A Neural Net-Based Approach to Software Metrics*

G. Boetticher, K. Srinivas, D. Eichmann†

Software Reuse Repository Lab
Department of Statistics and Computer Science
West Virginia University
{gdb, srini, eichmann}@cs.wvu.wvnet.edu

Correspondence to:

David Eichmann
SoRReL Group
Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506

email: eichmann@cs.wvu.wvnet.edu
fax: (304) 293-2272

* Submitted to the 4th International Conference on Tools With Artificial Intelligence, November 10-13, 1992, Arlington, Virginia.

† This work was supported in part by NASA as part of the Repository Based Software Engineering project, cooperative agreement NCC-9-16, project no. RICIS SE.43, subcontract no. 089.

Abstract

Software metrics provide an effective method for characterizing software. Metrics have traditionally been composed through the definition of an equation. This approach is limited by the fact that all the interrelationships among all the parameters be fully understood. This paper explores an alternative, neural network approach to modeling metrics. Experiments performed on two widely accepted metrics, McCabe and Halstead, indicate that the approach is sound, thus serving as the groundwork for further exploration into the analysis and design of software metrics.

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 — of both the processes and the artifacts associated with software. 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 comprehendible, easily incorporated into new systems, and behave as anticipated in those new systems. Unfortunately, no consensus currently exists on how to go about measuring a component's reusability. One reason for this is our less than complete understanding of software reuse, yet obviously it is useful to measure something that is not completely understood.

This paper describes a preliminary set of experiments 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.

We begin in section 2 by describing two of the more widely accepted software metrics and then in section 3 briefly discuss various neural network architectures and their applicability. Section 4 presents the actual experiment. We draw conclusions in section 5, and present prospects for future work in section 6.

2 – Software metrics

There are currently many different metrics for assessing software. Metrics may focus on lines of code, complexity [7, 8], volume[5], or cohesion [2, 3] to name a few. Among the many metrics (and their variants) that exist, the McCabe and Halstead metrics are probably the most widely recognized.

The McCabe metric measures the number of control paths through a program [7]. Also referred to as cyclomatic complexity, it is defined for a program G as [8]:

$$v(G) = \text{number of decision statements} + 1$$

assuming a single entry and exit for the program, or more generally as

$$v(G) = \text{Edges} - \text{Nodes} + 2 \cdot \text{Units}$$

where Edges, Nodes, and Units correspond respectively to the number of edges in the program flow graph, the number of nodes in the program flow graph, and the number of units (procedures and functions) in the program.

The Halstead metric measures a program's volume. There are actually several equations associated with Halstead metrics. Each of these equations is directly or indirectly derived from the following measures:

- n_1 the number of unique operators within a program (operators for this experiment include decision, math, and boolean symbols);
- N_1 the total number of operators within a program;
- n_2 the number of unique operands in a program (including procedure names, function names, variables (local and global), constants and data types); and
- N_2 the total number of operands in a program.

The measurements for a program are equal to the sum of the measurements for the individual modules.

Based on these four parameters, Halstead derived a set of equations, which include the follow-

ing (in which we are most interested):

Actual Length: $N = N_1 + N_2$

Program Volume: $V = N \cdot \log_2(n)$

Program Effort: $E = V / (2 \cdot n_2)$

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, modeling 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's understanding of a problem limits metric sophistication (i.e., complexity). For example, one reason why it is so hard to develop reuse metrics is that no one completely understands "design for reuse" issues.

The goal then is to find alternative methods for generating software metrics. Modeling 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.

We establish neural networks as a method for modeling software metrics by showing that we can model two widely accepted metrics, the McCabe and the Halstead metrics.

3 – 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. It seems only natural then to use a neural network in classifying software.

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

The back-propagation algorithm meets both of these criteria [9]. It works by calculating a partial first derivative of the overall error with respect to each weight. The back-propagation ends up taking infinitesimal steps down the gradient [4]. However, a major problem with the back-propagation algorithm is that it is exceedingly slow to converge [7]. Fahlman developed the quickprop algorithm as a way of using the higher-order derivatives in order to take advantage of the curvature [4]. 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 to ensure stability and continuity in our experiments when we moved to more complex domains in future work.

4 – Modeling Metrics with Neural Networks

As mentioned earlier, the goal of the experiment is to determine whether a neural network could be used as a tool to generate a software metric. In order to determine whether this is possible, the first step is to determine 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 accepted, 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).

