Metric-based Neural Network Classification Tool for Analyzing Large-Scale Software

Mr. Ray Paul, Assistant Technical Director, Software
U.S. Army Operational Evaluation Command

Abstract
Software developers use software metrics to understand and improve the software development process. In an effort to better manage software development, the U.S. Army is instituting a process to collect and interpret software metrics to disclose the information contained in the metrics. The neural network described to perform classification of software metrics is a three-layer, error back-propagation network. Using historical data, the neural network learns the relationship between certain metrics and a particular classification. The neural network will select the classification which best fits the input metrics. The capability of neural networks to classify non-linearly separable problem spaces gives neural networks an advantage over tree-based and linear network-based classification methods. When applied to actual software metrics taken from NASA software, the neural network correctly classified 100% of the data presented.

1: Introduction
Software metrics have been used to understand characteristics of software, manage software development, estimate software reliability, and in general to understand and improve the software development process. In 1990, a team of government and industry professionals met to form the Army's Software Test and Evaluation Panel (STEP). STEP had the broad charter to develop improved test and evaluation methods toward the goal of improving software in the Army. The panel's activities concerned both software that is embedded in materiel systems (e.g. the Patriot missile) and software that is used in traditional business systems (e.g. accounting system). One of the results of STEP was the identification of 12 metrics that offer early and detailed insights into the quality of software that is being designed for the Army. Hundreds of metrics have been identified and may be used at the option of the Program Manager (PM) of a major software development effort. A set of 12 core metrics were considered to constitute a broad measurement base of the products and processes in the software development lifecycle, and this set became mandatory in 1992 for all Army systems.

Understanding and using metrics data to guide the decision making process in a large software project can be a very difficult task. Therefore, it is not only important to provide tools that interpret metrics data in an informed manner, but it is also important to provide tools that can guide the process of developing new software by learning from historic data. Neural networks have been used in this manner for optimization, learning, pattern recognition and classification problems [1]. This report describes an implementation of a neural network-based classification tool, which will form an integral part of the metrics management tool being developed as a part of this project.

It is important to manage a high-quality software development process in such a way that those components that are likely to present problems are allocated the necessary resources. It is essential to identify such components early in the development process. For example, it may be important to identify those components that are likely to be "error-prone" so that sufficient resources can be allocated to them for testing. However, before development of the software itself, it is not possible to know all the metrics values for all components. The task of resource allocation and project management would be much easier with this prior knowledge. However, it may be possible to know certain metrics values for components in the early part of the process (e.g., design phase). The question one would like to answer is "using these values and historic data from other projects, can we identify those components that are likely to be error-prone?" Note that "error-prone" is just an example criterion. Someone else may be interested in identifying those components that require "high development effort". Hence, using the historic metrics data we would like to be able to answer these types of queries.

Answering these queries can be regarded as a metric-based classification problem. This report describes an implementation of a neural-network based approach for metric-based classification.
A black box model for such a classification problem is shown in Figure 1. The inputs to the model are the metrics and the associated empirical data, while the outputs are the target classes. Based on these target classes, the model uses several different software metrics to categorize past components as being within or outside a target class. The model is then applied to a project for which information about the desired property is unknown, and based on the past data, it identifies the component as being in the target class or not.

The application of such a classification technique using target classes is a three step process. Initially, a target class is defined which is simply a binary membership function on the objects or components being classified. Next, the model is "trained" to accurately classify a set of training objects, depending on whether or not they are in the target class. These training objects need to be representative of the population from which future objects will be selected, in order that those objects whose target classes membership is unknown and will be predicted by the model. Lastly, the model is applied to previously unseen objects and predicts whether or not they belong to the target class.

Based on this type of modeling approach using past metric data and target classes, we discuss a metric-based classification scheme, the Neural-Network Based Classification Model. This scheme has been successfully applied to data from a NASA production environment.

Other modeling techniques have been developed to perform the metric-based classification, including a Tree-based classification method [2, 3] and a linear network based classification method [4].

