Improving Credibility of Machine Learner Models in Software Engineering

Gary D. Boetticher
University of Houston – Clear Lake
2700 Bay Area Boulevard
Houston, Texas 77059
Voice: 1 281.283.3805
FAX: 1 281.283.3869
Boetticher@cl.uh.edu

Improving Credibility of Machine Learner Models in Software Engineering

ABSTRACT

Given a choice, software project managers frequently prefer traditional methods of making decisions rather than rely on empirical software engineering (empirical/machine learning-based models). One reason for this choice is the perceived lack of credibility associated with these models. To promote better empirical software engineering, a series of experiments are conducted on various NASA datasets to demonstrate the importance of assessing the ease/difficulty of a modeling situation. Each dataset is divided into 3 groups, a training set, and “nice/nasty” neighbor test sets. Using a nearest neighbor approach, “Nice neighbors” align closest to same class training instances. “Nasty neighbors” align to the opposite class training instances. The “Nice,” “Nasty” experiments average 94 and 20 percent accuracy respectively. Another set of experiments show how a 10-fold cross-validation is not sufficient in characterizing a dataset. Finally, a set of metric equations is proposed for improving the credibility assessment of empirical/machine learning models.

KEYWORDS

Data Forecasting, Data Mining, Data Mining Algorithms, Heuristics, Knowledge Discovery, Machine Learning, Software Management, Deterministic/Probabilistic Systems, Emerging Information technologies, Governmental IS, Decision Trees, Innovative Technology, Cost Estimation, Quality, Software Metrics, Software Quality, Time and Cost Estimation, Debugging, IS Maintenance, Program testing, Cost Estimation, Effort Estimation, Knowledge Classification, Measures, Meta-Analysis, Defect Prediction, Empirical Software Engineering

INTRODUCTION

Software Project Management: State Of Practice

Software project management has improved over the years. For example, the Standish Group, a consulting company, which has been studying IT management since 1994 noted in their latest release of the Chaos Chronicles (Standish 2003) that "2003 Project success rates improved by more than 100 percent over the 16 percent rate from 1994." Furthermore, "Project failures in 2003 declined to 15 percent of all projects. This is a decrease of more than half of the 31 percent in 1994."

Even with these successes, there are still significant opportunities for improvement in software project management. Table 1 shows several "state of practice" surveys collected in 2003 from IT companies in the United States (Standish 2003); South Africa (Sonnekus 2003); and the United Kingdom (Sauer 2003).

Table 1: State of Practice Surveys

Year	Successful	Challenged	Failure	Projects Surveyed
United States (Chaos Chronicles III)	34%	51%	15%	13,522
South Africa	43%	35%	22%	1,633
United Kingdom	16%	75%	9%	421

According to the Chaos Chronicles (Standish 2003) *successful projects* refers to projects that are completed on time and within budget with all features fully implemented; *project challenged* means that the projects are completed, but exceed budget, go over time, and/or are lacking some/all of the features and functions from the original specifications; and *project failures* are those projects which are abandoned and/or cancelled at some point.

Applying a weighted average to Table 1 results in 34 percent of the projects identified as successful, 50 percent are challenged, and 16 percent end up in failure. Thus, about one-third of the surveyed projects

end up as a complete success, half the projects fail to some extent, and one sixth end up as complete failures. Considering the role of computers in various industries, such as the airlines and banking, these are alarming numbers.

From a financial perspective[1], the lost dollar value for US projects in 2002 is estimated at $38 billion with another $17 billion in cost overruns for a total project waste of $55 billion against $255 billion in project spending (Standish 2003). Dalcher (Dalcher 2003) estimates the cost for low success rates at $150 billion per year attributable to wastage arising from IT project failures in the Unites States, with an additional $140 billion in the European Union. Irrespective of which estimate is adopted, it is evident that software project mismanagement results in an annual waste of billions of dollars.

Empirical Software Engineering

One of the keys for improving the chances of project development success is the application of empirical-based software engineering. **Empirical-based software engineering** is the process of collecting software metrics and using these metrics as a basis for constructing a model to help in the decision-making process.