Since our long term goal of the experiment is to determine whether a neural network can be used to model software reusability metrics, Ada, with its support for reuse (generics, unconstrained arrays, etc.) seemed a reasonable choice for our domain language. Furthermore, the ample supply of public domain Ada software available from repositories (e.g., [1]) provides a rich testbed from which to draw programs for analysis.

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

We ran three distinct experiments. The first experiment modeled the McCabe metric on single procedures, effectively fixing the unit variable at 1. The second experiment extended the first to the full McCabe metric, including the unit count in the input vector, and using complete packages as test data. The third experiment used the same test data in modeling the Halstead metric, but a different set of training vectors.

4.1 – Experiment A: A Neural McCabe metric for Procedures

In this experiment all vectors had a unit value of one, so the unit column was omitted. In building both the training and test sets all duplicate vectors and stub vectors (i.e., statements of the form "PROCEDURE XYZ IS SEPARATE") were removed. The input for all trials in this experiment contained 26 training vectors and 8 test vectors (the sets were disjoint). Each training vector corresponded to an Ada procedure and contained three numbers, the number of edges, the number of nodes, and the cyclomatic complexity value.

The goals of this first experiment were to establish whether a neural network can be used to model a very simple metric function (the McCabe metric on a procedure basis) and to examine the influence neural network architecture has on the results. The input ran under 6 different architectures: 2-1 (two input layers, no hidden layers, and one output layer), 2-1-1 (two input layers, one

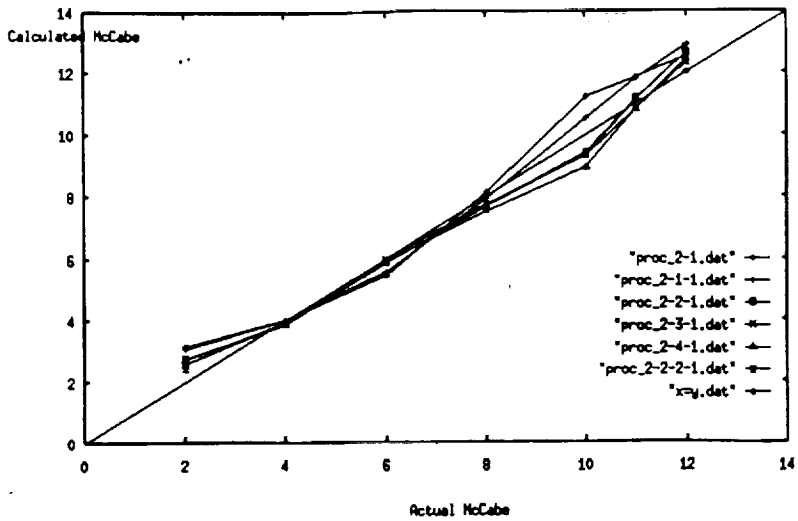

Figure 1: McCabe Results for Single Procedures

hidden layer, and one output layer), 2-2-1, 2-3-1, 2-4-1, and 2-2-2-1. In order to examine the impact of architecture, other parameters remained constant. Alpha, the learning rate, was set to 0.55 throughout the trials. An asymsigmoid squashing function (with a range of 0 to +1) was used to measure error. Finally, each trial was examined during epochs 1000, 5000, and 25,000. Figure 1 presents the results of these trials. In the graph, the neural calculated values are plotted against the actual values for the metric at 25,000 epochs*. In an ideal situation, all lines would converge to x = y, indicating an exact match between the actual McCabe metric (calculated using the traditional equation) on the x-axis, and the neural calculated McCabe metric on the y-axis.

This experiment provides good results considering the minimal architectures used. Most points tend to cluster towards the actual-calculated line regardless of architecture selection. This suggests that more complex architectures would not provide dramatic improvements in the results.

Considering that only 26 training vectors were used, the results were quite favorable, and we moved on to the next experiment.

* In fact, all figures in the paper correspond to the results following 25,000 epochs.

4.2 – Experiment B: A Neural McCabe Metric for Packages

The second experiment modeled the McCabe metrics on a package body basis. Changes in data involved the addition of another input column corresponding to the number of units (the number of procedures in an Ada package) and the selection of a slightly different set of training vectors, chosen to ensure coverage of the added input dimension.

