# Software Fault Prediction Using Fuzzy Clustering and Radial-Basis Function Network

Atchara Mahaweerawat Peraphon Sophasathit Chidchanok Lursinsap
Advanced Virtual and Intelligent Computing Center
Department of Mathematics, Faculty of Science
Chulalongkorn University, Bangkok 10330, Thailand
E-mail: matchara@lycos.com peraphon.s@chula.ac.th
lchidcha@pioneer.netserv.chula.ac.th

**Abstract**: This paper presents a new approach for predicting software faults by means of fuzzy clustering and radial-basis function techniques. We employed fuzzy subtractive clustering to divide historical and development data into clusters. Next we applied the radial-basis function network to predict software faults that occurred in the component residing in each cluster. In so doing, we were able to predict software faults reasonably accurate.

**Keywords**: fuzzy subtractive clustering, multilayer perceptron, back-propagation, radial-basis function, software fault prediction, regularization networks, generalized radial-basis function networks.

## 1. Introduction

Software reliability is the probability of failure free operations of a computer program executing in a specified environment for a specified time [8]. Software reliability is considered a software quality factor that aids in predicting software quality using standard predictive models. There are many approaches for predicting software quality, most of which yield some forms of quality indicators. One popular indicator is known as software fault. Software fault prediction utilizes historical and development data to arrive at a conclusive decision whether the software in question is at fault.

Recent research and development in fuzzy logic and neural networks proliferate the accuracy and efficiency of fault prediction considerably. Fuzzy logic is an extension of standard Boolean logic [9]. The fundamental concept is a fuzzy set in which each element in the set is characterized by its grade of membership of the set [6]. A membership function maps an element of the set in a given domain to an appropriate membership grade value, ranging between zero and one. Typical membership functions are the Gaussian distribution function, the sigmoid curve quadratic, and cubic polynomial curve, etc.

It is often the case that myriad of data are collected as a result of software reliability study. Applying fuzzy logic to such an overwhelming number of data points may be time-consuming and yield irrelevant estimators that do not reflect the inherent characteristics of the software faults. As a consequence, some preprocessing may be required to classify those representing data in clusters such that more accurate fault patterns can be extracted from each cluster.

Our definition of a cluster is a group of entities with similar properties [10]. Entities from different clusters cannot be similar. Thus, a set of points can be grouped into one or more clusters, depending on the properties of those points. Usually, the obvious property is distance among the points. Unfortunately, sorting these points into clusters where they belong is, in many cases, not straightforward since there are points that lie between different clusters of points which could belong to either adjacent clusters. This is where fuzzy clustering comes into play. To determine if a point belongs to a given cluster (and thus assigned the same membership grade value), every point in the cluster must lie within a given proximity. Thus, the distance between the point in question and a neighboring point must be smaller than the distance to the rest of the points. In addition, points in different clusters are assigned different grades of membership based on the selected membership function. Such an assignment calls for an algorithm that must be able to compute the membership grade value in a finite number of iterations. This is a tall order that ordinary distance algorithmic approach cannot accomplish.

Neural networks, on the other hand, introduce yet another data clustering technique called the Radial-Basis Function (RBF). The RBF technique is suitable for interpolation problem which is viewed as a curve fitting problem in high-dimensional space [7]. The learning process involved is equivalent to finding a surface in a multi-dimensional space that provides the best fit to the training data, with the criteria for 'best fit' being measured in some statistical sense. Thus, we employed the RBF technique to predict software fault for this study.

Based on the aforementioned techniques, we constructed two models in our experiment to classify software artifacts as high-risk which is likely to contain faults, and low-risk which is likely to be fault free. The first model utilized multilayer perceptron (MLP) with back-propagation algorithm [2], whereas the second model employed radial-basis function network. Software data were measured and collected using many different software metrics, namely, lines of code (*LOC*), non-comment lines of code (*NCLOC*), Halstead program length (*N*), Halstead volume (*V*), McCabe cyclomatic complexity (*V(G)*), Halstead number of unique operands ($n_2$), Halstead total number of operands ($N_2$), Henry&Kafura fan-in/fan-out ($fan_{in}/fan_{out}$), Henry&Kafura information flow (*IF*), and density of comments (*DC*). These software metrics can be derived by the followings:

- McCabe cyclomatic complexity (*V(G)*) [14]
  $V(G) = e - n + 2p$
  where
  *G* = program flow graph
  *e* = number of edges
  *n* = number of nodes
  *p* = number of unconnected paths
- $fan_{in}$: the $fan_{in}$ of a module M is the number of local flows that terminates at M, plus the number of data structures from which information is retrieved by M [3]
- $fan_{out}$: the $fan_{out}$ of a module M is the number of local flows that emanates from M, plus the number of data structures that is updated by M [3]
- The information flow (*IF*) is measured by the formula [3]
  $IF = (fanin \times fanout)$
- Halstead's software science [14] is defined as
  Length (*N*) $= N_1 + N_2 = n_1 log_2 (n_1) + n_2 log_2 (n_2)$
  Volume (*V*) $= N log_2 (n) = N log_2 (n_1 + n_2)$
  where
  $n_1$ = the number of distinct operators that appears in a program
  $n_2$ = the number of distinct operands that appears in a program
  $N_1$ = the total number of distinct operator occurences
  $N_2$ = the total number of distinct operand occurrences
- Density of comments (*DC*) [3]
  $DC = CLOC/LOC$
  where *CLOC* is the number of comment lines of program text.