Two common types of software metrics are project and product metrics. **Project metrics** refer to the estimated time, money, or resource effort needed in completing a software project. The Standish Group (Standish 2003) perceives software cost estimating as the *most effective way to avoid cost and schedule*. Furthermore, several studies (Chaos 2003 and Jones 1998) have shown that by using software cost-estimation techniques, the probability of completing a project successfully doubles. Thus, estimating the schedule, cost, and resources needed for the project is paramount for project success.

[1] All monetary amounts are depicted in U.S. dollars.

Product metrics are metrics extracted from software code and are frequently used for software defect prediction. Defect prediction is a very important area in the software development process. The reason is that a software defect dramatically escalates in cost over the software lifecycle. During the coding phase, finding and correcting defects costs $977 per defect (Boehm 2001). In the system-testing phase, the cost jumps to $7,136 per defect (Boehm 2001). If a defect survives to the maintenance phase, then the cost to find and remove increases to $14,102 (Boehm 2001).

From an industry perspective, Tassey (Tassey 2002) estimates that the annual cost of software defects in the United States is $59.5 billion and that “feasible” improvements to testing infrastructures could reduce the annual cost of software defects in the United States by $22.2 billion. The Sustainable Computing Consortium (SCC), an academic, government and business initiative to drive IT improvements, estimates that from a global perspective, defective computer systems cost companies $175 billion annually. Thus, there are major financial incentives for building models capable of predicting software defects.

There are primarily two methods for constructing models in empirical software engineering. The first adopts an **empirical approach** which emphasizes direct observation in the modeling process and results in one or more mathematical equations. Common examples of this approach include COCOMO I (Boehm 1981), COCOMO II (Boehm 2000), Function Points (Albrecht 1979; and 1983), SLIM (Putnam 1978) for effort estimation. The second method automates the observation process by using a **machine learning** approach to characterize relationships. Examples of machine learners include Bayesian Belief Networks (BBN), Case-Based Reasoners (CBR), decision tree learners, genetic programs, and neural networks. The by-product of applying these learners include mathematical equations, decision trees, decision rules, and a set of weights as in the case of a neural network. For the purpose of this paper, a

cursory description will suffice, further details regarding the application of machine learners in software engineering may be found at (Khoshgoftaar 2003; Pedrycz 2002; and Zhang 2005).

Although the financial incentives are huge, empirical software engineering has received modest acceptance by software practitioners. A project manager may estimate a project by using a human-based approach, an empirical approach, or a machine learning approach. Even though popular models, such as COCOMO or Function Points, have existed for more 25 or more years their application is rather low. In 2004, Jørgenson (2004) compiled a series of studies regarding the frequency of human-based estimation and finds it is used 83 percent (Hihn 1991), 62 percent (Heemstra 1991), 86 percent (Paynter 1996), 84 percent (Jørgenson 1997), and 72 percent (Kitchenham 2002). It is evident that human-based estimation is the dominant choice relative to empirical or machine learning-based estimation. Furthermore, Jørgenson (2004) states the empirical-based estimation ranges from about 7 to 26 percent and machine learning-based estimation is only about 4 to 12 percent.

A key question is "*Why haven't empirical-based models and particularly machine learner models gained greater acceptance by software practitioners?"*

One possible answer to this question is the difficulty in assessing the credibility of these models. A model may boast accurate results, but these results may not be realistic. Thus, the lack of perceived credibility makes it difficult for software practitioners to adopt a new technology within their software development process.

It may be argued that "credibility processes" exist in the form of *n*-fold cross validation and accuracy measures. Unfortunately, none of these approaches address the issue of a dataset's difficulty (how easy or hard is it to model). An empirical model that produces spectacular results may be the consequence of a test set that closely resembles a training set rather than the capability of empirical learner. The outcome

is a set of unrealistic models that is unlikely to be adopted by industry. Without a mechanism for assessing a dataset's difficulty, empirical/machine learning models lose credibility.

This paper introduces several credibility metrics which may be incorporated into the model formulation process with the goal of giving greater credibility. Before introducing these metrics, a series of experiments are conducted to demonstrate some of the difficulties in using a dependent variable for making sampling decisions. The **dependent variable**, also known as the **class variable**, is the attribute we are trying to predict from a set of independent (non-class) variables. The initial set of experiments demonstrates the importance of sampling on non-class attributes. These experiments examine the "best case" versus "worst case" for a NASA-based defect repository dataset.

