Nearest Neighbor Sampling for Better Defect Prediction

Gary D. Boetticher
University of Houston – Clear Lake
2700 Bay Area Boulevard
Houston, Texas 77059
1 281.283.3805
Boetticher@cl.uh.edu

ABSTRACT
An important step in building effective predictive models applies one or more sampling techniques. Traditional sampling techniques include random, stratified, systemic, and clustered. The problem with these techniques is that they focus on the **class attribute**, rather than the **non-class attributes**. For example, if a test instance's nearest neighbor is from the opposite class of the training set, then it seems doomed to misclassification. To illustrate this problem, this paper conducts 20 experiments on five different NASA defect datasets (CM1, JM1, KC1, KC2, PC1) using two different learners (J48 and Naïve Bayes). Each data set is divided into 3 groups, a training set, and "nice/nasty" neighbor test sets. Using a nearest neighbor approach, "Nice neighbors" consist of those test instances closest to class training instances. "Nasty neighbors" are closest to opposite class training instances. The "Nice" experiments average 94 percent accuracy and the "Nasty" experiments average 20 percent accuracy. Based on these results a new nearest neighbor sampling technique is proposed.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: *Metrics;* D.2.8 [**Software Engineering**]: *Testing and Debugging;* I.2.6 [**Artificial Intelligence**]: *Learning---Knowledge Acquisition*

General Terms
Measurement, Experimentation

Keywords
Empirical Software Engineering, Defect Prediction, Nearest Neighbor analysis, Decision Trees, NASA Data Repository

1. INTRODUCTION
Defect Prediction in Empirical-based Software Engineering seeks to build accurate, reliable data-driven models that will be embraced by project managers. To this end, public data repositories [10] and commercial quality public domain tools [1, 13] are readily available.

As researchers build models, their results serve as benchmarks for comparing/contrasting ideas and theories on tool usage, metric selection, and process definition. The lack of process standardization dampens repeatability in terms of results. A common example is the multiple ways of defining effort: Pred(X), MMRE, #Correct/TotalInstances. Even when two researchers use the same indicator (e.g. Pred(X), they both might assume different values for X (25 or 30 percent).

Another overlooked step in Empirical-based Software Engineering is the sampling process. Traditional approaches such as random, stratified, systemic, or clustered are widely accepted. Most of these approaches (random is the exception) assess a class (or dependent) variable in determining how to partition tuples into training/test sets. The problem is that non-class (independent) variables contain a lot of vital information which needs to be considered when partitioning tuples.

To demonstrate how non-class attributes dramatically impact the modeling process, a series of experiments are conducted against five NASA defect data sets. The experiments fall into two categories. The "Nice" experiments use a test set where the non-class attributes of the test data have nearest-neighbors in the training set and the same class value. The "Nasty" experiments use a test set where the non-class attributes of the test data have nearest-neighbors in the training set with an opposite class value. Experimental results for the "Nice" experiments are 97, 94, 91, 88, and 98 percent respectively. The "Nasty" experiments achieve accuracy rates of 6, 16, 23, 42, and 19 percent. These experiments are described and discussed in sections 2 through 5.

Based on these experiments and the corresponding discussion, section 6 describes two different nearest neighbor sampling approaches which could be incorporated into the data mining process. Benefits of using such a technique would be better data representation, better models, and higher probability that project managers will embrace Empirical Software Engineering practices.

Finally, sections 7 and 8 present conclusions and future directions.

2. NASA DATA SETS
All experiments use five public domain defect datasets from the NASA Metrics Data Program (MDP) and PROMISE repository [10]. These five data sets, referred to as CM1, JM1, KC1, KC2, and PC1, contain static code measures (e.g. Halstead, McCabe, LOC) along with defect rates. Table 1 provides a project description for each of these data sets.

