2006 International Joint Conference on Neural Networks
Sheraton Vancouver Wall Centre Hotel, Vancouver, BC, Canada
July 16-21, 2006

Neural Net Analysis of the Propensity for Change in Large Software Systems

Steven B. Morphet, James Fawcett, Kanat Bolazar, and Murat Gungor

Abstract—**A novel approach for analyzing the relationship between code metrics and change count histories is presented. Specifically, neural networks are employed to determine a mapping between metrics and change count. While these neural networks can be trained to a high degree of accuracy, their internal workings remain opaque to the user. As such, a fuzzy modeling approach is additionally employed to generate the rules governing the neural computation. These rules are linguistic in nature and are thus more easily interpreted by software project managers. Application of this method to Mozilla change data reveals the importance of fan-out, total lines of code and maximum cyclomatic complexity metrics in predicting amount of change per file.**

I. Introduction

This paper studies relationships between code metrics and change count histories for a large project. The analysis is file based. That is, we compute a variety of metrics for each source code file in several large libraries from the open-source Mozilla project, and relate them to the number of cumulative changes for each of those files in several builds. We use files because changes are recorded for files in the data we examined; and because files are the units of configuration management in large software projects.

Others, German [1] and [2], Huntley [3], have examined open-source project data but we have found no modeling of the reliability of metrics to measure potential for change, as reported here. Graves, et. al. [4] analyzed relationships between change metrics and predicted faults, using data from a telephone switching system. Our focus is on modeling change history using code metrics.

We chose Mozilla because it is large[1], accessible, and has provided a wealth of change data in its source code repository (CVS) database. The data presented here is drawn from the Windows build of Mozilla [5], for release 1.4.1.

The analysis uses a converging series of Neural Nets [6] [7] to model production of changes as functions of the metrics set, described below. Change data was extracted from the Mozilla CVS repositories and divided into training and testing sets. After building networks using the training data, the results were evaluated by comparing predictions of change made by the nets, using various metrics as inputs with actual change counts recorded in the Mozilla repository as outputs. In all analyses, some networks found significant relationships between a subset of the metrics used and change history, achieving accuracies above 90 percent, detecting files with high propensity for change, on the testing data sets. The results show that propensity for change is related to the chosen metrics and is dominated by two of them, fan-out and total lines of code.

The results are similar to those of a previous paper that used Multiple-Linear Regression as the analysis tool. We followed with this work to see if the nonlinear modeling properties of a three-layer neural network could provide improved accuracy, and found that that was indeed the case, although the basic conclusions regarding the significant metrics are the same. Finally, we produce a linguistic model of the network predictions, based on fuzzy rule sets [8] [9]. The linguistic model may be a more natural way for developers and managers to interpret results of analyses like this.

II. Incremental Neural Model

Three-layer (single hidden layer) feed-forward neural networks with hidden layer sigmoidal transfer functions are universal approximators; they are capable of approximating any continuous multivariate function, to any desired degree of accuracy, given a sufficient number of hidden layer nodes [10] [11].

Weights are usually initialized to random small values, and back-propagation training is a highly nonlinear deterministic process that causes neural networks that have similar weights to diverge during training. Similar to brain development, this training can be seen as organic growth, with hidden layer nodes (neurons) taking on responsibilities to detect certain recurrent patterns in data that correlate with output. This organic differentiation of the hidden layer nodes often does not have a single solution, even if we count all possible rearrangements of the hidden nodes as the same solution.

Two neural networks trained separately may have hidden layer nodes that detect different important patterns in input. Building a neural network incrementally takes advantage of the possibility that two separately trained neural networks may be able to compensate for and complement each other.

Building a neural network incrementally consists of four steps summarized below then expanded:

- Read and prepare the data: quantize, normalize, separate into training and testing sets.
- Train small $(m \times 2 \times 1)^2$ neural networks on remaining delta (output) that is not yet accounted for.