Next, a second set of experiments demonstrates that a *10*-fold cross validation may not be sufficient in building realistic models.

After demonstrating some of the inadequacies of current validation processes, several credibility metrics are introduced. These metrics measure the *difficulty* of a dataset. Adoption of these metric equations will lead to more realistic models, greater credibility to models, and an increased likelihood that software practitioners will embrace empirical software engineering.

RELATED RESEARCH

This section provides a general overview of different sampling/resampling techniques, a description of the K-nearest neighbor (KNN) algorithm, and an overview of assessing models for accuracy.

Traditional sampling typically adopts a **random**, **stratified**, **systematic**, or **clustered** approach. The randomized approach, which is the simplest, randomly selects tuples to be in the test set. One way to improve estimates obtained from random sampling is by arranging the population into strata or groups.

A non-class attribute (e.g. age, gender, or income) is used to stratify samples. Systematic sampling orders data by an attribute, then selects every i^{th} element on the list afterwards. Cluster sampling is a method of selecting sampling units in which the unit contains a cluster of elements (Kish 1965). Examples of cluster samples may include all students of one major from all university students, or residents from a particular state.

When extracting samples for a test set, systematic and clustering focus on the class (or dependent) variable. Stratified sampling may use a non-class variable, but typically this is limited to at most one of the non-class variables and does not consider the non-class attribute in light of the class attribute. The problem is that non-class (independent) variables contain a lot of vital information which needs to be considered when partitioning tuples intro training and test sets.

An ***n*-fold cross validation** is a resampling method for validating a model. For this technique, data is partitioned into *n*-classes, and *n* models are constructed with each of the *n*-classes rotated into the test set. *N*-fold cross validation addresses the issue of data distribution between training and test sets, but does not consider difficulty in modeling the training data.

The **K-nearest neighbor** (**KNN**) algorithm is a supervised learning algorithm which classifies a new instance based upon some distance formula (e.g. Euclidean). The new instance is classified to a category relative to some majority of K-nearest neighbors. Traditionally, the KNN algorithm is viewed as a machine learner, rather than a method for dividing training and test data.

Regarding approaches for measuring accuracy, Shepperd et al. (2000) discuss the merits of various methods for measuring accuracy including *TotalError*, *TotalAbsoluteError*, *TotalRelativeError*, *BalancedMMRE, MMRE, Pred(X), Mean Squared Error*, and R^2. There is validity in using one or more

of these accuracy methods. However, they do not provide any information regarding the difficulty in modeling a dataset.

A SIMPLE EXAMPLE

To illustrate why it is important to consider the relationship between non-class attributes over a training and test set, consider the data set in Table 2. It consists of three attributes: Source Lines of Code (**SLOC**); **v(g)**, also known as cyclomatic complexity, which equals the number of decision points within a program plus one; and *defects*. For this example, **defects** refer to the number of software defects present in a software component. If a software component has zero defects, then it is classified as an *A*, otherwise it is classified as a *B*.

Table 2: Simple Dataset

SLOC	v(g)	Defects
81	13	A
87	13	A
182	33	A
193	32	A
53	10	B
58	10	B
140	30	B
150	27	B

Figure 1 plots of these points. The points form 4 clusters of *A*s and *B*s respectively.

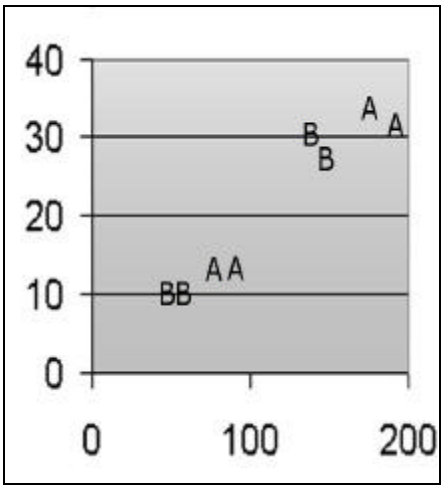

Figure 1: Plot of Tuples from Table 2

If the goal is to build a model for predicting software defects, a systematic sampling approach may be applied which generates the follow training and test sets:

Table 3A: Training Data

SLOC	v(g)	Defects
81	13	A
182	33	A
58	10	B
150	27	B

Table 3B: Test Data

SLOC	v(g)	Defects
87	13	A
193	32	A
53	10	B
140	30	B

Applying a nearest neighbor approach to the non-class attributes, it is clear that the *A*s in the test set match the *A*s in the training set. This is depicted as arrows between Tables 3A and 3B. The same is true for those defects in the *B* class. It is expected that such an experiment would produce very good results irrespective of the machine learner selected. Figure 2 highlights training samples with gray boxes.

