NASA-CR-190641

Using Neural Networks in Software Repositories

p. 35

David Eichmann, ed.
West Virginia University Research Corporation

6/1/92

N92-31597

Unclas

G3/63 0115109

(NASA-CR-190641) USING NEURAL NETWORKS IN SOFTWARE REPOSITORIES (Research Inst. for Computing and Information Systems) 35 p

Cooperative Agreement NCC 9-16
Research Activity No. SE.43

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division

ricis

Research Institute for Computing and Information Systems
University of Houston-Clear Lake

TECHNICAL REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

Using Neural Networks in Software Repositories

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by David Eichmann, Kankanahalli Srinivas, and G. Boetticher of West Virginia University. Dr. E. T. Dickerson served as RICIS research coordinator.

Funding was provided by the Information Technology Division, Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Ernest M. Fridge, III of the Information Technology Division, Information Systems Directorate, NASA/JSC.

SoRReL

West Virginia University

Software Reuse Repository Lab

Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
(304) 293-3607   email: sorrel@cs.wvu.wvnet.edu

Using Neural Networks in Software Repositories

David Eichmann

June 1, 1992

Introduction

This report consists of two papers, written approximately ten months apart. The first is an exploration of the use of neural network techniques to improve the effectiveness of retrieval in software repositories. The second relates a series of experiments conducted to evaluate the feasibility of using adaptive neural networks as a means of deriving (or more specifically, *learning*) measures on software.

Taken together, these two efforts illuminate a very promising mechanism supporting software infrastructures – one based upon a flexible and responsive technology.

Neural Network-Based Retrieval from Software Reuse Repositories*

David Eichmann & Kankanahalli Srinivas
Department of Statistics & Computer Science
West Virginia University
Morgantown, WV 26506
eichmann/srini@a.cs.wvu.wvnet.edu

October 7, 1991

Overview. A significant hurdle confronts the software reuser attempting to select candidate components from a software repository – discriminating between those components without resorting to inspection of the implementation(s). We outline an approach to this problem based upon neural networks which avoids requiring the repository administrators to define a conceptual closeness graph for the classification vocabulary.

1 Introduction

Reuse has long been an accepted principle in many scientific disciplines. Biologists use established laboratory instruments to record experimental results; chemists use standardized measuring devices. Engineers design based upon the availability of components that facilitate product development. It is unreasonable to expect an electrical engineer to design and develop the transistor from first principles every time one is required.

Software engineers, however, are frequently guilty of a comparable practice in their discipline. The reasons for this are as varied as the environments in which software is developed, but they usually include the following:

*To appear in *Neural Networks and Pattern Recognition in Human Computer Interfaces*, R. Beale and J. Findlay (eds.), Ellis Horwood Ltd., West Sussex, UK, due out March, 1992.

- a lack of development standards;
- the *not invented here* syndrome;
- poor programming language support for the mechanical act of reuse; and
- poor support in identifying, cataloging, and retrieving reuse candidates.

The first three items involve organization mentality, and will not be addressed here.[1] We instead focus upon the final item in this list, the nature of the repository itself, and more specifically upon the mechanisms provided for classification and retrieval of components from the repository.

The complexity of non-trivial software components and their supporting documentation easily qualifies reuse as a "wicked" problem – frequently intractable in both description and solution. We describe an approach that we are currently exploring for making classification and retrieval mechanisms more efficient and natural for the software reuser. This approach centers around the use of neural networks in support of imprecise classification and querying.

2 The Problem

A mature software repository can contain thousands of components, each with its own specification, interface, and typically, its own *vocabulary*. Consider the signatures presented in Figures 1 and 2 for a stack of integers and a queue of integers, respectively.

$$\begin{aligned}
\text{Create: } &\Longrightarrow \text{Stack}\\
\text{Push: Stack} \times \text{Integer} &\Longrightarrow \text{Stack}\\
\text{Pop: Stack} &\Longrightarrow \text{Stack}\\
\text{Top: Stack} &\Longrightarrow \text{Integer}\\
\text{Empty: Stack} &\Longrightarrow \text{Boolean}
\end{aligned}$$

Figure 1: Signature of a Stack

[1]Concerning language support – there *are* languages which readily support reuse, but they must be available to the programmers. Consider for a moment the inertia exhibited by FORTRAN and COBOL in commercial data processing. The very existence of such large bodies of code in languages ill-suited for reuse acts as an inhibitor for the movement of organizations towards better suited languages.

$$\begin{gathered}\text{Create: } \Longrightarrow \text{Queue}\\ \text{Enqueue: Queue} \times \text{Integer} \Longrightarrow \text{Queue}\\ \text{Dequeue: Queue} \Longrightarrow \text{Queue}\\ \text{Front: Queue} \Longrightarrow \text{Integer}\\ \text{Empty: Queue} \Longrightarrow \text{Boolean}\end{gathered}$$