We divided data into several groups using fuzzy subtractive clustering technique. These groups of data were then fed into the models for software fault prediction. The predictive quality results from both models were statistically compared to gauge their misclassification rate, quality achieved, and verification cost. It was found that the RBF model yielded more accurate prediction than the MLP model. As a consequence, we will improve the accuracy and performance in future studies on software fault predictions.

## 2. Software Reliability and Prediction Models

The proliferation of computer technology has brought about increasingly complicated software demand. As complexity grows, so does quality needs. Unfortunately, complexity works against quality, thus resulting in low reliability. New software development paradigms are employed to cope with such stringent requirements, for example, zero-defect software quality assurance, software fault detection, and software reliability measurement, etc.

Software reliability involves various precautions to guard against faulty operations, ranging from testing, production, and maintenance. As it is generally known that exhaustive test is in no way practical, failure is thus inevitable. In principle, we define software failure as the departure of the external results of program operation from equipment. Such discrepancies are referred to as faults.

A fault is the defect in the program that causes failure [8]. Faults can be classified based on [13]

- *Locality* which includes atomic component faults, composite component faults, system level faults (i.e., operator faults, replication faults), and external faults (i.e., environment faults, user faults);
- *Effect* which includes value faults (a result from a computation that does not meet the system specification), timing faults (a processor or service which is not delivered or completed within the specified time interval);
- *Cause* which includes immediate cause (such as resource depletion faults that involve a section of the system being unable to obtain the resources required to perform its task, logic faults that result from the system not behaving according to specification, physical faults that are caused by hardware breaks or mutation in executable software), ultimate cause (specification faults, configuration faults);
- *Duration* which includes persistent faults (that remain active for a significant period of time), transient faults (that remain active for a short period of time); and
- Effect on *system state* which is characterized by faults describing the system states.

Study [8] shows that there are two principal factors that affect the behavior of failure. The first factor is the number of faults in the software being executed, and the second factor concerns with the execution environment or operational profile of execution. The fact that software development process is still a human-oriented activity makes it impossible to produce fault-free software products. As such, faults are usually introduced when the code is being developed by programmers, during original design, adding new features, or design changes, or fault repair. Various efforts have been taken to remove new faults being introduced during maintenance, namely, regression test, cleanroom technique, etc. Removal process often takes place when the first fault is detected. This process, from a practical standpoint, is dependent on the efficiency which faults are found and removed.

To model software reliability, we considered the process involving the above principal factors from three viewpoints, namely, *fault introduction*, *fault removal*, and *the environment*. Fault introduction depends primarily on the characteristics of the developed code (code created or modified for the application) and the characteristics of the development process [8]. The most significant code charac-teristic is size, whereas the development process characteristics encompass software engineering technologies, tools, and levels of experience of personnel. Such a software reliability model specifies a general failure process dependency based on the aforementioned factors. This dependency can be defined by establishing the parameters of the model through estimation or prediction. The former rests on statistical techniques being applied to failure data taken from the programs, whereas the latter determines the properties of software product quality and the development process using fuzzy logic or neural network techniques.

In this paper, we employed the count of software faults and a few selected software fault prediction parameters (or metrics) as the bases for our reliability model. Software reliability analyzes can then be carried out by means of the proposed model operating on sample code. We hope that the proposed model will be able to predict some predominant characteristics and causes of software fault.

This paper is organized as follows. Section 3 discusses our approach for data clustering technique using fuzzy subtractive method. Section 4 establishes the fundamental of the proposed fault prediction model based on well-defined neural network principles. Section 5 derives the radial-basis membership function. Some evaluation criteria for the experiment are described in Section 6 and 7, respectively. The conclusion and future work are given in Section 8.

## 3. Fuzzy Subtractive Clustering

There are many fuzzy clustering techniques to group data points in clusters. One popular technique is Fuzzy C-Means clustering which is a simple and straightforward approach. This technique, however, requires two predefined clusters where every data point membership depends on a membership grade. The subtractive clustering, on the other hand, does not require a predefined number of clusters. The subtractive clustering is a fast, one-pass algorithm for estimating the number of clusters and cluster centers in a set of data [10]. Each data point is considered a potential cluster center, which is calculated from the density of the surrounding data points [1]. Given a group of data points *{$x_1$, $x_2$, . . . , $x_n$}* and all data points are normalized with respect to each variable (vector) associated with $x_i$, the initial potential value of the data point $x_i$ is defined as [4]

$$\begin{aligned} P_i &= \sum_{j=1}^{n} e^{-\frac{\|x_i - x_j\|^2}{(r_a/2)^2}} \\ &= \sum_{j=1}^{n} e^{-\alpha\|x_i - x_j\|^2} \end{aligned} \qquad (1)$$

where $\alpha = \frac{4}{r_a^2}$

$\|.\|$ is the Euclidean distance

$r_a$ is a positive constant

The constant $r_a$ is effectively a normalized radius defining the neighborhood where any data points outside this radius have little influence on the potential. The data point with the highest potential value is selected as the first cluster center.