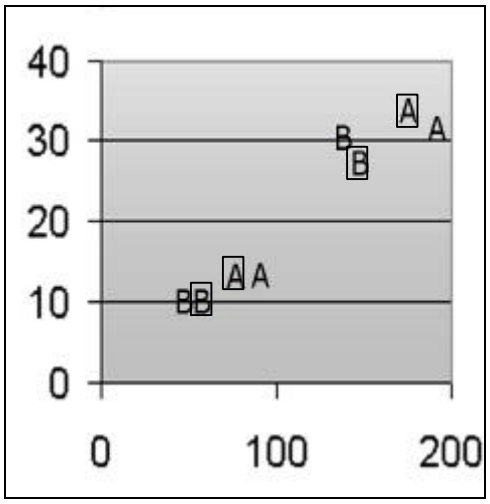

Figure 2: Plot of Training and Test Sets (Square Boxes Are Training Samples)

Suppose a second experiment is conducted using stratified sampling and produces the following training and test sets:

Table 4A: Training Data

SLOC	v(g)	Defects
81	13	A
87	13	A
140	30	B
150	27	B

Table 4B: Test Data

SLOC	v(g)	Defects
182	33	A
193	32	A
53	10	B
58	10	B

Applying a nearest neighbor approach to the non-class attributes, it is clear that the *A*s in the test set match a *B* instance in the training set. Also, the *B*s in the test set match an *A* instance in the training set. It is expected that such an experiment would generate very poor results irrespective of the machine learner selected. Figure 3 highlights training samples with gray boxes.

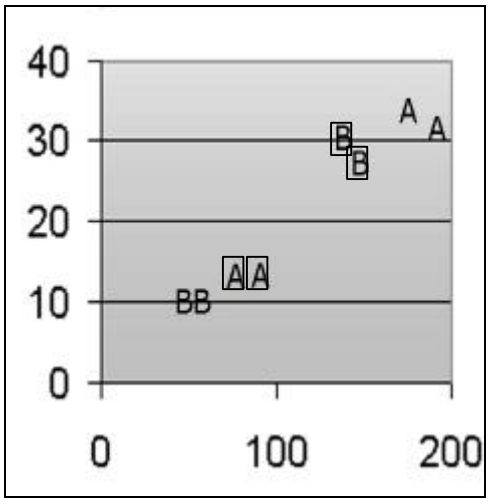

Figure 3: Plot of Training and Test Sets (Square Boxes Are Training Samples)

By ignoring nearest neighbors when building models, success/failure may be a function of the nearest neighbor distribution rather than the capability of the algorithm/machine-learner.

To illustrate the importance of considering nearest neighbor in training/test distribution, a series of experiments are conducted using NASA-based defect data. Although the focus is on defect data, the idea easily extends to other types of software engineering data (e.g. effort estimation data).

NASA DATA SETS

To demonstrate how non-class attributes dramatically impact the modeling process, a series of experiments are conducted against five NASA defect data sets. These 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 are 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.

All experiments use five public domain defect datasets from the NASA Metrics Data Program (MDP) and the PROMISE repository (Shirabad 2005). These five datasets, referred to as CM1, JM1, KC1, KC2,

and PC1, contain static code measures (e.g. Halstead, McCabe, LOC) along with defect rates. Table 5 provides a project description for each of these data sets.

Table 5. 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 (Menzies 2005a).

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 6 shows the general demographics of each of the data sets after pre-processing.

