Neural Network Approach for Software Defect Prediction Based on Quantitative and Qualitative Factors

Parvinder S. Sandhu, Suman Lata, and Dalveer Kaur Grewal

***Abstract*—These Quality of a software component can be expressed in terms of level of number of faults present in data. Quality estimations are made using fault data available from previously developed similar type of projects and the training data consisting of software measurements. In this paper, an attempt is made to use Batch Gradient Descent (BGD), Batch Gradient Descent with momentum (BGDWM), Variable Learning Rate (VLR), Variable Learning Rate training with momentum (VLRM) and Resilient Backpropagation (RB) based neural network approach to identify the relation between the various qualitative as well as quantitative factor of the modules with the number of faults present in the module that will be helpful for prediction of the level of number of faults present in the modules. The dataset used is elicited from 31 completed software projects in the consumer electronics industry. The data were gathered using a questionnaire distributed to managers of recent projects. The performance of the algorithms is recorded in terms of MAE, RMSE and Accuracy percentage values.**

***Index Terms*—Neural network, quantitive, qualittative, software fault, defect data, and software quality**

I. Introduction

A software fault is a defect that causes software failure in an executable product. In software engineering, the non-conformance of software to its requirements is commonly called a bug. Software Engineers distinguish between software faults, software failures and software bugs. In case of a failure, the software does not do what the user expects but on the other hand fault is a hidden programming error that may or may not actually manifest as a failure and the non-conformance of software to its requirements is commonly called a bug .

A software quality model is a useful tool for meeting the objectives of software reliability and software testing initiatives of different projects. Metrics available in the early lifecycle data can be used to verify the need for increased quality monitoring during the development. Different modeling techniques can be used to identify fault prone modules[1]-[11].

In this study, we investigate whether qualitative and quantitative factors can be used to identify level of number of faulty software modules using different neural network approaches.

Manuscript received February 22, 2012; revised March 28, 2012.

Parvinder S. Sandhu is with the Deptt. Of CSE and IT Rayat and Bahra Institute of Engg. and Bio-Technology, Mohali, India.

Suman Lata and Dalveer Kaur Grewal are with the Deptt. Of CSE/IT Lovely Professional University, Jalandhar, India.

II. Methodology

The methodology consists of the following steps:

A. Find the Quantitivative and Qualittiative attributes

The first step is to find the structural code and requirement attributes of software systems i.e. software metrics. The real-time defect data sets are taken from http://promisedata.org/repository. The Qualitative and quantitative dataset is about 31 projects completed in a consumer electronics company (one row per project). There is a mixture of qualitative attributes (these are measured on a 5 point ranked scale VL, L, M, H, VH) and quantitative attributes whose scale is stated [12].

Qualitative factors

The Quantitative factors are grouped under five topics [12]:

- Specification and Documentation process
- New Functionality
- Design and Development process
- Testing and Rework
- Project Management

Each factor is named and described by a question to be answered. The descriptive questions were specifically tailored for the organization providing the project data.

The following are the *Specification and documentation process* attributes [12]:

1) Relevant Experience of Spec and Doc Staff: How would you rate the experience and skill set of your team members for executing this project during the requirements and specifications phase?
2) Quality of Documentation inspected: How would you rate the quality of the requirements given by the client or other groups?
3) Regularity of Spec and Doc Reviews: Have all the Requirements, Design Documents and Test Specifications been reviewed in the project?
4) Standard Procedures Followed: In your opinion, how effective was the review procedure?
5) Quality of Documentation inspected: What was the review effectiveness in the project for the requirements phase?
6) Spec Defects Discovered in Review: In your opinion, is the defect density of spec reviews on the high side?
7) Requirements Stability: How stable were the requirements in your project?

The following are the details of the *new functionality* attributes [12]:

1) Complexity of new functionality: What was the complexity of the new development or new features that happened in your project?
2) Scale of New functionality implemented: How large was the extent of working on new functionality rather

than just enhancing the older functionalities in your project?

3) Total no. of Inputs and Outputs: For your product domain, would you rate the total no of outputs/inputs (newly developed / enhanced) as high?

The following are the Design and development process attributes [12]:

1) Relevant Development Staff Experience: How would you rate the experience and skill set of your team members for executing this project during the design and development phase?
2) Programmer capability: On an average, how would you assess the Quality of code produced by the team members?
3) Defined processes followed: What was the review effectiveness in the project for the Design and Development phase?
4) Development Staff motivation: What is your opinion about the motivation levels of your team members?

