Machine-Learning Techniques for Software Product Quality Assessment

Hakim Lounis
Department of Computer Science
Université du Québec à Montréal, Canada
lounis.hakim@uqam.ca

Lynda Ait-Mehedine
Department of Computer Science and Operations Research
Université de Montréal, Canada
aitmemel@iro.umontreal.ca

Abstract

Integration of metrics computation in most popular Computer-Aided Software Engineering (CASE) tools is a marked tendency. Software metrics provide quantitative means to control the software development and the quality of software products. The ISO/IEC international standard (14598) on software product quality states, "Internal metrics are of little value unless there is evidence that they are related to external quality". Many different approaches have been proposed to build such empirical assessment models. In this work, different Machine Learning (ML) algorithms are explored with regard to their capacities of producing assessment/predictive models, for three quality characteristics. The predictability of each model is then evaluated and their applicability in a decision-making system is discussed.

1. Introduction

Software metrics provide quantitative means to control the software development and the quality of software products. They are necessary to identify where the resources are needed, and are a crucial source of information for decision-making. Moreover, early availability of metrics is considered as a key factor to a successful management of software development.

A large number of object-oriented (OO) measures have been proposed in the literature [1] [3] [4] [15]. A particular emphasis has been given to the measurement of design artifacts, in order to help assess quality early on during the development process. Such measures are aimed at providing ways of assessing the quality of software. Such an assessment of design quality is objective, and the measurement can be automated. But how do we know what measures actually capture important quality aspects?

As it is stated by the ISO/IEC international standard (14598), internal metrics are especially helpful when they are related to external quality attributes, e.g., maintainability, reusability, usability, etc. Assessment models can take different forms depending on the building technique that is used. For example, they can be mathematical models (case of statistical techniques like linear and logistic regression) or Artificial Intelligence-based models (case of Machine-Learning –ML-techniques). In all cases, they allow affecting a value to a quality characteristic based on the values of a set of software measures, and they allow the detection of design and implementation anomalies early in the software life cycle. They also allow organizations that purchase software to better evaluate and compare the offers they receive.

In the balance of this paper, we explore for three different quality characteristics (i.e., maintenance cost, reusability, and fault-proneness), the capacities of ML techniques to produce usable and understandable assessment/predictive models. We first present in section 2 previous works in the same topic. We then introduce in section 3 the hypotheses we want to verify with selected ML algorithms. These algorithms are presented in section 4. Section 5 describes the empirical process we follow and provides the experimental results we have obtained. Finally, conclusions, current research and directions for future research are outlined.

2. Related Work

Many of the measures proposed and their relationships to external quality attributes have been the focus of little empirical investigation. However, different approaches have been proposed to build such empirical assessment models. Most of the work has been done using statistical techniques [4] [5]. As far as we know, Selby and Porter [6] have been the first to use a ML classification algorithm to automatically construct software quality models. They have used ID3 [7], a ML classification algorithm, to identify those product measures that are the best predictors of interface errors likely to be encountered during maintenance. After Selby & Porter, many others, e.g., [8], [9], have used ML classification algorithms to construct software quality predictive models.

More recently, we [10] have investigated ML algorithms with regard to their capabilities to accurately assess the

Proceedings of the Fourth International Conference on Quality Software (QSIC'04)
0-7695-2207-6/04 $ 20.00 IEEE

correctability of faulty software components. Three different families' algorithms have been analyzed on the same data than those used by [8], and FOIL [11], an inductive logic programming system, presented the best results from the point of view of model accuracy. On the other hand, in most above-mentioned techniques, the estimation process depends on threshold values that are derived from a sample base. This dependency raises the problem of representativity for the samples, as these often do not reflect the variety of real-life systems. Thus, in a previous study, we also worked on the identification of trends instead of the determination of specific thresholds. The problem of using precise metric thresholds values associated with the estimation models was then bypassed by replacing them with fuzzy thresholds [12].

