Fault Prediction in Object-Oriented Software Using Neural Network Techniques

Atchara Mahaweerawat*†, Peraphon Sophatsathit*, Chidchanok Lursinsap* and Petr Musilek‡

*Advanced Virtual and Intelligent Computing Center (AVIC)
Department of Mathematics, Faculty of Science
Chulalongkorn University, Bangkok 10330, Thailand
atchara.m@student.chula.ac.th, peraphon.s@chula.ac.th, lchidcha@chula.ac.th
† Department of Mathematics, Statistics, and Computer Science, Faculty of Science
Ubon Ratchathani University, Ubon Ratchathani 34190, Thailand
‡Facility for Advanced Computational Intelligence and Applications (FACIA)
Department of Electrical and Computer Engineering, Faculty of Engineering
University of Alberta W2-030 ECERF, Edmonton, Alberta T6G 2V4, Canada
Petr.Musilek@ualberta.ca

***Abstract*— To remain competitive in the dynamic world of software development, organizations must optimize the usage of their limited resources to deliver quality products on time and within budget. This requires prevention of fault introduction and quick discovery and repair of residual faults.**

In this paper a new approach for predicting and classification of faults in object-oriented software systems is introduced. In particular, faults due to the use of inheritance and polymorphism are considered as they account for significant portion of faults in object-oriented systems.

The proposed fault prediction model is based on supervised learning using Multilayer Perceptron Neural Network. The results of fault prediction are analyzed in terms of classification correctness and some other standard criteria.

Based on the results of classification, faulty classes are further analyzed and classified according to the particular type of fault. The classification model is based on clustering using Radial-Basis Function Neural Network. It is concluded, that the proposed model provides high accuracy in discrimination between faulty and fault-free classes.

Keywords: Software fault, Predictive model, Neural network, Radial-Basis Function, Back-Propagation Learning.

I. INTRODUCTION

Software reliability can be defined as the probability of failure-free operation of a computer program executing in a specified environment for a specified time [1]. It is often considered a software quality factor that can aid in predicting the overall quality of a software system using standard predictive models. Predictive models of software faults use historical and current development data to make predictions about faultiness of software subsystems/modules. Although software faults have been widely studied in both procedural and object-oriented programs, there are still many aspects of faults that remain unclear. This is true especially for object-oriented software systems, in which inheritance and polymorphism can cause a number of anomalies and fault types [2]. Unfortunately, existing techniques used to predict faults in procedural software are not generally applicable in object-oriented systems.

Some recent studies [3], [4], [5], [6], [7], [8], [9] report the use of object-oriented metrics to predict fault-proneness and number of faults by applying various statistical methods and neural network techniques. However, they generally stop at the problem of fault prediction without attempt to further characterize the faults likely present in the system. In this paper, a new method of fault prediction is introduced along with a method for classification of fault type. For the reasons mentioned earlier, faults due to inheritance and polymorphism are of special interest in this work.

The problem of predicting whether a software class is faulty is viewed as a binary classification problem in which the class is represented as a data point with coordinates described by object-oriented metrics and other parameters. The prediction of fault type in a faulty software class is then considered as a clustering problem in which each fault type is represented by a cluster prototype [10]. To solve the two problems, use of neural network techniques [11] is proposed. In particular, the classification problem is addressed using a Multilayer Perceptron (MLP) while the solution to clustering problem is based on Radial-Basis Function Network (RBFN).

The paper is organized as follows. Section II provides background information on the problem area including fault taxonomy, software metrics and neural network methods used in this study. Section III describes the data used in the experimental part of this work along with a new method for data preprocessing. Construction of the new fault prediction models and their result are presented in Section IV. The result obtained from the models are further discussed in Section V. Finally, main conclusion and directions of future work are given in Section VI.