Let $x_1^*$ be the first cluster location with the potential value $P_1^*$. The potential value of each data point $x_i$ is iteratively adjusted by [4, 15]

$$\begin{aligned} P_i &= P_i - P_1^* e^{\frac{-\|x_i - x_1^*\|^2}{(r_b/2)^2}} \\ &= P_i - P_1^* e^{-\beta\|x_i - x_1^*\|^2} \end{aligned} \qquad (2)$$

where $\beta = \frac{4}{r_b^2}$

$r_b$ is a positive constant

The constant $r_b$ is normally larger than $r_a$ and suggested to be $1.25 r_a$ [15] to avoid obtaining closely spaced cluster centers. The potential value of each data point near the first cluster center will be greatly reduced by Equation 2, and thus the data point will unlikely be chosen as the next cluster center [4].

The data point with the highest remaining potential is obtained and set as the next cluster center. The potential of the remaining data points can be recalculated according to their distance to the new cluster center. The general potential revision formula can be expressed as [4]

$$P_i = P_i - P_k^* e^{-\beta \|x_i - x_k^*\|^2} \qquad (3)$$

where $x_k^*$ is the location of the kth cluster center

$P_k^*$ is its potential value

The procedure of finding a new cluster center is repeated until a sufficient number of cluster centers are generated. The stopping criterion is that the remaining potential of all data points are lower than some fractions of the first cluster center potential $P_1^*$ which is usually set to $P_k^* < 0.15 P_1^*$ as suggested in [4, 15].

## 4. Multilayer Perceptron and Backpropagation Learning

Multilayer perceptrons (MLP) are multilayer feedforward networks [7]. The network consists of a set of sensory units that constitute the input layer, one or more hidden layers of computation nodes, and output layer of computation nodes. The input signal propagates through the network in a forward direction on a layer-by-layer basis.

### 4.1. Back-propagation Learning Algorithm

Back-propagation (BP) learning consists of two phases, namely, learning phase and working phase. In learning phase, a set of input patterns (training set) are presented to the input layer together with their corresponding desired output patterns. A small random initial weight value is assigned to each connection between nodes in the input layer and the hidden layer. As each input pattern is applied, the actual output is recorded. The weights between the output layer and the previous layer (hidden layer) are recalculated using the generalized delta rule which will be described later. The adjustment reduces the difference between the network actual outputs and the desired outputs for a given input pattern.

The input to each node for successive layers is the sum of the scalar products of the incoming vector components with their respective weights. Thus the input to a node *j* (Figure 1) is given by [12]

$$input_j = \sum_i w_{ji} \cdot out_i \qquad (4)$$

where $w_{ji}$ is the weight connecting node *i* to node *j* and $out_i$ is the output of node *i*. No calculation is performed at the input layer since it is just feeding the value of input patterns to the network. The output of a node *j* is, therefore,

$$out_j = f(input_j) \qquad (5)$$

and this output is sent to all nodes in subsequent layers. This computation is continued through all layers of the network until the output layer is reached. At that point, the output vector is generated.

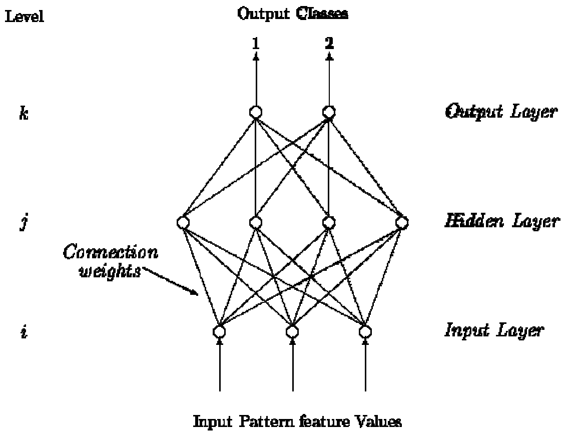

**Figure1: Architecture of Multilayer Perceptron.**

The function *f* is the activation function of each node. In this study, we employed the sigmoid activation function as follows:

$$f(x) = \frac{1}{1 + \exp(-x)} \qquad (6)$$

where *x* = $input_j$. Figure 2 illustrates sigmoid curve of a node acting like a thresholding device [12].