In the present work we reaffirm that machine-learning approaches, as they are classified in a well-accepted taxonomy [13], are of a significant help for the building of software quality assessment models. This statement is strengthened when the main purpose is the conception of a Knowledge-Based System (KBS) for quality assessment. KBSs are used in numerous application domains. They are used to reproduce an expert's reasoning and are based on two distinct components: knowledge and reasoning. Separation between these two levels of intervention makes it possible to offer a flexibility of operation that many traditional software approaches are missing. KBSs are presently an effective and useful solution to integrate the necessary analyses of software engineer experts and to meet the needs of the software industry, in terms of quality. It is especially true when we consider the recent progress done within the tools that help KBSs design.

The following section introduces the hypotheses we have verified with the ML techniques.

3. Working Hypotheses

In order to explore/confirm the relevance of the use of ML algorithms for assessment models production, we work on the assessment of three quality attributes starting from software products internal measures.

3.1. Assessing Corrective Maintenance Costs

The first hypothesis (let us call it, H_1) concerns the relevance of some internal product measures for assessing corrective maintenance costs. To do so, and as in [8] and [10], we have used (1) data collected on corrective maintenance activities for the Generalized Support Software reuse asset library located at the Flight Dynamics Division of NASA's GSFC and (2) internal product measures extracted directly from the faulty components of this library. The corrective maintenance data come from the maintenance of a library of reusable components. This library is known as the Generalized Support Software (GSS) reuse asset library. Component development began in 1993. Subsequent efforts focused on generating new components to populate the library and on implementing specification changes to satisfy mission requirements. The first application using this library was developed in early 1995. The asset library consists of 1K Ada83 components totaling approximately 515 KSLOC. In our study, the dependent variable is the total effort spent to isolate and correct a faulty component. Isolation and correction effort is measured on a 4-point ordinal scale: 1 hour, from 1 hour to 1 day, from 1 to 3 days, and more than 3 days. Once an error is found during configuration and testing, the maintainer finds the cause of the error, locates where the modifications are to be made, and determines that all effects of the change are accounted for. Then the maintainer modifies the design (if necessary), code, and documentation to correct the error. Once the maintainer fixes the error, the maintainer provides the names of the components changed (in our case the faulty components). To build the classification model, we have dichotomized the corrective maintenance cost into two categories: *low* and *high*.

On the other hand, the internal product measures, e.g., size metrics, cyclomatic complexity, Hastead's metrics, etc. are extracted thanks to the ASAP tool [14]. ASAP shares the same weaknesses of comparable tools: most of the measures are highly correlated to size and contrary to some of the metrics involved in the next sub-sections, they do not capture high-level internal software product attributes, e.g. modularity. The use of higher sophisticated measures could indeed improve the accuracy of the assessment models we want to generate. The construction of the tools to extract these measures in the framework of this study would also be cost-prohibitive. Finally, it is important to point out that the definition and/or validation of software measures is beyond the scope of this paper. For more details about the data, please see [10].

Finally, the hypothesis we state is the following:

> Hypothesis H_1: Measures extracted by the ASAP tool (size metrics, cyclomatic complexity, Hastead's metrics, etc.) affect corrective maintenance costs.

3.2. Assessing Reusability

Our concern is to automate the detection of potentially reusable components. The basic idea is that some internal attributes like inheritance, coupling or complexity can be good indicators of the possibility of reuse of a component. Establishing a direct relation between these attributes (which are measurable) and the potential of a component to be reusable (which is not directly measurable) can be an interesting track for the automation of detection.

Reusability is a complex factor, which is domain dependent. Some components are more reusable in one domain than in others. Our goal is not to search for a set of methods measuring reusability universally but to study some specific aspects and characteristics pertaining to OO programming languages, e.g. C++ that affect reusability. We propose four reusability hypotheses (H_{2a}, H_{2b}, H_{2c}, and H_{2d}) regarding the relationships between reusability and each of inheritance, coupling (at the code and design level), and complexity, respectively. Different aspects can be considered to measure empirically the reusability of a component depending on the adopted point of view. One aspect is the amount of work needed to reuse a component from a version of a system to another version of the same system. Another aspect is the amount of work needed to reuse a component from a system to another system of the same domain. This latter aspect was adopted as the empirical reusability measure for our experiments. To define the possible values for this measure, we worked with a team specializing in developing intelligent multiagents systems. The obtained values are:

- Totally reusable: means that the component is generic to a certain domain (in our case "intelligent multiagents systems").
- Reusable with minimum rework: means that less than 25% of the code needs to be altered to reuse the component in a new system of the same domain.
- Reusable with high amount of rework: means that more than 25% of the code needs to be changed before reusing the component in a new system of the same domain.
- Not reusable at all: means that the component is too specific to the system to be reused.

