How to Predict More with Less

Defect Prediction Using Machine Learners in an Implicitly Data Starved Domain

Kim Kaminsky
Department of Computer Science, University of Houston - Clear Lake
2700 Bay Area Boulevard
Houston, TX 77058 USA
kaminsky@cl.uh.edu

and

Gary D. Boetticher
Department of Computer Science, Department of Software Engineering, University of Houston - Clear Lake
2700 Bay Area Boulevard
Houston, TX 77058 USA
boetticher@cl.uh.edu

ABSTRACT
Many software organizations do not allocate enough resources for software quality. Therefore, the automated assessment of software, in terms of defect prediction, is an important issue to the software engineering community. Researchers seek to build accurate and reliable predictors using legacy data. However, it is a difficult task when one realizes that this is an "implicitly data starved" domain due to the skewed distribution of the data. The problem is heightened because the interesting instances (modules with moderate to high defect rates) represent a significant portion of the "starved data." One potential solution is to balance the data, referred to as equalized learning, in order to compensate for the skewness. This is accomplished by replicating the starved data in order to emphasize items of interest.

To assess the feasibility of this technique, a series of Genetic Programming experiments are conducted using a NASA-based repository of defect data.

Keywords
Equalized Learning, Genetic Programs, Machine Learning, Defect Prediction, NASA

1. INTRODUCTION

Defect prediction is an important research area in the Software Engineering community. It enables software practitioners to detect, track and resolve product anomalies that might affect human safety and lives. Additionally, defect prediction allows changes to be made earlier in the life-cycle process, thus lowering software costs and improving customer satisfaction.

Considering these benefits, many organizations tend to under-appropriate resources to their software quality group. Many theories could be postulated regarding the reasons for this situation. However, a core question is, *How to do more with less?* More specifically, *How to find more software defects using less effort?*

Answering this last question begins with an observation of process trends. As software organizations mature in their corresponding process, there are new initiatives to collect and analyze various forms of process, project, and product data. The intent is to perform empirical software engineering so that business and technical decisions are data driven.

One of the major obstacles in successfully achieving this initiative is the severe lack of data. Operating in a data starved domain makes it difficult to formulate reliable and accurate models.

Data starved domains assume two forms, they may be explicit or implicit. An explicit data starved domain contains relatively few instances of data. This occurs frequently in software engineering, particularly project estimation. This is understandable in the context of competitive project estimation. If a corporation discovers a method to improve their accuracy in project estimation, they will be extremely reluctant to share their proprietary information and lose their competitive advantage. Also, software project development takes such a long time to complete. Thus, the number of project completion instances is sparse within an organization.

Implicit data starved domains typically possess a sufficient amount of data. The problem is the lack of instance diversity where certain attributes exhibit highly skewed distributions. In the case of software defects there might be a 100 to 1 ratio of instances to unique values. Furthermore, in the case of "interesting values" (e.g.

defect rates of 2 or higher), this ratio could be higher. It is not unusual for 50 percent of the data to be concentrated within 5 percent of the distribution curve.

These obstacles force researchers to consider innovative ways of solving this problem. One approach, which has received a lot of attention, is the application of machine learners.

Machine learners, such as genetic programs (GPs), artificial neural networks (ANNs), case-based reasoners (CBRs), and rule inductors (RIs), are ideally suited to this type of problem because they handle noisy and incomplete data very well. At the core of these learners is induction, which is a process of inferring a generalization based upon a sample set. Thus defect prediction utilizing software metrics fits this paradigm.

Among the possible learners available, Genetic Programs offer the advantage of providing human-readable solutions in the form of polynomial equations. Therefore, this paper focuses upon the Genetic Program machine learner.

Ideally, a Genetic Program works best if all possible solutions to a problem are equally represented in the training data set. This paper explores the possibility of populating a GP training data set with replicated instances of higher value defects. Thus, if twenty percent of the data represents software modules with zero defects, but only two percent of the data represents software modules with ten defects, copies may be made of the data with ten defects until there is equal representation within the training data set. This type of data equalization will allow the Genetic Program to treat data with higher defect values with equivalent importance. Therefore, the resulting program solution should predict modules with higher defects as well as it predicts modules with lower defects.