PROMISE'05 May 15, 2005 St. Louis, Missouri, USA

Table 1. Project Description for Each Data Set

Project	Source Code	Description
CM1	C	NASA spacecraft instrument
KC1	C++	Storage management for receiving/processing ground data
KC2	C++	Science data processing. No software overlap with KC1.
JM1	C	Real-time predictive ground system
PC1	C	Flight software for earth orbiting satellite

Each 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 class attribute for each data set refers to propensity for defects. The original MDP data set contains numeric values for the defects, while the PROMISE data sets convert the numeric values to Boolean values where *TRUE* means a component has 1 or more defects and *FALSE* equates to zero defects. The reason for the conversion is that the numeric distribution displayed signs of an implicitly data-starved domain (many data instances, but few of interest) where less than 1 percent of the data has more than 5 defects [5].

3. DATA PREPROCESSING

Data pre-processing removes all duplicate tuples from each data set along with those tuples that have questionable values (e.g. LOC equal to 1.1). Table 2 shows the general demographics of each of the data sets after pre-processing.

Table 2. Data Pre-Processing Demographics

Project	Original Size	Size w/ No Bad, No Dups	0 Defects	1+ Defects	% Defects
CM1	498	441	393	48	10.9%
JM1	10,885	8911	6904	2007	22.5%
KC1	2109	1211	896	315	26.0%
KC2	522	374	269	105	28.1%
PC1	1109	953	883	70	7.3%

4. EXPERIMENTS

4.1 Training and Test Set Formulation

To assess the impact of nearest-neighbor sampling upon the experimental process twenty experiments are conducted on each of the five data sets.

For each experiment, a training set is constructed by extracting 40 percent of data from a given data set. Selecting the 40 percent uses stratified sampling, in that it maintains the ratio between Defect/Non-defect data. As an example, JM1 has 8911 records, 2007 of which, or 22.5 percent, have 1 or more defects. An corresponding training set for the JM1 project contains 3564 records, 803 (or 22.5 percent) of which are classified as having 1 or more defects (TRUE).

It could be argued that a greater percentage (more than 40 percent) of the data could be committed to the training set. There are several reasons for choosing only 40 percent. First, Menzies claims that only a small portion of the data is needed to build a model [5]. Second, since the data is essentially a two-class problem, there was no concern about whether each class would receive sufficient representation. Finally, it is necessary to insure that there is sufficient amount of test data for assessing the results.

Once a training set is established, the remaining 60 percent of the data is partitioned into two test groups. Prior to splitting the test data, all of the non-class attributes are normalized by dividing each value by the *Difference* ($Maximum_k$ - $Minimum_k$ for each column k). This guarantees that each column receives equal weighting. The next step loops through all the test records. Each test record is compared with every training record to determine the minimum Euclidean Distance for all of the non-class attributes. If the training and test tuples with the smallest Euclidean Distance share the same class values (TRUE/TRUE or FALSE/FALSE), then add the test record to the Nice Neighbor test set, otherwise add it to the Nasty Neighbor test set. Figure 1 shows the corresponding algorithm.

Essentially, this is the K-Nearest Neighbor algorithm that determines a test tuples closest match in the training set. Nearest neighbors from the same class are considered "Nice," otherwise they are classified as "Nasty."

```
For j=1 to test.record_count
   minimumDistance = 9999999
   For i=1 to train.record_count
     Dist = 0
     For k=1 to train.column_count - 1
       Dist = Dist + (train_ik – test_jk)^2
     end k
     if (abs(Dist) < abs(minimumDistance))
       then if Train_i.defect = Test_j.defect
         then minimumDistance = Dist
         else minimumDistance = -Dist
   end i;
   if minimumDistance > 0
     then Add_To_Nice_Neighbors
   if minimumDistance < 0
     else Add_To_Nasty_Neighbors
   if minimumDistance = 0
      then if Train_i.defect = Test_j.defect
        then Add_To_Nice_Neighbors
        else Add_To_Nasty_Neighbors
end i:
```

Figure 1: Nice/Nasty Neighbor Algorithm

All experiments use the training data to build a model between the non-class attributes (e.g. *size*, *complexity*, *vocabulary*) and the class attribute *defect* (which is either *True* or *False*).

