]. A holographic neuron holds a correlation matrix that enables memorizing stimulus-response associations. Individual associations are learned deterministically in one non-iterative transformation. Holographic neurons internally work with complex numbers in polar notation so that the magnitude (from 0.0 to 1.0) is interpreted as the confidence level of data, and the phase (from 0 to $2\pi$) serves as the actual data value.

In our study, input data were converted to the range $[0, 2\pi]$ using a sigmoid function and interpreted as phase orientation of complex values with a unity magnitude. On the other hand, the response was converted using a linear interpolation. These conversion methods provided the maximum symmetry in the distribution of data. We trained the network to obtain a maximum error of 0.1 for each observation.

## 4. Evaluation Criteria

To evaluate the predictive models we used a set of criteria that are based on the analysis of categorical data. In our study we have two variables, real risk and predicted risk, that can assume only two discrete values, low and high, in a nominal scale. Thus the data can be represented by a two-dimensional contingency table, shown in Table 3, with one row for each level of the variable real risk and one column for each level of the variable predicted risk. The intersections of rows and columns contain the frequency of observations ($n_{ij}$) corresponding to the combination of variables. Row totals ($n_{i\bullet}$) and column totals ($n_{\bullet j}$) correspond to the frequency of observations for each of the variables. In our context, the first row contains low-risk components, i.e., with no faults, while the second row contains high-risk components, including at least one fault. The first column contains components that the models classify as low-risk, while the second column contains components classified as high-risk.

The evaluation criteria are predictive validity, misclassification rate, achieved quality and verification cost. We use the criterion of predictive validity for assessment, since we determine the absolute worth of a predictive model by looking at its statistical significance. A model that does not meet the criterion of predictive validity should be rejected. The remaining criteria are used for comparison, taking into account that the choice between the accepted models depends from the perspective of the software engineering manager. In practice one might be more interested in achieving better quality even at a high verification cost, or be satisfied with lower quality, sparing verification effort.

| | Predicted Risk | | |
|---|---|---|---|
| Real Risk | *low* | *high* | |
| *low* | $n_{11}$ | $n_{12}$ | $n_{1\bullet}$ |
| *high* | $n_{21}$ | $n_{22}$ | $n_{2\bullet}$ |
| | $n_{\bullet 1}$ | $n_{\bullet 2}$ | $n$ |

Table 3. Two-dimensional contingency table

## 4.1 Predictive validity

Predictive validity is the capability of the model to predict the future component behavior from present and past behavior. The present and past behavior are represented by data in the training set while the future behavior of components is described by data in the testing set. Having data represented by a contingency table, we apply the predictive validity by testing the null hypothesis of no association between the row variable (real risk) and the column variable (predicted risk), i.e., the predictive model is not able to discriminate low-risk components from high-risk components. The alternative hypothesis is one of general association. A chi-square ($\chi^2$ ) statistic [6] with a distribution of one degree of freedom is applied to test the null hypothesis.

## 4.2 Misclassification rate

For our predictive models, which classify components as either low-risk or high-risk, two misclassification errors are possible. A Type 1 error is made when a high-risk component is classified as low-risk, while a Type 2 when a low-risk component is classified as high-risk. It is desirable to have both types of error small. However, since the two types of errors are not independent, software engineering managers should consider their different implications. As a result of a Type 1 error, an actual high-risk component could pass quality control. This would cause the release of a lower quality product and more fix effort when a failure happens. As a result of a Type 2 error, an actual low-risk component will receive more testing and inspection effort than needed.

In the contingency table, the number of Type 1 and Type 2 errors is given, respectively, by $n_{21}$ and $n_{12}$. We use the following measures of misclassification [26]:

- Proportion of Type 1: $P_1 = n_{21} / n$
- Proportion of Type 2: $P_2 = n_{12} / n$
- Proportion of Type 1 + Type 2: $P_{12} = (n_{21} + n_{12}) / n$

## 4.3 Quality achieved

We are interested in measuring how effective the predictive models are in terms of the quality achieved after the components classified as high-risk have undergone an extra verification activity. We suppose that the verification will be so exhaustive as to find all the faults in the components that are actually high-risk. So if all the high-risk components are properly classified, all defects will be removed by the extra verification, and perfect quality will be achieved. However, quality will be degraded with each high-risk component that is not identified.

We measure the criterion of achieved quality using the completeness measure [3] which is the percentage of faulty components that have been actually classified as such by the model.

