Machine learning method for software quality model building

Maurício Amaral de Almeida[1] and Stan Matwin
School of Information Technology and Engineering
University of Ottawa
150 Louis Pasteur, Ottawa
Ontario, K1N 6N5 Canada
{malmeida,stan}@csi.uottawa.ca

Abstract

Software quality prediction can be cast as a concept learning problem. In this paper, we discuss the full cycle of an application of Machine Learning to software quality prediction. As it often happens in real-life applications, significant part of the project was devoted to activities outside the learning process: data acquisition, feature engineering, labeling of the examples, etc. We believe that in projects that reach out to real data (rather than rely on the prepared data sets from the existing repositories), these activities often decide about the success or a failure of the project. The method proposed here is applied to a set of real-life COBOL programs and some discussion on the results is presented.

Keywords: Learning applications, software metrics, software quality, symbolic learning.

1. Introduction

Lack of adequate tools to evaluate and estimate software quality is one of the main challenges in Software Engineering. Historically, there have been numerous attempts to develop robust, analytical models based on the knowledge of factors influencing software quality [Kan95]. Many of these approaches adapt to software the general engineering reliability models (e.g. The Raleigh Model [Kan95]). The results, however, have not produced tools whose use has proven itself in practice. A different approach posited that rather than looking for analytical models, one should treat the problem as an empirical experiment, and try to predict quality based on past experience. The existing models (e.g. [Khoshgoftaar et al. 98]) typically use statistical approaches, e.g. regression techniques, on

[1] Faculdade de Tecnologia de São Paulo, São Paulo, Brazil.

data that represents the past experience of some organization concerning selected aspects of software quality. In our experience, statistical models have two drawbacks. Firstly, these methods behave like “black boxes”. The user enters the values of the model’s parameters, and obtains an answer quantifying some aspect of software quality. It is difficult for the user to see the connection between the inputs of the model and the quantitative answers it provides: interpretability of the model is questionable. Secondly, the empirical data available about both the organization, and the software, may be drastically different from the kind of organization and the kind of software whose quality is to be predicted. Consequently, these methods are often criticized for being based on assumptions that do not apply to the project at hand, and for the difficulties with the interpretation of their estimates.

We propose an approach that derives simple estimates of selected aspects of software quality. The method that we have developed builds a quality model based on the organization’s own past experience with similar type of software projects. The underlying model-building technique is concept induction from examples. It inherits from the specific induction technique the immediate interpretability of the results. We show how a problem of software quality prediction can be cast as a concept learning problem. This allows us to draw on a rich body of techniques for learning concepts from examples. Low maintainability, for instance, becomes a concept to be learned from positive and negative examples. These are software components whose maintenance effort can be labeled as low or high.

In this paper, we discuss the full cycle of an application of Machine Learning. As it often happens in real-life applications, significant part of the project was devoted to activities outside the learning process: data acquisition, feature engineering, labeling of the examples, etc. We believe that in projects that reach out to real data (rather than rely on the prepared data sets from the existing repositories), these activities often decide about the success or a failure of the project. Consequently, it is important to present and discuss them, with the goal of producing generally recognized methodologies for data mining activities that precede and follow the learning step.

2. Prior and related work

There is a small, but growing body of work in which ML is applied in the software quality prediction task. Selby and Porter [Porter90] have used ID3 to identify the attributes that are the best predictor of interface errors likely to be encountered during maintenance. [Briand93] built cost models for isolation and correction using an approach that combines statistical and machine learning classification methods [Briand92]. In this work, values of metrics were used for the first time to represent the properties of the code believed relevant to the task. The metrics were extracted by the ASAP tool form a set of ADA components from the NASA SEL. Jorgensen [Jorgensen95] has constructed a predictive model of the cost of corrective maintenance using the approach of [Briand92] for generating a logical classification model, and has compared the symbolic ML approach to neural networks and lest square regression. Basili [Basili97] has built maintenance cost using the C5.4 and metrics extracted by the AMADEUS tool. Cohen [Cohen97] presents a comparison

between the FLIPPER and the FOIL using dada from class declarations in C++.

Almeida, Lounis and Melo [Almeida98] have continued the work by [Basili97] comparing the models built with C4.5 with models built with NewID, CN2 and FOIL. The metrics was extracted with the ASAP tool from a set of ADA components from the NASA SEL.