TABLE I

FAULT AND ANOMALIES DUE TO INHERITANCE AND POLYMORPHISM

Acronym	Fault/Anomaly
SDA	State Definition Anomaly (possible post-condition violation)
SDIH	State Definition Inconsistency (due to state variable hiding)
SDI	State Definition Incorrectly (possible post-condition violation)
IISD	Indirect Inconsistent State Definition
SVA	State Visibility Anomaly

II. BACKGROUND

A. Fault Categories and Software Metrics

Inheritance and polymorphism provide many benefits in creativity, efficiency and reuse but they can cause a number of anomalies and faults [2]. This study focuses on five fault types incurred by the use of polymorphism shown in Table I.

An indicative measurement of faults that provides quantitative description of certain characteristics of software products and process is code metrics. In this study, a set of object-oriented metrics [3] has been considered. These metrics are extracted from source code using the software tool Understanding C++ [12].

A number of parametric measurements are introduced as faulty causes, namely, number of appearances of syntactic fault pattern [13], syntactic and structural measures. These measures are categorized with the above fault types shown in Table II. Details of each metric can be found in [12], [14].

B. Neural Networks

This study employs two neural network techniques as the underlying mechanisms for fault prediction, namely, Multilayer Perceptron (MLP) and Radial-Basis Function Networks (RBFN). The former helps cluster input data into appropriate fault categories, whereas the latter computes, via curve-fitting approximation, fault type so obtained. Procedural details of both techniques can be found in [11].

III. DATA DESCRIPTION AND PREPROCESSING

The experiments have been carried out using 3,000 C++ classes from different sources: complete applications, individual algorithms, sample programs and various other sources on the Internet. The classes were written by different developers. The size of the classes varies between 100 and 500 lines of code. Such composition of experimental data provides a good mixture necessary for obtaining general predictive models.

Of all the 3,000 classes, half of them will be representative of faulty samples and the other half fault-free samples. The faulty samples were divided into 5 groups of 300 classes, having each fault type code listed in Table I inserted according to syntactic patterns in [13]. All faulty and fault-free samples are measured with 60 software metrics and fault parameters given in Table II and [12].

The data are nomalized to 0 and 1, and randomly grouped into three sets, namely, A, B, and C. Each group is divided into an 800-class training set and a 200-class test set. Table III shows the number of software classes in each fault type in each set.

TABLE II

FAULT/ANOMALY TYPES INDITIFIED BY SYNTACTIC PATTERNS AND PARAMETERS

Pattern/ Parameter	Fault Type				
	SDA	SDIH	SDI	IISD	SVA
ECE	X				
ECI	X				X
ECR	X				
EDIV	X		X	X	
RCE	X		X	X	
RCI	X		X		X
RCR	X		X		
RCOM	X		X		
RDIV	X	X	X		
RDUV		X			
NMI	X	X	X		X
NME	X		X	X	X
NMR	X	X	X	X	X
DepIV	X				
DiffOvrrI			X		
DiffDef			X		
NDTRAM	X		X		
NDVRAM	X		X		
NDTRM	X		X		
NDVRM	X		X		
OVrrMet	X		X		
NTIMet	X		X		
NVIMet	X		X		
NTOVrrMet	X		X		
NVOVrrMet	X		X		
IdenVar		X			
ImRef		X			
IPriV					X
RIpriV					X

TABLE III

TRAINING AND TEST DATA SETS

Fault Category	A		B		C	
	training	test	training	test	training	test
Fault-free	400	100	273	75	284	87
SDA	80	20	116	14	91	21
SDIH	80	20	102	27	112	19
SDI	80	20	123	20	89	26
IISD	80	20	95	33	98	22
SVA	80	20	91	31	126	25
Total	800	200	800	200	800	200

A. Data Preprocessing

All 60 software metrics and fault parameters were applied to the experimental data. However, not all software metrics and fault parameters contributed to faultiness of software classes. Therefore, it was necessary to select only the relevant metrics and fault parameters in order to filter out the irrelevant ones. An algorithm to select the relevant attributes is proposed. The algorithm is based on the relative difference between values of each metric applied to faulty and fault-free classes in the training set. In the following discussion, both software metrics

and fault parameters are simply referred to as metrics.