- Completeness: $C = n_{22} / n_{2\bullet}$

## 4.4 Verification cost

Quality is achieved by increasing the cost of verification due to an extra effort in inspection and testing for the components that have been flagged as high-risk. We measure the verification cost by using two indicators. The former, inspection [26], measures the overall cost by considering the percentage of components that should be verified. The latter, wasted inspection, is the percentage of components that do not contain faults but have been verified because they have been incorrectly classified.

- Inspection: $I = n_{\bullet 2} / n$
- Wasted Inspection: $WI = n_{12} / n_{\bullet 2}$

## 5. Results

We applied the evaluation criteria on the testing set and analyzed the resulting data.

Table 4 shows the associations of the predictions and the real behavior of the components. The rightmost two columns show the chi-square values and the probabilities of incorrectly rejecting the null hypothesis, that is incorrectly saying that there is a significant association. The most popular probability value used as a threshold to establish significance is 0.05. If *p* is less than 0.05 there is a significant association and it is correct to reject the null hypothesis. Since all the probability values are much higher than 0.05 we must accept the null hypothesis of no association between predicted risk and real risk.

Table 5 shows the results of comparing the predictive models to each other with respect to the remaining criteria. All the data are represented as percentages. The first three columns of data show the misclassification rates. Recall that a random prediction should have a proportion of Type 1 + Type 2 errors of 50 percent, and proportions of Type 1 and Type 2 errors of 25 percent each. In this study the proportions of Type 1 + Type 2 errors ranges between 46 and 59 percent. Discriminant analysis and logistic regression, when applied in conjunction with principal component analysis, have high proportions of Type 2 error (respectively 41 and 46 percent) in comparison with the proportions of Type 1 error (respectively 15 and 13 percent). On the other hand, the other models have balanced values of Type 1 and Type 2 error, ranging between 20 and 28 percent.

| Modeling Techniques | $\chi^2$ | $p^*$ |
|---|---|---|
| Discriminant analysis | 0.244 | 0.621 |
| Principal components + Discriminant analysis | 0.685 | 0.408 |
| Logistic regression | 0.648 | 0.421 |
| Principal components + Logistic regression | 1.761 | 0.184 |
| Logical classification model | 0.215 | 0.643 |
| Layered neural network | 0.648 | 0.421 |
| Holographic network | 0.227 | 0.634 |

$p^*$ is the probability of incorrectly rejecting the null hypothesis (no association)

Table 4 Assessment of predictive models

| | Misclassification rate | | | Achvd quality | Verification cost | |
|---|---|---|---|---|---|---|
| Modeling Techniques | $P_1$ | $P_2$ | $P_{12}$ | $C$ | $I$ | $WI$ |
| Discriminant analysis | 28.21 | 25.64 | 53.85 | 42.11 | 46.15 | 55.56 |
| Principal comp. + Discriminant analysis | 15.38 | 41.03 | 56.41 | 68.42 | 74.36 | 55.17 |
| Logistic regression | 28.21 | 28.21 | 56.41 | 42.11 | 48.72 | 57.89 |
| Principal comp. + Logistic regression | 12.82 | 46.15 | 58.97 | 73.68 | 82.05 | 56.25 |
| Logical classification model | 25.64 | 20.51 | 46.15 | 47.37 | 43.59 | 47.06 |
| Layered neural network | 28.21 | 28.21 | 56.41 | 42.11 | 48.72 | 57.89 |
| Holographic network | 25.64 | 28.21 | 53.85 | 47.37 | 51.28 | 55.00 |

Table 5 Comparison of predictive models

Looking at the achieved quality and verification cost results, it is possible to better interpret the misclassification results. The highest values of quality correspond to the models built with principal component analysis followed by either discriminant analysis or logistic regression (completeness is, respectively, 68 and 74 percent). However, these high values of achieved quality are obtained by inspecting the great majority of components (inspection is, respectively, 74 and 82 percent), thus wasting more than one half of the verification effort (wasted inspection is, respectively, 55 and 56 percent). None of the other models discovers even half of the high-risk components and waste nearly half or more of the verification effort.

## 6. Related Work

Some empirical studies, relevant to this work, are summarized in the following.

Briand et al. [4] presented an experiment for predicting high-risk components using two logical classification models (Optimized Set Reduction and classification tree) and two logistic regression models (with and without principal components). Design and code metrics were collected from 146 components of a 260 KLOC system. OSR classifications were found to be the most complete (96 percent) and correct (92 percent), where correctness is the complement of our wasted inspection. The classification tree was more complete (82 percent) and correct (83 percent) than logistic regression models. The use of principal components improved the accuracy of logistic regression, from 67 to 71 percent completeness and from 77 to 80 percent correctness.