It is anticipated that this approach will result in GP models that are better able to recognize software with high defect counts. Consequently, this will lead to fixing software problems prior to release.

In 1993, NASA established the Independent Verification and Validation (IV&V) Facility as part of an agency-wide strategy to provide the highest achievable levels of safety and cost-effectiveness for mission critical software. As part of this initiative, NASA's IV&V has established a Metrics Data Program (MDP) Data Repository. This repository contains metrics and associated defect rates at the function/method level.

This paper addresses the implicit data starvation issue using data from NASA's MDP data repository. The data, which is significantly skewed towards lower-end values, is equalized to represent each defect value equally. Two sets of experiments are conducted, one on the original data and a second on the equalized data set. Performing these experiments will establish whether equalization is a feasible approach to the implicit data starvation issue.

The paper is organized as follows: Section 2 discusses related research in terms of machine learners applied to defect detection and data preparation. Section 3 describes the NASA project dataset. Section 4 offers an overview of equalized learning. Section 5 describes a series of Genetic Programming experiments. The results are discussed in section 6. Finally, sections 7 and 8 present conclusions and future directions.

2. RELATED WORK

Data Defect Prediction and Machine Learners

Most statistical and software metric-based models struggle with formulating accurate defect predictions [3]. Fenton [3] considers the poor quality of the data as a major cause for the problem. One approach to address this problem is the application of expert systems and machine learners [1, 2, 3, 7, 11, and 14] in formulating a predictor. Such an approach is plausible since expert systems and machine learners handle noisy data and uncertainty rather well.

After considering various reasons for the disappointing results predicting software defects, Fenton [3] concludes that a Bayesian Belief Network would be a possible solution to the problem. Fenton built a tool based on this paradigm called AID, (Assess, Improve, Decide). AID was tested on 28 projects from the Philips Software Centre. The results of this study were promising, however proper training requires great deal of data [4].

The application of Neural Networks to the problem of defect detection has received a great deal of attention. Neural Networks have successfully been applied to predict defects in a chemical processing plant. The results were 10 to 20 times better than the application of traditional methods [12].

Hochmann extends the use of Neural Networks to software defects [7]. Thirty classification models were built with an equal distribution of fault-prone and nonfault-prone software modules. In the first study, a Genetic Algorithm develops optimal back-propagation networks to detect software defects [7]. In the second study, an Evolutionary Neural Network, (ENN), are compared to Discriminate Analysis. The error rates for Discriminate Analysis were much higher than for ENNs. A *z*-test supported the statistical significance of these findings [7].

Evett et al. [1] apply Genetic Programs against several industrial-level repositories in predicting software faults. Their operator set includes *add*, *subtract*, *multiply*, *divide*, *sine*, *cosine*, *exponentiation*, and *logarithms*. The

operands consist of various product metrics including *Halstead's vocabulary*, *McCabe's cyclomatic complexity*, and *Lines of Code (LOC)*. The author's assess their GP models by ranking data sets according to defect counts and comparing the top *n* percent of actual and predicted models where *n* ranges from 75 to 90 percent.

Machine Learners and Data Preparation

Recognizing the problems with Neural Networks, when applied to software defects and the multicollinearity of the data involved, Neumann [11] developed an enhanced technique for risk categorization called PCA-ANN. This approach combines statistics, pattern recognition and Neural Networks. PCA-ANN essentially enhances the Neural Network by working with the input data to orthogonalize and normalize the data. The enhanced Neural Network performed significantly better than a pure Neural Network [11]. Hochmann also recognizing the importance of data preparation used Principal Component Analysis to prepare the data set [7]. Finally, Mizuno proposes a data equalization method to improve the performance of an Artificial Neural Network in technical analysis of the stock market [10].

3. NASA PROJECT DATASET

The data for the machine learning experiments originate from a NASA project, which will be referred to as "KC2." KC2 is a collection of C++ programs containing over 3000 "c" functions. The analysis focuses only on those functions created by NASA developers. This means COTS-based metrics are pruned from the data set. After eliminating redundant data, the final tally consists of metrics from 379 "c" functions.