Based on the difference error term or $\delta$ term in the output layer, the weight can be computed by [7]

$$w_{kj}(n+1) = w_{kj}(n) + \eta(\delta_k out_k) \qquad (7)$$

where $w_{kj}(n+1)$ and $w_{kj}(n)$ are the weights connecting nodes *k* and *j* at iteration *(n+1)* and *n*, respectively, $\eta$ is a learning rate parameter. The $\delta$ terms of hidden layer nodes are computed and the weights connecting the hidden layer to the previous layer (another hidden layer or input layer) are updated accordingly. This calculation is repeated until all weights have been adjusted.

The $\delta$ term in previous equation is often referred to as the rate of change of error with respect to the input of node *k*, and can be written as [12]

$$\delta_k = (d_k - out_k) f'(input_k) \qquad (8)$$

for nodes in the output layer, and

$$\delta_j = f'(input_k) \sum_k \delta_k w_{kj} \qquad (9)$$

for nodes in the hidden layers, where $d_k$ is the desired output of node $k$.

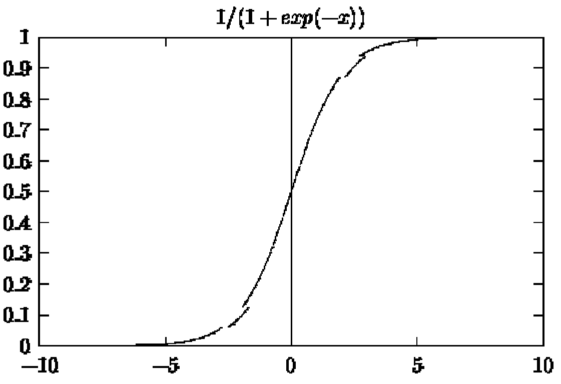

**Figure 2: The Sigmoid Activation Function.**

The fundamental of back-propagation algorithm employed in training phase is a gradient descent optimization procedure which minimizes the mean squared error between network output and the desired output of all input patterns $P$ as follows [12]:

$$E = \frac{1}{2P}\sum_p \sum_k (d_k - out_k)^2 \qquad (10)$$

The training set is presented iteratively to operate the network, whereby the weights are updated until their values become stabilized according to the following criteria:

- a user-defined error tolerance is achieved, or
- a maximum number of iterations is reached.

## 4.2. Applying Cross Validation to Stopping Criteria

In the training phase of a neural network, the important goal is to obtain optimal generalization performance. Generalization performance means small errors on examples not seen during training [5]. Standard neural network architectures, such as the multilayer perceptron (MLP), are prone to overfitting. Such a behavior is often the case while the network seems to perform better (where the errors on the training set decrease). Errors begin to surface at some point during training and become worse, e.g., the errors on unseen examples increase.

There are two ways to overcome overfitting [5]. The first way is to reduce the number of dimensions of the parameter space or the effective size of each dimension. The other way is early stopping. Early stopping can be carried out either interactively or automatically, i.e., based on human judgement or automatic stopping criteria, respectively.

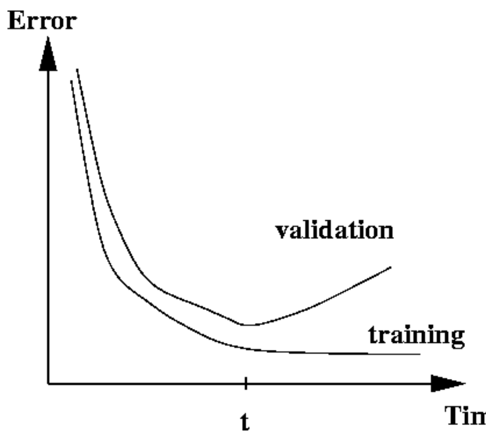

**Figure 3: Idealized Training and Generalization Error Curves.(Vertical:Errors;Horizontal:Time)**

Figure 3 shows the evolution overtime of the per-example error on a training set and on a test set not used for training. The following steps describe how to apply cross validation to early stopping [5]:

- split the training data into a training set and a cross validation set,
- train only on the training set and evaluate the per-example error on the validation set once in a while,
- stop training as soon as the error on the cross validation set is higher than it was checked the last time, and
- use the weights the network had in that previous step as the result of the training run.

This method uses cross validation set to simulate the behavior on the test set, assuming that the error on both sets will be similar.

There are a number of plausible proposed stopping criteria that were classified by [5] into three classes based on the definitions below. Given

- $E$ be the objective function (error function) of the training algorithm,
- $E_{tr}(t)$ be the average error per-example over the training set, measured after epoch $t$,
- $E_{va}(t)$ be the corresponding error on the validation set and is used by the stopping criterion,
- $E_{te}(t)$ be the corresponding error on the test set, and
- $E_{opt}(t)$ be the lowest validation set error obtained in epochs up to $t$:

$$E_{opt}(t) = \min_{t' \le t} E_{va}(t')$$

The generalization loss at epoch $t$ is the relative percentage increase of validation error over the minimum-so-far (in percent) [5]:

$$GL(t) = 100 \times \left( \frac{E_{va}(t)}{E_{opt}(t)} - 1 \right) \qquad (11)$$