Table 6: 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%

NEAREST NEIGHBOR EXPERIMENTS

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. Using stratified sampling to select 40 percent of the data maintains the ratio between Defect/Non-defect data. As an example, JM1 has 8911 records, 2007 (22.5 percent) of which have 1 or more defects. A corresponding training set for the JM1 project contains 3564 records, 803 (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 (Menzies 2005b). 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 the test record is added to the "Nice Neighbor" test set, otherwise add it to the "Nasty Neighbor" test set. Figure 4 shows the corresponding algorithm.

Essentially, this is the K-Nearest Neighbor algorithm that determines a test tuple's 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 j;
```

Figure 4: Nice/Nasty Neighbor Algorithm

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

After constructing 300 data sets (1 training set and 2 tests sets; 20 trials per software project; 5 software projects), attention focuses on data mining tool selection.

Data Mining Tool Selection

Since the data contains 20-plus attributes and only two class values (TRUE/FALSE), the most reasonable data mining tool in this situation 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 recursively continues on the two or 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 (Witten 2000). WEKA is an open-source machine learning workbench. Implemented in Java, WEKA incorporates many popular machine learners and is widely used for practical work in machine learning. According to a recent KDD poll (KDD 2005), 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. There are reasons for adopting these tools. First, these tools performed very well in more than 1000 data mining experiments conducted by the author. Second, success in using these particular learners was noted Menzies, et al. (Menzies 2005b) 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

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

for all classes c_i and tries to predict the class which has the highest probability. The classifier is considered naïve (Rish 2001) since it assumes that the frequencies of the different attributes are independent.

A second learner, J48, is based on Quinlan's C4.5 (Quinlin 1992).

Assessment Criteria

The assessment criterion uses four 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.
- **NF**, which is the probability of missing an alarm. This is defined as the probability of incorrectly identifying a module where the fault was missed divided by the total number of modules with 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 7.

Table 7. Definition of the Confusion Matrix

	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:

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

PF is defined as:

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

NF is defined as:

$$PF = A / (A + B) \quad (4)$$

and *Acc* is defined as:

$$Acc = (A + D) / (A + B + C + D) \quad (5)$$

Based on the example in Table 7, the corresponding values would be:

$$PD = 50 / (50 + 200) = 20\% \quad (6)$$

$$PF = 100 / (100 + 900) = 10\% \quad (7)$$

$$NF = 200 / (200 + 50) = 80\% \quad (8)$$

$$Acc = (50 + 900) / (50 + 100 + 200 + 900) = 76\% \quad (9)$$

Results

Table 8 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 and how much of the solution space is covered by the JM1 data set, it would seem that a tuple in the test set would have difficulty aligning to a specific tuple in the training set.

Table 8. 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%
Average	**94.4%**	**93.6%**	**18.7%**	**21.2%**

Regarding *PD*, the results as expressed in Table 9 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 9 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 9. 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 10. 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 10. 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 11 and 12. 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 having 1 or more defects in the "Nice" test sets discount the results.

Analyzing the "Nasty" data sets in Tables 11 and 12 reveal that 97.2 percent (e.g. (50 + 249)/(50+249+2+7)) of the data contains 1 or more defects.

Table 11. Confusion Matrix, Nice Test Set (Rounded)

J48		**Naïve Bayes**	
2	3	3	2
58	1021	68	1011

Table 12. 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 6, the percentage of defects to the 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 6). 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 13 and 14 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 13. 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 14. 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 sub-experiments confirm the results achieved in tables 11 and 12.

10-Fold/Duplicates Experiment

The next experiment demonstrates that a *10*-fold cross validation does not provide a total perspective regarding the validation of a dataset.

This experiment uses the five NASA datasets described in the earlier sections. For each dataset, two types of experiments are conducted: the first uses the original dataset (less bad data) with duplicates, and a second where duplicates are removed. Note that each original dataset had only one bad data sample.

Each experiment uses a *10*-fold cross validation for each of the 20 trials. A defect prediction model is constructed based on the C4.5 learner from WEKA (J48) using the default settings.