The experiment ranged over five different architectures (3-3-1, 3-5-1, 3-10-1, 3-5-5-1, and 3+5-5-1 (hidden layers are connected to all previous layers)) and four training sets (16, 32, 48, and 64 vectors). Each smaller training set is a subset of the larger training set, and training and test sets were always disjoint. Alpha remained constant at 0.55 throughout the trials. Once again, we used an asymsigmoid squashing function in every trial. Data was gathered at epochs 1000, 5000, and 25,000.

We selected vectors for the test suite to ensure variety both in the number of units in the program and in the nature of the program (number crunching programs tend to provide higher cyclomatic complexity values than I/O-bound programs). For a given package body, its cyclomatic complexity is equal to the sum of the cyclomatic complexities for all its procedures.

Some packages contained stub procedures. These stub procedures generate an edge value of zero and a node value of one and thus produce a cyclomatic complexity of 1. Stub procedures did not seem to adversely affect the training set.

The four figures below depict the results first when neural network architectures remain constant and training set size varies and second when training set size remains constant and neural network architectures vary.

As the training set increases, the results converge towards the x = y line, indicating a strong correspondence to the actual McCabe metric. This behavior occurs in all architectures; we show the 3-3-1 architecture in Figure 2, and the 3+5-5-1 architecture in Figure 3. Except for the initial

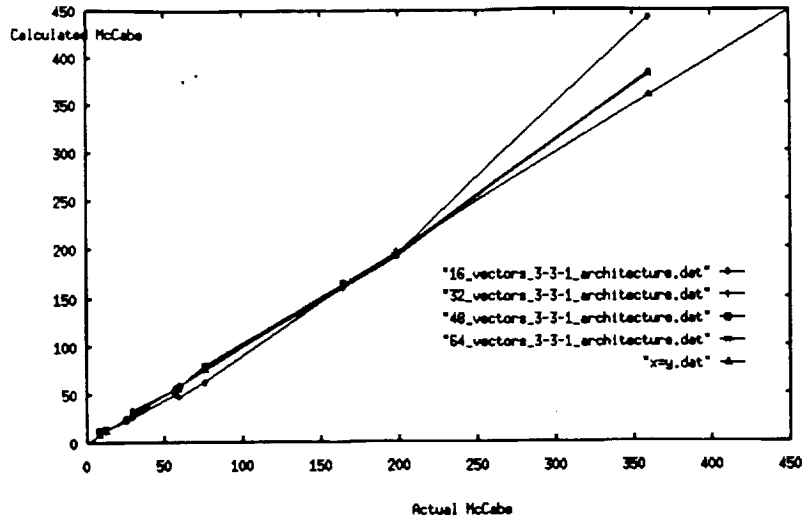

Figure 2: The 3-3-1 Architecture

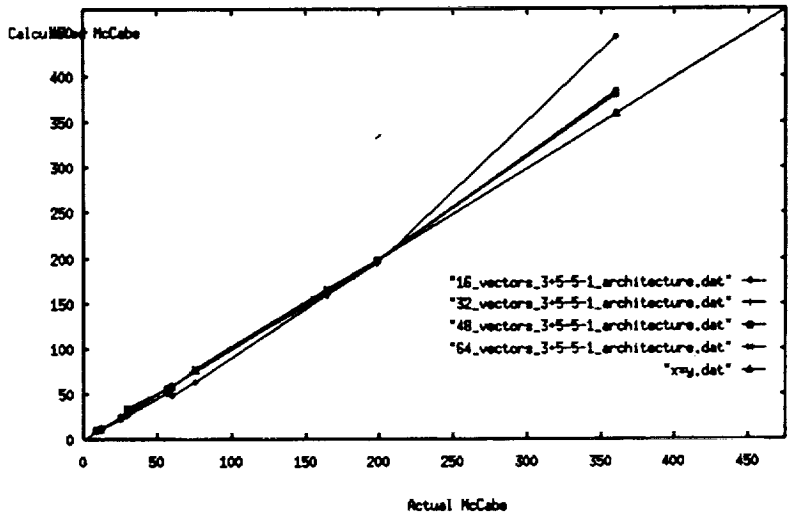

Figure 3: The 3+5-5-1 Architecture

improvement after 16 vectors, there is no significant improvement of results in the other three trials. This suggests that relatively low numbers of training vectors are required for good performance.

Furthermore, as shown in Figure 4 for 16 training vectors and Figure 5 for 64 training vectors, network architecture had virtually no effect on the results. These strong results are not surprising, given the linear nature of the McCabe metric.