After constructing 150 data sets (1 training, 2 sets per trial; 20 trials per software project; 5 software projects), attention focuses on data mining tool selection.

4.2 Data Mining Tool Selection

Since the data contains 20-plus attributes and only two class values (TRUE/FALSE), the most reasonable choice of data mining tools is a decision tree learner. A decision tree selects an attribute which best divides the data into two homogenous groups (based on class value). The split selection continues recursively on the two (and sometimes more) subtrees until all children of a split are totally homogenous (or the bin dips below a prescribed threshold). Decision tree learners are described as greedy in that they do not look ahead (2 or more subtree levels) due to the associated computational complexity.

One of the most popular Public Domain Data Mining tools is the Waikato Environment for Knowledge Analysis (WEKA) tool [13]. WEKA is an open-source machine learning workbench, implemented in Java, it incorporates many popular machine learners and is widely used for practical work in machine learning. According to an a recent KDD poll [3], Weka was rated number two in terms of preferred usage as compared to other commercial and public domain tools.

Within WEKA, there are many learners available. The experiments specifically use the Naïve Bayes and J48 learners for analysis. The primary reason for adopting these tools is previous success achieved by Menzies et al. [6] in their analysis of the NASA defect repositories.

A Naïve Bayes classifier uses a probabilistic approach to assign the most likely class for a particular instance. For a given instance $_x$, a naïve Bayes classifier computes the conditional probability

$$P(C = c_i \mid x) = P(C = c_i \mid A_1 = a_{i1},\ldots A_n = a_{in}) \qquad (1)$$

for all classes c_i and tries to predict the class which has the highest probability. The classifier is considered naïve [9] since it assumes that the frequencies of the different attributes are independent.

A second learner, J48, is based on Quinlan's C4.5 [8].

4.3 Assessment Criteria

The assessment criterion uses three metrics in all experiments to describe the results. They are:

- **PD**, which is the probability of detection. This is the probability of identifying a module with a fault divided by the total number of modules with faults.
- **PF**, which is the probability of a false alarm, This is defined as the probability of incorrectly identifying a module with a fault divided by the total number of modules with no faults.
- **Acc**, which is the accuracy. This is the probability of correctly identifying faulty and non-faulty modules divided by the total number of modules under consideration.

Each of these metrics is based on simple equations constructed from WEKA's confusion matrix as illustrate by Table 3.

Table 3. Definition of Accuracy

	A Defect is Detected.	**A Defect is not Detected.**
A Defect is Present.	**A = 50** Predicted=TRUE Actual= TRUE	**B = 200** Predicted= FALSE Actual= TRUE
A Defect is not Present.	**C = 100** Predicted= TRUE Actual=FALSE	**D = 900** Predicted= FALSE Actual= FALSE

PD is defined as:

$$PD = A / (A+B) \qquad (2)$$

PF is defined as:

$$PF = C / (C + D) \qquad (3)$$

and *Acc* is defined as:

$$Acc = (A + D) / (A + B + C + D) \qquad (4)$$

Based on the example in Table 3, the corresponding values would be:

$$PD = 50 / (50 + 200) = 20\% \qquad (5)$$

$$PF = 100 / (100 + 900) = 10\% \qquad (6)$$

$$Acc = (50 + 900) / (50 + 100 + 200 + 900) = 76\% \qquad (7)$$

4.4 Results

Table 4 shows the accuracy results of the 20 experiments per project. As might be expected, the "Nice" test set did very well for all five projects for both machine learners averaging about 94 percent accuracy. Its counterpart, the "Nasty" test set, did not fare very well, averaging about 20 accuracy.

It is interesting to note that the JM1 data set, with 7 to 20 times more tuples than any of the other projects, is above the overall average for the "Nice" data sets, and below the overall average on the "Nasty" data sets. Considering the large number of tuples in this data set, it would seem that a nearest neighbor approach would be diluted by the large number of tuples.