On the other hand, the internal characteristics we measure are divided into three categories: inheritance, coupling, and complexity.

The inheritance metrics we have used are the following: depth of inheritance tree *(DIT),* height of inheritance tree (HIT), number of ancestors (NOA), number of children (NOC), number of inherited methods (NMI), and number of polymorphic methods (NOP).

In the case of coupling metrics, Briand & al. [1] proposed a comprehensive suite of metrics to measure the level of class coupling during the design of OO systems. The set of metrics quantifies the different OO design mechanisms (for instance, friendship between classes, specialization, and aggregation). There are three different facets, or modalities, of coupling between classes in OO systems. They are referred as locus, type and relationship. Locus refers to the expected locus of impact; i.e., whether the impact of change flows towards a class (import) or away from a class (export). A class c exports impact to its friends and descendants, and imports impact from its ancestors and classes that have c as their friend. On the other hand, type refers to the type of interaction between classes (or their elements): It may be Class-Attribute (CA) interaction, Class-Method (CM) interaction, or Method-Method (MM) interaction. Lastly, relationship refers to the type of relationship between two classes. It can be friendship, inheritance, or other (neither).

Combining these three facets, 18 different types of coupling are derived as listed in the following table:

Table 1. Design Coupling Metrics

Symbol	Metrics
IFCAIC	Inverse Friend CA Import Coupling
ACAIC	Ancestors CA Import Coupling
OCAIC	Others CA Import Coupling
FCAEC	Friend CA Export Coupling
DCAEC	Descendant CA Export Coupling
OCAEC	Others CA Export Coupling
IFCMIC	Inverse Friend CM Import Coupling
ACMIC	Ancestors CM Import Coupling
OCMIC	Others CM Import Coupling
FCMEC	Friend CM Export Coupling
DCMEC	Descendant CM Export Coupling
OCMEC	Others CM Export Coupling
IFMMIC	Inverse Friend MM Import Coupling
AMMIC	Ancestors MM Import Coupling
OMMIC	Others MM Import Coupling
FMMEC	Friend MM Export Coupling
DMMEC	Descendant MM Export Coupling
OMMEC	Others MM Export Coupling

On the other hand, in [15], we proposed a comprehensive suite of metrics to quantify coupling at the code level, in modular software systems written in the C/C++ programming language. In this work, a *module* is considered as a collection of *units*, collected in a file and its associated header. A program *unit* is one or more contiguous program statements having a name by which other parts of the system can invoke it. There are two different kinds of module interconnections: (i) common interconnection (if the modules are to be used together in a useful way, there may be some external references, in which the code of one module refers to a location in another module. This reference may be to a data location defined in one module and used in another), and, (ii) unit-call interconnection (an entry point of a unit appears in the code of one module and is called from another module unit.

An other information is used to distinguish between module interconnections: the kind of information shared by interconnected modules (parameters or global areas), the type to which belongs the shared information (scalar, structure, class, etc.), and, the use made of this shared information (the shared information is used on the right

side of an assignment statement, in an output statement, in a predicate statement, in an assignment to another variable, and that variable is then used in a predicate statement), or, the shared information is modified).

Combining these criteria, we obtain 24 coupling metrics (some combinations are not possible). The detailed description of these metrics can be found in [15].

Finally, the complexity metrics we have used, are consigned in table 2. For more information about the data, see [16].

The four obtained hypotheses are the following:

Hypothesis H_{2a}: A class position in the class hierarchy affects its reusability.

Hypothesis H_{2b}: The code coupling between a class and its environment affects its reusability.

Hypothesis H_{2c}: The design coupling between a class and its environment affects its reusability.

Hypothesis H_{2d}: A component complexity affects its reusability.