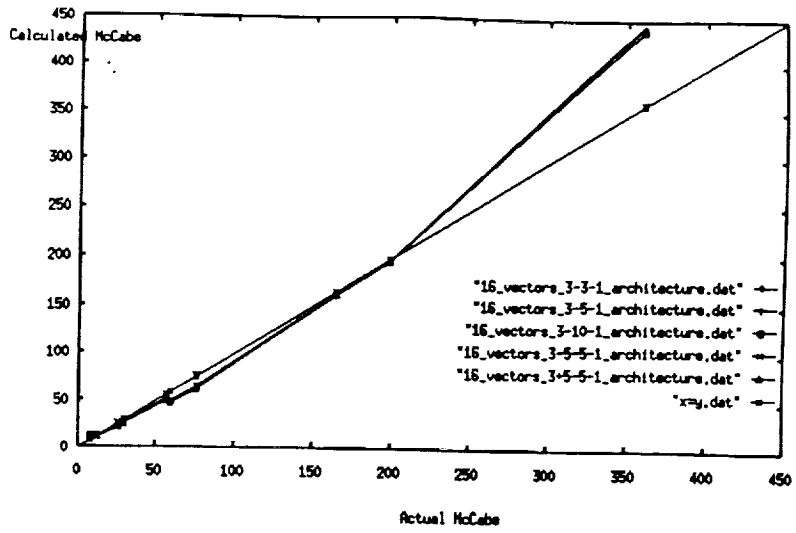

Figure 4: 16 Training Vectors for all Architectures

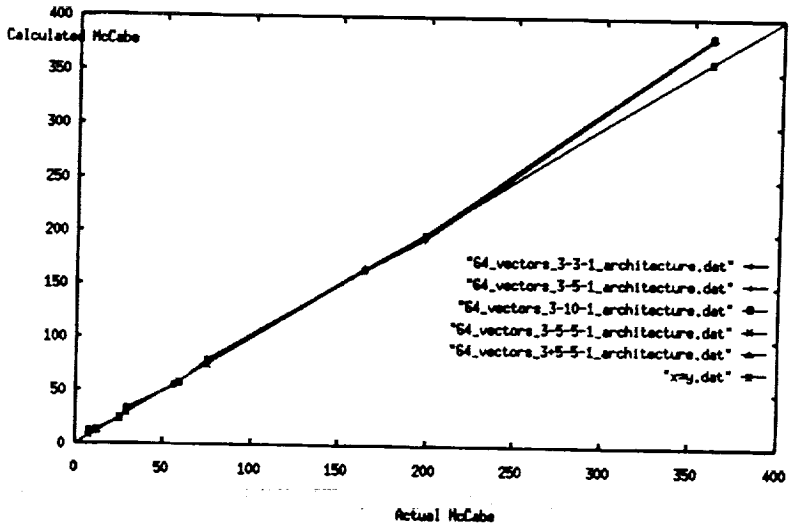

Figure 5: 64 Training Vectors for all Architectures

4.3 – Experiment C: A Neural Halstead Metric for Packages

Based upon the results of the first two experiments, we assumed for this experiment that if the experiment worked for packages, then it also worked for procedures, and further, that the increasing the number of training set vectors improves upon the results. Therefore, the focus of this experiment was on varying neural network architectures over a fixed-size training set.