The following are the *Testing and Rework* attributes [12]:

5) Testing Process Well Defined: How effective was the testing process adopted by your project?
6) Staff Experience –Unit Test: What was the level of software test competence of those performing the unit test?
7) Staff Experience –Independent Test: How would you rate the experience and skill set of the independent test engineers (Integration, functional or subsystem testing, Alpha, Beta)?
8) Quality of Documented Test Cases: What was the extent of the defects that were found using formal testing against the intuitive/random testing?

The following are the *Project Management* attributes [12] :

1) Dev. Staff Training Quality: What is the coverage of the identified project / process related trainings as well as trainings identified as per the roles, by the team members?
2) Configuration Management: How effective is the project's document management and configuration management?
3) Project Planning: Has the project planning been done adequately?
4) Scale of Distributed Communication: How many sites/groups were involved in the project?
5) Stakeholder involvement: To what extent were the key project stakeholders involved?
6) Customer involvement: How good was customer interaction in the project?
7) Vendor Management: How would you rate the Vendor /Sub-contractor Management (if applicable)?
8) Internal communication/ interaction: How would you the rate the quality of internal interactions / communication within the team?
9) Process Maturity: What's your opinion about process maturity in the project?

Qualitative data are expressed on a 5-point ordinal scale. The ordinal values used are: Very High, High, Medium, Low, Very Low. The data values were gathered using a questionnaire, which was completed by the project manager, project quality manger or other senior project staff. The number of faults present in the modules is also expressed on a 5-point ordinal scale: Very High, High, Medium, Low and Very Low.

Quantitative Factors

The following are the Quantitative factors are [12]:

- Software size: the size, in KLoC of the developed code and the development language
- Effort: development effort measured in person hours for the software development, from specification review to unit test

B. Analyze, Refine Metrics and Normalize the Metric Values

In the next step the metrics are analyzed, refined and normalized and then used for modeling of fault prediction in software systems.

C. Explore Different Neural Network Techniques

It is very important to find the suitable algorithm for modeling of software components into different levels of fault severity in software systems. The following five Neural Network algorithms are experimented:

- Batch Gradient Descent
- Batch Gradient Descent with momentum
- Variable Learning Rate
- Variable Learning Rate training with momentum
- Resilient Backpropagation

1) Backpropagation Algorithm:

There are many variations of the backpropagation algorithm, several of which are described in the literature. The simplest implementation of backpropagation learning updates the network weights and biases in the direction in which the performance function decreases most rapidly, the negative of the gradient. One iteration of this algorithm can be written as:

$$x_{k+1} = x_k - \alpha_k g_k \tag{1}$$

where, x_k is a vector of current weights and biases, g_k is the current gradient, and α_k is the learning rate.

There are two different ways in which this gradient descent algorithm can be implemented: incremental mode and batch mode. In incremental mode, the gradient is computed and the weights are updated after each input is applied to the network. In batch mode, all the inputs are applied to the network before the weights are updated.

In batch mode the weights and biases of the network are updated only after the entire training set has been applied to the network. The gradients calculated at each training example are added together to determine the change in the weights and biases.

In batch steepest descent algorithm weights and biases are updated in the direction of the negative gradient of the performance function.

Hence, Batch Gradient Descent without momentum Training Algorithm can train any network as long as its weight, net input, and transfer functions have derivative functions. Backpropagation is used to calculate derivatives of performance PERF with respect to the weight and bias variables X. Each variable is adjusted according to gradient descent:

$$\partial X = l_r \times \frac{\partial PERF}{\partial X} \tag{2}$$

where l_r is the learning rate .

Training stops when any of these conditions occurs:

1) The maximum number of Epochs (repetitions) is

reached.

2) The maximum amount of Time to train has been exceeded.

3) Performance has been minimized to the Performance Goal.

4) The performance gradient falls below Minimum Performance Gradient value.

5) Validation performance has increased more than Maximum number of validation Failures value

Gradient descent with momentum backpropagation is a network training function that updates weight and bias values according to gradient descent with momentum. It can train any network as long as its weight, net input, and transfer functions have derivative functions.