Figure 2: Signature of A Queue

These signatures are isomorphic up to renaming, and thus exemplify what we have come to refer to as the *vocabulary problem*. Software reusers implicitly associate distinct semantics with particular names, for example, pop and enqueue. Thus, by the choice of names, a component developer can mislead reusers as to the semantics of components, or provide no means of discriminating between components. Figure 3, for example, appears to be equally applicable as a signature for both stack and queue, primarily due to the neutral nature of the names used.

$$\begin{gathered}\text{Create: } \Longrightarrow \text{Sequence}\\ \text{Insert: Sequence} \times \text{Integer} \Longrightarrow \text{Sequence}\\ \text{Remove: Sequence} \Longrightarrow \text{Sequence}\\ \text{Current: Sequence} \Longrightarrow \text{Integer}\\ \text{Empty: Sequence} \Longrightarrow \text{Boolean}\end{gathered}$$

Figure 3: Signature of a Sequence

3 Software Classification

Retrieval mechanisms for software repositories have traditionally provided some sort of classification structure in support of user queries. Keyword-based retrieval is perhaps the most common of these classification structures, but keywords are ill-suited to domains with rich structure and complex semantics. This section lays out the principle representational problems in software classification and selected solutions to them.

3.1 Literary Warrant

Library scientists use *literary warrant* for the classification of texts. Representative samples drawn from the set of works generate a set of descriptive terms, which in turn generate a classification of the works as a whole. The adequacy of the classification system hinges a great deal on the initial choice of samples.

With appropriate tools, literary warrant in software need not restrict itself to a sample of the body of works. Rather, it can examine each of the individual works in turn, providing vocabularies for each of them. This may indeed be *required* in repositories where the component coverage in a particular area is sparse.

3.2 Conceptual Closeness

The vocabulary of terms built up through literary warrant typically contains a great deal of semantic overlap words whose meanings are the same, or at least similar. For instance, two components, one implementing a stack and the other a queue might both be characterized with the word insert, corresponding to push and enqueue, respectively, as discussed in section 2.

Synonym ambiguity is commonly resolved through the construction of a restricted vocabulary, tightly controlled by the repository administrators. Repository users must learn this restricted vocabulary, or rely upon the assistance of consultants already familiar with it. It is rarely the case, however, that the choice is between two synonyms. More typically it is between words which have similar, but distinct, meanings (e.g., insert, push, and enqueue, as above).

3.3 Algebraic Specification

While not really a classification technique, algebraic specification techniques (e.g., [GH78]) partially (and unintentionally) overcome the vocabulary problem through inclusion of behavioral axioms into the specification. The main objection to the use of algebraic specifications in reuse is the need to actually *write* and *comprehend* the specifications. The traditional examples in the literature rarely exceed the complexity of the above Figures. Also, algebraic techniques poorly address issues such as performance and concurrency.

A repository containing algebraic specifications depends upon the expertise of the reusers browsing the repository; small repositories are easily understood whereas it is unreasonable to require a reuser to examine all components in a large repository for suitability.

3.4 Basic Faceted Classification

Basic faceted classification begins by using domain analysis (aka literary warrant) "to derive faceted classification schemes of domain specific objects." The classifier not only derives terms for grouping, but also identifies a vocabulary that serves as the values that populate those groups. From the software perspective, the groupings, or *facets* become a taxonomy for the software.

Prieto-Díaz and Freeman identified six facets: function, object, medium, system type, functional area, and setting [PDF87]. Each software component in the repository has a value assigned for each of these facets. The software reuser locates software components by specifying facet values that are descriptive of the software desired. In the event that a given user query has no matches in the repository, the query may be relaxed by wild-carding particular facets in the query, thereby generalizing it.

The primary drawback in this approach is the flatness and homogeneity of the classification structure. A general-purpose reuse system might contain not only reusable components, but also design documents, formal specifications, and perhaps vendor product information. Basic faceted classification creates a single tuple space for all entries, resulting in numerous facets, tuples with many "not applicable" entries for those facets, and frequent wildcarding in user queries.

A number of reuse repository projects have incorporated faceted classification as a retrieval mechanism (e.g., [Gue87][Atk]), but they primarily address the vocabulary problem through a keyword control board, charged with creating a controlled vocabulary for classification.

Gagliano, et. al. computed conceptual closeness measures to define a semantic distance between two facet values [GOF$^+$88]. The two principle limitations to this approach are the static nature of the distance metrics and the lack of inter-facet dependencies; each of the facets had its own closeness matrix.

3.5 Lattice-Based Faceted Classification

Eichmann and Atkins extended basic faceted classification by incorporating a lattice as the principle structuring mechanism in the classification scheme [EA90]. As shown in Figure 4, there are two major sublattices making up the overall lattice.