A high generalization loss directly indicates overfitting, hence the stopping criteria of training. Consequently, the first class of stopping criteria ($GL_\alpha$) is to stop as soon as the generalization loss exceeds a certain threshold [5], that is,

$GL_\alpha$ : stop after first epoch *t* with *GL(t)* $> \alpha$

As training error still decreases rapidly, no overfitting takes place. When the error decrease rate begins to level, overfitting will likely occur. A training strip of length *k* is defined as a sequence of *k* epochs numbered *n+1,...,n+k*, where *n* is divisible by *k*. The training progress (in per thousand) measured after such a training strip is [5]

$$P_k = 1000 \times \left( \frac{\sum_{t'=t-k+1}^{t} E_{tr}(t')}{k \times \min_{t'=t-k+1}^{t} E_{tr}(t')} \right) \quad (12)$$

The above measures the average training error for the duration when the strip becomes larger than the minimum strip.

The second class of stopping criteria is defined as the quotient of generalization loss and progresses as follows [5]:

$PQ_\alpha$ : stop after first end-of-strip epoch *t* with $\frac{GL(t)}{P_k(t)} > \alpha$

Cross validation errors are measured only at the end of each strip, depending on the sign of changes in the generalization.

The third class of stopping criteria is based on the notion of stopping when generalization errors increase in *s* successive strips [5], that is,

$UP_s$: stop after epoch *t* if $UP_{s-1}$ stops after epoch *t-k* and $E_{va}(t) > E_{va}(t-k)$

$UP_1$: stop after first end-of-strip epoch *t* with $E_{va}(t) > E_{va}(t-k)$

This definition means that when the validation error has increased not only once, but during *s* consecutive strips and assume that such increases indicate the beginning of final overfitting, independent of how large the increases actually are.

From all stopping criteria classes (*GL*, *PQ*, and *UP*), we employed the class *PQ* as the stopping criteria of the training under back-propagation learning approach.

## 5. Radial-Basis Function Networks (RBFN)

### 5.1. The Interpolation Problem

The radial-basis function networks (RBFN) concerns with the problem of curve-fitting by approximation in high-dimensional spaces. The technique is used in our training process as it is equivalent to finding an interpolating surface in the multidimensional space that provides the best fit to the training data, measured by pre-selected statistical criteria.

The curve-fitting or interpolation problem can be stated as follows [7]:

Given a set of *N* different points $\{x_i \in \Re^{m_0} | i = 1,2,\dots,N\}$ and a corresponding set of *N* real numbers $\{d_i \in R^1 | i = 1,2,\dots,N\}$, a function $F : \Re^N \to \Re^1$ that satisfies the interpolation conditions is given by

$$F(x_i) = d_i \qquad i = 1,2,\dots,N \quad (13)$$

### 5.2. Radial-Basis Functions

The radial-basis functions (RBF) technique suggests that the interpolation function $F$ should be constructed as follows [7]:

$$F(x) = \sum_{i=1}^{N} w_i \varphi(\|x - x_i\|) \quad (14)$$

where $\{\varphi(\|x - x_i\|) | i = 1,2,\dots,N\}$ is a set of *N* arbitrary (generally nonlinear functions) radial-basis functions, and $\|\cdot\|$ is the Euclidean norm. The known data points $\{x_i \in \Re^{m_0} | i = 1,2,\dots,N\}$ are defined to be the centers of the radial-basis function.

Inserting the interpolation conditions of Equation 13 in Equation 14, we get a set of simultaneously linear equations with unknown coefficients (or weights) of the expansion $w_i$

$$\begin{bmatrix} \varphi_{11} & \varphi_{12} & \cdots & \varphi_{1N} \\ \varphi_{21} & \varphi_{22} & \cdots & \varphi_{2N} \\ \vdots & \vdots & & \vdots \\ \varphi_{N1} & \varphi_{N2} & \cdots & \varphi_{NN} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \end{bmatrix} \quad (15)$$

where

$$\varphi_{ji} = \varphi(\|x_j - x_i\|) \quad (j,i) = 1,2,\dots,N \quad (16)$$

Let

$d = [d_1, d_2, \ldots, d_N]^T$ be desired output vector

$w = [w_1, w_2, \ldots, w_N]^T$ be linear weight vector

$N$ is the size of the training sample

$\Phi$ denote an N-by-N matrix with element $\varphi_{ji}$

$$\Phi = \{\varphi_{ji} \mid (j,i) = 1, 2, \ldots, N\} \tag{17}$$

This matrix is called the *interpolation matrix* and Equation 15 can be written in compact form [7]

$$\Phi w = d \tag{18}$$

The unknown weights($w$) can be obtained by solving the following linear equation:

$$w = \Phi^{+} d \tag{19}$$

where $\Phi^{+}$ is the pseudoinverse of $\Phi$

$$\Phi^{+} = \left(\Phi^T \Phi\right)^{-1} \Phi^T \tag{20}$$

There are many functions that can be used in radial-basis function, for example,

1. Multiquadrics

   $\varphi(r) = (r^2 + c^2)^{\frac{1}{2}}$ for some $c>0$ and $r \in \Re$

2. Inverse multiquadrics

   $\varphi(r) = \dfrac{1}{(r^2 + c^2)^{\frac{1}{2}}}$ for some $c>0$ and $r \in \Re$