Backpropagation is used to calculate derivatives of performance PERF with respect to the weight and bias variables X. Each variable is adjusted according to gradient descent with momentum:

$$\partial X = m_c \times \partial X_{PERF} + l_r \times (1 - m_c) \times \frac{\partial PERF}{\partial X} \quad (3)$$

where ∂X_{PERF} he previous change to the weight or bias, l_r is the learning rate and m_c is the momentum constant. Training stops when any of these conditions mentioned in Batch Gradient Descent without momentum Training Algorithm occurs.

2) Variable Learning Rate:

With standard steepest descent, the learning rate is held constant throughout training. The performance of the algorithm is very sensitive to the proper setting of the learning rate. If the learning rate is set too high, the algorithm can oscillate and become unstable. If the learning rate is too small, the algorithm takes too long to converge. It is not practical to determine the optimal setting for the learning rate before training, and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface.

Variable Learning Rate without momentum is a network training function that updates weight and bias values according to gradient descent with adaptive learning rate. Here, Backpropagation is used to calculate derivatives of performance DPERF with respect to the weight and bias variables X. Each variable is adjusted according to gradient descent:

$$\partial X = l_r \times \frac{\partial PERF}{\partial X} \quad (4)$$

Each of epoch, if performance decreases toward the goal, then the learning rate is increased by the factor lr_inc. If performance increases by more than the factor max_perf_inc, the learning rate is adjusted by the factor lr_dec and the change, which increased the performance, is not made.

Training stops when any of these conditions mentioned in Batch Gradient Descent without momentum Training Algorithm occurs.

Gradient descent w/momentum and adaptive lr backpropagation or Variable Learning Rate Training with momentum is a network training function that updates weight and bias values according to gradient descent momentum and an adaptive learning rate.

In this algorithm Backpropagation is used to calculate derivatives of performance PERF with respect to the weight and bias variables X. Each variable is adjusted according to the gradient descent with momentum:

$$\partial X = m_c \times \partial X_{PERF} + l_r \times m_c \times \frac{\partial PERF}{\partial X} \quad (5)$$

where ∂X_{PERF} he previous change to the weight or bias, l_r is the learning rate and m_c is the momentum constant.

For each epoch, if performance decreases toward the goal, then the learning rate is increased by the factor lr_inc. If performance increases by more than the factor max_perf_inc, the learning rate is adjusted by the factor lr_dec and the change, which increased the performance, is not made.

Training stops when any of these conditions mentioned in Batch Gradient Descent without momentum Training Algorithm occurs.

3) Resilient Backpropagation:

Multilayer networks typically use sigmoid transfer functions in the hidden layers. These functions are often called "squashing" functions, because they compress an infinite input range into a finite output range. Sigmoid functions are characterized by the fact that their slopes must approach zero as the input gets large. This causes a problem when you use steepest descent to train a multilayer network with sigmoid functions, because the gradient can have a very small magnitude and, therefore, cause small changes in the weights and biases, even though the weights and biases are far from their optimal values.

The purpose of the resilient backpropagation training algorithm is to eliminate these harmful effects of the magnitudes of the partial derivatives. Only the sign of the derivative is used to determine the direction of the weight update; the magnitude of the derivative has no effect on the weight update. The size of the weight change is determined by a separate update value. The update value for each weight and bias is increased by some factor whenever the derivative of the performance function with respect to that weight has the same sign for two successive iterations. Resilient Backpropagation can train any network as long as its weight, net input, and transfer functions have derivative functions. In this algorithm Backpropagation is used to calculate derivatives of performance PERF with respect to the weight and bias variables X. Each variable is adjusted according to the following equation:

$$\partial X = \Delta X \times sign(g_X) \quad (6)$$

where the elements of ΔX are all initialized to 0 and g_X is the gradient. At each iteration the elements of ΔX are modified. If an element of g_X changes sign from one iteration to the next, then the corresponding element of deltaX is decreased by delta_dec. If an element of gX maintains the same sign from one iteration to the next, then the corresponding element of deltaX is increased by delta_inc [13].