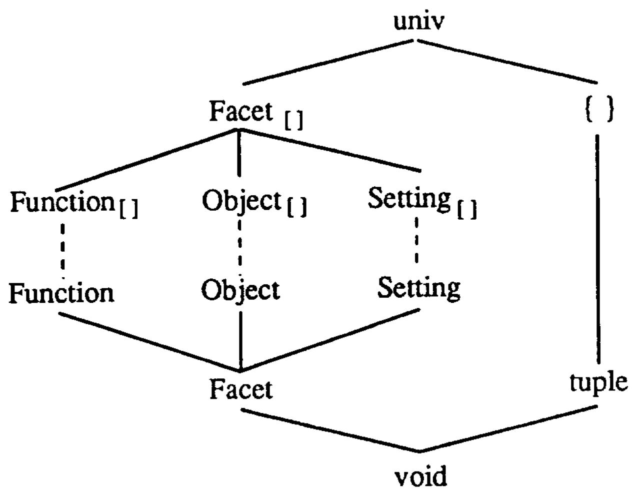

Figure 4: The Type Lattice

On the left is the sublattice comprised of sets of facet values (for clarity, shown here with only three facets), partially ordered by the subset relation. The *Facet*$_{[]}$ vertex in the lattice represents the empty facet set, while the *Facet* vertex represents the set of all facet values in the classification scheme. Each member of the power set of all facet values falls somewhere within this sublattice.

On the right is the tuple sublattice, containing facet set components, and partially ordered by the subtype relation [Eic89]. The vertex denotes the empty tuple. The *tuple* vertex denotes the tuple containing all possible facet components, with each component containing all the values for that facet. Adding facet values to a component or adding a new component to a tuple instance moves the tuple instance down through the lattice.

Queries to a repository supporting lattice-based faceted classification are similar to those to one supporting basic faceted classification, with two important distinctions – query tuples can mention as many or as few facets as the reuser wishes, thereby avoiding the need for wildcarding, and classifiers can similarly classify a given component with as many or as few facets as are needed for precise characterization of the component.

Lattice-based faceted classification avoids conceptual closeness issues through the specification of sets of facet values in the classification of components. If there are a number of semantically close facet values that all characterize the component, all are included in the facet instance for that component. This avoids the need to generate closeness metrics for facet values, but it also may result in reuser confusion about just what the component does.

3.6 Towards Adaptive Classification and Retrieval

The principle failing in the methods described so far is the static nature of the classification. Once a component has been classified, it remains unchanged until the repository administrators see fit to change it. This is unlikely to occur unless those same administrators closely track reuser retrieval success, and more importantly, retrieval *failure* – particularly in those cases where there are components in the repository matching reuser requirements, but those components were not identified during the query session.

Manual adjustment of closeness metrics becomes increasingly unreasonable as the scale of the repository increases. The number of connections in the conceptual graph is combinatorially explosive. The principle design goal in our work is the creation of an ***adaptive*** query mechanism – one capable of altering its behavior based upon implicit user feedback. This feedback appears in two guises; *failed queries,* addressed by widening the scope of the query; and *reuser refusals,* cases where candidate components were presented to the reuser, but not selected for retrieval. The lattice provides a nice structure for the former, but a different approach is required for the latter.

4 Our Approach

We are currently designing a new retrieval mechanism using previous work described in [EA90] as a starting point, and employing neural networks to address the vocabulary and refusal problems. The motivations behind using neural networks include:

- **Associative Retrieval from Noisy and Incomplete Cues:** Traditional methods for component retrieval are based on strict pattern matching methods such as unification. In other words, the query should contain exact information about the component(s) in the repository. Since exact information about components is usually not known, queries fail in cases where exact matching does not occur. Associative retrieval based on neural networks uses relaxation, retrieving components based on partial/approximate/best matches. This is sometimes referred to as *data fault tolerance* and is ideally suited for our problem domain.

- **Classification and Optimization by Adaptation:** In approaches using the conceptual closeness measure, the problem of defining correlations between various components and assigning a numerical correlation value rests upon the designer or the administrator of the repository. Designers idiosyncratically arrive at these correlations and their values, which may not be appropriate from the perspective of the software retriever/reuser. It is our belief that the best way to arrive at these correlations and their values is for the system to learn them in responding to user queries.

We also intend to use another adaptation strategy for optimizing the retrieval of similar repetitive queries. Since in most situations, reusers repeatedly issue similar queries, the system will adapt to these queries by weight adjustment. The weight adjustment will settle the relaxation process quickly in response to these repetitive queries and hence result in faster retrieval. The effect here is similar to that of *caching* frequently issued queries. Note, however, that once the system has learned that two concepts are conceptually close, we want it to remember this, irrespective of how often the reusers inquire about it.

- **Massive Parallelism:** The neurocomputing paradigm is characterized by *asynchronous, massively parallel, simple* computations. Since neural networks are massively parallel, retrieval from large repositories is possible, using the fast associative search techniques that are natural and inherent in these networks.