3. Methodology

We view software quality as a multi-dimensional concept, consisting of such properties of the software as modifiability, changeability, error proneness, etc. The approach we are proposing is to treat each quality dimension as an *unknown* function from the observable attributes of the software to a value in a selected quality dimension. This function clearly depends on the characteristics of the software organization that develops and maintains the software. We treat these attributes, such as

- development tools for software and hardware;
- Software project methodologies;
- Software life cycle;
- Specific characteristics of the software development team

as non-observable: not only are they often unavailable or uncertain, but their precise influence on software quality is difficult to quantify. We will refer to the collection of non-observables as context. We rely on experience, which is represented as a certain number of quality values, known from past experience, and the observables that resulted in these values. We are then predicting the unknown values of the quality function, using ML techniques.

In this manner, we do not need to work with the context. We assume that it implicitly influences the observables, and that this influence is captured by the induction process. Consequently, ML-generated software quality models are adaptable to a given context that influences the quality.

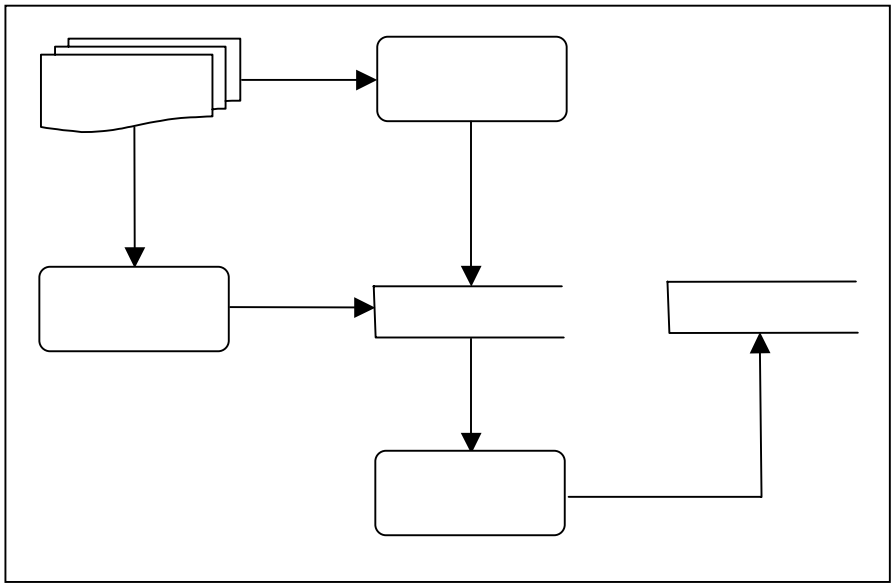


Figure 1. Quality model construction process.

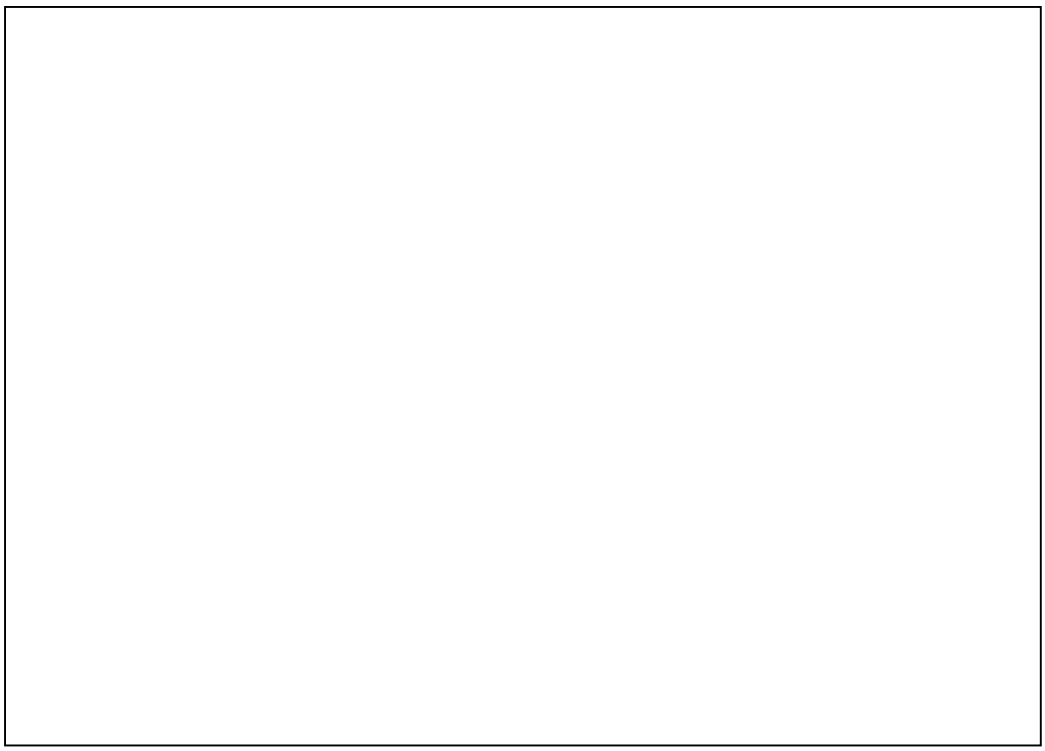
Fig. 1 presents the whole process. The starting point is the source code, divided into units (modules, procedures, files, whatever are the units in a given programming language). The experience with the quality of that software is represented by the quality labels, known from the past (e.g. how long it took to perform a given maintenance request on a particular unit). Units of source code are mapped into vector of software metrics values. These vectors, with their corresponding quality labels, are fed into an inductive learner, which treats the prediction problem as a classification problem: quality is discretized into a small number of discrete values of labels. A ML system produces a model (e.g. a decision tree, or a set of rules).