3. Gaussian functions

   $\varphi(r) = e^{\left(-\frac{r^2}{2\sigma^2}\right)}$ for some $\sigma > 0$ and $r \in \Re$

The multivariate Gaussian function gives two important properties that make it a proper choice for building radial-basis function [7] in our work, i.e., it is both translation and rotation invariant. The multivariate Gaussian function with these properties is also called *Green's function*, which has the form:

$$G(x, x_i) = \exp\left(-\frac{\|x - x_i\|^2}{2\sigma_i^2}\right) \tag{21}$$

## 5.3. Regularization Networks

Referring to the RBF network structure, the regularization network consists of three layers, namely,

- *input layer* which is composed of a number of input nodes equal to the dimension $m_0$ of the input vector $x$,
- *hidden layer* which is composed of nonlinear units that are connected directly to all nodes in the input layer, and
- *output layer* which is composed of a single linear unit fully connected to the hidden layer.

There is one hidden unit for each data point $x_i$, $i = 1, 2, \ldots, N$, where $N$ is the size of the training sample, Green's function is used as the activation function of the individual hidden unit. Therefore the output of the $i^{th}$ hidden unit is $G(x,x_i)$. The output of the network is a linearly weighted sum of the outputs of the hidden units.

*The regularization network models the interpolation function F as a linear superposition (linear weighted sum) of multivariate Gaussian functions which number is equal to the number of the given input example N* [11], that is,

$$F(x) = \sum_{i=1}^{N} w_i G(x, x_i) \tag{22}$$

substituting Equation 21 in Equation 22, we get

$$F(x) = \sum_{i=1}^{N} w_i \exp\left(-\frac{\|x - x_i\|^2}{2\sigma_i^2}\right) \tag{23}$$

where $w_i$ is the weight corresponding to connecting link $i$ from a node in the hidden layer to an output node.

## 5.4. Generalized RBF Networks

In the regularization networks, the number of Green functions is equal to the number of the training examples. This is computationally inefficient in practice since it may require a combination of very large number of basis functions.

In real world situations, finding the linear basis function weight needs to invert a very large $N \times N$ matrix which is quite computationally complex. To overcome this problem, the network complexity needs to be reduced. This means that we will attempt to find a solution that approximates the solution produced by the regularization networks. Let $F^*(x)$ be the approximated solution [7], that is,

$$F^*(x) = \sum_{i=1}^{m_1} w_i \varphi_i(x) \tag{24}$$

where $\{\varphi_i(x \mid i = 1,2,\ldots,m_1)\}$ is a new set of basis functions. Typically, the number of basis functions is less than the number of data points ($m_1 \le N$), and the $w_i$ form a new set of weights. We get

$$\varphi_i(x) = G(\|x - t_i\|),\ i = 1,2,\ldots,m_1 \tag{25}$$

where the set of centers $\{t_i \mid i = 1,2,\ldots,m_1\}$ must be determined. In order to reduce the number of nodes in the hidden layer, we will redefine $F^*(x)$ as

$$F^*(x) = \sum_{i}^{m_1} w_i G(x, t_i) = \sum_{i=1}^{m_1} w_i G(\|x - t_i\|) \tag{26}$$

The result is referred to as the Generalized RBF Networks. As it will become clear in the experiment that the number of cluster centers obtained from this model is less than the number of nodes (training patterns) as oppose to the equal numbers of cluster centers and pattern nodes in the regularization networks. As such, the network complexity is reduced.

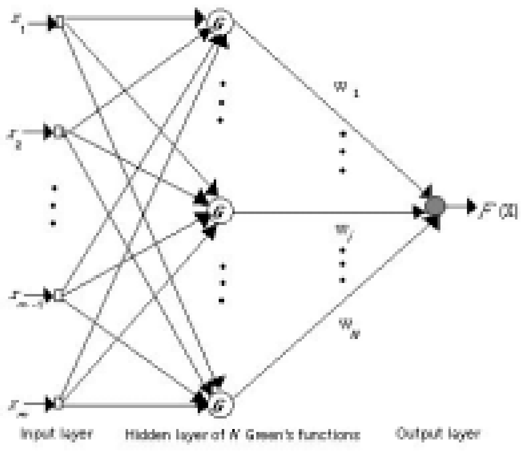

**Figure 4: Regularization Network**

## 6. Evaluation Criteria

Two models are built to assess our conjecture to classify software components as high-risk or low-risk in terms of fault-prone prediction. The first model is a multilayer perceptron and the second a radial-basis function. The following criteria [3] are applied for our experiment evaluation.