1) Set initial weight of each metric to accentuate its importance.

$$W_i^{(t)} = 0 \tag{1}$$

where

$W_i^{(t)}$ is the weight value of metric i
i $= \{1, 2, \ldots, m\}$
m is the number of metrics
t is the iteration number

2) Establish a pair of fault-free and faulty classes from the training set.

$$X = \{x_1, x_2, \ldots, x_m\}, Y = \{y_1, y_2, \ldots, y_m\} \tag{2}$$

where
X is a faulty class consists of m metrics
Y is a fault-free class consists of m metrics
For example, set up half the data in the training set to be fault-free class (C1) and the remaining half as faulty class (C2), arranging them correspondingly as depicted in Figure 1 and Figure 2.

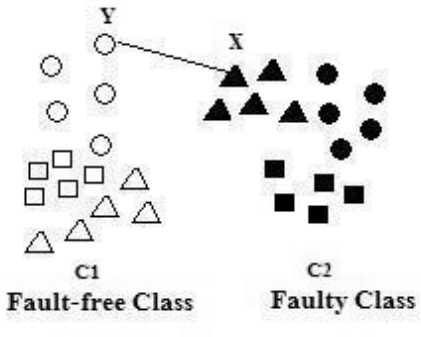

Fig. 1. A pair of fault-free and faulty classes

3) Calculate the relative difference of values of each metric pair from step 2.

$$D_i = \frac{|x_i - y_i|}{(x_i + y_i)} \times 100 \tag{3}$$

where
D_i is the relative difference of values of metric i of their respective classes
x_i is the value of metric i of the faulty class
y_i is the value of metric i of the fault-free class
This will prevent metrics intermix among their corresponding applicable domain.

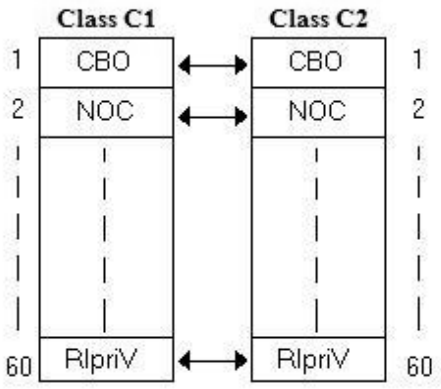

Fig. 2. Metric comparison between a fault-free class and a faulty class

4) Adjust the weight value of each metric

$$\begin{aligned} &\textbf{IF} && D_i \geq \beta \\ &\textbf{THEN} && W_i^{(t)} = W_i^{(t-1)} + 1 \\ &\textbf{ELSE} && W_i^{(t)} = W_i^{(t-1)} - 1 \end{aligned} \tag{4}$$

where $\beta = 50$ is a threshold value
5) Repeat step 2 through step 4 until all fault-free classes match with all faulty classes of the training set.
6) Consider the weight value of each metric, replacing negative values with zero

$$\begin{aligned} &\textbf{IF} && W_i < 0 \\ &\textbf{THEN} && W_i = 0 \end{aligned} \tag{5}$$

7) Normalize all weight values

$$W_i = \frac{W_i - min}{max - min} \tag{6}$$

where max and min are the maximum and minimum weight values, respectively.
8) Select the metrics with weight values above the selected threshold.

After applying the selection algorithm with threshold value 0.5, 34 relevant metrics were obtained from set A, B and C. Notice that the selected metrics from set A are the same as those from set C, while the set of selected metrics from set B is different. The metrics are shown in Table IV as the measures for fault prediction model construction discussed in Section IV.

IV. Fault Prediction Models

A. Faultiness prediction

To predict the faulty class, a predictive model has been constructed using MLP with back-propagation learning algorithm.