3.1. Case selection.

When building dedicated quality models one should pay special attention to the definition of the specific conditions under which the model is being built. Fig. 2 presents a partial list of aspects to be considered in the selection of the cases for the model building.

The first aspect to be considered is the attribute (the dimension of quality) to be modeled. Enough reliable historical information about that attribute is necessary for model induction.

The context should be well defined, and all cases must belong to the same context to assure the comprehensibility, reliability and applicability of the model. The context characteristics to be considered include the software and hardware used, the team, and the project. The software component aspects to be considered include the granularity, the accessibility, and the quantity of the components.



3.2. Metrics selection and extraction

After the cases selection the metrics to be used should be chosen. In many cases the metrics have already being extracted by the software organization, and we do not have even access to the source code of the components.

The metrics to be chosen depends on the programming languages, the available extraction tools, the attribute to be modeled, and others. Usually classical metrics [Fenton96], [Kan95] are included in the metrics set, such as:

- Size metrics (KLOC, Function points, ...)
- Code documentation metrics (comment lines, blank lines, ...)
- Halstead metrics [Halstead77]
- Cyclomatic complexity [McCabe76]
- Specific syntactic structures (IF-THEN-ELSE, WHILE, DO-WHILE, FOR, ...)
- Structural metrics (Fan-in, Fan-out)
- Object oriented metrics (methods/class, inheritance tree depth, coupling, ...)
- Data structure metrics (Number of operands [Halstead77], DATA [Boehm81]).

In our experiment, we have used 19 metrics, based on the metrics from the list above.

3.3. Labeling

Due to a number of factors, i.e. the granularity and imprecision of the available information, difficulty to automatically retrieve the available information, and incoherent data, in our experience the labeling is the hardest phase of the quality model building. For instance, it is quite common that the historical data are presented in a different granularity then the one used in the metrics extraction.

The source code, extraction methods, and evaluation methods for the labeling process must be clearly established. Finally the classes and a mapping scheme must be defined. Frequently the classes are defined as “high” and “low” for many of the most common dimensions of software quality modeling like maintainability, faut-proness, changeabilirty, and others.

In our experiment, there was a need to attribute the cost to units of smaller granularity than the units for which cost data was available. We have achieved this by attributing costs to

sub-units proportionally to the length of the code. This was consistent with the experts' (maintainers') intuition about the influence of the size of units maintained to the cost of maintenance.

3.4. Data analysis

After the metrics extraction and the case labeling, the data set should be analyzed to assure the data quality and consequent reliability of the model. The tools to analyze the data include the outlier analysis, the pareto chart and the distribution of the classes and attributes. A deeper discussion of the techniques for analyzing software data can be found in [Fenton96].

An outlier in a class might indicate noise in the class that can be eliminated. Alternatively, outliers of the whole set might indicate cases whose behavior is so atypical that they should not be used to build de model. A pareto chart consists in plotting the histogram of the classes to guarantee that there are enough data in each class to build a model. [Kubat97] [Kubat98] have some interesting proposals to treat situations in which classes are highly imbalanced.