[1] There are 6193 source code files in the Windows build for version 1.4.1

[2] The network has m input neurons serving a fan-out function, 2 hidden neurons employing a non-linear transfer function, and one output neuron employing a linear transfer function.

0-7803-9490-9/06/$20.00/©2006 IEEE

- Combine these small networks into a larger network, but only select those that improve the predictive ability as evidenced by production of good results on the test set.
- Perform a final pruning of the overall network.

A. Step 1

We analyzed all the files of one Mozilla module, using the following metrics for each file[3]:

Fan-in and fan-out require a dependency graph to first be produced. A dependency graph is a directed graph with files as the nodes and their dependencies as the edges. Fan-in is the number of incoming edges to a node, while fan-out is the number of outgoing edges from a node. (These are inter-file metrics.)

TABLE I
EMPLOYED METRICS

Metric	Description
fi	fan-in (number of dependent files)
fo	fan-out (number of files depended upon)
mcc	maximum CC (cyclomatic complexity)
tcc	total CC
acc	average CC
mfs	maximum function size
afs	average function size
fc	function count
scs	strong component size (mutual dependency set size)
godc	global object declaration count
tloc	total lines of code

Software metrics employed in this analysis.

The other metrics are intra-file metrics discovered by static analysis of code.

Most central files used in Mozilla have been in Mozilla for many years. The eleven metrics constitute the inputs to our neural network. The neural output is change count (*cc*). This is the change count over the whole life of a file, and it is extracted from the change log in the repository.

We quantize this change count by separating those files that have 25 or more changes over their whole life, and those files that have fewer than 25 changes. The goal is for the team leader and project manager to discover problem files in the system. Once trained, this neural network should be able to predict which files need many changes and may need special attention.

With the eleven inputs metrics, and quantized change count as the output, we create a set of input-output vectors. 80% of this data set is used in training; the remaining 20% is the test set (to be used later to improve the generalizing ability of the neural network).

B. Step 2

Using the training data set, we train an $11 \times 2 \times 1$ feed-forward neural network, consisting of two hidden nodes employing a sigmoidal transfer functions, and a single output node employing a linear transfer function.

Even after training (via back-propagation in this case), the output of this small neural network will quite probably not match the expected output completely. If the mean squared error between the difference replaced the output training vector and a new $11 \times 2 \times 1$ neural network with sigmoidal hidden layer nodes and a linear output node is created and trained.

This process repeats until the mean-squared error falls below a predetermined threshold.

C. Step 3

At the completion of step 2, *N* small neural networks have been created. These networks are combined into a single larger network.

Since every small neural network has the same input set, and each has only a single output neuron employing a linear transfer function, combining two (or more) of these networks into a larger network is a straightforward process. Specifically, the hidden neurons of the second network are moved into the first network. These new hidden neurons are then connected to the input nodes and output neuron using the corresponding weight values found in the second network. Given any input vector *I*, presenting *I* to the first and second small networks and adding their outputs will result in exactly the same value as presenting *I* to the larger network derived from combining the two small neural networks.

This process repeats with the combined network from the previous step becoming the first small network, and the second small network is the next $11 \times 2 \times 1$ neural network derived in step 2.

As a point of optimization, if the resulting neural network derived from combining two smaller networks does not demonstrate improved predictive ability (as measured by the performance on the test set), the newly derived network is rejected and processing continues with the previously derived network and the next small network. For example, if twenty $11 \times 2 \times 1$ neural networks are created but only four of these networks improve results on the test set, the combined network is an $11 \times 8 \times 1$ network with hidden layer nodes copied from the four $11 \times 2 \times 1$ neural networks, and a linear output node (derived as a linear combination of the four output nodes from the $11 \times 2 \times 1$ neural networks).

D. Step 4

As a last greedy step, each hidden layer neuron is removed in turn and the accuracy of the overall network is reassessed on the test data. If the modified network's performance is unacceptable, the hidden neuron is reinserted; otherwise, the hidden neuron is left off. In either case, processing continues with the next hidden neuron.