5 System Architecture

In this section, we describe some of the potential neural-network architectures and discuss their strengths and limitations in employing them for our task.

5.1 Hopfield Networks

These networks can be used as content-addressable or associative memories. Initially the weights in the network are set using representative samples from all the exemplar classes. After this initialization, the input pattern I is presented to the network. The network then iterates and converges to a output. This output represents the exemplar class which matches the input pattern best.

Although this network has many properties that are desirable for our system, some of the serious limitations in our context include:

1. The networks have limited capacity [Lip87] and may converge to novel spurious patterns.

2. They result in unstable exemplar patterns if many bits are shared among multiple exemplar patterns.

3. There are no algorithms to incrementally train these networks, i.e., to adjust the initial weights in a manner that creates a specific alteration in subsequent query responses. This is important for our application, since we seek an architecture capable of adapting over time to user feedback.

5.2 Supervised Learning Algorithms

Many good *supervised* learning algorithms exist, including backpropagation [RHW86], cascade correlation and others, but they cannot be used in this context because our problem requires an *unsupervised* learning algorithm. Hence, we are investigating unsupervised learning architectures, such as Adaptive Resonance Theory (ART) [Gro88].

5.3 ART

ART belongs to a class of learning architectures known as *competitive* learning models [Gro88][CG88]. The competitive learning models are usually characterized by a network consisting of two layers L_1 and L_2. The input pattern I is fed into layer L_1 where it is normalized. The normalized input is fed forward to layer L_2 through the weighted interconnection links that forms an adaptive filter. Layer L_2 is organized as a *winner-take-all network* [FB82][Sri91][BSD90]. The network layer L_2 is usually organized as a mutually inhibitory network wherein each unit in the network inhibits every other unit in the network through a value proportional to the strength of its activation. Layer L_2 has the task of selecting the network node a_{max}, receiving the maximum total input from L_1. The node a_{max} is said to cluster or code the input pattern I.

In the ART system the input pattern I is fed in to the lower layer L_1. This input is normalized and is fed forward to layer L_2. This results in a network node n_{max} of layer L_2 being selected by virtue of it having the maximum activation value among all the nodes in the layer. This node n_{max} represents the hypothesis H put forth by the network about the particular classification of the input I. Now a *matching* phase occurs wherein the hypothesis H and the input I are matched, with the quality of the required match controlled by the *vigilance parameter*.

If the quality of match is worse than the value specified in the vigilance parameter, a mismatch occurs and the layer L_2 is reset thereby deactivating node n_{max}. The input I activates another node and the above process recurs, comparing another hypothesis or forming a new hypothesis about the input pattern I. New hypotheses are formed by learning new classes and recruiting new uncommitted nodes to represent these classes.

Some of the properties of ART that makes it an potential choice for our task include

1. Real-time (on-line) learning;
2. Unsupervised learning;
3. Fast adaptive search for best match as opposed to strict match; and
4. Variable error criterion which can be fine-tuned by appropriately setting the *vigilance* parameter.

However, one of the limitations of ART for our particular task arises from its inability to distinguish the queries for particular components by users, from the component classes which form the exemplar classes. Another limitation arises from the fact that only one exemplar class is chosen at a time which represents the *best* match, rather than choosing a collection of *close* matches for reuser consideration.

Our proposed system will operate in two phases. The first, *loading* phase populates the repository with components. The second, *retrieval* phase identifies candidate components in response to user queries. The distinguishing factor between the two phases is the value of the vigilance parameter. In the loading phase, the system will employ a high vigilance value. This ensures the formation of separate categories for each of the components in the repository. In the retrieval phase, the system will employ a low vigilance value, thereby retrieving components that best match the query.

We also intend to modify the winner-take-all network layer of the ART to choose k winners instead of one. This is extremely useful in our context because there may be multiple software components which meet the user specifications. The software reuser may select a subset $m \leq k$ of these components based upon requirements. The system should associate these m components with the user query and retrieve them for subsequent queries having similar input specifications. This can be achieved by associating small initial weights on the lateral links of the winner-take-all network and modifying them appropriately based on user feedback (i.e., reuser refusals).

6 Discussion

6.1 Our Placement in the User-Based Framework

Discussions in the workshop placed our work in the region of user intention / no feedback in the user-based framework. Upon further reflection, we have slightly altered our perspective. While this placement is certainly proper in the strict context of a single user query, it is not accurate in the broader context of a community of users accessing the repository over time.

As the system is rewarded for providing true hits to users and punished for providing false hits, there is a consensual drift, providing feedback for subsequent user queries. Thus, viewing the amortized effect of user behavior, rather than the immediate effect of user behavior, our system shifts down towards passive observation and left towards immediate feedback.[2] The net result is that our system occupies two distinct points in the framework, one for the semantics involved in the immediate query query and one for the semantics involved in the aggregate behavior of the repository over time.