Since we need to discretize the quality into a small number of label values, some discretization process is necessary. In our experiments the median of the values of the attribute that is being modeled is used as the boundary between the high and low labels.

A presentation of the de distribution of each of the attributes, containing maximums, minimums, average, standard deviation, median, upper and lower quartiles and the histogram may be a valuable tool in the interpretation of the model. In section 4, we present these measures obtained in our experiment.

3.5. Model evaluation

The model validation should be done in two different dimensions: the accuracy and the comprehensibility. As long as the model may be used to predict the behavior of unseen cases the accuracy, precision and recall are relevant attributes for evaluating the model and should be considered. Comprehensibility and meaningfulness of the model are harder to evaluate. To evaluate the latter, the model should be taken back to the software organization where its utility can be better evaluated. We present specific accuracy results in sec. 4.

3.6. Model application.

Software quality models may by applied in a number of ways, for example:

- Evaluating the software components generated inside the organization.
- Evaluating the software components generated by a third party.
- Construction of guidelines for the development process.

Firstly, the evaluation and prediction of software components generated inside de organization is the most obvious application of the software quality models.

Secondly, sometimes when dealing with sub-contracting of software, one whishes to be able to evaluate quality attributes of the software product before accepting it. In such cases during the relation with each sub-contacted it is possible to build quality models that relates the internal attributes of the sub-contracted software with its external behavior in the context of the contracting organization.

Thirdly, the analysis of the model may allow a better comprehension of the reasons that leads some quality aspect to be bellow the accepted level, allowing the organization to build or refine it's software development guidelines.

It is important to notice that although the method presented here was primarily exemplified using metrics of internal attributes of the software source code, this method can be extended to any metric of any phase of the software life-cycle.

Figure 3. represents a long-term view of the application of the software quality models in the software development process. A life-cycle model can be viewed as processes transforming software descriptions. The software quality model generation method proposed here can be applied to any software description, and may be used both to evaluate and to help creating guidelines for the process. In that way a software quality optimization cycle can be built.