TABLE IV

THE FILTERED METRICS AND WEIGHT VALUES FROM DIFFERENT DATA SETS

No.	A		B		C	
	Metric	**Weight**	**Metric**	**Weight**	**Metric**	**Weight**
1	CBO	0.6629	CBO	0.7118	CBO	0.6764
2	NOC	0.9620	NOC	0.9532	NOC	0.9580
3	CountDeclInstance VariablePrivate	0.6143	CountDeclInstance VariablePrivate	0.9922	CountDeclInstance VariablePrivate	0.5161
4	CountDeclInstance VariableProtected	0.9824	CountDeclInstance VariableProtected	0.9648	CountDeclInstance VariableProtected	0.9862
5	CountDeclInstance VariablePublic	0.9760	CountDeclInstance VariablePublic	0.9948	CountDeclInstance VariablePublic	0.9762
6	CountDecl MethodPrivate	0.9739	CountDecl MethodPrivate	0.6400	CountDecl MethodPrivate	0.9978
7	NOD	0.9746	CountDecl MethodProtected	0.9760	NOD	0.9764
8	IndBase	0.8882	IndBase	0.8940	IndBase	0.8847
9	ECE	0.9988	ECE	0.9983	ECE	0.9976
10	ECI	0.9307	ECI	0.9010	ECI	0.9112
11	ECR	0.9964	ECR	0.9977	ECR	0.9957
12	EDIV	0.9104	EDIV	0.9182	EDIV	0.9307
13	RCE	0.8670	RCE	0.9080	RCE	0.9101
14	RCI	0.8314	RCI	0.8618	RCI	0.8364
15	RCR	0.9905	RCR	0.9940	RCR	0.9950
16	RCOM	0.8847	RCOM	0.8746	RCOM	0.8672
17	RDIV	0.8801	RDIV	0.8414	RDIV	0.8466
18	RUIV	0.6753	RUIV	0.5996	RUIV	0.5758
19	NDTRAM	0.5405	NDTRAM	0.5369	NDTRAM	0.5772
20	NDVRAM	0.6199	NDVRAM	0.5935	NDVRAM	0.6151
21	NDTRM	0.8002	NDTRM	0.7711	NDTRM	0.7753
22	NDVRM	0.8318	NDVRM	0.7991	NDVRM	0.8137
23	ImRef	0.9993	ImRef	0.9999	ImRef	0.9999
24	IdenVar	0.9517	IdenVar	0.9568	IdenVar	0.9601
25	DiffDef	1.0000	DiffDef	1.0000	DiffDef	1.0000
26	DiffOvrrI	0.9969	DiffOvrrI	0.9988	DiffOvrrI	0.9996
27	OVrrMet	0.9861	OVrrMet	0.9912	OVrrMet	0.9814
28	NTIMet	0.9847	NTIMet	0.9898	NTIMet	0.9788
29	NVIMet	0.9878	NVIMet	0.9907	NVIMet	0.9806
30	NTOVrrMet	0.9847	NTOVrrMet	0.9898	NTOVrrMet	0.9788
31	NVOVrrMet	0.9886	NVOVrrMet	0.9916	NVOVrrMet	0.9822
32	DepIV	0.9994	DepIV	0.9993	DepIV	0.9989
33	IPriV	0.9670	IPriV	0.9474	IPriV	0.9652
34	RIpriV	1.0000	RIpriV	1.0000	RIpriV	0.9999

Three MLP models were constructed to represent the three data set in Table III. The MLP model A, B, and C are trained with their corresponding data set and filtered metrics listed in Table IV. The objective of the models is to correctly classify the data points into fault-free and faulty groups shown in Figure 3.

The output value expected from the output node of each model is zero for the fault-free class and one for the faulty class. The learning rate of 0.35 with the help of the sigmoid function in weight adjustment to yield the correct output value.