The KC2 data set contains twenty-one software product metrics based on the product's size, complexity and vocabulary. The size metrics include *total lines of code*, *executable lines of code*, *lines of comments*, *blank lines*, *number of lines containing both code and comments*, and *branch count*. Another three metrics are based on the product's complexity. These include *cyclomatic complexity*, *essential complexity*, and *module design complexity*. The other twelve metrics are vocabulary metrics. The vocabulary metrics include *Halstead length*, *Halstead volume*, *Halstead level*, *Halstead difficulty*, *Halstead intelligent content*, *Halstead programming effort*, *Halstead error estimate*, *Halstead programming time*, *number of unique operators*, *number of unique operands*, *total operators*, and *total operands*.

The KC2 data set also contains the defect count for each module. The majority of the defect values range from 0 to 2. There are 272 instances of zero defects in the modules, 56 instances of one defect, and 25 instances of two defects. Of the remaining 24 module instances, nine have three defects, four have four defects, five have five defects, two have six defects, one has eight defects, one has ten defects, one has eleven defects, and one has thirteen defects. Thus, ninety percent of the defect data is concentrated in 27 percent of the defect value points. The defect distribution is illustrated in Figure 1.

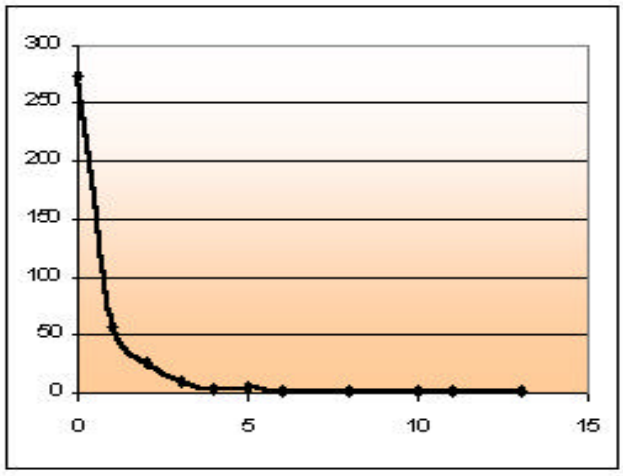

Figure 1: Data Distribution for KC2

4. EQUALIZED LEARNING

Equalized learning is a process of balancing instance frequency within a data set. It involves three steps. The first step calculates a histogram by counting the learning samples in each category. The second step uses information about the importance of each category. Learning samples in these categories are duplicated to modify the histogram. The third step involves shuffling the samples. [10].

For the KC2 data set, every sample is considered equally important. It is equally important to predict zero defects, as it is to predict thirteen defects. If a zero defect module is incorrectly predicted to have many defects time will be wasted trying to detect defects that do not exist. On the other hand, if a thirteen defect module is predicted to have zero defects, no time will be spent fixing the module, which may cost a great deal when the errors show up during software execution. Therefore, the KC2 data set treats all instances with equal weighting.

Balancing the KC2 data set involves determining the most frequently occurring instance, in this case functions with zero defects (272 unique samples). Next, duplicates of the other instances are added so those have a collective total approaching 272 instances. This may be expressed as follows:

Num. of Duplicates = Floor (Max Inst. / Attribute Inst.) *Eq. 1*

where

Maximum Instances refers to the attribute with the highest instance count;

Attribute Instance is the instance count of a particular attribute; and

Floor is a mathematical floor function.

5. MACHINE LEARNING EXPERIMENTS

Overview of Genetic Programs

Genetic Programs solve problems by genetically breeding a population of individuals, or chromosomes, over a series of generations. Inspired by theories of evolution, Genetic Programs use the analogy of evolutionary operators on chromosomes to optimize a fitness function. A fitness function assesses the goodness of a *chromosome*. The goodness of a chromosome serves as the basis for propagation decisions.

An implementation of a Genetic Program starts with a population of individuals, usually represented as trees. Each tree, or chromosome, represents a potential solution to the given problem. Each node on the tree represents a *gene*, or some trait within a problem. Programmatically, each gene corresponds to either an operator or an operand. Collectively, the set of the genes would make up a mathematical expression or function.

Collectively the set of chromosomes, which represent potential solutions, are known as a *population*. This population 'reproduces' to create a future generation. Each iteration of a Genetic Program produces a new generation of individuals.