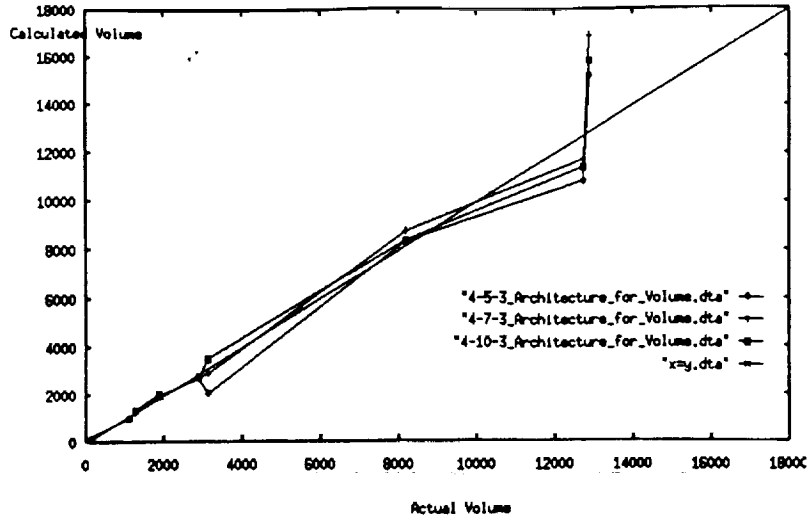

Figure 6: Volume Results, Broad Architectures

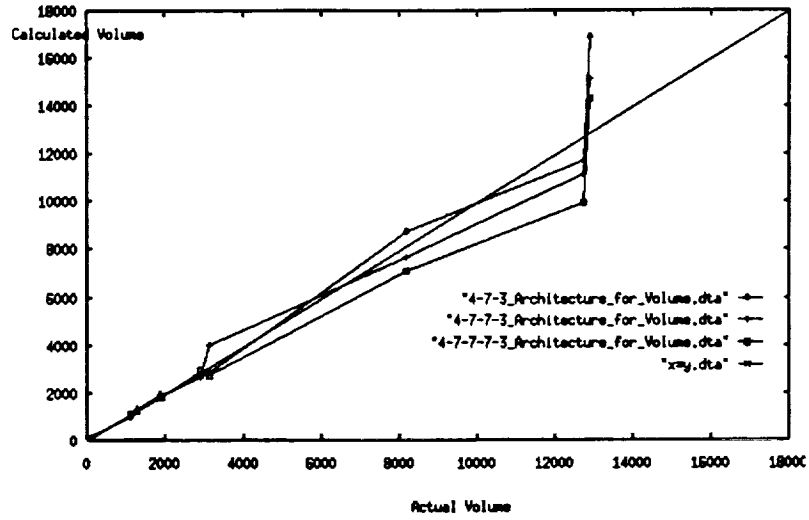

Figure 7: Volume Results, Deep Architectures

The experiment ranged over seven different neural network architectures broken into three groups: broad, shallow architectures (4-5-3, 4-7-3, and 4-10-3), narrow, deep architectures (4-7-7-3 and 4-7-7-7-3), and narrow, deep architectures with hidden layers that connected to all previous layers (4+7-7-3 and 4+7+7-7-3). We formed these three groups in order to discover whether there was any connection between the complexity of an architecture and its ability to model a metric.

Figures 6, 7, and 8 present the results for the Halstead volume for broad, deep, and connected

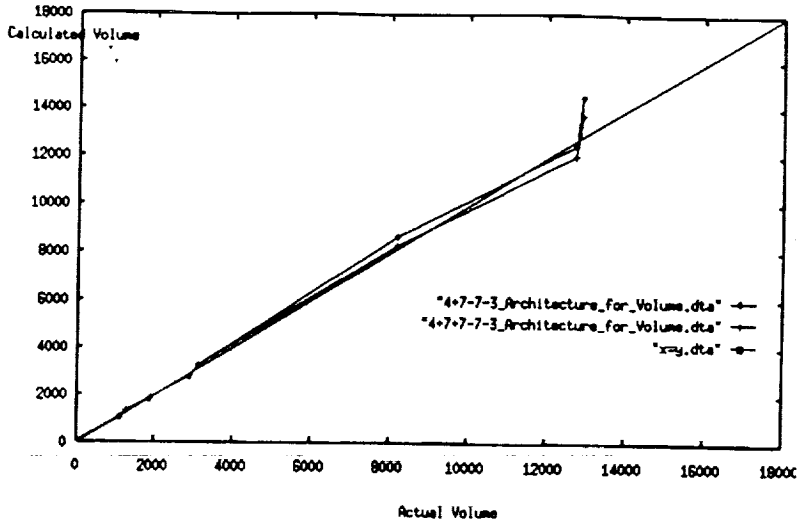

Figure 8: Volume Results, Connected Architectures

architectures, respectively. Note that both the broad and deep architectures do moderately well at matching the actual Halstead volume metric, but the connected architecture performs significantly better. Furthermore, there is no significant advantage for a five versus four layer connected architecture, indicating that connecting multiple layers may be a sufficient condition for adequately modeling the metric.

This pattern of performance also held for the Halstead length metric and the Halstead effort metric, so we show only the results for the connected architecture in Figure 9 and Figure 10, respectively.