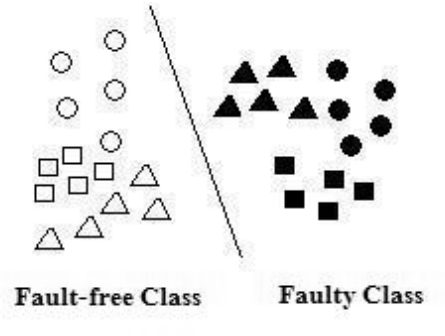

Fig. 3. Faultiness Classification

After the training process is completed, the model is re-applied to classify the test data. The output values so obtained ranging between 0 and 1 which are indecisive for data classification. Setting an acceptance ratio at 0.55, a data point is classified as a faulty class if the output of MLP is greater than this value. Otherwise, it is a fault-free class. Since the structure of model A, B, and C are similar, each model can be cross applied to other train and test data accoring to selected metrics as inputs of each model. The results of correctness percentage from all models are given in Table V.

From Table V, model A yields the highest correctness percentage when applying other data set. Model B yields the highest correctness percentage only from it's own test data, while model C yields the lowest correctness percentage from all test data. Based on the results of model A and B, it is reasonable to combine the selected metrics from both models to construct a new faultiness prediction model, having 35 combined metrics as listed in Table VI. Since model A yields the best result when appling with other data sets, data set A is used to build the new faultiness prediction model, encompassing 35 input nodes in input layer, 15 hidden nodes in hidden layer, and 1 output node in output layer as shown in Figure 4. All input values are normalized to [0, 1].

The new faultiness preidiction model with combined metrics was trained and tested with test set A, data set B, and data set C, yielding the results in Table VII. These results were further evaluated by a few selected criteria [15] as follows:

- **Type 1 error (T1):** This error occurs when a faulty class is classified as fault-free; T1 = 5.32%
- **Type 2 error (T2):** This error occurs when a fault-free

TABLE V

RESULT FROM FAULTINESS PREDICTION MODELS

Test set	Model		
	A	B	C
A	92.50%	92.70%	91.80%
B	94.40%	92.50%	91.60%
C	89.90%	91.40%	91.01%

TABLE VI

THE COMBINED FILTERED METRICS

No.	Metric
1	CBO
2	NOC
3	CountDeclInstanceVariablePrivate
4	CountDeclInstanceVariableProtected
5	CountDeclInstanceVariablePublic
6	CountDeclMethodPrivate
7	CountDeclMethodProtected
8	NOD
9	IndBase
10	ECE
11	ECI
12	ECR
13	EDIV
14	RCE
15	RCI
16	RCR
17	RCOM
18	RDIV
19	RUIV
20	NDTRAM
21	NDVRAM
22	NDTRM
23	NDVRM
24	ImRef
25	IdenVar
26	DiffDef
27	DiffOvrrI
28	OVrrMet
29	NTIMet
30	NVIMet
31	NTOVrrMet
32	NVOVrrMet
33	DepIV
34	IPriV
35	RIpriV

TABLE VII

RESULT FROM THE FAULTINESS PREDICTION MODEL WITH COMBINED METRICS

Test set	Correctness percentage
A	94.00%
B	92.80%
C	92.10%

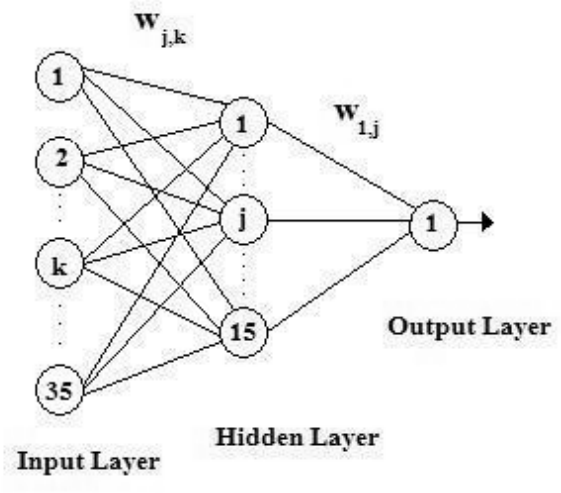

Fig. 4. Structure of Faultiness Prediction Model with combined metrics

TABLE VIII
METRIC GROUP

Range of total weight values	Metric Group
$6 \leq \texttt{tw} < 6.5$	M1
$6.5 \leq \texttt{tw} < 7$	M2
$7 \leq \texttt{tw} < 7.5$	M3
$7.5 \leq \texttt{tw}$	M4

class is classified as faulty; T2 = 2.09%

- **Quality achieved (C):** If all faulty classes are properly classified, defects will be removed by extra verification; C = 91.53%
- **Inspection (I):** Inspection measures the overall verification cost by considering the percentage of classes that should be verified; I = 59.55%
- **Waste Inspection (WI):** Waste inspection is the percentage of classes that do not contain faults but are verified because they have been classified incorrectly; WI = 3.51%