III. Sensitivity Analysis

[3] These metrics were chosen because they were computable from the stored information in the Mozilla source code repository.

In order to reduce the number of inputs used to train the neural network, sensitivity analysis has been conducted on the original 11 inputs. Sensitivity analysis is performed in the following manner:

- Given a dataset, the base accuracy of the neural network is established.
- Next data associated with a given input is randomized
- This new dataset is presented to the neural network
- The output accuracy of the neural network is compared against the base accuracy (and a running total is maintained)
- Steps 2 through 4 are repeated for the same input until the running total (of that input) stabilizes
- The entire process repeats for all inputs

The data associated with an input having no effect on the neural network's output can be completely randomized without effecting accuracy. Likewise, the data associated with inputs having little effect on the neural network's output can also be randomized with little effect on neural network accuracy. Multiple neural networks are trained on the data and sensitivity analysis is performed on each. The following table shows the results of marginalizing each input in turn for each of 10 trained neural networks.

From table II, the "strong component size" input is the least important of the inputs (accuracy is effected the least). As such, it was eliminated, and the training process rerun. Proceeding in this fashion, the process of sensitivity analysis/retraining continues until the networks can no longer reliably train. The final results of input elimination by sensitivity analysis are shown below.

TABLE II
SENSITIVITY ANALYSIS

NN	BASE ACCURACY	FI	FO	MCC	TCC	ACC
1	0.9444	0.8737	0.2961	0.3287	0.9121	0.6243
2	0.9222	0.8616	0.3288	0.5208	0.3631	0.8808
3	0.9222	0.8612	0.3037	0.2148	0.2137	0.3970
4	0.9222	0.8198	0.2647	0.3036	0.2561	0.7252
5	0.9333	0.8854	0.2668	0.3014	0.2260	0.8956
6	0.9666	0.6640	0.3076	0.3524	0.3921	0.3965
7	0.9222	0.9177	0.3638	0.3262	0.6707	0.8885
8	0.9333	0.6235	0.2986	0.2020	0.4966	0.8548
9	0.8666	0.8105	0.3216	0.2677	0.7394	0.8370
10	0.9000	0.8764	0.3155	0.5230	0.2448	0.8393
	Average:	*0.8194*	*0.3067*	*0.3341*	*0.4514*	*0.7339*
	MFS	AFS	FC	SCS	GODC	TLOC
1	0.9034	0.9153	0.4964	0.9153	0.8883	0.2708
2	0.8911	0.6230	0.9057	0.9025	0.9166	0.4721
3	0.8720	0.5422	0.8937	0.9016	0.8666	0.2414
4	0.8964	0.6777	0.8896	0.8740	0.9004	0.9213
5	0.9054	0.5012	0.8974	0.8993	0.8376	0.9122
6	0.2317	0.4371	0.3371	0.7775	0.5160	0.4344
7	0.3273	0.8887	0.8433	0.9002	0.6572	0.2485
8	0.9233	0.5465	0.3275	0.9064	0.6257	0.5787
9	0.8194	0.5632	0.8510	0.8644	0.4653	0.7091
10	0.5341	0.5841	0.8482	0.8753	0.8431	0.5445
Av:	*0.7304*	*0.6279*	*0.7290*	*0.8816*	*0.7517*	*0.5333*

Results of sensitivity analysis for neural networks 1 through 10.

Eliminating "maximum cyclomatic complexity" results in a data set that is extremely difficult to train on. Therefore, after sensitivity analysis is complete, the set of training inputs consists of:

TABLE III
SENSITIVE INPUTS

NN	BASE ACCURACY	FO	MCC	TLOC
81	0.9888	0.3123	0.9466	0.2056
82	0.9333	0.3041	0.3726	0.3748
83	0.9666	0.3694	0.6021	0.2766
84	0.9444	0.3208	0.2501	0.2920
85	0.9111	0.3437	0.8774	0.3261
86	0.9444	0.3076	0.5931	0.2732
87	0.9444	0.3591	0.8900	0.3872
88	0.9333	0.3396	0.7525	0.4544
89	0.9666	0.3614	0.6025	0.2710
90	0.9444	0.3152	0.2452	0.2950
	Average:	*0.3333*	*0.6132*	*0.3156*

Most sensitive inputs for neural networks 81 through 90.