5 – Conclusions

The experimental results clearly indicate that a neural network approach for modeling metrics is feasible. In all experiments 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

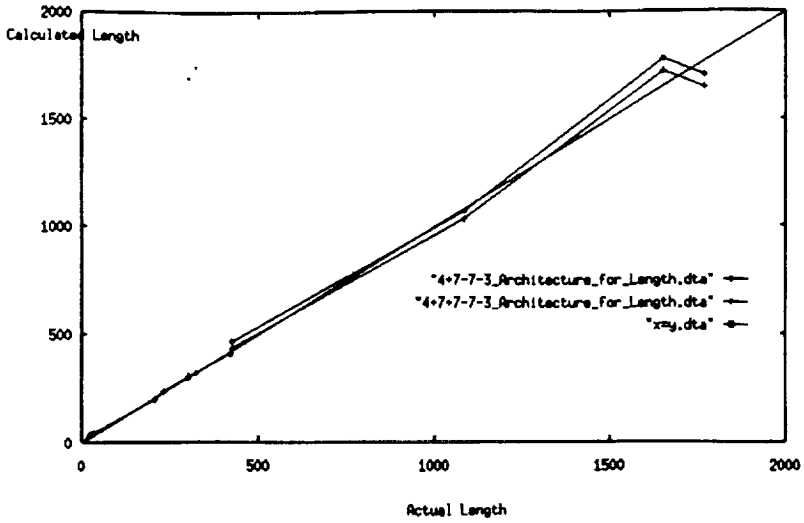

Figure 9: Length Results, Connected Architectures

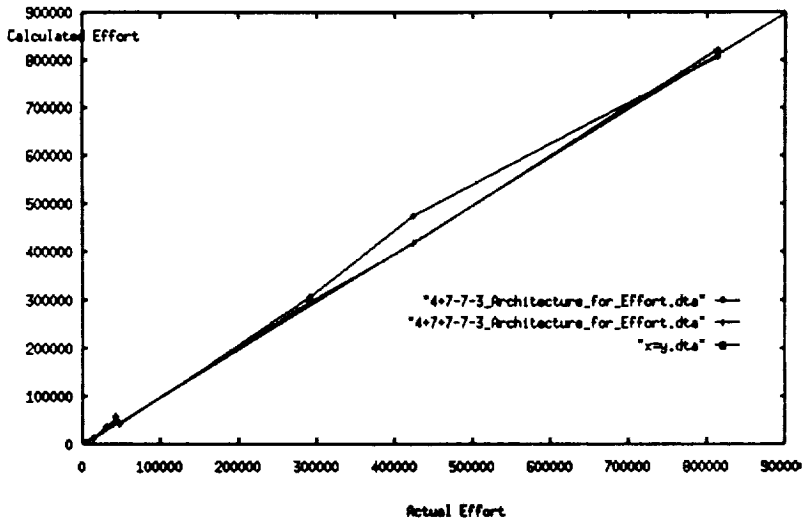

Figure 10: Effort Results, Connected Architectures

the oscillation.

6 – Future work

Applying this work to other existing metrics is an obvious extension, but we feel that the development of new metrics by applying neural approaches is much more significant. In particular, expanding this work to the development of a reusability metric offers great promise. Effective re-

use is only possible with effective assessment and classification. Since no easy algorithmic solutions currently exist, we've turned to neural networks to support the derivation of reusability metrics. Unsupervised learning provides interesting possibilities for this domain, letting the algorithm create its own clusters and avoiding the need for significant human intervention.

Coverage and accuracy are important aspects of developing a neural network to model a software reuse metric. McCabe and Halstead metrics are interesting and useful, but they do not provide coverage regarding reusability. We need to expand the number of parameters in the data set in order to provide adequate coverage with respect to reusability of a component. We also would like to improve the accuracy of answers by enlarging our data sets to include possibly hundreds of training set vectors. This will need to be a requirement when exploring more complex metric scenarios, and the cost of such extended training is easily borne over the expected usage of the metric.

Finally, it is possible to explore alternative neural network models. For example, the cascade correlation model [5] dynamically builds the neural network architecture, automating much of the process described here.

References

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

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

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

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

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

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

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

[8] Li, H.F. and Cheung, W.K., "An Empirical Study of Software Metrics," *IEEE Transactions on Software Engineering*, (13)6, June 1987, p. 697-708

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

[10] McCabe, T.J., "A complexity measure," *IEEE Transactions on Software Engineering*, (SE-2) 4, Dec. 1976, p. 308-320.

[11] McCabe, T.J., "Design Complexity Measurement and Testing," *Communications of the ACM*, (32)12, December 1989, p. 1415-1425.