The tree-based method essentially generates a decision tree in which each internal node represents a partitioning function for a metric. Each link which emanates from an internal node represents a partition of the metric value of the node. The leaf nodes do not partition objects, they label the objects they encounter as members or non-members of the target class. Each path from the root node to a leaf node, therefore, represents a unique vector whose elements correspond to the partition values associated with each link in that path. Once a tree is constructed, it can classify new components by starting at the root node and following the path that corresponds to the component's metric values. For details refer to [2, 3]. The advantages of a neural network based classification model over a linear network model are described in Section 2.4.
model consists of an input layer, and output layer and a number of hidden layers. The number of hidden layers is dependent on the user. Typically, a single hidden layer can solve any classification problem.

To understand its functionality, let us consider the input to the model. The raw training-set data is recoded into relatively small groups of discrete values. The recoding process maps metric values into mutually exclusive, exhaustive ranges by transforming all data into nominal values. This grouping into discrete ranges can be done using non-parametric methods such as quartiles, distribution-sensitive least-weight-subsequence functions, or clustering methods [2,3]. This report borrows examples from [2] and [4] to explain the working of the neural network approach so that the differences between approaches can be more easily understood. The appropriate method is part of the calibration process that customizes the model building technique to different projects and environments.

Figure 3 shows the input form for the network model in Figure 2. Each of the software components used from the past metric data has information about several different metrics, such as number of lines, function calls, design effort, etc. Consider a single such metric, the number of lines. Different software components have different values for this metric. Rather that deal with a large number of these different values, the idea is to partition the entire set of possible values into discrete ranges and to quantify the value of the metric as falling into a certain range of values. For example, in Figure 3, we have the following partitions: 1 £ n < 20, 20 £ n < 50, 50 £ n < 100, and 100 £ n < 200, where n is the number-of-lines metric. A component which has 43 lines, would fall into the second range. This pre-processing is uniform for all metrics considered. The key difference is that the number of such partitions for a given metric can be specified by the user. Furthermore, the partitions are mutually exclusive. This provides the basis for generating orthogonal basis vectors for each partition. In this example, the orthogonal basis vectors for the number of lines metric are (1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0) and (0, 0, 0, 1). Since the particular component has 43 lines, it would fall into partition block 2 and the basis vector is (0, 1, 0, 0). These basis vectors which are formed for each of the metrics are the input to the network. The next stage in the network is a set of 'weights' which have to be learned using the past metric data. At the start, these weights are initialized to be 0. A weighted sum is then obtained and a bias is added, giving the output value. The bias is set to 0 initially. Note that we have chosen initial weights to be 0. However, any other random weight can be used as a starting weight.

2.1 Learning Phase

This phase involves training the network using past metric data to learn the set of weights. For each set of past metric data of a component, a 'target class' is specified that reflects the goals and circumstances of a particular project. These user-specifiable target classes define the properties that developers want to predict about the system and its components. Examples of these properties are components likely to have a high number of faults, high development effort, or faults in a certain class. The model building process then characterizes software artifacts from previous systems (components, subsystems, or processes) which meet the target class criteria.

The orthogonal basis vectors generated for a component are given as input to the network model and the output value is obtained. This output value is then compared with the target class specification. We have a positive instance if the output value is greater than 0.9, and if the component is in the target class, or if the output value is less than 0.1 and if the component is not in the target class. If not, we have a negative instance. If we have a negative instance, the set of weights, along with the bias, have to be corrected. In case of a positive instance, the weights remain the same. The predicted mapping is correct if the calculation made on the network's current set of link weights $W_{ij}$ classifies the component correctly. In this case the link weights are not modified. The predicted mapping is incorrect if the calculation based on the network's current set of link weights $W_{ij}$ classifies the component incorrectly. In this case the link weights are modified. In general, a link weight contributes to a misclassification if it maps a 1 in the input pattern to an incorrect 1 (in the target class) or 0 (not in the target class) in the output pattern. This weight change is in increments of +6 or -6, corresponding respectively to whether the network incorrectly classified a component as 0 (the component is predicted as not in the target class but actually is), or 1 (the component is predicted as in the target class but actually is.
The bias is considered as a weight whose input is always 1.