### 6.1 Misclassification rate

There are two misclassification errors, namely, type 1 and type 2 errors. A type 1 error occurs when a high-risk component is classified as low-risk, while a type 2 error occurs when a low-risk component is classified as high-risk. Misclassification is measured as a ratio of type 1 or type 2 propagation errors to the predicted total errors. Simply put,

Propagation of Type 1: $P_1 = \dfrac{n_{hl}}{n_{tot}}$

Propagation of Type 2: $P_2 = \dfrac{n_{lh}}{n_{tot}}$ (27)

Propagation of Type 1 + Type 2: $P_{12} = \dfrac{n_{hl} + n_{lh}}{n_{tot}}$

where $n_{hl}$ is the number of type 1 errors
$n_{lh}$ is the number of type 2 errors
$n_{tot}$ is the number of software components used in the classification

### 6.2 Quality achieved

If all the high-risk components are properly classified by the models, defects will be removed by the extra verification, and perfect quality will be achieved. Unfortunately, absolute prediction is in no way attained. Thus, the completeness measure will be used to determine the level of achieved quality completeness as follows:

$$C = \frac{n_{hh}}{n_{rh}} \tag{28}$$

where $n_{hh}$ is the number of faulty components that have been actually classified by the models
$n_{rh}$ is the number of faulty components

### 6.3 Verification cost

Two indicators are used to measure the verification cost, namely, inspection and wasted inspection. Inspection (*I*) measures the overall cost by considering the percentage of components that should be verified. Waste inspection (*WI*) is the percentage of components that do not contain faults but have been verified because they have been incorrectly classified. Both verification costs are defined as follows:

$$I = \frac{n_{ph}}{n_{tot}} \qquad WI = \frac{n_{lh}}{n_{ph}} \tag{29}$$

where $n_{ph}$ is the number of components that have been classified as high-risk class
$n_{lh}$ is the number of low-risk components that have been classified as high-risk class

$n_{tot}$ is the number of all components

## 7. The Experiment

The data obtained for this study were collected based on predetermined software metrics as described in Section 1 to characterize 118 software components. The data were then divided into training (88 software components) and test (30 software components) data sets. Each data point consisted of 11 variables for 11 software metrics. By applying the subtractive clustering, we obtained 66 cluster centers from 88 training patterns. We thus employed two models to classify the components into fault-prone class (one or more faults) and fault-free class (no fault), namely, the MLP and the RBF models.

### 7.1. The Multilayer Perceptron Network Model

The first model is constructed from multilayer perceptron network with back-propagation learning algorithm. The training data set is randomly divided to two groups, i.e., 59 data points for training group and 29 data points for validation group. The training group is used to adjust the weights, while the validation group is for error evaluation during training process. The criteria for automatic early stopping in training phase are the $PQ_\alpha$ class described in Section 4.

The MLP model consists of 11 input nodes in the input layer, 100 nodes in the hidden layer, and 1 output node in the output layer. Each hidden node and output node has one bias which is equal to 1.

The output expected from the model is zero ($y$ = 0) for the fault-free class (low-risk class) and one ($y$ = 1) for the fault-prone class (high-risk class). The learning rate ($\eta$ = 0.65) and the sigmoid function are used in weight adjustment. The criteria are reached at the 250th training iteration with $PQ$ = 0.0323.

When the training process is completed, the model is re-applied to classify the validation group. The output values so obtained range between 0 and 1 which are indecisive for component classification. By applying the acception ratio, if the calculated output is greater than or equal to 0.55, the component is a high-risk class. Otherwise, it is a low-risk class. The approach yields a 60% accurate prediction of the test data.

### 7.2. The Radial-Basis Function Network Model

In the work of [1], cluster centers are used to construct fuzzy rules with Sugeno-type. Assuming that the data point is in 2-dimension space ($x_1$, $x_2$) and $y$ is a dependent variable or output. Given a cluster center $\left(x_1^*, x_2^*, y^*\right)$, fault regression is predicted according to the following premises:

$$\begin{aligned}&\mathit{IF}\ x_1\ \mathit{IS\ CLOSE\ TO}\ x_1^*\\&\mathit{AND}\ x_2\ \mathit{IS\ CLOSE\ TO}\ x_2^*\\&\mathit{THEN}\ y = a_0 + a_1x_1 + a_2x_2\end{aligned}$$

The relation IS CLOSE TO is implemented as a Gaussian membership function and linear least squares estimation is applied to estimate the parameter $a_j$. This method groups the data points in to proper clusters in which they belong.

The output can be calculated using the equation

$$y(x) = \sum_{i=1}^{66} w_i G(\|x - t_i\|) \qquad (30)$$

From the above equation, it is apparent that Euclidean distance, along with proper weight, serves as the model's IS CLOSE TO relation. Since the calculated output value is a real number close to 0 or 1, it is rounded to the nearest integer (0 or 1). This simple adjustment yields an 83% prediction accuracy of the test data.