Porter [20] presented an application of classification trees to data collected from 1400 components of six FORTRAN projects in a NASA environment. For each component, 19 attributes were measured, capturing information spanning from design specifications to implementation. He measured the mean accuracy across all tree applications according to completeness (82 percent) and to the percentage of components whose target class membership is correctly identified (72 percent), that is the complement of the Proportion of Type 1 and Type 2 error.

Munson and Khoshgoftaar [19] detected faulty components by applying principal component analysis and discriminant analysis to discriminate between programs with less than five faults and programs having five or more faults. The data set included 327 program modules from two distinct Ada projects of a command and control communication system. They collected 14 metrics, including Halstead’s metrics together with other code metrics. Applying discriminant analysis with principal components resulted in correctly recognizing 79 percent of the modules with a total misclassification rate of 5 percent.

Khoshgoftaar et al. [15] again applied principal component analysis and discriminant analysis to identify fault-prone modules (modules with five or more faults) in a large telecommunications system. They used 1980 modules consisting of 194 new, 917 reused but modified, and 869 reused without modification. For product metrics, they used 3 call-graph-based metrics and 6 control-flow-graph-based metrics. They also used reuse information as additional categorical predictor variables. They classified 38.0 percent of the modules as fault prone when using product metrics only, and 31.4 percent when including the reuse variables too. The real percentage of faulty modules was 12.1 percent. The Proportion of Type 1 error (Type II misclassification rate in their study) was 21.25 percent with product metrics only, and 13.75 percent with also reuse variables. The Proportion of Type 2 error (Type I misclassification rate in their study) was, respectively, 32.4 percent and 23.8 percent. Finally, the Proportion of Type 1 and Type 2 error combined was 31.1 percent using only product metrics and 22.6 including the reuse variables.

## 7. Lessons Learned

This empirical investigation of the modeling techniques for identifying high-risk components has taught us three main lessons:

- Principal component analysis does not always produce a better input for predictive models.

In our study, we built two classification models for both discriminant analysis and logistic regression. The first pair of models was based on the eleven original product measures, while the second pair used the three domain metrics that had been generated from the principal component analysis. An unexpected result of the models using orthogonal domain metrics is that the good performance in achieved quality was exclusively the result of classifying many components to be high-risk.

- It is not always possible to successfully predict the future behavior of software products.

Despite the variegated selection of modeling techniques, no model satisfied the criterion of predictive validity, that is no model was able to discriminate between components with faults and components without faults. This result is in contrast with the software measurement literature which always reports successful results in recognizing fault-prone components from product measures. The previous section provides some examples.

- Predictive modeling techniques are only as good as the data they are based on.

The relationship between software product measures and the presence of faults cannot be considered an assumption that holds for any data set and project. An assumption is a statement that is postulated to be true without the need to be verified. Past positive findings at showing correlation between product measures and number of faults have built a confidence that this relationship is a general property. However, the underlying phenomena continue to be poorly understood and we do not really know what findings can be reused across environments and projects. Whereas the research underlying the validation of software product measures as internal attributes of software quality is not novel, it is only within the past few years that researchers have begun to worry about a rigorous and local validation [5, 9, 17, 25]. Predictive models are very attractive to build but they can be a waste of time if we rely on false assumptions instead of building a local process for selecting valid predictors.

## Acknowledgments

We would like to thank the students from the University of Bari for providing the fault data used in this study, Aurora Lonigro and Giulia Festino for their support in processing and analyzing the data, and Carolyn Seaman for her suggestions and comments on a first draft of this paper.

## References

[1] Agresti, A., Categorical Data Analysis, John Wiley & Sons, New York, 1990.

[2] Boetticher, G., Srinivas, K., and Eichmann, D., A neural net-based approach to software metrics, in *Proc. 5th Int. Conf. Software Eng. and Knowledge Eng.*, 271-274, 1993.

[3] Briand, L. C., Thomas, W. M., and Hetmanski, C. J., Modeling and managing risk early in software development, in *Proc. 15th Int. Conf. Sofware Eng.*, 55-65, 1993.

[4] Briand, L. C., Basili, V. R., and Hetmanski, C. J., Developing interpretable models with optimized set reduction for identifying high-risk software components, *IEEE Trans. Software Eng.*, 19 (11), 1028-1044, November (1993).

