Evolutionary Decision Trees and Software Metrics for Module Defects Identification

Monica Chiș

Abstract—Software metric is a measure of some property of a piece of software or its specification. The aim of this paper is to present an application of evolutionary decision trees in software engineering in order to classify the software modules that have or have not one or more reported defects. For this some metrics are used for detecting the class of modules with defects or without defects.

Keywords—Evolutionary decision trees, decision trees, software metrics.

I. INTRODUCTION

SOFTWARE engineering describes the collection of techniques that apply an engineering approach to the construction and support of software products.

Software engineering activities include managing, costing, planning, modeling, analyzing, specifying, designing, implementing, testing and maintaining software products. Whereas computer science provides the theoretical foundations for building software, software engineering focuses on implementing software in a controlled and scientific way [1].

Software metrics is a term that embraces many activities, all of which involve some degree of software measurement [1] such as: cost and effort estimation, productivity, measures and models, data collection, quality models and measures, reliability models, performance evaluation and models, structural and complexity metrics, capability-maturity assessment, management by metrics, evaluation of methods and tools.

The paper presents an application of evolutionary decision trees in software engineering for reporting modules with defects. In order to do this 5 different lines of code measures, 3 McCabe metrics, 4 base Halstead measures, a branch count are used. Identified modules that have one or more defects can be re-designed or tested and maintained more cautiously and any other special care can be devoted to these modules.

The rest of this paper is organized as follows. Section 2 briefly discusses the software metrics used in this paper. Section 3 introduces a brief description of decision trees. Section 4 contains the experimental evaluation of this method. Section 5 gives some conclusions and suggestion for future work on this direction.

Manuscript received January 31, 2008.
M. Chiș is with Avram Iancu University Cluj-Napoca, Romania (e-mail: monicachis@clicknet.ro).

II. SOFTWARE METRICS

In order to frame our contribution in the proper context we begin with a review of the concept of software metrics and introduction on the related work.

Software metrics is a term that embraces many activities, all of which involve some degree of software measurement [1] such as: cost and effort estimation, productivity, measures and models, data collection, quality models and measures, reliability models, performance evaluation and models, structural and complexity metrics, capability-maturity assessment, management by metrics, evaluation of methods and tools.

In this section, the software metrics problem is presented. A classification of software metrics is presented. The most used software metrics are analyzed.

The product software metrics deal with the characteristics of source code for a software project. Product software metrics are subdivided in: Size Metrics, Complexity Metrics, Halstead’s Software Metrics.

A. Size Metrics

Size Metrics are represented by a number of metrics attempt to quantify software “size”.

For a software application it is easy to measure the number of lines of codes for quantifying software size. We discuss here a little bit about some aspects of software size [1]. Each product of software development is a physical entity. In this acceptance, it can be described in terms of its size. Ideally, the idea was to define a set of attributes for software size analogous to human height and weight. Each attribute captures a key aspect of software size. Fenton [1] suggest the following software size aspects:

1) length: physical size of the product
2) functionality: functions supplied by the product to the user
3) complexity
4) problem complexity: the complexity of the underlying problem
5) algorithmic complexity: efficiency of the algorithm
6) structural complexity: algorithm structure
7) cognitive complexity: understandability of software

The most commonly used measure for the length of a code source of a program is the number of lines of code (LOC) [1]. The abbreviation NCLOC is used to represent a non-commented source line of code. NCLOC is also sometimes referred to as effective lines of code (ELOC). NCLOC is therefore a measure of the uncommented length.

The commented length is also a valid measure, depending
on whether or not line documentation is considered to be a part of programming effort. The abbreviation CLOC is used to represent a commented source line of code [1].

By measuring NCLOC and CLOC separately we can define:

\[ \text{total length (LOC) = NCLOC + CLOC} \]  \hspace{1cm} (1)

It is useful to separate comment lines and other lines (NCLOC). KLOC is used to denote thousands of lines of code.

Generally, it is better to address how the followings are handled:
1) blank lines
2) comment lines (CLOC)
3) data declarations or other commands
4) lines that contains several separate instructions
5) lines programs generated by a tool

The entity CLOC/LOC is then a measure of the density of comments in a program.

The purpose of software is to provide certain functionality for solving some specific problems or to perform certain tasks. Efficient design provides the functionality with lower implementation effort and fewer LOCs. Therefore, using LOC data to measure software productivity is like using the weight of an airplane to measure its speed and capability. In addition to the level of languages issue, LOC data do not reflect no coding work such as the creation of requirements, specifications, and user manuals. The LOC results are so misleading in productivity studies that Jones states "using lines of code for productivity studies involving multiple languages and full life cycle activities should be viewed as professional malpractice" [3],[4],[5],[6],[7],[8],[9].

For each metrics program is better to establish exactly what enter the LOC. Because the confusion existing according with the types of lines of code that are counting in LOC, could be a variation until 500%). LOC has the disadvantage that could be calculated exactly only in a very advanced phase of project. Programming Languages have different expressiveness that this metric depend a lot the language used [10]. A source line of code (SLOC) is a term used in the most of software metrics program.