This process is repeated for each iteration through the training data. The training ends when the network can correctly classify 100% of the components in the training set or no further improvement is achieved.

2.2 Weight Adjustment

As mentioned earlier, we have a set of components along with their target classes. Our aim is to find the set of weights that give a 100% successful classification of the training set.

The net input to a layer \( j \) is given by:

\[
net_j = \sum w_{ji} o_i
\]

The output of node \( j \) is given by:

\[
o_j = f (net_j)
\]

where \( f \) is the activation function. Typically, a sigmoidal activation function is used [1]:

\[
o_j = \frac{1}{1 + e^{- \left( net_j + \theta_j \right) / \theta}}
\]

where the parameter \( \theta \) serves as a threshold or bias. The effect of \( \theta \) is to modify the shape of the sigmoid. A low value of \( \theta \) tends to make the sigmoid take on the characteristics of a threshold logic unit, and a high value of \( \theta \) results in a more gently varying function.

We present the input pattern and adjust the set of all connecting links and also the thresholds in the nodes such that the desired outputs are obtained in the output nodes. This is repeated for all the input patterns until the net has a single set of weights and biases that satisfy all input training patterns.

For each pattern, in general, the outputs \( o_k \) will not be the same as the target values \( t_k \). The square of the error for each pattern is

\[
E_p = \frac{1}{2} \sum_k (t_k - o_k)^2
\]

and the average system error is

\[
E = \frac{1}{2P} \sum_P \sum_k (t_k - o_k)^2
\]

Using this, the incremental weight changes \( \Delta w_{kj} \) are made proportional to \( -\partial E / \partial w_{kj} \):

\[
\Delta w_{kj} = \eta \frac{\partial E}{\partial w_{kj}}
\]

However, since \( E \), the error, is expressed in terms of the outputs \( o_k \), each of which is the nonlinear output of the node \( k \), i.e.,

\[
o_k = f (net_k)
\]

where \( net_k \) is the input to the \( k \)th node and by definition is the weighted linear sum of all the outputs from the previous layer:

\[
net_k = \sum w_{kj} o_j
\]

The partial derivative \( \partial E / \partial w_{kj} \) can be evaluated using the chain rule

\[
\frac{\partial E}{\partial w_{kj}} = \left( \frac{\partial E}{\partial net_k} \right) \left( \frac{\partial net_k}{\partial w_{kj}} \right)
\]

Since

\[
\frac{\partial net_k}{\partial w_{kj}} = o_j
\]

we now define

\[
\delta_k = - \frac{\partial E}{\partial net_k}
\]

and write

\[
\Delta w_{kj} = \eta \delta_k o_j
\]

for any output layer node \( k \), and

\[
\delta_k = o_k (1 - o_k) \sum \delta_k w_{kj}
\]

for any hidden layer unit.
The thresholds are learned in the same manner as are other weights. They are regarded as a weight from a unit that always has an output value of unity.

As might be expected from such steepest-descent methods, rapid learning might result in oscillations. Hence, the weight change is modified to include a sort of momentum term:

\[ \Delta w_{ji}(n+1) = \eta \delta_j o_i + \alpha \Delta w_{ji}(n) \]  

(14)

where the quantity \((n+1)\) is used to indicate the \((n+1)\)th step, and \(\alpha\) is a proportionality constant.

2.3 Output Phase

Once the learning phase is complete, we have a set of weights for the network. The basis vectors are then generated for a new component and its target class predicted using the learned weights.

Section 3 illustrates the recoding for a hypothetical example. This recorded data is given as input to train the model to learn the link weights. Appendix A gives the output from program execution for the classification model.

2.4 Advantages

The neural network-based classification model satisfies the requirements of the classification problem. The main advantage of using a neural-network based model as shown in Figure 2 is that the pattern classes do not have to be linearly separable. Linearly separable means that there is a straight line that separates the two pattern classes as being in the target class and not in the target class. Figure 4 shows patterns that are linearly separable while Figure 5 shows patterns that are not linearly separable.