In the implementation first the network is created and training is performed on the training data. Thereafter the trained network is tested by testing data in the testing phase. The results of the different algorithms are expressed in terms of *MAE*, *RMSE* and *Accuracy* values. The details of the different criteria used are in next step. The following steps will be followed to train a Neural Network:

- Load the data
- Divide data into Training, Validation and Test data
- Set number of hidden neurons
- Training is accomplished by sending a given set of inputs through the network and comparing the results with a set of target outputs.
- If there is a difference between the actual and target outputs, the weights are adjusted to produce a set of outputs closer to the target values.
- Network weights are determined by adding an error correction value to the old weight.
- The amount of correction is determined
- This Training procedure is repeated until the network performance no longer improves.
- If the network is successfully trained, it can then be given new sets of input and generally produce correct results on its own

D. Comparison of Algorithms

The comparisons are made on the basis of the more accuracy and least value of *MAE and RMSE* error values. *Accuracy* value of the prediction model is the major criteria used for comparison. The mean absolute error is chosen as the standard error. The technique having lower value of mean absolute error is chosen as the best fault prediction technique.

1) Mean Absolute Error

Mean absolute error, MAE is the average of the difference between predicted and actual value in all test cases; it is the average prediction error [13]. The formula for calculating MAE is given in equation 7.

$$\frac{|a_1-c_1|+|a_2-c_2|+\ldots+|a_n-c_n|}{n} \quad (7)$$

Assuming that the actual output is *a*, expected output is *c*.

2) Root Mean-Squared Error

RMSE is frequently used measure of differences between values predicted by a model or estimator and the values actually observed from the thing being modeled or estimated [13]. It is just the square root of the mean square error as shown in equation 8.

$$\sqrt{\frac{(a_1-c_1)^2+(a_2-c_2)^2+\ldots+(a_n-c_n)^2}{n}} \quad (8)$$

The mean-squared error is one of the most commonly used measures of success for numeric prediction. This value is computed by taking the average of the squared differences between each computed value and its corresponding correct value. The root mean-squared error is simply the square root of the mean-squared-error. The root mean-squared error gives the error value the same dimensionality as the actual and predicted values.

The mean absolute error and root mean squared error is calculated for each machine learning algorithm i.e. Neural Network.

III. Results And Discussions

The proposed Neural based methodology is implemented in MATLAB 7.4. MATLAB (Matrix Laboratory) environment is one such facility which lends a high performance language for technical computing. The Batch Gradient Descent (BGD), Batch Gradient Descent with momentum (BGDWM), Variable Learning Rate (VLR), Variable Learning Rate training with momentum (VLRM) and Resilient Backpropagation (RB) algorithms are experimented for training a neural network separately. The above said algorithms are applied on the dataset and the performance of the above algorithms is shown as Error v/s Epoch graph in the figures 1 to 5.