Table 2. Complexity Metrics

Symbol	Definition
WMC	the complexity of a class defined as the sum of the complexities of the methods of this class.
RFC	the size of the Response Set of a class, defined as the set of methods in the class together with the set of methods called by the class's methods.
NOM	the number of local methods in a class.
CIS	the number of public methods in a class.
NPM	the average of the number of parameters per method in a class.
NOT	the number of trivial methods, which is marked as inline in C++, in a class.
NOP	the number of methods that can exhibit polymorphic behavior, e.g., virtual methods in C++.
NOD	the number of attributes in a classes.
NAD	the number of user defined objects, i.e. abstract data types, used as attributes in a class and are therefore necessary to instantiate an object instance of the class.
NRA	the number of pointers and references used as attributes in a class.
NPA	the number of attributes that are declared as public in a class.
NOO	the number of overloaded operator methods defined in a class.
CSB	the size of the objects in bytes that will be created from a class declaration. The size is computed by summing the size of all attributes declared in a class.

3.3. Assessing Fault-Proneness

Finally, the goal of the third experiment is to empirically investigate the relationships between object-oriented design measures and fault-proneness at the class level. We therefore need to select a suitable and practical measure of fault-proneness as the dependent variable for our study. For instance, in a non-faulty component, there was not any change of corrective type and in a faulty one, there were one or more changes of corrective type during the development/maintenance phase. The measures of inheritance, coupling, and cohesion identified in a literature survey on object-oriented design measures are the independent variables used in this study. The definition of the 28 coupling metrics (including those of table 1) and the 10 cohesion ones (including *LCOM*, *TCC, LCC*, etc.) could be found in [17]**.** The inheritance metrics are 11: we added 5 more metrics comparatively to those presented in section 3.2: *AID* (average inheritance depth), *NOD* (number of descendants), *NMO* (number of overridden methods), *NMA* (number of methods added, new methods), and *SIX* (specialization index).

We focus on design measurement since we want the measurement-based models investigated in this study to be usable at early stages of software development. Furthermore, we only use measures defined at the class level since this is also the granularity at which the fault data could realistically be collected.

We obtain then 3 more hypotheses called H_{3a}, H_{3b}, and H_{3c}:

Hypothesis H_{3a}: A class position in the class hierarchy affects its fault-proneness.

Hypothesis H_{3b}: The design coupling between a class and its environment affects its fault-proneness.

Hypothesis H_{3c}: Class cohesion affects its fault-proneness.

To verify these 8 hypotheses, we explore how different ML algorithms, belonging to different approaches, could help us build accurate assessment models. This is the content of the next section.

4. Machine-Learning Approaches

Machine Learning (ML) is one of the most prolific subfield of Artificial Intelligence (AI) where many approaches have been developed. The taxonomy defined by [13] and given in figure 1 illustrates only a subset of them, in which we have chosen seven algorithms.

Most of the work done in machine learning has focused on supervised machine learning algorithms. Starting from the description of classified examples, these algorithms produce definitions for each class. The most popular approach is an induction one - the divide and conquer

approach -. In this approach, a decision tree generally represents the induced knowledge. It is the case of algorithms like ID3 [7], CART [18], ASSISTANT [19] and C4.5 [20]. This algorithm could summarize their principle:

If all the examples are of the same class
Then- Create a leaf labeled by the class name;
Else
- Select a test based on one attribute;
- Divide the training set into subsets, each associated to one of the possible values of the tested attribute;
- Apply the same procedure to each subset;
Endif.

The key step of the algorithm above is the selection of the "best" attribute to obtain compact trees with high predictive accuracy. Information-based heuristics have provided effective guidance for the division process.

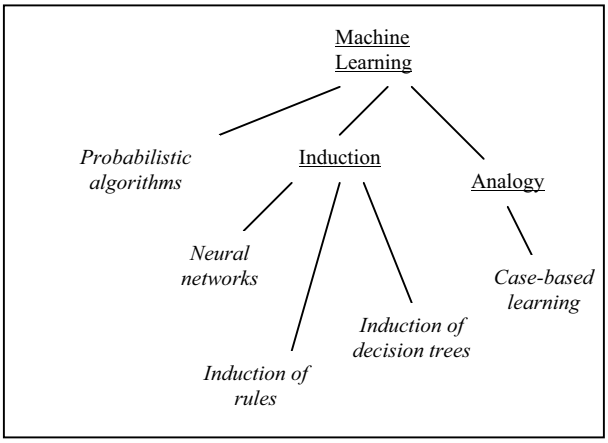

Figure 1. ML Taxonomy

OC1 (Oblique Classifier 1) is also a decision tree induction system designed for applications where the instances have numeric (continuous) feature values, which is the case in our study. However, It builds decision trees that contain linear combinations of one or more attributes at each internal node:

$$\sum_{i=1}^{i=k} a_i x_{i+} \; a_{k+1}$$

These trees then partition the space of examples with both oblique and axis-parallel hyper planes. More details on this algorithm are given in [21].