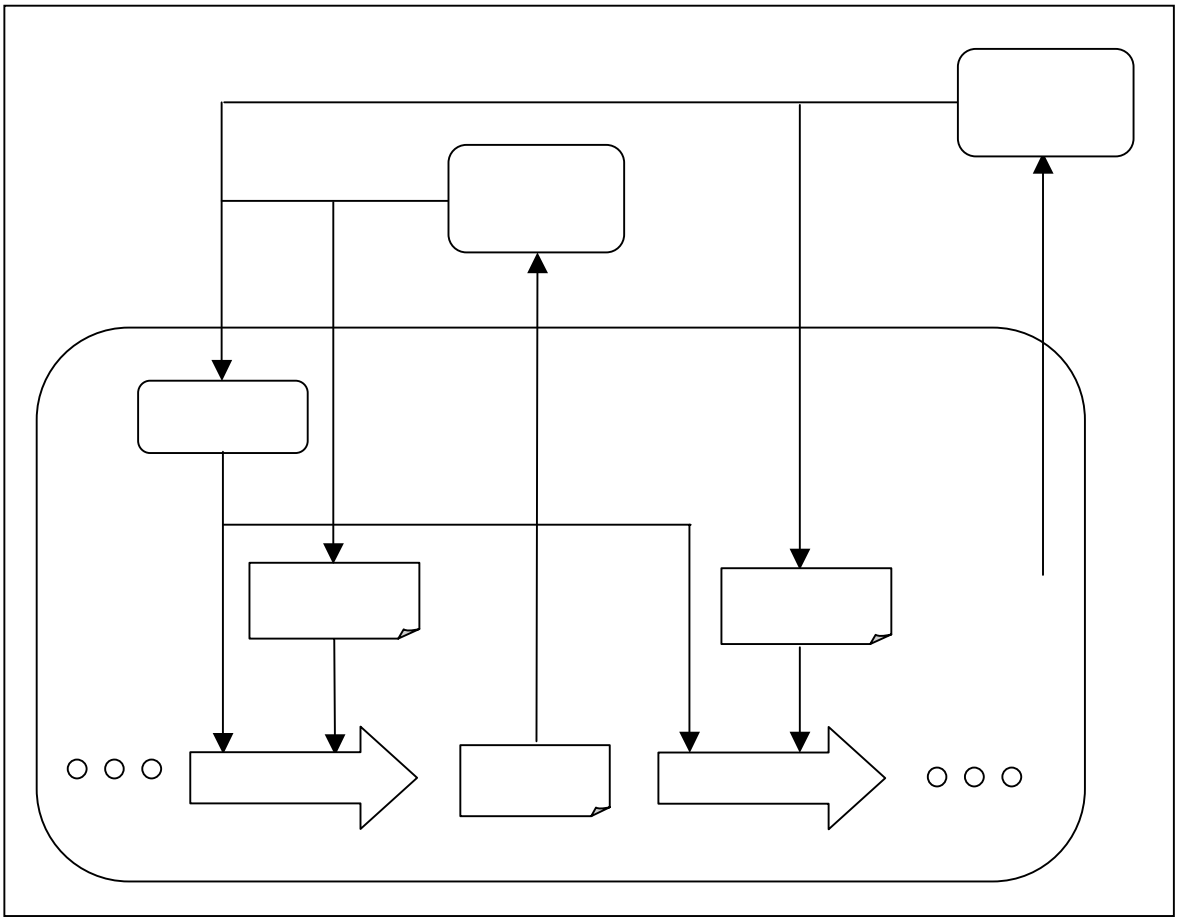


Figure 3. Long-term view of the application of the software quality models in the software development process

4. A specific application

The method proposed above was applied to a set of programs from Bell Canada, the company responsible for the local and phone traffic, as well as for the majority (60%) of long-distance and overseas market in Canada.

Bell Canada has implemented a set of functional changes in the billing system. The system is composed of hundreds of COBOL programs from which 355 were altered. The metrics extracted form those programs are presented in the figure 4.

Large values in the standard deviation column, when compared with the average, of the table indicate considerable degree of variance in the data. This large variance makes us believe that the sample programs were well selected.

During the change of the billing system the cost of specification and development of the changes were registered to entire sets of programs. The attribute to be modeled was the chagebility of the programs. To do so, the cost for changing was spread among the individual programs. Different forms of spreading the cost were considered. The chosen method was to divide the cost attributed to each program set in a weighed way to each program in the set, using as weight the "Total lines of Code".

$$\text{Alteration cost}_i = \frac{\text{Set cost} \cdot \text{Total lines of code}_i}{\sum_{Set} \text{Total lines of code}}$$

The dichotomization of the cost in two classes was made using the median of the costs. Alteration cost above the median was classified as "high" otherwise they were categorized as "low". In that manner 176 programs were classified as "high" and 179 as "low".

	Average	Std. Deviation	Median	Max.	Min.
Cyclomatic complexity	43.0507	81.28723	16	851	1
Declarative length	178.6056	214.7458	121	2066	18
Declarative statements	136.631	178.6303	87	1917	10
Declarative comments	19.16338	36.95701	12	378	0
Declarative comments density	0.164557	0.159099	0.123596	1.333333	0
Declarative blank lines	22.81127	22.241	17	226	0
Declarative blank lines density	0.208776	0.112239	0.187845	0.73913	0
Executable length	510.2676	813.6223	281	9939	17
Executable statements	371.5859	554.7937	205	5458	5
Executable comments	68.12113	254.8285	31	4470	1

Executable comments density	0.230784	0.237322	0.168	2.4	0.006452
Executable blank lines	70.56056	95.77846	37	686	0
Executable blank lines density	0.208938	0.111368	0.193548	0.650794	0
Total source lines	906.5634	1105.779	572	11290	105
Total Lines of code	513.0761	664.4598	313	5814	23
Total Lines of comments	298.2648	406.127	185	5464	51
Total Blank lines	95.22254	111.43	58	792	1
Reserved words	724.4761	1017.704	428	9667	20
Unconditional branches	5	22.04003	0	264	0

Five different models were generated using for NewID, CN2, C4.5, C4.5 rules and FOIL [Almeida98]. The models were tested using the one-out cross validation for NewID, CN2 and C4.5 and 3-fold cross validations for FOIL. The results are presented in the figure 5.