6.2 The Relationship to Gestural Recognition

Beale [BE], Rubine [Rub], and Zhao [Zha], the other occupants of the Novel Input category of the task-based framework, respectively address sign language recognition, drawing geometric figures, and diagram editing – all interpreting imprecise human gestures and mapping them to a precise application domain. They all address the inability of humans to accurately repeat physical movement.

Our mechanism, on the other hand, accepts a precisely phrased user query and adapts it to an imprecise application domain. Ignoring the issue of poor typing skills, our user community can accurately repeat a given user intention (query) any number of times, and we know exactly what that intention is. The challenge in our domain occurs when that intention has no exact match in the system. It's similar to Rubine's system offering to draw a square or a hexagram (or perhaps even a five-sided star) when the user gestured a pentagram, but the system had no training in pentagram gestures.

[2]or more precisely, non-immediate feedback.

6.3 Directions for Future Research

Options available to us at this point in our work lie in two general directions, further extending repository semantics and exploring the application of neural networks to these types of application domains.

With respect to the former, the classification scheme described here is restricted to facets and tuples containing facets. In other work, the classification scheme was first extended to include signatures for abstract data types [Eic91a] and then further extended to support axioms in a second phase in the query process [Eic91b]. A merger of that work with that described here has appeal – particularly the imprecise matching of signatures.

With respect to the latter, we are interested in studying the tradeoffs between individual user adaptation versus the consensual adaptation described above. These two actually are the extremes in a continuum of user groupings. This coupled with an additional dimension of user expertise forms a state space of user behavior where the system might more heavily weight certain semantic connections for experts and other semantic connections for novices. This will require the development of new algorithms for relaxation.

7 Conclusions

Our approach extends previous work in component retrieval by incrementally adapting the conceptual closeness weights based upon actual use, rather than an administrator's assumptions. Neural networks provide a quite suitable framework for supporting this adaptation. Reuse repository retrieval provides a unique and challenging application domain for neural networking techniques.

This approach effectively adds an additional dimension to the conceptual space formed by the type lattice. This additional dimension allows traversal from one vertex to another using the adapted closeness weights derived from user activity, rather than the partial orders used in defining the lattice. The resulting retrieval mechanism supports both well-defined lattice-constrained queries and ill-defined neural-network constrained queries in the same framework.

References

[Atk] J. Atkins. private communication.

[BE] R. Beale and A. D. N. Edwards. Gestures and neural networks in human-computer interaction. in this volume.

[BSD90] J. Barden, K. Srinivas, and D. Dharmavaratha. Winner-take-all networks: Time-based versus activation-based mechanisms for various selection goals. In *IEEE International Conference on Circuits and Systems*, pages 215–218, New Orleans, LA, May 1990.

[CG88] G. A. Carpenter and S. Grossberg. The art of adaptive pattern recognition by a self-organizing neural network. *IEEE Computer*, 21(3):77–88, 1988.

[EA90] D. Eichmann and J. Atkins. Design of a lattice-based faceted classification system. In *Second International Conference on Software Engineering and Knowledge Engineering*, pages 90–97, Skokie, IL, June 1990.

[Eic89] D. Eichmann. *Polymorphic Extensions to the Relational Model.* PhD dissertation, The University of Iowa, Dept. of Computer Science, August 1989.

[Eic91a] D. Eichmann. A hybrid approach to software repository retrieval: Blending faceted classification and type signatures. In *Third International Conference of Software Engineering and Knowledge Engineering*, pages 236–240, Skokie, IL, June 1991.

[Eic91b] D. Eichmann. Selecting reusable components using algebraic specifications. In *Second International Conference on Algebraic Methodology and Software Technology*, pages 37–40, Iowa City, IA, May 1991. to appear in AMAST'91, Workshops in Computing Series, Springer-Verlag.

[FB82] J. A. Feldman and D. H. Ballard. Connectionist models and their properties. *Cognitive Science*, 6:205–254, 1982.

[GH78] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GOF$^+$88] R. Gagliano, G. S. Owen, M. D. Fraser, K. N. King, and P. A. Honkanen. Tools for managing a library of reusable ada components. In *Workshop on Ada Reuse and Metrics*, Atlanta, GA, June 1988.

[Gro88] S. Grossberg, editor. *Neural Networks and Natural Intelligence.* MIT Press, Cambridge, MA, 1988.

[Gue87] E. Guerrieri. On classification schemes and reusability measurements for reusable software components. SofTech Technical Report IP-256, SofTech, Inc., Waltham, MA, 1987.

[Lip87] R. P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4:4–22, 1987.

[PDF87] R. Prieto-Díaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, 1987.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1. MIT Press, Cambridge, MA, 1986.

[Rub] D. Rubine. Criteria for gesture recognition technologies. in this volume.

[Sri91] K. Srinivas. *Selection in Massively Parallel Connectionist Networks.* PhD dissertation, New Mexico State University, Dept. of Computer Science, 1991.