**B. Complexity Metrics**

The cyclomatic complexity metrics [11] are described below. For any given computer program, its control flow graph, G, could be drawn. Each node of G corresponds to block of sequential code and each arc corresponds to a branch of decision in program. The cyclomatic complexity of such a graph can be computed by a simple formula from graph theory, as:

\[ v(G) = e - n - 2 \]  \hspace{1cm} (2)

where:
\[ e \] is the number of edges

McCabe [11] proposed that \( v(G) \) could be used as a measure of program complexity.

Halstead metric are described below. Halstead [12] proposed a unified set of metrics that apply to several aspects of programs, as well as to the overall software production effort.

Some of this product metrics are: program vocabulary metrics (\( n \)), program length metrics (\( N \)), program volume metrics (\( V \)).

According to Halstead [12] computer programs can be visualized as a sequences of tokens, each token being classified as either an operator or operand.

He has defined the program vocabulary (\( n \)), of a program as:

\[ n = n_1 + n_2 \]  \hspace{1cm} (3)

where:
\[ n_1 \] is the number of unique operators in the program;
\[ n_2 \] is the number of unique operands in the program;
\[ n \] is the total numbers of unique tokens from which the program has been constructed [12].

**Program length** (\( N \)) is the count of the total number of operators and operands in the program

\[ N = N_1 + N_2 \]  \hspace{1cm} (4)

where:
\[ N_1 \] is the total number of operators in the program;
\[ N_2 \] is the total number of operands in the program

\( N \) represents a clearly measure of the program’s size. Halstead considers \( N' \) an estimated value for \( N \) calculated with the formula presented below:

\[ N' = n_1 \log_2 n_1 + n_2 \log_2 n_2 \]  \hspace{1cm} (5)

Program Volume (\( V \)) is a measurement of program size. \( V \) is the measure of the storage volume required to represent the program.

\[ V = N' \cdot \log_2 n \]  \hspace{1cm} (6)

Between LOC, \( N \) and \( V \) there is a linearly related.

**III. DECISION TREES**

Inductive inference is the process of moving from concrete examples to general models, where the goal is to learn how to classify objects by analyzing a set of instances (already solved cases) whose classes are known. Instances are typically...
represented as attribute-value vectors. Learning input consists
of a set of such vectors, each belonging to a known class, and
the output consists of a mapping from attribute values to
classes. This mapping should accurately classify both the
given instances and other unseen instances.

Decision tree learning, used in data mining and machine
learning, uses a decision tree as a predictive model which
maps observations about an item to conclusions about the
item's target value. More descriptive names for such tree
models are classification trees or regression trees. In these
tree structures, leaves represent classifications and branches
represent conjunctions of features that lead to those
classifications.

In decision theory and decision analysis, a decision tree is a
graph or model of decisions and their possible consequences,
including chance event outcomes, resource costs, and utility.
It can be used to create a plan to reach a goal. Decision trees
are constructed in order to help with making decisions. A
decision tree is a special form of tree structure. Another use of
trees is as a descriptive means for calculating conditional
probabilities.

Decision trees [12] is formalism for expressing such
mappings and consists of tests or attribute nodes linked to two
or more sub-trees and leafs or decision nodes labeled with a
class which means the decision. A test node computes some
outcome based on the attribute values of an instance, where
each possible outcome is associated with one of the sub-trees.
An instance is classified by starting at the root node of the
tree. If this node is a test, the outcome for the instance is
determined and the process continues using the appropriate
sub-tree. When a leaf is eventually encountered, its label gives
the predicted class of the instance.

Evolutionary algorithms are adaptive heuristic search
methods which may be used to solve all kinds of complex
search and optimization problems. They are based on the
evolutionary ideas of natural selection and genetic processes
of biological organisms. Evolutionary algorithms are able to
evolve solutions to real-world problems, if they have been
suitably encoded. They are often capable of finding optimal
solutions even in the most complex of search spaces or at least
they offer significant benefits over other search and
optimization techniques.

The traditional decision trees' induction methods contain
several disadvantages. In this paper the power of evolutionary
algorithms to induct the decision trees is used. Evolutionary
decision support model that evolves decision trees in a multi-
population genetic algorithm SAEDT: self-adapting
evolutionary decision trees [13] is used.

Many experiments have shown the advantages of such
approach over the traditional heuristic approach for building
decision trees, which include better generalization, higher
accuracy, possibility of more than one solution, efficient
approach to missing and noisy data, etc.

In SAEDT algorithm [13] individuals are presented like
directly as decision trees. All intermediate solutions are
feasible, no information is lost because of conversion between

<table>
<thead>
<tr>
<th>TABLE I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classification Tasks</td>
</tr>
<tr>
<td>Training observation</td>
</tr>
<tr>
<td>Test Observations</td>
</tr>
<tr>
<td>Predictors</td>
</tr>
<tr>
<td>Classes</td>
</tr>
<tr>
<td>Majority Class</td>
</tr>
<tr>
<td>% misclassified if Majority Class is used as Predicted Class</td>
</tr>
</tbody>
</table>

IV. DETECTING MODULE DEFECTS USING DECISION TREES

In order to test decision trees in predicting potentially
modules that contain defects a real dataset are used the dataset
[14] contains 2109 software modules. A set of 21 attributes,
containing various software complexity measures and metrics
have been used for each software module.