The induction of rules is another important ML approach. Because the structure underlying many real-world datasets is quite rudimentary, and just one attribute is sufficient to determine the class of an instance quite accurately, it turns out that simple rules frequently achieve surprisingly high accuracy. The pseudo-code for such an approach (called one rule) is the following:

For each attribute
For each value of that attribute, make a rule as follows:
- Count how often each class appears;
- Find the most frequent class;
- Make the rule assign that class to this attribute value;
Calculate the error rate of the rules;
Choose the rules with the smallest error rate.

The covering technique illustrates a more sophisticated approach. It represents classification knowledge as a disjunctive logical expression defining each class. CN2 [22] is an instance of such a family. The following algorithm could summarize it:

- Find a conjunction that is satisfied by some examples of the target class, but no examples from another class;
- Append this conjunction as one disjunct of the logical expression being developed;
- Remove all examples that satisfy this conjunction and, if there are still some remaining examples of the target class, repeat the process.

On the other hand, the multilayer perceptron is probably the most widely used neural network architecture to solve classification problems with supervised learning. It consists of an input layer, one or more hidden layers, and an output layer of neurons that feed one another via synaptic weights. The input layer simply holds the data to be classified. The outputs of the hidden and output layers are computed, for each neuron, by calculating the sum of its inputs multiplied by its synaptic weights and by passing the result through an output function. The synaptic weights of the hidden and output neurons are determined by a training procedure where examples of the patterns to be learned are presented successively, and where the weights are adjusted so as to minimize the error between the obtained classification results and the desired ones. The standard training procedure for the multilayer perceptron uses the backpropagation algorithm [23] or one of its derivatives. For instance, the resilient backpropagation (RPROP) algorithm [24] has faster learning and a better overall performance than regular backpropagation or backpropagation with a momentum [23].

Case-Based Learning (CBL) is the most popular technique in the learning by analogy paradigm. The training examples are stored verbatim, and a distance function is used to determine which member(s) of the training set is closest to an unknown test instance. Once the nearest training instance(s) has/have been located, its class is predicted for the test instance. Although there are multiple choices, most instance-based learners use Euclidian

distance. Despite several practical problems, it is considered as a good compromise.

Lastly, representing probabilistic approaches, Bayesian techniques have long been used in the field of pattern recognition, but only recently have they been taken seriously by ML researchers [25] and made to work on data sets with redundant attributes and numeric attributes. Such a probabilistic algorithm is trained by estimating the conditional probability distributions of each attribute, given a class label. The classification of a case, represented by a set of values for each attribute, is accomplished by computing the posterior probability of each class label, given the attributes values, using Bayes' theorem. The case is assigned to the class with the highest posterior probability. The following formula corresponds to the probability of the class value $C=cj$ given the set of attribute values $e_k=\{A_1=a_{1k}, \ldots, A_m=a_{mk}\}$:

$$p(c_j / e_k) = \frac{\prod_{k=1}^{m} p(a_{ik} / c_j)\ p(c_j)}{\sum_{h=1}^{c} \prod_{k=1}^{m} p(a_{ik} / c_h)\ p(c_h)}$$

Recent empirical evaluations have found the Bayesian algorithms to be accurate [26] and very efficient at handling large databases (e.g., data mining tasks). The simplifying assumptions underpinning the Bayesian algorithms are that the classes are mutually exclusive and exhaustive and that the attributes are conditionally independent once the class is known. It is a great stumbling block; however, some attempts are being made to apply Bayesian analysis without assuming independence.

The next section will present results we have obtained by using seven ML algorithms, chosen in the previous taxonomy, on data related to corrective maintenance costs, reusability, and fault-proneness.