The total weight value of each input node indicates how important a particular input node is for faultiness prediction. All metrics are then distributed to 4 groups based on the range of weight values as shown in Table VIII and Table IX. The metrics group M1 has the least effect while the metrics group M4 has the greatest effect on faultiness prediction.

B. Fault Type Identification

Fault type identification model is based on RBFN technique as mentioned earlier. The objective of the model is to cluster faulty classes into groups based on fault type as shown in the example of three fault types in Figure 5. The model consists of 35 input nodes in the input layer, a number of hidden nodes in the hidden layer (this number is determined during the training process), and 5 output nodes in the output layer that form an output vector. The output vector denotes the type of fault

TABLE IX
DISTRIBUTION OF METRIC GROUPING BASED ON TOTAL WEIGHT VALUE BETWEEN INPUT LAYER AND HIDDEN LAYER

Input node	Metric	Metric Group			
		M1	M2	M3	M4
1	CBO			X	
2	NOC			X	
3	CountDeclInstance VariablePrivate				X
4	CountDeclInstance VariableProtected			X	
5	CountDeclInstance VariablePublic			X	
6	CountDecl MethodPrivate			X	
7	CountDecl MethodProtected			X	
8	NOD			X	
9	IndBase				X
10	ECE			X	
11	ECI			X	
12	ECR		X		
13	EDIV			X	
14	RCE			X	
15	RCI			X	
16	RCR				X
17	RCOM				X
18	RDIV			X	
19	RUIV		X		
20	NDTRAM				X
21	NDVRAM			X	
22	NDTRM			X	
23	NDVRM		X		
24	ImRef		X		
25	IdenVar	X			
26	DiffDef				X
27	DiffOvrrI				X
28	OVrrMet		X		
29	NTIMet			X	
30	NVIMet				X
31	NTOVrrMet			X	
32	NVOVrrMet				X
33	DepIV			X	
34	IPriV			X	
35	RIpriV			X	

in binary format as '10000', '01000', '00100', '00010', and '00001', representing SDIH, IISD, SVA, SDA, and SDI faults, respectively.

During the experiment, training data were used to generate the weights between the hidden layer and the output layer. If the network yields low accuracy, the number of hidden node will be incremented by one. This restructuring by node-plus-one progression continues until the desired accuracy is acquired or the number of hidden nodes reaches the number of training data points. At which point, reorganization must be done by repeating the attribute selection algorithm and proceed along the same steps described. The implication of this reorganization is that some, or all, selected metrics do not contribute to the faulty behavior of software components, whereby prediction accuracy will fall short of the acceptable range.

Based on the above procedures, the proposed model yields

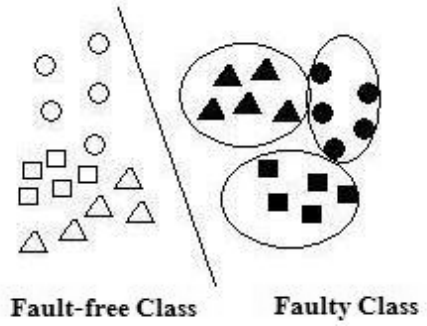

Fig. 5. Fault Type Clustering

TABLE X
RESULT FROM THE FAULT TYPE PREDICTIVE MODEL

Fault Category	Predicted Fault Type				
	SDA	SDIH	SDI	IISD	SVA
SDA	193	9	44	7	9
SDIH	5	264	6	2	3
SDI	36	10	222	7	3
IISD	10	3	9	239	7
SVA	5	2	2	4	280

an 90.05% prediction accuracy on test data from set A, all data from data set B and C. The results shown in Table X relate the actual number of each type of fault and the results of classification.