This process of creating a population of individuals, selecting the fittest individuals within this population, and recombining these individuals to produce better solutions continues until an acceptable solution is found. Figure 2 shows the general Genetic Programming algorithm.

```
The Genetic Program

Initialize the Population
Evaluate Fitness
Repeat
    Select Parents
    Perform Crossover
    Perform Mutation
Until a Desired Fitness is Reached
```

Figure 2: GP Algorithm [5, 6]

After the population has been initialized and the fitness of each individual has been evaluated, the selection of parent chromosomes occurs. During selection, the fittest individuals are selected to engage in reproduction. A *fitness function* evaluates the individuals and ranks them in terms of performance. The best individuals recombine their genetic material or a particular fit individual may be cloned. The poorest performing individuals do not reproduce at all, but are eliminated from the population.

The fitness function is defined as follows:

$$Fitness = 1 + e^{(7*(1-n-k)/(n-1) * Se^2 / Sy^2)} \qquad Eq.\ 2$$

where

k corresponds to the number of inputs;

n represents the number of valid results;

Se is the standard error; and

Sy equals the standard deviation.

This allows a range of values from 1 through 1067. These values are scaled to span 1 through 1000.

Once two individuals within a population have been chosen, several reproductive, or *recombination*, operations may occur. One example is *crossover*. Crossover takes two trees, chooses a random branch, and then crosses over the genetic material. Crossover occurs at one point, or at several points, within the chromosome.

Mutation also alters the genetic material of a chromosome. Mutation prevents a solution from falling into a local minima or maxima, which is a problem experienced by most optimization algorithms [6, 13]. While crossover exchanges genetic material between two chromosomes, mutation randomly selects and changes some genes within one chromosome and passes this change onto the offspring. Mutation randomly selects a node in the tree and changes the genetic material. Mutation promotes diversity within a population by randomly adding in gene variations.

This process repeats itself until the optimal fitness level is reached. Each iteration of the Genetic Program is called a generation. Sometimes a Genetic Program will not find a perfect solution, or the desired fitness level is not reached, so program termination is determined by the number of generations.

Genetic Programs solve optimization and induction problems particularly well. While machine learners, in general, excel at these types of problems, Genetic Programs provide the additional advantage of a symbolic high-level representation of the problem solution. A high-level representation facilitates a faster and easier interpretation of the data [8].

Experiment Description

To verify the accuracy of the equalized learning approach, we conduct two Genetic Programming experiments. The first experiment analyzes the original

KC2 data set (after the duplicates are removed). As described above, this data set consists of 379 samples. The second experiment equalizes this KC2 data set. Overall, there are 3,013 samples in the equalized data set.

Each experiment consists of twenty trials. Repeating each experiment twenty times creates enough samples for comparing results. In order to maintain experimental consistency, all GP settings are the same for all runs. The GP settings include maximum 50 generations; chromosome length of 2000 characters; and population size of 1000 chromosomes.

Results

The difference between the two sets of experiments is significant. The unequalized data set achieves an average fitness value of 12, while the equalized data produces an average fitness value of 81.4. A *t*-test shows that these results are statistically significant as indicated below.

Table 1: Experiment 1 t-Test Results

	Original	**Equalized**
Mean	12	81.4
Variance	10.42105	1061.095
Observations	20	20
Pearson Correlation	0.67619	
Hypothesized Mean Difference	0	
Df	19	
t Stat	-10.1811	
P(T<=t) one-tail	1.97E-09	
t Critical one-tail	1.729131	

As observed in Table 1, the T value is significantly less than 1%, the null hypothesis may be rejected. Of course, the null hypothesis is that there is no significant difference between the two groups. Thus, it is verified that there is a significant statistical difference between the two groups.

6. DISCUSSION

These experiments demonstrate that it is possible to compensate for underrepresented values in a data set. Furthermore, data equalization is a valuable endeavor in the data mining process. This method proves especially useful when it is precisely the underrepresented data that most need to be forecasted and identified.

A negative aspect to the data equalization technique is that it vastly increases the size of the learning set. Genetic Programs, and some other machine learners, may encounter performance problems. This process slows the GP training process, and this factor that should be considered prior to equalizing data. If the Genetic Program can formulate an acceptable answer without equalization, it may not be worth the time and effort to equalize the data. In some cases, if the data set is very large, equalization may become at best a burdensome task, and at worst infeasible.