The paper analyzed the decision trees build with this software metrics.

The 21 attributes are described in the theoretical chapter of the paper. These attributes are:

1. loc - number of McCabe's line count of code
2. v(g) - number of McCabe "cyclomatic complexity"
3. ev(g) - number of McCabe "essential complexity"
4. iv(g) - number of McCabe "design complexity"
5. n - number of Halstead total operators + operands
6. v - Halstead "volume"
7. l - Halstead "program length"
8. d - Halstead "difficulty"
9. i - Halstead "intelligence"
10. e - Halstead "effort"
11. b - Halstead
12. t - Halstead's time estimator
13. IOCCode - Halstead's line count
14. IOComment - Halstead's count of lines of comments
15. IOBlank - Halstead's count of blank lines
16. IOCCodeAndComment – Number of code e comment
17. uniq_Op - number of unique operators
18. uniq_Opnd – number of unique operands
19. total_Op - total operators
20. total_Opnd - total operands
21. branchCount of the flow graph

From all 2099 modules 2089 have been randomly selected for the training set, and the remaining 529 modules has been selected for the testing set. Several decision trees have been induced for predicting modules with defects. The results of classification using induced decision tree is presented below:

Confusion Matrix for training data is listed below for training and validation:

<table>
<thead>
<tr>
<th>True class</th>
<th>Predicted Class</th>
<th>Predicted Class</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>1607</td>
<td>4</td>
<td>1611</td>
</tr>
<tr>
<td>TRUE</td>
<td>204</td>
<td>89</td>
<td>293</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1811</td>
<td>93</td>
<td>1904</td>
</tr>
</tbody>
</table>

Confusion Matrix for test data is listed below for training and validation:

<table>
<thead>
<tr>
<th>True class</th>
<th>Predicted Class</th>
<th>Predicted Class</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>166</td>
<td>6</td>
<td>172</td>
</tr>
<tr>
<td>TRUE</td>
<td>30</td>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>TOTAL</td>
<td>196</td>
<td>9</td>
<td>205</td>
</tr>
</tbody>
</table>

The accuracy of test data is 83.44 %. The accuracy of training data is 89.08 %.

The decision rules are:

Rule0  Problems = FALSE
Rule1  IF loComment >= 22 THEN Problems = TRUE
Rule2  IF loComment < 22 AND uniq_op >= 29 THEN Problems = TRUE
Rule3  IF Loc >= 286 THEN Problems = TRUE
Rule4  IF Loblank >= 35 THEN Problems = TRUE
Rule5  IF uniq_op >= 60 THEN Problems = TRUE
Rule6  IF v(g) >= 19 THEN Problems = FALSE
Rule7  IF b >= 0.74 THEN Problems = TRUE
Rule8  IF d >= 38.54 THEN Problems = TRUE
Rule9  IF uniq_op >= 59 THEN Problems = TRUE
Rule10 IF LOCEC >= 4 THEN Problems = FALSE
Rule11 IF uniq_op < 24 THEN Problems = FALSE
Rule12 IF d >= 27.42 THEN Problems = FALSE
Rule13 IF d >= 27.12 THEN Problems = TRUE
Rule14 IF uniq_op >= 21 THEN Problems = TRUE
Rule15 IF v(g) >= 15 THEN Problems = TRUE
Rule16 IF loCode >= 4 THEN Problems = TRUE
Rule17 IF i >= 32.63 AND n < 176 THEN Problems = FALSE
Rule18 IF tot_op >= 197 THEN Problems = TRUE
Rule19 IF n >= 176 THEN Problems = TRUE
Rule20 IF loCode < 4 THEN Problems = FALSE
Rule21 IF Loblank >= 1 THEN Problems = FALSE
Rule22 IF Loblank >= 16 THEN Problems = TRUE
Rule23 IF uniq_op >= 22 THEN Problems = TRUE
Rule24 IF n < 10 THEN Problems = FALSE
Rule25 IF Loc < 39 AND Loc >= 7 THEN Problems = FALSE
Rule26 IF loCode >= 86 THEN Problems = TRUE
Rule27 IF tot_oper >= 68 THEN Problems = TRUE
Rule28 IF loComment >= 16 THEN Problems = TRUE
Rule29 IF Loc >= 59 THEN Problems = FALSE

The term Problems is false if module has defects and is true if the module has no defects.

V. CONCLUSION AND FUTURE WORK

In this paper some results with software metrics used for detecting if a software module has or not has defects are presented. The proposed approach is considered to be useful in order to detecting defects in other data set. The decision rules are useful for develop new rules in software defect identification.

The future work will be to use another software metrics for detecting defects and for quality evaluation of a software metrics using decision trees.

The application of the rules to another data set can classify the module.

Decision trees are very powerful tools for classifying the software module using software metric. We develop new algorithm in order to classify using some new metrics.

REFERENCES