From the model's structure, the weights assigned to the hidden layer and the output layer of the structure of fault type model. The weight value of each hidden node designates which output node should have effect on. The maximum weight value obtained from all hidden nodes that exert on a given output node indicates the dominance of the hidden node.

To explore which metrics dominate the fault type of a given hidden node that represents all 35 metrics, an algorithm is proposed as follows:

1) Choose a fault type to find a set of representative metrics, for example, SDIH fault.
2) Find the hidden nodes that effect the fault type from the results. There are 2 hidden nodes in this case.
3) Identify the set of classes from the training data where the selected fault is originated. There are 80 classes from the training data that contain SDIH fault.
4) Calculate the difference between each metric of a training class and the same metric of a hidden node (note that each class has 35 metrics, so does each hidden node).

$$V_i = |c_i - h_i| \tag{7}$$

where
V_i is the difference of values of metric i of the class and the hidden node
c_i is the value of metric i of the class
h_i is the value of metric i of the hidden node

5) Repeat step 4 for the selected fault type until all classes and hidden nodes are considered.
6) For each fault type, caculate the total difference of each metric value from Step 5.

$$TotV_i = \sum_{j=1}^{m}\sum_{k=1}^{n} V_i^{(j,k)} \tag{8}$$

where
$TotV_i$ is the total difference of value of metric i of all classes and hidden nodes
$V_i^{(j,k)}$ is the difference of value of metric i of training class k and hidden node j
m is the number of hidden nodes for the selected fault type
n is the number of training classes for the selected fault type

7) Nomalize all total difference values

$$TotV_i = \frac{TotV_i - min}{max - min} \tag{9}$$

where max and min are the maximum and minimum total difference values, respectively.
8) repeat Steps 1-7 above until all fault types are considered.

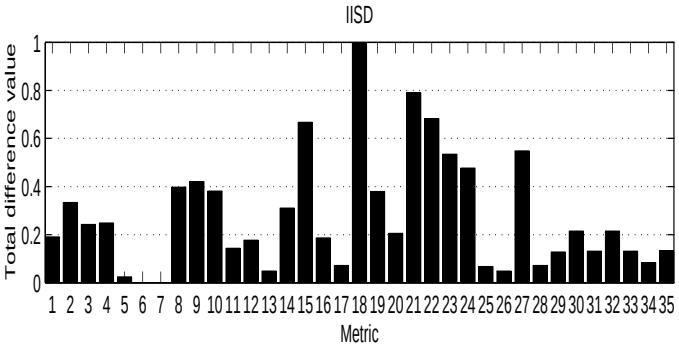

Fig. 6. The total difference of value of each metric between hidden nodes and training classes with IISD fault

Figure 6 and Figure 7 show the effects of IISD and SDA metrics have on particular fault types. The zero total difference value means that the corresponding metrics of that training class and hidden node are the same and thus has no effect on the fault type. On the other hand, if the total difference metric between the training classes and the hidden nodes is high, the metric will likely contribute to the fault prediction of the software.

V. DISCUSSION

The proposed software metric attribute selection algorithm proved to be effective in determining the significance of each metric and characterization of software faultiness. Based on the two predictive models, the proposed approach is able to predict faultiness of a class with more than 90% accuracy.

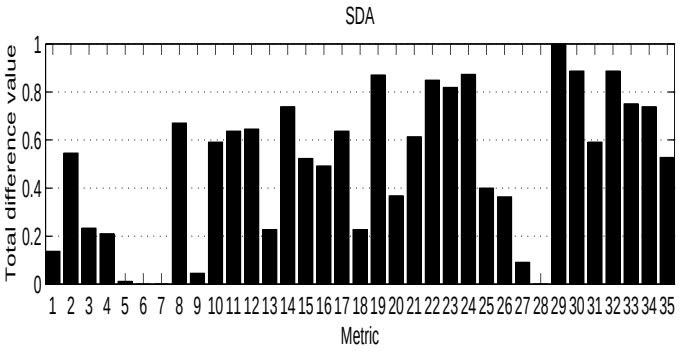

Fig. 7. The total difference of value of each metric between hidden nodes and training classes with SDA fault

According to the evaluation criteria, the faulty classes can be detected in 91.53% of test cases, the inspection cost for verification is 59.55%, and the waste cost is 3.51%. Only 2.09% of faulty classes were undetected.

