Using Neural Networks In Software Metrics

Dr. Eng. Math. Ion I. Bucur
MSc. Eng. Nicolae Begnescu
University "Politehnica" of Bucharest
nbegnescu@gmail.com

Abstract

Software metrics provide effective methods for characterizing software. Metrics have traditionally been composed through the definition of an equation, but this approach is limited by the fact that all the interrelationships among all the parameters be fully understood. Derivation of a polynomial providing the desired characteristics is a substantial challenge.
In this paper instead of using conventional methods for obtaining software metrics, we will try to use a neural network for that purpose. Experiments performed in the past on two widely known metrics, McCabe and Halstead, indicate that this approach is feasible.

1. Introduction

As software engineering matures into a true engineering discipline, there is an increasing need for a corresponding maturity in repeatability, assessment, and measurement —both of the processes and of the artifacts associated with software. Repeatability of process is inherent to a true engineering discipline.

Repeatability of artifact takes natural form in the notion of software reuse, whether of code or of some other artifact resulting from a development or maintenance process. Accurate assessment of a component's quality and reusability are critical to a successful reuse effort. Components must be easily comprehensible, easily incorporated into new systems, and behave as anticipated in those new systems.
We will try to determine whether neural networks can model known software metrics. If they can, then neural networks can also serve as a tool to create new metrics. Establishing a set of measures raises questions of coverage (whether the metric covers all features), weightings of the measures, accuracy of the measures, and applicability over various application domains.
The appeal of a neural approach lies in a neural network's ability to model a function without the need to have knowledge of that function, thereby providing an opportunity to provide an assessment in some form, even if it is as simple as *this* component is reusable, and *that* component is not.

If all we seek is an assessment of a component, a properly trained neural network produces results comparable to a traditional algorithmic implementation of a multi-variable polynomial. In fact, while much of the research in metrics is concerned with the derivation of the polynomial, the result of evaluating that polynomial is frequently the true goal of the research.
In [1] has been demonstrated that neural networks are capable of modeling both the linearity of the McCabe metric and the more complex Halstead measure with a reasonable amount of accuracy, and hence applicable to metrics of unknown simplicity or complexity, such as a reusability metric.

2. Popular software metrics

There are currently many different metrics for assessing software. Metrics may focus on lines of code, complexity [2, 3], volume [4], or cohesion [5, 6] to name a few. Among the many metrics (and their variants) that exist, the McCabe and Halstead metrics are probably the most widely known, frequently appearing in introductory material on the subject. More current and complete coverage of this area appear in work such as [12].

Traditionally, software metrics are generated by extracting values from a program and substituting them into an equation. In certain instances, equations may be merged together using some weighted average scheme. This approach works well for simple metrics, but as our models become more sophisticated, deriving metrics with equations becomes harder. The traditional process requires the developer to completely understand the relationship among all the variables in the proposed metric.
This demand on a designer understands of a problem limits metric sophistication (i.e., complexity). One reason why it is so hard to develop reuse metrics, for example, is that no one completely understands "design for reuse" issues.

The goal then is to find alternative methods for generating software metrics. Deriving a metric using a neural network has several advantages. The developer need only to determine the endpoints (inputs and output) and can disregard (to an extent) the path taken. Unlike the traditional approach, where the developer is saddled with the burden of relating terms, a neural network automatically creates relationships among metric terms. Traditionalists might argue that you must fully understand the nuances among terms, but full understanding frequently takes a long time, particularly when there are numerous variables involved.

2.1   Halstead metric

Halstead`s Metric is a linguistic one. It is based on arguments derived from common sense, information theory and psychology.
The metric is based on two easily measured properties of the program, which are:
* $n1$ = the number of distinct operators in the program (keywords).
* $n2$ = the number of distinct operands in the program (data objects).
Halstead metric defines Program Length, by the following:

H = n1 *log n1 + n2 *log n2

When calculating Halstead length, paired operators such as BEGIN .. END, DO .. UNTIL , FOR .. NEXT are treated as a single operator.
Also the following properties can be counted easily:
* $N1$ = actual number of operators
* $N2$ = actual number of operands
The Actual Halstead Length is $N = N1 + N2$.
Halstead length may be computed using the $n1$ and $n2$ parameters (the sum of which Halstead defines as the program's vocabulary), even though the program has not been written yet.
For example : if a program is written using 20 keywords and uses a total of 30 data objects, its Halstead length equals 233.6, and it has been empirically observed that this estimates compares with the actual length quite closely. This even happens to hold when the program is subdivided into modules (subroutines).
Halstead Metric gives birth to a bug prediction formula:

B = (N1 + N2) log(n1 + n2) / 3000

For instance, a program that accesses 75 data objects a total of 1300 times and uses 150 operators a total of 1200 times, should be expected to have about 6.5 bugs.