However, these experiments have shown, that for reasonably sized data sets, data equalization is indeed a valuable tool.

7. CONCLUSIONS

These experiments show that data equalization is a feasible approach for implicitly starved data when applied to a Genetic Program. Poor quality data does not necessarily have to imply a poor solution to the problem.

Using empirical data from NASA's IV&V facility, two separate experiments were performed. One ran the Genetic Program with the original data resulting in a fitness value of 12. The second set of experiment ran the GP with the data set equalized and resulted in a fitness value of 81.4. A *t*-test verified that the difference is statistically significant.

Equalizing the data helps compensate for under-represented values in the data set. This approach offers very good potential for improving the performance of machine learners in implicitly data starved environments.

8. FUTURE DIRECTIONS

The process of equalizing data in an implicitly starved environment has proven to be valuable when applied to a Genetic Program in the defect detection domain. Other research has found it useful when applied to neural networks in the financial forecasting domain [10].

The amount of work done in this area remains limited to a few studies. It seems logical to assume that this methodology could be equally useful when applied to other machine learners, as well as other domains. More research done in this area would help validate the conclusions drawn in this paper.

Another possible area of research involves the effects of data equalization on the performance of the machine learner. It would be useful to know in which situations this methodology is appropriate and when it is infeasible due to the size of the data and its effects upon performance.

Finally, it would be useful to apply to the results from one NASA defect data set to other NASA defect data sets to determine if the solution is transferable. Also, a comparison can be made with the solution obtained from the original data set and from the equalized data set when applied to a different data set.

9. REFERENCES

[1] Cohen, William W., Devanbu, Prem., *A Comparative Study of Inductive Logic Programming Methods for*

Software Fault Prediction, Proc. 14th International Conference on Machine Learning., 1997.

[2] Evett, Matthew., Khoshgoftar, Taghi., *GP-based Software Quality Prediction.*, Genetic Programming 1998: Proceedings of the Third Annual Conference, 1998.

[3] Fenton, Norman E. *A Critique of Software Defect Prediction Models.* IEEE Transactions on Software Engineering, Vol. 25, No 3, May/June 1999.

[4] Fenton, Norman E., Krause, Paul., Neil, Martin. *A Probabilistic Model for Software Defect Prediction*, for submission to IEEE Transactions in Software Engineering. 2001.

[5] Goldberg, David. (1989) *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, USA.

[6] Grefenstette, John J. *Incorporating Problem Specific Knowledge into Genetic Algorithms.* In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, Chapter 4, pages 42-60. Morgan Kaufmann Publishers, Inc., 1987.

[7] Hochmann, J.P., Hudepohl, E.B., Allen, T.M., Khoshgoftaar, R. *Evolutionary Neural Networks: A Robust Approach to Software Reliability*, Eighth International Symposium on Software Reliability Engineering (ISSRE '97), pages 13-26, 1997.

[8] Koza, John. (1992) *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, Massachusetts.

[9] Metrics Data Program, NASA IV&V facility. Information Datasets are available at: http://mdp.ivv.nasa.gov/about.html

[10] Mizuno, et al., Application of Neural Network To Technical Analysis of Stock Market Prediction, Studies in Informatics and Control (With Emphasis on Useful Applications of Advanced Technology), 7 2, June 1998. This paper may be downloaded at: www.neuralstocks.com/Kimoto.pdf

[11] Neumann, Donald E., *An Enhanced Neural Network Technique for Software Risk Analysis***,** IEEE Transactions on Software Engineering, pages 904-912, 2002.

[12] Stites, Robert L, Ward, Bryan, Walters, Robert V. *Defect Prediction with Neural Networks*, Proceedings of the conference on Analysis of neural network applications, pages 199 – 206, ACM Press, 1991.

[13] Whitley, Darrel. (1993) *A Genetic Algorithm Tutorial*. Technical Report CS-93-103, November 10, 1993. Department of Computer Science, Colorado State University.

[14] Zhang, Du. *Applying Machine Learning Algorithms in Software Development*, Proceedings of the 2000 Monterey Workshop on Modeling Software System Structures in a Fastly Moving Scenario, pages 275-291, 2000.