[Zha] R. Zhao. On the graphical gesture recognition for diagram editing. in this volume.

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 following (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

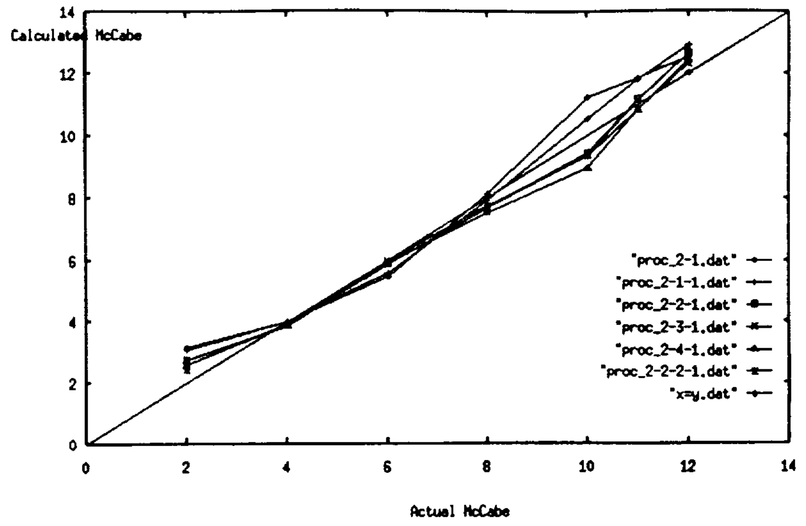

Figure 1: McCabe Results for Single Procedures

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

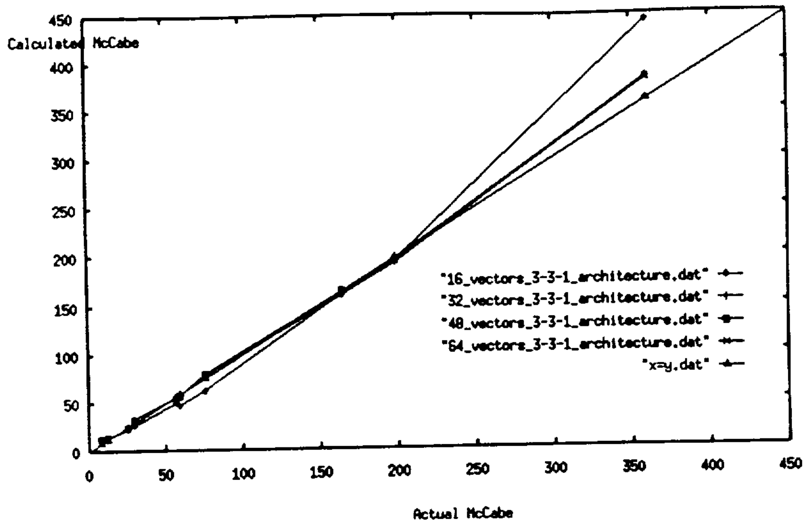

Figure 2: The 3-3-1 Architecture

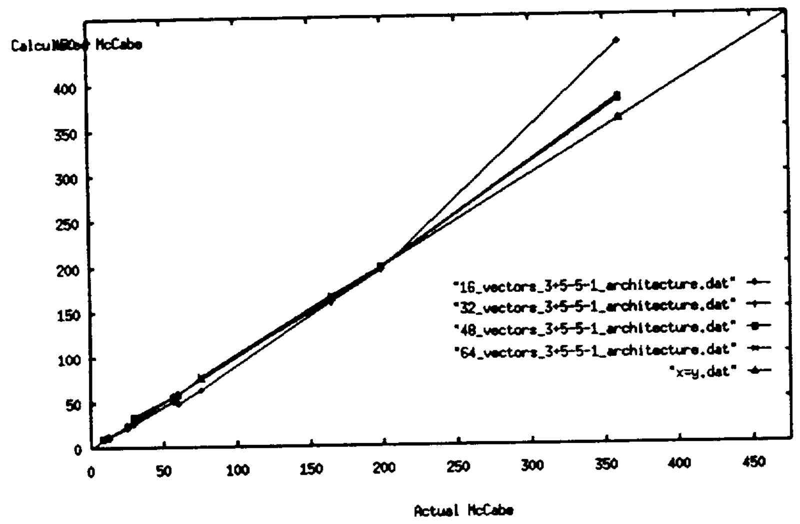

Figure 3: The 3+5-5-1 Architecture

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.