In the addition to the above equations, Halstead also derived equations to estimate the programming effort, measured in time units, and the calculated results seem to correlate with the experience as well.

The time prediction is nonlinear, for instance:

- 120 - statement program should take 2.4 hours
- 1000 - statement program should take 230 hours
- 2000 - statement should take 1023 hours

2.2 McCabe's metric. McCabe's Cyclomatic Complexity

McCabe's metric is based on determining how complicated the control flow a program (procedure or function) is. The control flow is represented as directed graph.

Nodes represent one or more procedural statements. Each conditional instruction is mapped to a distinct node.

Edges (arcs) indicate potential flow from one node to another. An edge must terminate at a node. A predicate node is a node with two or more edges emanating from it.
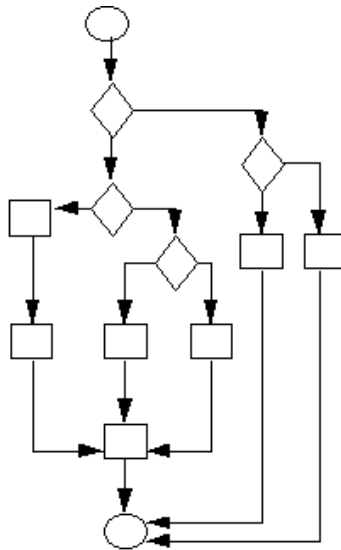

Figure 1. Control Flowgraph of the program P

Let G be the control Flowgraph of the program P, G has e edges and n nodes, then V(P)=e-n+2, V(P) is the number of linearly independent paths in G.

We could simplify, if d is the number of decision nodes in G then V(P)=d+1.

2.3 COCOMO

The original COCOMO Model has now been superseded by COCOMO 2. However, a brief review of the original COCOMO model provides insight into the evolution of software estimation approaches. This paper has been excerpted from SEPA, 4/e.

The hierarchy of COCOMO models takes the following form:

- Model 1. The Basic COCOMO model is a static, single-valued model that computes software development effort (and cost) as a function of program size expressed in estimated lines of code (LOC).

- Model 2. The Intermediate COCOMO model computes software development effort as a function of program size and a set of "cost drivers" that include subjective assessments of product, hardware, personnel and project attributes.
- Model 3. The Advanced COCOMO model incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.

To illustrate the COCOMO model, we present an overview of the Basic and Intermediate versions.

The COCOMO models are defined for three classes of software projects. Using Boehm's terminology these are: (1) organic mode – relatively small, simple software projects in which small teams with good application experience work to a set of less than rigid requirements (e.g., a thermal analysis program developed for a heat transfer group); (2) semi-detached mode – an intermediate (in size and complexity) software project in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements (e.g., a transaction processing system with fixed requirements for terminal hardware and data base software); (3) embedded mode – a software project that must be developed within a set of tight hardware, software and operational constraints (e.g., flight control software for aircraft).

The Basic COCOMO equations take the form:

$$E = a_b KLOC^{b_b}$$

$$D = c_b E^{d_b}$$

where E is the effort applied in person-months, D is the development time in chronological months and KLOC is the estimated number of delivered lines of code for the project (express in thousands). The coefficients a_b and c_b and the exponent b_b and d_b are given in Table 1.

Software Project	a_b	b_b	c_b	d_b
organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
embedded	3.6	1.20	2.5	0.32

Table 1. Basic COCOMO Model

The basic model is extended to consider a set of "cost driver attributes" that can be grouped into four major categories:

1. *Product attributes*
a. required software reliability
b. size of application data base
c. complexity of the product
2. *Hardware attributes*
a. run-time performance constraints
b. memory constraints
c. volatility of the virtual machine environment
d. required turnaround time

3. *Personnel attributes*

a. analyst capability
b. software engineer capability
c. applications experience
d. virtual machine experience
e. programming language experience
4. *Project attributes*
a. use of software tools
b. application of software engineering methods
c. required development schedule

Each of the 15 attributes is rated on a 6 point scale that ranges from "very low" to "extra high" (in importance or value). Based on the rating, an effort multiplier is determined from tables published by Boehm, and the product of all effort multipliers results is an *effort adjustment factor* (EAF). Typical values for EAF range from 0.9 to 1.4.

The intermediate COCOMO model takes the form:

$$E = a_i KLOC^{b_i} * EAF$$

where *E* is the effort applied in person-months and *KLOC* is the estimated number of delivered lines of code for the project. The coefficient a_i and the exponent b_i are given in Table 2.

Software project	a_i	b_i
organic	3.2	1.05
Semi-detached	3.0	1.12
embedded	2.8	1.20

Table 2. Intermediate COCOMO model

COCOMO represents a comprehensive empirical model for software estimation. However, Boehm's own comments about COCOMO (and by extension all models) should be heeded:

Today, a software cost estimation model is doing well if it can estimate software development costs within 20% of actual costs, 70% of the time, and on its own turf (that is, within the class of projects to which it has been calibrated ... This is not as precise as we might like, but it is accurate enough to provide a good deal of help in software engineering economic analysis and decision making.