Table 4. Accuracy Results from All the Experiments

	Nice Test Set		**Nasty Test Set**	
	J48	**Naïve Bayes**	**J48**	**Naïve Bayes**
CM1	97.4%	88.3%	6.2%	37.4%
JM1	94.6%	94.8%	16.3%	17.7%
KC1	90.9%	87.5%	22.8%	30.9%
KC2	88.3%	94.1%	42.3%	36.0%
PC1	97.8%	91.9%	19.8%	35.8%
Overall Average	94.4%	93.6%	18.7%	21.2%

Regarding *PD*, the results as expressed in Table 5 for the "Nice" test set are superior to the "Nasty" test set for the learners. An overall weighted average is preferred over a regular average in order not to bias the results towards those experiments with very few defect samples.

The results in Table 5 can be misleading. 76 of the 100 "Nice" test sets contained zero defect tuples. Of the remaining 24 "Nice" test sets, only 2 of these 24 had 20 or more samples with defects.

Table 5. Probability of Detection Results

	Nice Test Set		Nasty Test Set	
	J48	**Naïve Bayes**	**J48**	**Naïve Bayes**
CM1	0.0%	0.0%	5.9%	37.4%
JM1	71.9%	92.9%	14.9%	16.4%
KC1	45.8%	87.5%	22.7%	31.0%
KC2	100.0%	100.0%	42.2%	36.0%
PC1	11.7%	75.0%	10.9%	30.8%
Overall Weighted Average	**45.6%**	**60.8%**	**16.8%**	**20.0%**

The "Nice" test set did very well at handling false alarms as depicted in Table 6. The "Nasty" test set triggered alarms about 18 to 37 percent of the time depending upon learner. Overall, the sample size is small for the "Nasty" test sets. 90 percent (from the 100 experiments) of the "Nasty" test sets contain zero instances of non-defective data. For the remaining 10 "Nasty" data sets, only 3 contain 10 or more instances of non-defective modules.

Table 6. Probability of False Alarms Results

	Nice Test Set		Nasty Test Set	
	J48	**Naïve Bayes**	**J48**	**Naïve Bayes**
CM1	2.6%	11.7%	0.0%	50.0%
JM1	5.1%	5.0%	61.7%	66.1%
KC1	9.0%	12.5%	46.4%	91.7%
KC2	11.8%	5.9%	0.0%	50.0%
PC1	1.7%	7.9%	1.9%	70.6%
Overall Weighted Average	**5.4%**	**6.3%**	**18.5%**	**37.1%**

To better understand these results, consider Tables 7 and 8. These tables show the weighted averages (rounded) of all confusion matrices for all 100 experiments (20 per test group). In the "Nice" test data, 99.6 percent are defined as having no defects (FALSE). Less than 1 percent of the tuples actually contain defects. Although the "Nice" test sets fared better than the "Nasty" test sets regarding defect detection, the relatively few samples defect samples in the "Nice" test sets discount the results.

Analyzing the "Nasty" data sets in Tables 7 and 8 reveal that 97.2 percent (e.g. (50 + 249)/(50+249+2+7)) of the data contains 1 or more defects.

Table 7. Confusion Matrix, Nice Test Set (Rounded)

J48		Naïve Bayes	
2	3	3	2
58	1021	68	1011

Table 8. Confusion Matrix, Nasty Test Set (Rounded)

J48		Naïve Bayes	
50	249	60	241
2	7	3	5

Referring back to the right-most column of Table 2, the percentage of defects to total number of modules ranged from 7.3 to 28.1 percent. Considering that all training sets maintained their respective project ratio of defects to total components, it is quite surprising that the "Nice" and "Nasty" data sets would average such high proportions of non-defective and defective components respectively.