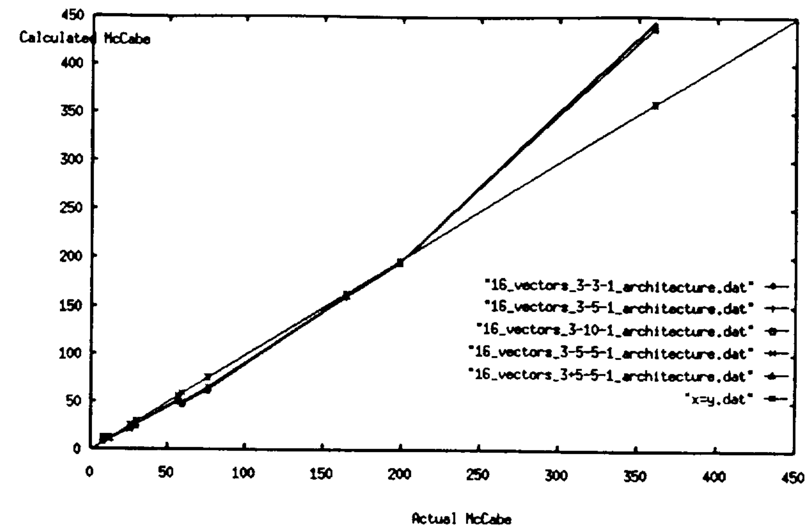

Figure 4: 16 Training Vectors for all Architectures

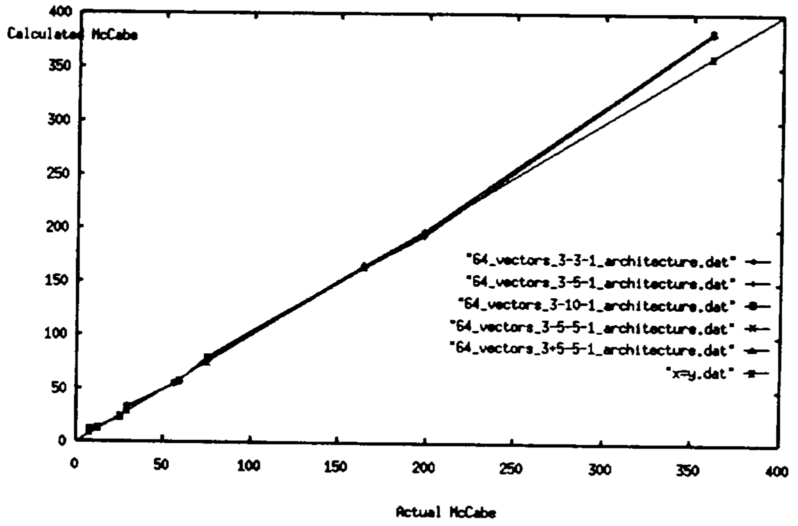

Figure 5: 64 Training Vectors for all Architectures

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.