3. Neural networks and the metrics modeling using neural networks

Neural networks by their very nature support modeling. In particular, there are many applications of neural network algorithms in solving classification problems, even where the classification boundaries are not clearly defined and where multiple boundaries exist and we desire the best among the alternatives. It seems natural then to use a neural network in classifying software.

There were two principle criteria determining which neural network to use. First, we needed a supervised neural network, since the answers are known. Second, the network needed to be able to classify.

The back-propagation algorithm meets both of these criteria. It works by calculating a partial first derivative of the overall error with respect to each weight, taking small steps down a gradient. However, a major problem with the back-propagation algorithm is that it is exceedingly slow to converge. Fahlman developed the quickprop algorithm as a way of using the higher-order derivatives in order to take advantage of the curvature.

The quickprop algorithm uses second order derivatives in a fashion similar to Newton's method. From previous experiments we found the quickprop algorithm to clearly outperform a standard back-propagation neural network. While an argument could be made for employing other types of neural models, due to the linear nature of several metrics, we chose quickprop.

The first step is determining whether a neural network can model existing metrics, in this case McCabe and Halstead. These two were chosen not from a belief that they are particularly good measures, but rather because they are widely known, public domain programs exist to generate the metric values, and the fact that the McCabe and Halstead metrics are representative of major metric domains (complexity and volume, respectively).

Finally, programs from several distinct application domains (e.g., abstract data types, program editors, numeric utilities and system oriented programs) were included in the test suite to ensure variety.

The neural network is trained with required data types and the main objectives are software classification after training.

4. Conclusions

The experimental results clearly indicate that a neural network approach for modeling metrics is feasible. In all experiments [1] the results corresponded well with the actual values calculated by traditional methods. Both the data set and the neural network architecture reached performance saturation points in the McCabe metric.

In the Halstead experiment, the fact that the results oscillated over the actual-calculated line indicate that the neural network was attempting to model the desired values. Adding more training vectors, especially ones containing larger values, would smooth out the oscillation.

5. References

[1] Boetticher, G., Srinivas, K., Eichmann D., *A Neural Net-Based Approach to Software Metrics*, *Proceedings of the Fifth International Conference on Software Engineering and Knowledge Engineering*, San Francisco, CA, June 16-18, 1993, pages 271-274.

[2] McCabe, T.J., "A complexity measure," *IEEE Transactions on Software Engineering,* vol. SE-2, no.4, pp. 308-320, Dec. 1976.

[3] McCabe, T.J., "Design Complexity Measurement and Testing," *Communications of the ACM,* vol. 32, no. 12, pp. 1415-1425, December 1989.

[4] Halstead, M.H., *Elements of Software Science,* North-Holland (Elsevier Computer Science Library), New York, 1977.

[5] Emerson, T. J., "A Discriminant Metric for Module Cohesion," *Proc. 7th International Conference on Software Engineering,* Los Alamitos, California, IEEE Computer Society, 1984, pp. 294-303.

[6] Emerson, T. J., "Program Testing, Path Coverage, and the Cohesion Metric," *Proc. of the 8th Annual Computer Software and Applications Conference,* IEEE Computer Society, pp. 421-431.

[7] Zuse, H., *Software Complexity: Measures and Methods,* Walter de Gruyter, Berlin, 1991.

[8] Conn, R., "The Ada Software Repository and Software Reusability," *Proc. of the Fifth Annual Joint* Conference on Ada Technology and Washington Ada *Symposium,* 1987, pp. 45-53.

[9] Fahlman, S. E., *An Empirical Study of Learning Speed in Back-Propagation Networks,* Tech Report CMUCS-88-162, Carnegie Mellon University, September, 1988.

[10] Fahlman, S. E. and Lebiere, M., *The Cascade-Correlation Learning Architecture,* Tech Report CMU-CS-90-100, Carnegie Mellon University, August 1991.

[11] Hertz, J., Krogh A., Palmer, R. G., *Introduction to the Theory of Neural Computation,* Addison Wesley, New York, 1991.

[12] Li, H.F. and Cheung, W.K., "An Empirical Study of Software Metrics," *IEEE Transactions on Software Engineering,* vol. 13, no. 6, pp. 697-708, June 1987.

[13] Lippmann, R. P. "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine,* pp. 4-22, April 1987.

[14] Zuse, H., *Software Complexity: Measures and Methods,* Walter de Gruyter, Berlin, 1991.

[15] Boehm, B., *Software Engineering Economics,* Prentice-Hall, 1981.

[16] Putnam, L. and W. Myers, *Measures for Excellence,* Yourdon Press, 1992.