Figure 4: Linearly-Separable Pattern Classes

Figure 5: Pattern Classes not Linearly-Separable

In case of a linear network, 100% successful classification for the training set is not guaranteed. The problem is solved by a non-linear model, such as the neural network-based classification model. It is quite complex in nature as it is non-linear and since the number of links is large, the set of weights to be learned are high.

3. Recoding for a Hypothetical Example

This section presents an example of a hypothetical project where the developers have experienced a series of interface faults in their correct software release. The example shows how the network-based classification is used to localize the components most likely to have interface faults in the next release. The example is shown to illustrate the working of the model in detail using a very small data set.

The metrics used in the example are component function, data bindings, design revision. The target class is based on the number of interface errors. Components with greater than 5 interface faults constitute the target class. In Table 1, components in the target class are shown as '+' while those not in the target class are denoted as '-'. These components serve as the training set for building the network-based model. To recode the raw metric data, the partitions used are shown in Table 2. The recoded metric data for the components is as shown in Table 3.

4. Summary

The neural network based classification technique has been applied to the NASA data. For this set of data, 100% correct classification was obtained.

The way this network is used for identifying components of some other project is as follows. It is assumed that the metric data which represents the input nodes of the network...
Table 1: Raw Metric Data for Training-Set Components [2,4]

<table>
<thead>
<tr>
<th>Component Function</th>
<th>Data Bindings</th>
<th>Design Revisions</th>
<th>Interface Errors</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>A User Interface</td>
<td>2</td>
<td>11</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>B User Interface</td>
<td>9</td>
<td>9</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>C File Management</td>
<td>6</td>
<td>11</td>
<td>10</td>
<td>+</td>
</tr>
<tr>
<td>D User Interface</td>
<td>13</td>
<td>0</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>E File Management</td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>F User Interface</td>
<td>15</td>
<td>4</td>
<td>9</td>
<td>+</td>
</tr>
<tr>
<td>G Process Control</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>H Process Control</td>
<td>15</td>
<td>10</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>I Process Control</td>
<td>20</td>
<td>5</td>
<td>6</td>
<td>+</td>
</tr>
<tr>
<td>J User Interface</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>K File Management</td>
<td>17</td>
<td>1</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>L File Management</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2: Partitions for Metrics of Training-Set Components [2,4]

<table>
<thead>
<tr>
<th>Component Function</th>
<th>Data Bindings</th>
<th>Design Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Management</td>
<td>0 ≤ x ≤ 7</td>
<td>0 ≤ x ≤ 3</td>
</tr>
<tr>
<td>User Interface</td>
<td>8 ≤ x ≤ 14</td>
<td>4 ≤ x ≤ 8</td>
</tr>
<tr>
<td>Process Control</td>
<td>x ≥ 15</td>
<td>x ≥ 9</td>
</tr>
</tbody>
</table>

Table 3: Recorded Metric Data for Training-Set Components [2,4]

<table>
<thead>
<tr>
<th>Component</th>
<th>Data Bindings</th>
<th>Design Revisions</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>010</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>010</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>100</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>010</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>100</td>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>010</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>001</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>001</td>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>001</td>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>J</td>
<td>010</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>K</td>
<td>100</td>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>100</td>
<td>001</td>
<td>0</td>
</tr>
</tbody>
</table>

is also available for the new project. For each metric, depending on which range, (in the above case which quartile) it falls into, the corresponding pattern is assigned to it. All the input patterns are then presented to the network. The network, using the method described in Section 2, computes whether the component under question belongs to the target class or not. Following this method, a set of components belonging to a project can be classified into two subsets; namely, one representing high development effort and the other one representing low development effort.

It should be noted that it is essential to have the input metric data for the new components to be classified. Suppose only a subset of metrics is available. In such a case, the network can be regenerated from only that subset from the historical data, and then be used to classify the new set of components.

A typical interactive session for network generation and learning phase is illustrated in the Appendix. We have shown only the hypothetical example in order not to overwhelm the reader with the tremendous amount generated when a large project is used to generate the network. Currently, the Neural Network tool is not integrated with the Xmetric interface. However, in the next phase we plan to integrate the Network tool with Xmetric interface.

References