### 7.3. Model Evaluation

To evaluate the plausibility of the models, the evaluation criteria [3] are used to gauge their misclassification rate, quality achieved, and verification cost in percentage based on the relationships established earlier.

- Misclassification rate parameters is obtained from Equation 27.
  Model with multilayer perceptron:
  $P_1 = 0, \quad P_2 = 40.00, \quad P_{12} = 40.00$
  Model with radial-basis function network:
  $P_1 = 6.67, \quad P_2 = 10.00, \quad P_{12} = 16.67$
- Quality achieved is computed from Equation 28 for completeness ($C$).
  Model with multilayer perceptron:
  $C = 100$
  Model with radial-basis function network:
  $C = 88.24$
- Verification cost is based on the cost ratio in Equation 29.
  Model with multilayer perceptron:
  $I = 96.67, \quad WI = 41.38$
  Model with radial-basis function network:
  $I = 60.00, \quad WI = 16.67$

From the above results, misclassification rate of the fault-prone MLP model is higher than that of the fault-prone RBF model. The rate at which a low-risk component is misclassified as a high-risk component is higher in the MLP model, while there is no misclassified high-risk component as low-risk class in the RBF model. Although the completeness of the MLP model is higher than the RBF model, it expends more wasted cost for inspection. The more important aspect, perhaps, is that the RBF model can accurately predict fault-prone component up to 83% as oppose to 60% in MLP model. From this advantage, the fault-prone RBF model can be used as a basis to improve fault prediction in the fault-free RBF model.

## 8. Conclusion

Predicting software fault requires enormous amount of data. Analyzing these data is a major undertaking which must be carried out with care. We proposed a systematic approach to categorize closely related data using fuzzy subtractive method before the actual analysis process took place with the help of Radial-Basis Function technique. With proper fine-tuning, the number of faults for each component can be determined and applied to subsequent predictive quality. A notable characteristic of data that effects the number of clusters is fault scattering pattern. If the faults of the component under investigation do not correspond to the software metrics, fault prediction will not be accurate and meaningful.

We envisioned several approaches for reliability model improvement. First and foremost is metrics selection. It is difficult to find a proper mix of metrics for the model parameters. Second, determining meaningful features to extract fault pattern for subsequent analysis is not a straightforward formulation. Third, deriving proper membership function and algorithmic training procedure is an important step for efficient model construction. Accurate predictions obtained from such a good reliability model will be conducive toward higher software process efficiency and product quality.

The interpretation of fault component from the model entails the quality of component reuse in that fault-free components can be validated to comply with system specifications prior to implementation. The issues of how fault is interpreted, misclassificaton, and accuracy of fault prediction remain to be investigated in our future work.

## References

[1] Yuan, X., Khoshgoftaar, T.M., Allen, E.B., and Ganesant, K., *An application of fuzzy clustering to software quality prediction,* Proc. 2000 IEEE Symposium on Application-Specific Systems and Software Engineering Technology, 85-90, 2000.

[2] Khoshgoftaar, T.M., Pandya, A.S., and More, H.B., *A neural network approach for predicting software development faults,* Proc. Third International Symposium on Software Reliability Engineering, 83-89, 1992.

[3] Lanubile, F., *Evaluating predictive models derived from software measures,* J. Systems and Software, **38**(1), 225-234, 1996.

[4] Chiu, S., *Method and software for extracting fuzzy classification rules by subtractive clustering,* Fuzzy Information Proceeding Society, Biennial Conference of the North American, 461-465, 1996.

[5] Prechelt, L., *Automatic early stopping using cross validation: quantifying the criteria,* Neural Networks, **11**(1), 761-767, 1998.

[6] Klir, G.J., and Folger, T.A., *Fuzzy Sets, Uncertainty and Information,* Prentice-Hall, 355p., 1988.

[7] Haykin, S., *Neural Networks,* Prentice Hall, 156-312, 1999.

[8] Musa, J.D., Iannino, A., and Okumoto, K., *Software Reliability Measurement, Prediction, Application,* McGraw-Hill, 1987.

[9] Zadeh, L.A., *Knowledge representation in fuzzy logic,* IEEE Transactions on Knowledge and Data Engineering, **1**(1), 89-100, 1998.

[10] Eklund, P., Kallin, L., and Riissanen, T., *Fuzzy Systems,* Lecture notes prepared for courses at the Department of Computing Science at Umeå University, Sweden, February, 2000.

[11] Nikolaev, N.I., *http://homepages.goldsmiths.ac.uk/nikolaev/311rbf.html*, 2002.

[12] Kanellopoulos, I., *http://ams.egeo.sai.jrc.it/eurostat/Lot16-SUPCOM95/node7.html,* 1997.

[13] Center For High Integrity Software Systems Assurance, *http:// hissa.nist.gov/chissa/ SEI_Framework/framework_1.html*, 1995.

[14] Kan, S.H., *Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995.

[15] Chiu, S.L., *Fuzzy model identification based on cluster estimation*, Journal of Intelligent and Fuzzy Systems, **2**(3), 1994.