NewID

	High	Low	Completeness
High	134	42	76.14%
Low	39	140	78.21%
			Accuracy
Correctness	77.46%	76.92%	77.18%

CN2

	High	Low	Completeness
High	131	45	74.43%
Low	44	135	75.42%
			Accuracy
Correctness	74.86%	75.00%	75.00%

C4.5 Tree

	High	Low	Completeness
High	135	41	76.70%
Low	40	139	77.65%
			Accuracy
Correctness	77.43%	77.22%	77.18%

C4.5 rules

	High	Low	Completeness
High	135	41	76.70%
Low	40	139	77.65%
			Accuracy
Correctness	77.43%	77.22%	77.18%

FOIL

	High	Low	Completeness
High	89	90	48,86%
Low	49	130	72.63%
			Accuracy
Correctness	63.70%	59.09%	64.48%

Figure 5. Evaluation of the models in the Bell Canada experiment

5. Conclusion

The results shown in sec. 4 indicate that the correctness and completeness do not differ significantly between the different learningt systems used. The values obtained for C4.5 (trees and rules), NewID, and CN2 are virtually identical. The results for FOIL are lower, likely due to the fact that there are no truly relational attributes among the metrics that represent the programs, and therefore FOIL is used as a non-relational learning system.

A criterion under which the results between different learners differ is comprehensibility. The difference, however, is difficult to quantify, as there are no generally agreed upon measures to express comprehensibility [IJCAI95]. To us, the C4.5 rules results seemed most comprehensible.

Our experience on this real-life Data Mining project indicates that the choice of the learning system matters much less than the whole process, which is summarized bellow:

1. Case selection.
2. Attribute selection and extraction.
3. Labeling.
4. Data analysis.
5. Model generation by the learning program.
6. Model evaluation.
7. Model application.

This is consistent with the overall conclusion of the recent review of Machine Learning applications [Provost98]. In our paper, we have indicated how we dealt with some of these issues. We believe that more work in this area needs to be done, leading to a systematic approach by which ML techniques can be applied to the problems of this kind.

Bibliography

[Briand92] Briand, L.; Basili, V.; Hetmansky, C. J. A pattern recognition approach for software engineering data analysis IEEE transactions on software engineering, n. 18, v. 1, Nov. 1992.

[Briand93] Briand, L.; Thomas, W. M.; Hetmanski, C. J. Modeling and managing risk early in software development In proceedings IEEE 15th International Conference on software engineering, Baltmore, May 1993.

[Boehm81] Boehm, B. W. Software engineering economics, Prentice Hall, Englewood Cliffs, NJ 1981.

[Fenton96] Fenton, N. E.; Pfleeger, S. L. Software metrics: A rigorous & practical approach International Thomson computer press, London, 1996.

[Halstead77] Halstead, M. H. Elements of software science Elsevier Nonth Holand, New York, 1977.

[IJCAI95] IJCAI'95 workshop Machine learning and comprehensibility 14th International joint conference on artificial intelligence, Montreal, 1995.

[Jorgensen95] Jorgensen, M. Experience with the accuracy of software maintenance task effort prediction models IEEE Transactions on software engineering, v. 21, n. 8, p. 674-81, Aug. 1995.

[Kan95] Kan, S. H. Metrics and models in software quality engineering. 1. Ed. Readings , Massachusets, Addison-Wesley Publishing Company, 1995.

[Khoshgoftaar98] Khoshgoftaar, T. M. at all Using process history to predict software quality IEEE Computer p. 66-72, April 1998.

[Kubat97] Kubat, M.; Matwin, S. Addressing the course of imbalanced training sets: One-side selection In Fisher Jr., D. H. Machine leaning proceedings of the fourteenth international conference (ICML '97) Nasville, Tenesse, July 8-12, 1997, p. 179-85, Morgan Kaufman, San Francisco, Califonia, 1997.

[Kubat98] Kubat, M.; Holte, R. C.; Matwin, S. Machine learning for the detection of oil spills in satellite radar images Machine learning, n. 30, p. 195-215.

[Michalski83] Michalski, R. S. A theory and methodology of inductive learning endgames In Carbonel, J.G.; Michalski, R. S.; Mitchell, T. M. Machine learning. An artificial intelligence approach, v.1, Tioga, Palo Alto, California, p. 83-134, 1983.

[McCabe76] McCabe, T. J. A complexity measure IEEE Transactions on software engineering, v. 2, n.4 p. 308-20, Dec 1976.

[Porter90] Porter, A.; Selby, R. Empiricaly guided software development using metric-based classification trees IEEE Software, v. 7, n. 2, p. 463-81, March 1991.

[Provost98] Provost, F.; Kohavi, R. On applied research in machine learning Machine learning, v. 30, n.2/3, p. 127-32, 1998.