[5] Briand, L., El Eman, K., and Morasca, S., Theoretical and empirical validation of software product measures, ISERN-95-03, International Software Engineering Research Network, 1995.

[6] Conover, W. J., *Practical Nonparametric Statistics*, Wiley, New York, 1971.

[7] Dillon, W. R., and Goldstein, M., Multivariate Analysis: Methods and Applications, John Wiley & Sons, New York, 1984.

[8] Esteva, J. C., and Reynolds, R. G., Identifying reusable software components by induction, *Int. J. Software Eng. and Knowledge Eng.*, 1 (3), 271-292 (1991).

[9] Fenton, N. E., Software measurement: a necessary scientific basis, *IEEE Trans. Software Eng.*, 20 (3), 199-206, March (1994).

[10] Jensen, G., Quality control in manufacturing based on fuzzy classification, in *Frontier Decision Support Concepts* (V. L. Plantamura, B. Soucek, G. Visaggio, eds.), John Wiley & Sons, New York, 107-118, 1994.

[11] Karunanithi, N., Whitley, D., and Malaiya, Y. K., Prediction of software reliability using connectionists models, *IEEE Trans. Software Eng.*, 18 (7), 563-573, July (1992).

[12] Karunanithi, N., Whitley, D., and Malaiya, Y. K., Using neural networks in reliability prediction, *IEEE Software*, 53-59, July (1992).

[13] Khoshgoftaar, T. M., Munson, J. C., Bhattacharya, B. B., and Richardson G. D., Predictive modeling techniques of software quality from software measures, *IEEE Trans. Software Eng.*, 18 (11), 979-987, November (1992).

[14] Khoshgoftaar, T. M., Lanning, D. L., and Munson, J. C., A comparative study of predictive models for program changes during system testing and maintenance, in *Proc. Conf. Software Maintenance*, 72-79, 1993.

[15] Khoshgoftaar, T. M., Allen, E. B., Kalaichelvan, K. S., and Goel, N., Early quality prediction: a case study in telecommunications, *IEEE Software*, 65-71, January (1996).

[16] Khoshgoftaar, T. M., and Szabo, R. M., Improving code churn prediction during the system test and maintenance phases, in *Proc. of the Int. Conf. Software Maintenance*, 58-67, 1994.

[17] Kitchenham, B., Pfleeger, S. L., and Fenton, N., Towards a framework for software measurement validation, *IEEE Trans. Software Eng.*, 21 (12), 929-943, December (1995).

[18] Lanubile, F., and Visaggio, G., Quality evaluation on software reengineering based on fuzzy classification, in *Frontier Decision Support Concepts* (V. L. Plantamura, B. Soucek, G. Visaggio, eds.), John Wiley & Sons, New York, 119-134, 1994.

[19] Munson, J. C., and Khoshgoftaar, T. M., The detection of fault-prone programs, *IEEE Trans. Software Eng.*, 18 (5), 423-433, May (1992).

[20] Porter, A. A., Developing and analyzing classification rules for predicting faulty software components, in *Proc. 5th Int. Conf. Software Eng. and Knowledge Eng.*, 453-461, 1993.

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

[22] Quinlan, J. R., Induction of decision trees, *Machine Learning*, 1 (1), 81-106 (1986).

[23] Quinlan, J. R., *C4.5: Programs for Machine Learning*, Morgan Kauffman Publishers, San Mateo, CA, 1993.

[24] Rumelhart, D., Hinton, G., and Williams, R., Learning internal representations by error propagation, in *Parallel Distribuited Processing*, vol.I, MIT Press, Cambridge, MA, 318-362, 1986.

[25] Schneidewind, N. F., Methodology for validating software metrics, *IEEE Trans. Software Eng.*, 18 (5), 410-422, May (1992).

[26] Schneidewind, N. F., Validating metrics for ensuring Space Shuttle Flight software quality, *Computer*, 50-57, August (1994).

[27] Selby, R. W., and Porter, A. A., Learning from examples: generation and evaluation of decision trees for software resource analysis, *IEEE Trans. Software Eng.*, 14 (12), 1743-1757, December (1988).

[28] Soucek, B., Sutherland, J., and Visaggio, G., Holographic decision support system: credit scoring based on quality metrics, in *Frontier Decision Support Concepts* (V. L. Plantamura, B. Soucek, G. Visaggio, eds.), John Wiley & Sons, New York, 171-182, 1994.

[29] Sutherland, J., A holographic model of memory, learning and expression, *Int. J. Neural Syst.*, 1 (3), 259-267 (1990).