To better understand these results, two additional experiments are conducted using the KC1 data set. The first experiment randomly allocates 60 percent of the data to the training set, while the second allocates 50 percent of the data. Both experiments maintain a defect/non-defective ratio of 26 percent (see Table 2). For both experiments the test data is divided into 8 groups using a 3-nearest neighbor approach. For each test vector, its 3 closest neighbors from the training set are determined. These neighbors are ranked based on first, second, to third closest neighbor. A "P" means that there is a positive match (same class), and an "N" means there is a negative match (opposite class). Thus, a "PPN" means that the first and second closest matches are from the same class and the third closest match is from the opposite class. Thus, the best case would be a "PPP" where the 3 closest training vectors are all from the same class.

Tables 9 and 10 show the results from these experiments. It is interesting to note that all 8 bins contain homogenous (all TRUEs, or all FALSEs) data. There is a general trend for the bin configuration to change from all non-defective tuples (all FALSEs) to all defective tuples (TRUEs) as the neighbor status changes from all positives (PPP) to all negatives (NNN). Also, the accuracy seems positively correlated to the nearest neighbor classifications.

Table 9. KC1 Data, KNN=3, 60 Percent of Training Data

			Accuracy	
Neighbor Description	**# of TRUEs**	**# of FALSEs**	**J48**	**Naïve Bayes**
PPP	None	None	NA	NA
PPN	0	354	88	90
PNP	0	5	40	20
NPP	None	None	NA	NA
PNN	3	0	100	0
NPN	13	0	31	100
NNP	110	0	25	28
NNN	None	None	NA	NA

Table 10. KC1 Data, KNN=3, 50 Percent of Training Data

			Accuracy	
Neighbor Description	**# of TRUEs**	**# of FALSEs**	**J48**	**Naïve Bayes**
PPP	0	19	89	84
PPN	0	417	91	91
PNP	0	13	23	0
NPP	None	None	NA	NA
PNN	None	None	NA	NA
NPN	18	0	100	100
NNP	132	0	20	20
NNN	7	0	0	29

These last two experiments confirm the results achieved in tables 7 and 8.

5. DISCUSSION

In general, project managers are reluctant to embrace empirical-based models in their decision-making process. Jurgenson [2] estimates more than 80 percent of all effort estimation is human-based and less than 4 percent is machine learning based. If predictive model builders in Software Engineering are going to have any hope of gaining favor with project managers, then it is critical that the modeling process be understood very well.

Using an n-nearest neighbor modeling approach provides an opportunity to do an in-depth study of why test vectors are misclassified. For example, normally Table 10 would be merged into one test set. The corresponding confusion matrices would be:

Table 11: KC1 Data, Percent of Training Data

J48		**Naïve Bayes**	
44	113	46	111
51	398	52	397

At this level of granularity, the modeler would be unable to recognize that 18 of the test vectors are classified with 100 percent accuracy even though the first and third closest neighbors are from the opposite class (row NPN in Table 10) and proceed to conduct further research.

6. NEAREST NEIGHBOR-BASED SAMPLING

It is evident that nearest neighbor test data distribution dramatically impacts upon experimental results. The question is *How does a modeler incorporate nearest neighbor sampling to generate realistic models?*

There are at least two possible solutions. One of which adapts to an organization's current data mining process; the second solution offers an alternative process.

In the first approach a data miner determines the nearest neighbor for each of the tuple in the test set (based on non-class attributes), relative to the training set. If the test tuple's nearest neighbor in the training set shares the same class instance value, then add *1* to a variable called *Matches*. Define a metric called *Experimental Difficulty* (*Exp_Difficulty*) as follows:

$$Exp\_Difficulty = 1 - Matches / Total\_Test\_Instances \quad (8)$$

The *Experimental Difficulty* provides a qualitative assessment of the ease/difficulty a data miner encounters for a given experiment. Synthesizing this metric with an accuracy metric would offer a more realistic assessment of results. For example, a "*Exp_Difficulty * Accuracy"* would give a more complete picture regarding the goodness of a model leading to better model selection and more credible models.