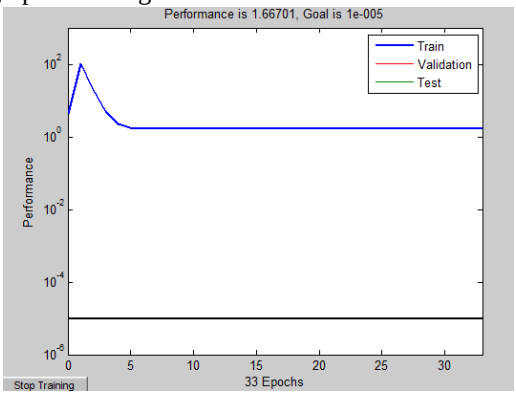

Fig. 1. Performance of the batch gradient descent algorithm for fault dataset

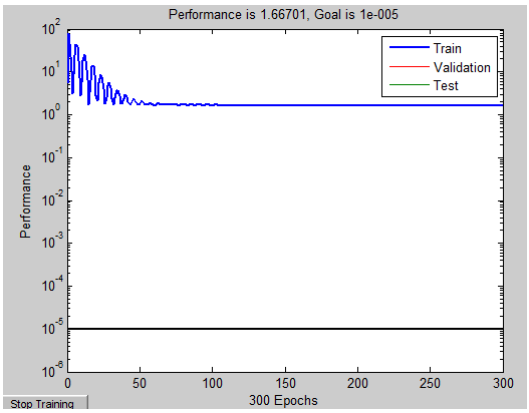

Fig. 2. Performance of the batch gradient descent with momentum algorithm

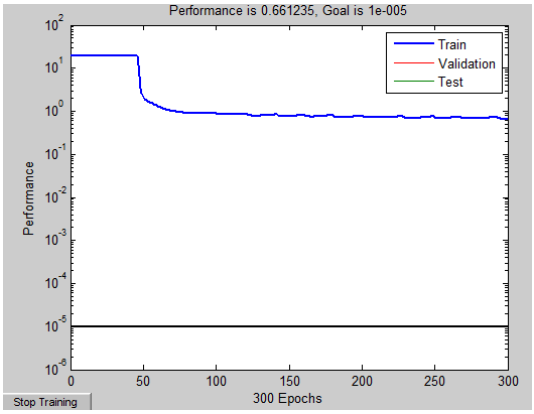

Fig. 3. Performance of the variable learning rate algorithm

During the training of Neural Network with Batch Gradient Descent (BGD) algorithm the Mean Square Error (MSE) value stabilizes at 1.66701 after 5th epoch of training and the training stops after 35 epochs as shown in figure 1. In case of Batch Gradient Descent with momentum (BGDWM) algorithm the MSE value stabiles to 1.66701 after 70th epoch as shown in figure 2. As observed from

figures 3 and 4 showing training with Variable Learning Rate (VLR) and Variable Learning Rate with momentum (VLRM) algorithms the MSE value stabiles after 250th epoch at 0.661235 and 0.525382 values respectively. The Resilient Backpropagation (RB) algorithm produces the MSE value 1.66701, as shown in figure 5, which is equal to the MSE value evidenced in the BGD and BGDWM algorithms.

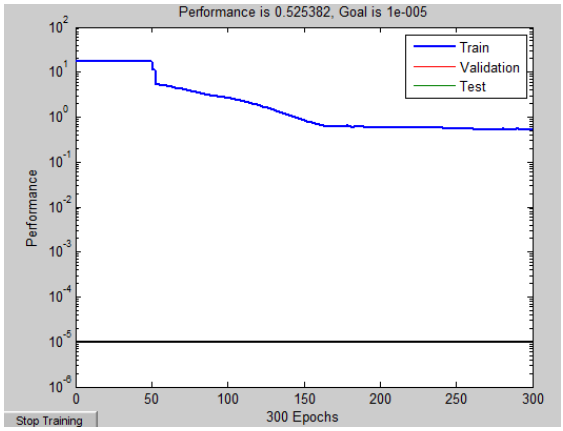

Fig. 4. Performance of the variable learning rate with momentum algorithm for fault dataset

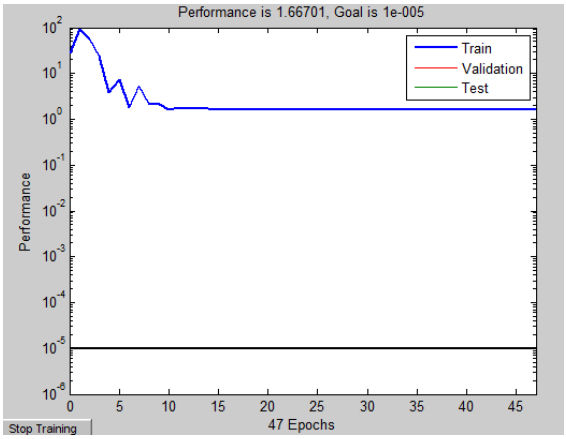

Fig. 5. Performance of the resilient backpropagation algorithm for fault dataset

TABLE I: RESULTS OF DIFFERENT NEURAL NETWORK BASED ALGORITHMS FOR PREDICTION OF FAULTS

Sr. No.	Algorithm	*MAE*	*RMSE*	*Accuracy %*
1	Batch Gradient Descent without momentum	1.1405	1.2911	41.9355
2	Batch Gradient Descent with momentum	1.1405	1.2911	41.9355
3	Variable Learning Rate without momentum	0.6541	0.8132	54.8387
4	Variable Learning Rate training with momentum	0.5445	0.7248	77.4194
5	Resilient Backpropagation	1.1405	1.2911	41.9355

The MAE, RMSE and Accuracy % of the five algorithms experimented is shown in the Table I. The Batch Gradient Descent without momentum algorithm, Batch Gradient Descent with momentum and Resilient Backpropagation algorithms shows 1.1405, 1.2911 and 41.9355 as MAE, RMSE and Accuracy% values. Whereas Variable Learning Rate without momentum algorithm have produced 0.6541, 0.8132 and 54.8387 as MAE, RMSE and Accuracy% values. In case of Variable Learning Rate training with momentum algorithm MAE, RMSE and Accuracy% values calculated are 0.5445, 0.7248 and 77.4194 respectively.