5. Hypotheses Verification

We propose to verify the eight hypotheses stated in section 3 and related to three different quality attributes. From the ML taxonomy presented in section 4, we select seven algorithms representing different approaches (see table 3). The computation of models accuracy is done thanks to a cross-validation procedure. It is helpful when the amount of data for training and testing is limited, which is our case; we try a fixed number of approximately equal partitions of the data, and each in turn is used for testing while the remainder is used for training. At the end, every instance has been used exactly once for testing.

Figure 2 illustrates the different steps of our empirical process. Table 3 summarizes the accuracies computed for each of the 8 models.

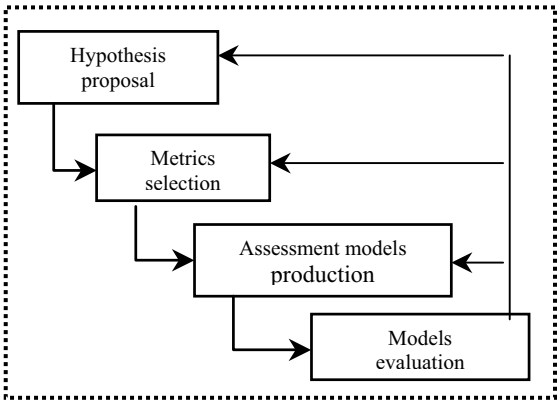

Figure 2. Empirical Process

The accuracies computed for yet studied ML algorithms are confirmed by this study. As we were expecting, C4.5 (decision tree), CN2 (covering rules), and the RPROP neural network present the best results in terms of predictive power.

Table 3. Computed Accuracies (%)

Stated Hypotheses / **ML algos**	H_1	H_{2a}	H_{2b}	H_{2c}	H_{2d}	H_{3a}	H_{3b}	H_{3c}
C4.5	66	73.8	86.9	88.1	89.3	73.8	77.5	73.7
OC1	68.8	48.8	51.2	53.6	54.8	72.5	73.8	70.9
One-rule	58.5	45.2	53.6	56	42.9	72.5	76.3	65.8
CN2	54	67.3	62.5	63.1	60	87.5	78.5	72.2
CBL	56.7	54.8	66.7	63.1	60.7	73.8	71.3	70.9
RPROP neural net	56.7	56.3	75	75	68.8	81.3	75	81.3
Naïve Bayes	54.9	45.2	46.4	57.1	45.2	36.2	68.8	70.9

The light grey cells show accuracies comprising between 60% and 75% and the dark grey ones contain accuracies greater than 75%. Even, simplistic algorithms (e.g., one rule and naïve Bayes) show some results higher than 60%! On the other hand, a basic CBL algorithm shows promising results; these results could be improved by changing, for instance, the distance function. Lastly, OC1 gives some interesting results, mainly for hypotheses regarding the fault-proneness assessment.

One distinction between C4.5 and CN2 from one part, and RPROP from the other part, is on the induced knowledge. The two first produce rules or decision trees that could be exploited by a knowledge-based system in a decision making process. RPROP doesn't produce any kind of explicit knowledge. However, it allows predicting a quality attribute, giving some internal measures values.

What about the hypotheses we try to verify?

H_{3b} and H_{3a} obtain pretty high accuracies. We conclude that for the system we have studied, the design coupling between a class and its environment, and the class position in the class hierarchy, affect the class fault-proneness, as we defined it. Moreover, some ML algorithms (e.g., RPROP) identify cohesion as a factor to consider when we deal with fault-proneness.

On the other hand, design and code coupling metrics seem to be relevant also to assess reusability of C++ medium size applications. It is also the case for the complexity of a class; it could be relevant to predict its reusability, for similar applications.

Finally, and despite the fact that C4.5 and CN2 scores are close to 70%, we consider that the results are not significant to conclude anything about the relevance of ASAP extracted metrics for corrective maintenance costs assessment.

In terms of selected internal metrics, we found a certain uniformity between the different models, and we confirm some results previously published. For instance, various ML algorithms select CSB (size of the object in bytes) as a complexity metric having an impact on the reusability of the component. It is also the case for design coupling metrics like OCAEC and OMMEC (export coupling). They are selected by numerous ML algorithms as relevant for assessing the reusability of a C++ class.