A second approach starts with the whole data set prior to partitioning into training and test sets. For each tuple in the data set its nearest neighbor (with respect to the non-class attributes) is determined. Add 1 to the *Match* variable if a tuple's nearest neighbor is from the same class. Modifying equation 8, results in the following equation:

$$Overall\_Difficulty = 1 - Matches / Total\_Data\_Instances \quad (9)$$

This gives an idea of the overall difficulty of the data set. The data modeler may partition the data in order to increase (or decrease) *Experimental_Difficulty*. In the context of industrial-based benchmarks, a data modeler might also choose to partition the data so that the *Experimental_Difficulty* coincides to a value adopted by another researcher. This lends greater credibility to comparing experimental results.

7. CONCLUSIONS

There has been significant amount of research in the area of machine learners in defect prediction [4, 7, 11, and 12]. What distinguishes this work from prior efforts is the demonstrating of nearest-neighbor analysis for gaining a deeper understanding of how data relates to each other, and thus the need for developing more representative models.

The experiments also demonstrate that a biased data distribution dramatically impacts test results. Ignoring the data distribution discounts the results achieved by a learner and reducing the chances that industry will embrace empirical-based Software Engineering.

Based upon the experiments, two nearest-neighbor sampling approaches are presented in terms of how they may be incorporated into a data mining process.

8. FUTURE DIRECTIONS

This work could be extended to examine data sets that contain more than 2 classes (e.g. actual defect counts). For example, the NASA data sets may be divided into four classes, (0, 1, 2, 3+ defects).

Also, additional research could be conducted using the 3-Nearest Neighbor, 5-Nearest Neighbor. The last two experiments produced some interesting results worthy of additional study.

9. REFERENCES

[1] Boetticher, G., Data Mining Software Tools (2005), CSCI5833 Data Mining Tools and Techniques, Department of Computer Science, University of Houston – Clear Lake, Houston, Texas. Available at:

http://nas.cl.uh.edu/boetticher/CSCI5931%20Data%20Mining.html

[2] Jorgensen, M., "A review of studies on Expert Estimation of Software Development Effort," *Journal of Systems and Software*, 70 (1-2), Pp. 37-60, 2004.

[3] KDD Nuggets Website, Polls: Data Mining Tools You Regularly Use, Knowledge Discovery and Data Mining Poll, http://www.kdnuggets.com/polls/data_mining_tools_2002_june2.htm

[4] Khoshgoftaar, T.M., and E.B. Allen, "Model software quality with classification trees," in *Recent Advances in Reliability and Quality Engineering*, H. Pham, Ed. 2001, pp. 247–270, World Scientific.

[5] Menzies, Tim, Personal Conversation, February 2, 2005.

[6] Menzies, T., Raffo D., Setamanit, S., DiStefano, J., Chapman, R., Why Mine Repositories, Submitted to*: Transactions on Software Engineering*, 2005.

[7] Porter, A.A., and R.W. Selby, "Empirically guided software development using metric-based classification trees," *IEEE Software*, pp. 46–54, March 1990.

[8] Quinlin, J.R., C4.5: Programs for machine learning. California: Morgan Kaufmann, 1992.

[9] Rish, I., An empirical study of the naive Bayes classifier, T.J. Watson Center, *IJCAI-01 Workshop on Empirical Methods in Artificial Intelligence*, Seattle, 2001.

[10] Shirabad, J.S., and Menzies, T.J. (2005) The PROMISE Repository of Software Engineering Databases School of Information Technology and Engineering, University of Ottawa, Canada . Available: http://promise.site.uottawa.ca/SERepository

[11] Srinivasan, K., and D. Fisher, "Machine learning approaches to estimating software development effort," *IEEE Trans. Soft. Eng.*, pp. 126–137, February 1995.

[12] Tian, J., and M.V. Zelkowitz, "Complexity measure evaluation and selection," *IEEE Transaction on Software Engineering*, vol. 21, no. 8, pp. 641–649, Aug. 1995.

[13] Witten, Ian, Eibe Franks, "Data Mining: Practical machine learning tools with Java implementations," Morgan Kaufmann, San Francisco, 2000.