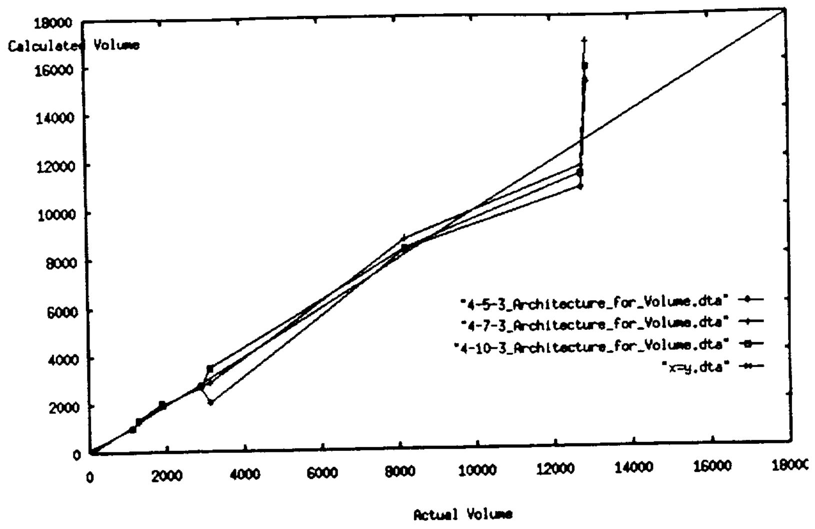

Figure 6: Volume Results, Broad Architectures

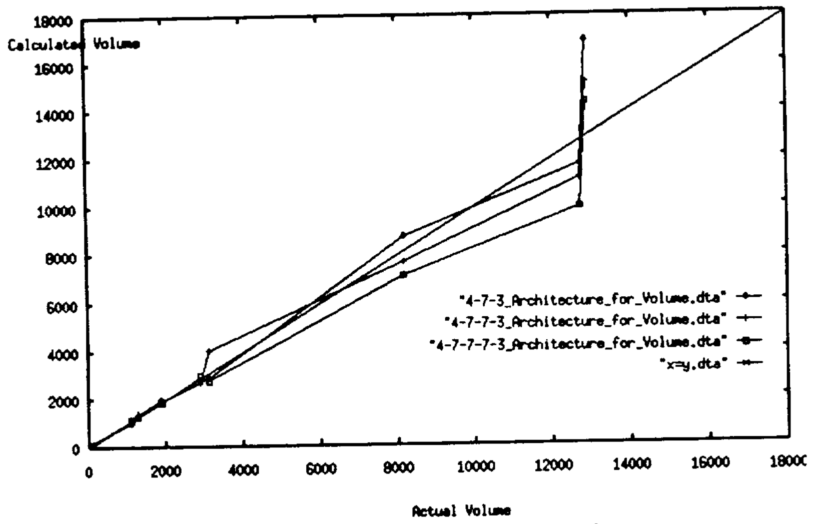

Figure 7: Volume Results, Deep Architectures

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

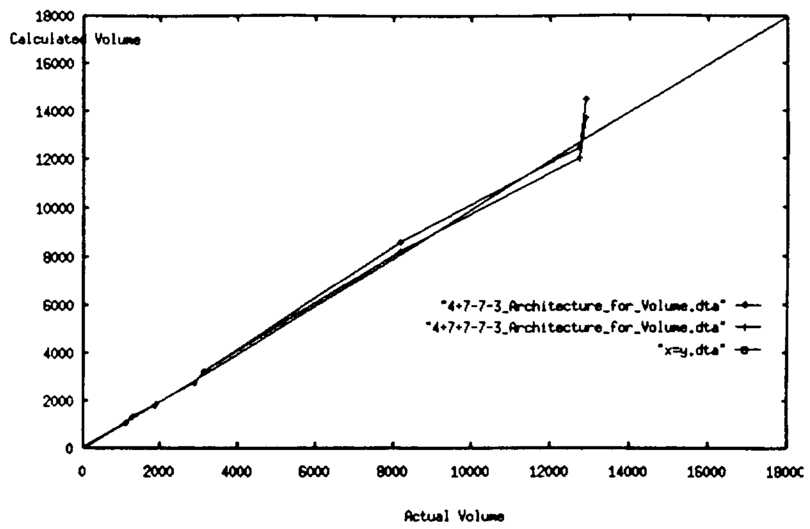

Figure 8: Volume Results, Connected Architectures

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

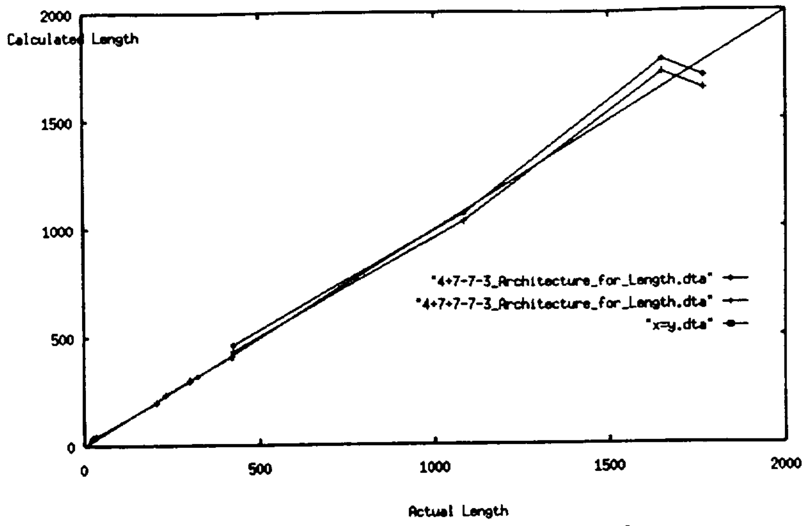

Figure 9: Length Results, Connected Architectures

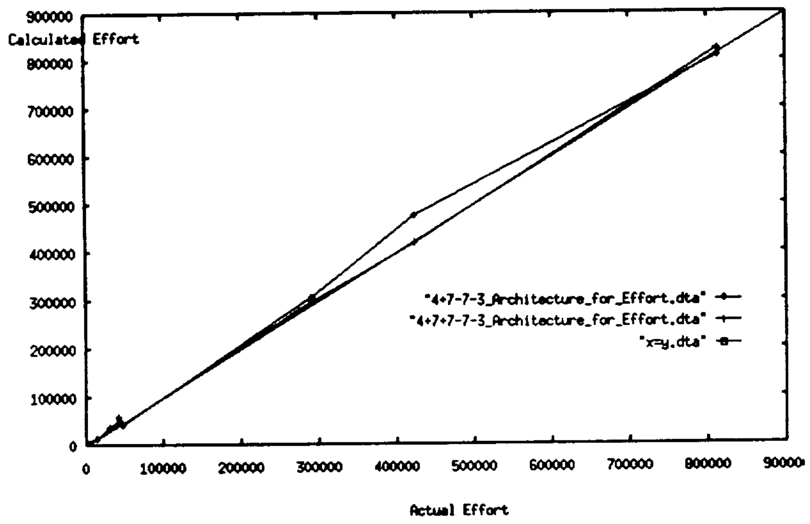

Figure 10: Effort Results, Connected Architectures

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 reuse 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.