IV. FUZZY MODEL

The principal objective of neural networks is to model functional mappings describing the relationship between inputs and outputs within data sets. While neural networks often train on these mappings successfully, they nevertheless remain black boxes. The desire to generate linguistically interpretable rule sets from black box models has led to a unique fuzzy model extraction technique.

TABLE IV
EMPLOYED METRICS

METRIC	DESCRIPTION
fo	fan-out (number of files depended upon)
mcc	maximum CC (cyclomatic complexity)
tloc	total lines of code

Software metrics employed after sensitivity analysis.

The methods that extract fuzzy system from neural networks, presented in the literature, may be divided into two categories: pedagogical – extracting global relationships from the inputs and outputs of the network directly, and decompositional – analyzing individual neurons within the network. In this paper, a decompositional modeling approach for extracting fuzzy systems from neural networks is employed. The approach may be summarized as follows:

- Individual fuzzy systems are extracted from each neuron in the network's hidden layer.
- Since the results of pedagogical methods are typically more interpretable than decompositional methods (one rule set as opposed to many), a method for

combining the individual fuzzy systems is applied.

- In the final step, the number of rules is reduced and the antecedents of any remaining rules are shortened (provided that the output of the fuzzy system is not adversely affected).

Since a decompositional approach is used to model each hidden neuron and later combined into an overall solution, the resultant fuzzy system is more truly representational of the inner functioning of the neural network than standard pedagogical methods.

In the development of fuzzy systems, one of the most critical issues in the evaluation process is defuzzification. The YAD defuzzifier employs an iterative process to determine the crisp value. Algorithmically, YAD repeatedly splits the fuzzy set, computing the running total of differences in mean degrees of membership of these split sets. YAD is linear in nature; that is, it is defined in terms of the addition of fuzzy set values. As such, it possesses the additive property, which means that adding the results of two defuzzified fuzzy sets equivalent to adding the fuzzy sets then defuzzifying. The additive property facilitates the combining of these fuzzy systems into a single system that models the behavior of the entire network.

Each hidden layer neuron in the neural network is modeled via a single input/single output (SISO) fuzzy decision tree (FDT). A SISO FDT has the topology of a decision tree, but each branch of the SISO FDT fires to a greater or lesser extent based on the degree of membership of the corresponding branch expressions. (That is, each branch has an associated expression of the form "input is membership function," which, when evaluated in fuzzy terms, produces a degree of membership associated with the branch.) Combining the branches' degrees of membership along the path from the root to the leaf node of interest provides an overall weighting factor for the SISO rule set associated with the leaf node.

To determine the output of a SISO FDT, evaluation proceeds from the root to each terminal node. Each leaf is treated as a special case of a multiple input/single output (MISO) system – special in the sense that each rule has only one input. As per MISO evaluation, the degrees of membership of each rule's antecedent subterms are computed and weighted, and then fuzzy conjunction is applied. The leaf weighting is then factored in. The remainder of the evaluation proceeds in much the same fashion as MISO evaluation.

Viewing a neural network at the level of the neuron, modeling proceeds as follows:

- Consider a hidden neuron and its input weights to comprise the black box to be modeled.
- Determine the number and shape of all input and output membership functions. The number and shape of all input membership functions across SISO systems are identical.
- Determine the weights associated with rules of the SISO FDT.

Replacing each hidden layer neuron in the neural network with the SISO FDT generated in the above steps yields a network of SISO FDTs.