On the other hand, the classic CBO metric [3] (the number of other classes to which a class is coupled), and methods invocation, e.g., ACMIC (ancestor class method import coupling), RFC_{OO} (the number of methods that can potentially be executed in response to a message received by an object of that class), and IH-ICP (the number of ancestor methods invocation in a class, weighted by the number of parameters of the invoked methods) are identified as relevant design coupling measures for fault-proneness assessment. In the case of inheritance measures, NMI (the number of methods inherited), NMO (the number of overridden methods), and DIT (the maximum length from the class to a root) are the three that are stated relevant in our experiment on fault-proneness. Finally, $LCOM_2$, $LCOM_5$, and $LCOM_1$ are the definitions of cohesion identified as relevant by the ML algorithms for assessing fault-proneness.

Of course, more experiments on more data extracted from various systems, are needed to confirm such conclusions. However, ML algorithms presented in this study are undeniable alternatives for software quality attributes assessment/prediction.

6. Conclusion and Perspectives

The first important conclusion is relative to the accuracies obtained by the machine-learned assessment models. They are comparable for those obtained with other techniques, and even sometimes, higher! Further more, we know that for some ML algorithms, e.g., C4.5 and CN2, the results will be much higher with non-continuous numeric data. Our past work on the fuzzyfication of such measures [12] is an example of such a data pre-processing step. It also gives a solution to the problematic use of precise metric thresholds values. We are now working on a pre-processing process. It tends to translate continue attributes into discrete ones, based on statistical considerations.

The main strength of such ML produced models is that we can incorporate them in an AI decision-making process, where, a KBS architecture keeps a good separation between what we consider as an expert knowledge (the produced models) and the procedures that exploit this knowledge. We are currently implementing an object-oriented knowledge-based architecture, which allows us (i) analyze OO source code, (ii) compute internal metrics, (iii) produce ML-based assessment models, and (iv) exploit these models by an inference system. Even the KBS is object-oriented, as we use the ILOG Jrules[1] API for build it. The following figure illustrates such an architecture.

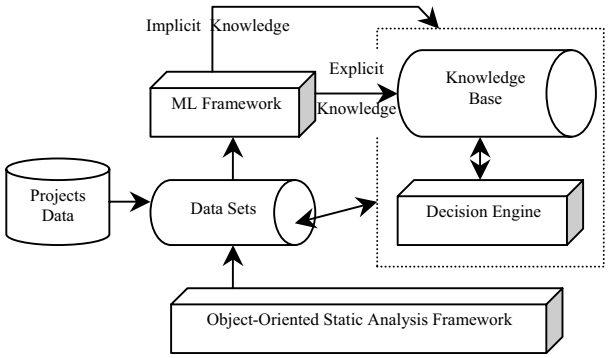

Figure 3. An OO KBS for quality assessment

The distinction we make between explicit and implicit knowledge results from the type of ML technique we use. In case of a symbolic technique that induce rules or trees, e.g., CN2 or C4.5, we talk about explicit knowledge. It is a piece of knowledge that can be stored in a knowledge base, and then, used by a decision engine. On the other hand, an artificial neural network can only assess the values of quality attributes given internal measures of a target application (it is a black box!): in this case, we talk about implicit knowledge that is used within the assessment process.

The rule and tree below are instances of explicit and understandable knowledge produced respectively by CN2 and C4.5.

[1] ILOG: www.ilog.com

```
IF   IH-ICP > 31.00
     THEN  Class is fault-prone
```

```
CBO <= 1: not fault-prone (4.0)
CBO > 1
|  IH-ICP <= 18
|      |  DCMEC <= 0: fault-prone (62.0/10.0)
|      |  DCMEC > 0
|      |          |  ICP <= 9: not fault-prone (4.0)
|      |          |  ICP > 9: fault-prone (2.0)
|  IH-ICP > 18
|      |  ACMIC <= 2: not fault-prone (6.0)
|      |  ACMIC > 2: fault-prone (2.0)

Number of Leaves:     6
Size of the tree:     11
```

However, they have to be confirmed and generalized by more experiments on more software data, extracted from various and representative applications.

7. References

[1] L. Briand, P. Devanbu, W. Melo, "An Investigation into Coupling Measures for C++", Proceedings of ICSE '97, Boston, USA, 1997.