IV. CONCLUSION

Prediction of Level of faults in modules supports software quality engineering through improved scheduling and project control. It is a key step towards steering the software testing and improving the effectiveness of the whole process. Fault prediction is used to improve software process control and achieve high software reliability.

In this study, we investigate whether qualitative and quantitative factors can be used to identify level of number of faulty software modules. We compare the performance of Batch Gradient Descent (BGD), Batch Gradient Descent with momentum (BGDWM), Variable Learning Rate (VLR), Variable Learning Rate training with momentum (VLRM) and Resilient Backpropagation (RB) based Neural Network for the fault dataset. Variable Learning Rate training with momentum algorithm shows best results among the five algorithms experimented with least values of MAE and RMSE calculated as 0.5445 and 0.7248 respectively. The Accuracy% of prediction of the level of number of faults for Variable Learning Rate training with momentum algorithm is also highest i.e. 77.4194.

The performance of Batch Gradient Descent without momentum algorithm, Batch Gradient Descent with momentum and Resilient Backpropagation algorithms comes out to be the same for the fault dataset used and the results of the Variable Learning Rate without momentum are second best but much below than the best one.

It is therefore, concluded the Variable Learning Rate with momentum based neural network model is implemented and the best algorithm for classification of the software components into different level of number of faults present in the modules of the software systems.

The future work can be extended in following directions:

- Most important attribute can be found for fault prediction and this work can be extended to further programming languages. More algorithms can be evaluated and then we can find the best algorithm.
- Further investigation can be done and the impact of attributes on the fault prediction can be found.
- Other dimensions of quality of software can be considered for mapping the relation of attributes and fault tolerance.

REFERENCES

[1] Y. Jiang, B. Cukic, and T. Menzies, "Fault Prediction Using Early Lifecycle Data," ISSRE 2007, the 18th IEEE Symposium on Software Reliability Engineering, *IEEE Computer Society*, Sweden, 2007, pp. 237-246.

[2] N. Seliya, T. M. Khoshgoftaar, and S. Zhong, "Analyzing software quality with limited fault-proneness defect data," in *proceedings of the Ninth IEEE international Symposium on High Asssurance System Engineering*, Germany, 2005, pp. 89-98.

[3] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on Software Engineering*, vol. 18, issue: 5, pp. 423-433, 1992.

[4] P. Bellini, "Comparing Fault-Proneness Estimation Models," 10th IEEE International Conference on Engineering of Complex *Computer Systems (ICECCS'05)*, China, 2005, pp. 205-214.

[5] F. Lanubile, A. Lonigro, and G. Visaggio, "Comparing Models for Identifying Fault-Prone Software Components," in *Proceedings of Seventh International Conference on Software Engineering and Knowledge Engineering*, USA, 1995, pp. 12-19.

[6] N. E. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE Transactions on Software Engineering*, vol. 25, issue: 5, pp. 675-689, 1999.

[7] Runeson, C. Wohlin, and M. C. Ohlsson, "A Proposal for Comparison of Models for Identification of Fault-Proneness," *Journal of System and Software*, vol. 56, issue: 3, pp. 301–320, 2001

[8] Runeson, C. Wohlin, and M. C. Ohlsson, "A Proposal for Comparison of Models for Identification of Fault-Proneness," *Journal of System and Software*, vol. 56, issue: 3, pp. 301–320, 2001.

[9] V. U. B. Challagulla, F. B. Bastani, I. L. Yen, and Pau1, "Empirical assessment of machine learning based software defect prediction techniques," *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, USA, pp. 263-270, 2005.

[10] S. Basu, A. Banerjee, and R. Moorey, "Semi-Supervised Clusering by Seeding," in *Proceedings of the 19th International Conference on Machine Learning*, Sydney, 2002, pp. 19-26

[11] C. E. Brodely and M. A. Friedl, "Identifying mislabeled training Data." *Journal of Artificial Intelligence Research*, vol. 11, pp.131-167, 1999.

[12] Norman Fenton, Martin Neil, William Marsh, Peter Hearty, Lukasz Radlinski, and Paul Krause, "Project Data Incorporating Qualitative Factors for Improved Software Defect," in *Proceedings of the PROMISE workshop*, 2007.

[13] Mathworks [Online]. Available: www.mathworks.com/help