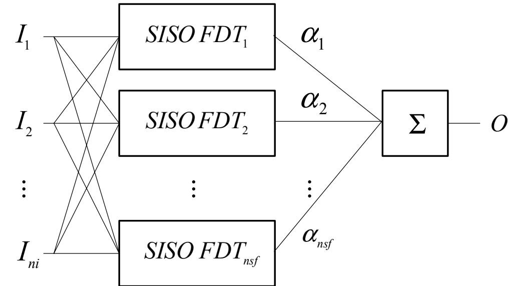

Fig. 1. The network of SISO FDTs used to model the Mozilla data set.

Given the properties of the defuzzifier, the alpha terms may be pushed into the rule weights of the SISO FDTs, resulting in a new network where all the links between the SISO FDTs and the summation unit have a weight of 1.0.

Additionally, since the input and output membership functions in the above SISO FDTs are identical, any two SISO FDTs can be combined into a single SISO FDT in a mathematically exact way. Applying this approach iteratively, a single SISO FDT may be derived that is mathematically equivalent to the network of SISO FDTs. As such, both the behavioral accuracy of a decompositional approach and the linguistic interpretability of a pedagogical approach are achieved.

V. Case Study

The above technique was applied twice to the three input (fo, mcc, and tcc) neural network trained on the Mozilla dataset. The first application specified two input and two output membership functions for the linguistic model. A typical rule set generated is:

- if fo low ∧ mcc low ∧ tloc high, cc high (2.2405)
- if fo low ∧ mcc high ∧ tloc high, cc high (1.5447)
- if fo high ∧ mcc high ∧ tloc high, cc high (1.3763)
- if fo high ∧ mcc high ∧ tloc low, cc high (1.3763)
- if fo high ∧ mcc low ∧ tloc high, cc high (1.2850)
- if fo low ∧ mcc low ∧ tloc low, cc low (1.1230)
- if fo low ∧ mcc low ∧ tloc low, cc high (0.9226)
- if fo low ∧ mcc high ∧ tloc low, cc low (0.4273)

In other words, if either the fan out or the total lines of code is high, then the change count is high. Restated, if both the fan out and total lines of code are low, then the change count is low. This is an intuitively appealing result, especially since maximum cyclomatic complexity is the least sensitive of the three inputs (as per the above sensitivity analysis). Retraining the neural network and reapplying this

process resulted in similar rule sets. The second application specified three input and two output membership functions for the linguistic model. A typical rule set generated is:

- if fo low ∧ mcc low ∧ tloc low, cc low (2.0484)
- if fo low ∧ mcc high ∧ tloc high, cc high (1.9967)
- if fo low ∧ mcc med ∧ tloc high, cc high (1.8653)
- if fo high ∧ mcc low ∧ tloc high, cc high (1.7276)
- if fo med ∧ mcc low ∧ tloc high, cc high (1.5543)
- if fo low ∧ mcc low ∧ tloc med, cc high (1.5026)
- if fo med ∧ mcc high ∧ tloc med, cc high (1.4292)
- if fo med ∧ mcc med ∧ tloc low, cc high (1.3763)
- if fo high ∧ mcc med ∧ tloc low, cc high (1.3763)
- if fo high ∧ mcc high ∧ tloc low, cc high (1.3763)
- if fo med ∧ mcc med ∧ tloc high, cc high (1.3763)
- if fo high ∧ mcc med ∧ tloc high, cc high (1.3763)
- if fo high ∧ mcc high ∧ tloc high, cc high (1.3763)
- if fo med ∧ mcc med ∧ tloc med, cc high (1.3763)
- if fo high ∧ mcc med ∧ tloc med, cc high (1.3763)
- if fo high ∧ mcc high ∧ tloc med, cc high (1.3763)
- if fo high ∧ mcc low ∧ tloc low, cc high (1.3653)
- if fo med ∧ mcc high ∧ tloc high, cc high (1.3507)
- if fo low ∧ mcc low ∧ tloc high, cc high (1.3151)
- if fo med ∧ mcc high ∧ tloc low, cc high (1.1904)
- if fo med ∧ mcc low ∧ tloc low, cc high (1.0541)
- if fo med ∧ mcc low ∧ tloc med, cc high (1.0095)
- if fo high ∧ mcc low ∧ tloc med, cc high (0.6523)
- if fo low ∧ mcc med ∧ tloc low, cc low (0.5908)
- if fo low ∧ mcc med ∧ tloc med, cc high (0.3685)
- if fo low ∧ mcc high ∧ tloc med, cc high (0.0977)
- if fo low ∧ mcc high ∧ tloc low, cc high (0.0248)