In addition, faulty type prediction yields an 90.05% of test cases. Closer examination reveals that some misclassification accured in all fault types are shown in Table X. The reason being is that there are overlapping parameters amoung fault types. Such inherent correlation induces erroneous classifcation obtained from the proposed neural network techniques. We envision that some forms of fine grained metric preprocessing should be carried out to alleviate the aforementioned caveats.

VI. Conclusion

The application of neural networks in predicting software faults requires enormous amounts of data. Analyzing the data is a major undertaking that must be carried out with the help of proper models. This study proposes an algorithm for metrics selection and a systematic approach to categorize closely related data using neural networks. MLP neural network with back-propagation learning algorithm has been used to identify faulty classes, while RBF neural network to categorize the faults according to several defined fault types.

Some approaches have been explored to improve the predictive model. The first possibility is to add more parameters, however it is very difficult to find a proper set of parameters that can represent characteristic of each fault type. Second, proper data classification technique would enhance not only the efficiency of the training process, but also the performance of the predictive model in terms of precision. Accurate predictions obtained from such a good reliability model would lead to higher efficiency of software process and quality of resulting software products.

Acknowledgments

This research is financially supported by The Office of Higher Education Commission, Ministry of Education, Thailand.

References

[1] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability Measurement, Prediction, Application*. the United States of America: McGraw-Hill Book Company, 1987.

[2] J. Offutt and R. Alexander, "A fault model for subtype inheritance and polymorphism," in *12th International Symposium on Software Reliability Engineering*, November 2001, pp. 84 – 95.

[3] M. M. T. Thwin and T.-S. Quah, "Application of neural network for predicting software development faults using object-oriented design metrics," in *Proceedings of the 9th International Conference on Neural Information Processing*, November 2002, pp. 2312 – 2316.

[4] T. Kamiya, S. Kusumoto, and K. Inoue, "Prediction of fault-proneness at early phase in object-oriented development," in *Proceedings of the Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, May 1999, pp. 253 – 258.

[5] L. Emam, J. W*ü*st, and J. W. Daly, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, pp. 63 – 75, 2001.

[6] L. Briand, J. W*ü*st, and J. W. Daly, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, pp. 245 – 273, 2000.

[7] L. C. Briand, J. W*ü*st, and J. W. Daly, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 706 – 720, 2002.

[8] D. Glasberg and K. E. Emam, "Validating object-oriented design metrics on a commercial java application," Technical Report NRC/ERB-1080, September 2000.

[9] Y. Mao, H. A. Sahraoui, and H. Lounis, "Reusability hypothesis verification using machine learning techniques: a case study," in *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, October 1998, pp. 84 – 93.

[10] P. Eklund and L. Kallin, "Fuzzy systems," Febuary 2000, lecture Notes prepared for courses at the Department of Computing Science at Umeå University, Sweden.

[11] S. Haykin, *Neural Networks*. the United States of America: Prentice Hall, 1999.

[12] "*Understand for C++*Scientific Toolworks, Inc., St. George, Utah, `http://www.scitools.com`."

[13] R. T. Alexander, J. Offutt, and J. M. Bieman, "Syntactic fault patterns in oo programs," in *Eight International Conference on Engineering of Complex Computer Software*, December 2002, pp. 193 – 202.

[14] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476 – 493, 1994.

[15] F. Lanubile, "Evaluating predictive models derived from software measure," *Journal of Systems and Software*, vol. 38, no. 1, pp. 225 – 234, 1996.