[2] J.M. Bieman, B.-K. Kang, "Cohesion and Reuse in an Object-Oriented System", in Proc. ACM Symp. Software Reusability (SSR'94), 259-262, 1995.

[3] S.R. Chidamber, C.F. Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, 20 (6), 476-493, 1994.

[4] W. Li, S. Henry, "Object-Oriented Metrics that Predict Maintainability", J. Systems and Software, 23 (2), 111-122, 1993.

[5] T.M. Khoshgoftaar, J.C. Munson, "Predicting Software Development Errors Using Software Complexity Metrics", in IEEE journal on selected Areas in Communications, vol.8, n.2, February 1990.

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

[7] J.R. Quinlan, "Induction of decision tree", in Machine Learning journal 1, p 81-106, 1986.

[8] V. Basili, Condon, K. El Emam, R. B. Hendrick, W. L. Melo, "Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components", in Proc. of the IEEE 19th Int'l. Conf. on S/W Eng., Boston, 1997.

[9] M. Jorgensen, "Experience with the Accuracy of Software Maintenance Task Effort Prediction Models", in IEEE TSE, 21(8):674-681, August 1995.

[10] M.A. De Almeida, H. Lounis, W. Melo, "An Investigation on the Use of ML Models for Estimating Software Correctability", in the Int. Journal of Software Engineering and Knowledge Engineering, October 1999.

[11] J.R. Quinlan, "Learning Logical Definitions from Relations", in ML journal, vol 5, n°3, p 239-266, 1990.

[12] H.A. Sahraoui, M. Boukadoum, H. Lounis, "Building Quality Estimation models with Fuzzy Threshold Values", in "L'objet", volume 7, number 4, 2001.

[13] Y. Kodratoff, "Apprentissage symbolique : une approche de l'IA", tome 1&2, Cépaduès-Editions, 1994.

[14] D. Doubleday, "ASAP: An Ada Static Source Code Analyzer Program", in CS-TR-1895, Computer Science Dept, University of Maryland, College Park, MD, 1995.

[15] H. Lounis, W. L. Melo, "Identifying and Measuring Coupling in Modular Systems", in Proc. of the 8th International Conference on Software Technology ICST'97, p. 23-40, Curitiba, Brasil, 1997.

[16] Y. Mao, H.A. Sahraoui, H. Lounis, "Reusability Hypothesis Verification Using Machine Learning Techniques: A Case Study", in Proc. of the 13th IEEE International Automated Software Engineering Conference, p. 84-93, Hawai, October 13-16, 1998.

[17] L.C. Briand, J. Wust, H. Lounis, "Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs", in Emp. Soft. Engineering, an Int. Journal, 6 (1):11-58, 2001, Kluwer Academic Publishers.

[18] L. Breiman, J. Friedman, R. Olshen, C. Stone, "Classification and Regression Trees". Published by Wadsworth, 1984.

[19] B. Cestnik, I. Bratko, I. Kononenko, "ASSISTANT 86: a knowledge elicitation tool for sophisticated users", Progress in machine learning, Sigma Press, 1987.

[20] J.R. Quinlan, "C4.5: Programs for Machine Learning", Morgan Kaufmann Publishers, 1993.

[21] S.K. Murthy, S. Kasif, S. Salzberg, "A System for Induction of Oblique Decision Trees", in Journal of Artificial Intelligence Research 2, 1-32, august 1994.

[22] P. Clark, T. Niblett, "The CN2 induction algorithm", in ML Journal, 3(4):261-283, 1989.

[23] D. E. Rumelhart, J. L. McClelland, ed., Parallel Distributed Processing, Explorations in the Micro structure of Cognition; vol. 1: Foundations, MIT press, 1986.

[24] M. Riedmiller, H. Braun, "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm", in Proc. of the IEEE Intl. Conf. on Neural Networks, San Francisco, CA, 1993.

[25] P. Langley, W. Iba, K. Thompson, "An analysis of Bayesian Classifiers", in Proc. of the National Conference on Artificial Intelligence, 1992.

[26] R. Kohavi, "Scaling up the accuracy of naive-Bayes classifiers: a decision-tree hybrid", in Proc. of the 2nd Int. Conference on Knowledge Discovery and Data Mining, 1996.