In other words, if the fan out is low and the maximum cyclomatic complexity is not high and the total lines of code is low, then the change count is low. This is also an intuitively appealing result that further refines the result discovered earlier. The addition of other membership functions, such as "VERY LOW" and "VERY HIGH" reveal that fan out is the most sensitive input of the three (a verification of the sensitivity analysis hypothesis).

These linguistically interpretable rules clearly show what the Mozilla project managers and team leaders should look for while assessing and managing risk in a project. Ideally, risk should be contained by proactive measures: Throughout the development, both fan-out and total lines of code (and possibly, also, maximum cyclomatic complexity) per file should be measured, and neither metric should be allowed to go above a limit, possibly by selectively refactoring high-risk code.

VI. Conclusions and Future Work

This paper presents a novel approach for analyzing the relationship between code metrics and change count histories. In order to capture the [potentially] non-linear mapping between metrics and change count, a neural network is utilized. Since neural networks function as black boxes, a fuzzy modeling approach is employed to extract the rule sets governing the neural computation. The Mozilla dataset is used as a case study, and, as such, several software metrics are gathered. Sensitivity analysis reveals that only three of the measured metrics are necessary to a train neural network to high accuracy. This is largely attributable to the metrics selected (a function of the data available). A fuzzy rule set is then generated for the neural network using a novel decompositional approach. The interpretation of the rule set is straightforward and intuitively appealing. While the fidelity of the measured metrics is relatively low, the potential of the model is demonstrated. Future work includes running a similar test against a different base of metrics (we are currently looking into different metrics correlated with level of development effort).

References

[1] German, D., Hindle A., and Jordan N. (2004). "Visualizing the evolution of software using softChange," Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. 336-341.

[2] German, D. and Mockus A. (2003). "Automating the Measurement of Open source Projects," ICSE 2003, 3rd Workshop on Open Source Software Engineering.

[3] Huntley, C. (2003). "Organizational Learning in Open-Source Software Projects: An Analysis of Debugging Data," IEEE Transactions on Engineering Management, 50(4), pp. 485-493.

[4] Graves et al. (2000). "Predicting Fault Incidence Using Software Change History," IEEE Transactions on Software Engineering, 26(7), pp. 653-661.

[5] Mozilla on Microsoft Windows 32-bit Platforms, www.mozilla.org/build/win32.html.

[6] Stephens, L. (May 2004). "Advanced Statistics Demystified," McGraw Hill Incorporated.

[7] Huck, S. (2000). "Reading Statistics and Research," Addison Wesley Longman.

[8] Morphet, S.B. "Modeling Neural Networks via Linguistically Interpretable Fuzzy Inference Systems." Doctoral Dissertation, Syracuse University, Computer Engineering Department. May 2004.

[9] Morphet, S.B., and L.B. Morphet. "Combining Single Input/Single Output Fuzzy Decision Tree." Submitted to the World Congress on Computational Intelligence 2006.

[10] Cybenko, G. (1989). "Approximation by Superpositions of a Sigmoidal Function," Mathematics of Control, Signals and Systems, 2(4), 303-314. Correction appears in 5 (1995) p.455.

[11] Hornik, K., Stinchcombe, M., and White, H. (1989). "Multilayer Feedforward Networks are Universal Approximators," Neural Networks, 2(5), pp. 359-366.