Table 15 shows the results from these experiments. For all five NASA datasets, the learner produces a better model with the inclusion of duplicates. A t-test shows that these differences are statistically significant for all five NASA datasets.

Table 15. Duplicate/No Duplicate Experimental Results

	Accuracy Average of 29 Runs	
	With Duplicates	**No Duplicates**
CM1	88.07%	87.46%
JM1	79.68%	76.56%
KC1	84.29%	74.03%
KC2	81.65%	76.22%
PC1	93.43%	91.65%

In these experiments, the duplicates are the "nice neighbors" described in the previous set of experiments. Performing a *10*-fold cross validation insures that datasets with duplicates will have a distinct advantage over datasets without duplicates.

DISCUSSION

In general, project managers are reluctant to embrace empirical-based models in their decision-making process. Jørgensen (2004) estimates more than 80 percent of all effort estimation is human-based and only about 4 percent is machine learning-based. If empirical software engineering is going to have any hope of gaining favor with project managers, then it is critical that the modeling process be understood very well.

The first set of experiments shows the extreme range of answers in corresponding best case/worst case scenarios. These experiments clearly indicate how significantly nearest-neighbor sampling influences the results despite the fact that no dataset had any duplicates.

The second set of experiments reflects a more realistic situation where an empirical software engineer may (or may not) include duplicates in the modeling process. The difference in results is statistically significant. For some learners, the issue of duplicates is not a problem. However, as seen with the C4.5 learner, it is a problem.

In order to avoid building artificial models, perhaps the best approach would be to not allow duplicates within datasets. Another attribute could be added to the dataset, *Number of Duplicates*, so that information regarding duplicates is not lost.

This solves the issue of duplicates within datasets, however it does not address the issue regarding the synergy between testing and training datasets (nice versus nasty test sets). The next section addresses this issue.

BETTER CREDIBILITY THROUGH NEAREST NEIGHBOR-BASED SAMPLING

As demonstrated in the first set of experiments, it is evident that nearest neighbor test data distribution dramatically impacts experimental results. The question is *How may nearest neighbor sampling be incorporated into the project development process in order to generate realistic models?*

There are at least two possible solutions: one of which adapts to an organization's current software engineering processes; the second solution offers an alternative process.

In the first approach, a software engineer determines the nearest neighbor for each of the tuples 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*. *Matches* will be used to define a metric called *Experimental Difficulty* (*Exp_Difficulty*) as follows:

$$Exp\_Difficulty = 1 - Matches / Total\_Test\_Instances \quad (10)$$

The *Experimental Difficulty* provides a qualitative assessment of the ease/difficulty for modeling a given data set. Combining this metric with an accuracy metric would offer a more realistic assessment of the 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. For an *n*-fold cross validation, the *Experimental Difficulty* could be calculated for each fold, then averaged over the *n* folds. 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 10, results in the following equation:

$$Overall\_Difficulty = 1 - Matches / Total\_Data\_Instances \quad (11)$$

This gives an idea of the overall difficulty of the data set. A software engineer may partition the data in order to increase (or decrease) *Experimental_Difficulty*. In the context of industrial-based benchmarks, the *Experimental_Difficulty* may be adjusted to coincide to a value adopted by another researcher. This lends greater credibility to comparing experimental results.

Considering an estimated 99 percent of the world's datasets are proprietary, this approach provides an additional benchmark for those models constructed on private datasets. Furthermore, these metric equations provide a means for assessing the robustness of the results.

CONCLUSIONS

Most datasets are proprietary in nature, making it impossible to replicate results in this situation. As demonstrated by the NASA experiments, not all data distributions result in similar results. This work extends previous research in defect prediction (Khoshgoftaar 2003; Porter 1990; Srinivasan 1995; and Tian 1995) by conducting nearest-neighbor analysis for gaining a deeper understanding of how datasets relate to each other, and thus the need for developing more realistic empirical/machine learning-based models. In the first set of NASA experiments, the "Nice" dataset experiments (easy datasets to model) resulted in an average accuracy of 94.0 percent and the "Nasty" dataset experiments (difficult datasets to model) produced an average accuracy of 19.5 percent. These results suggest that success in modeling a training data set may be attributable to the ease/difficulty of the data set, rather than the capability of the machine learner.

Including duplicates within a dataset reduces the difficulty of a dataset since similar tuples may appear in both the training and test sets. This research proposes removing duplicates in order to eliminate any bias. Finally, this work proposes a set of metric equations for measuring the difficulty of a dataset. Benefits of using these metric equations include:

- **The creation of more realistic models.** These metrics will help the Software Engineering community better gauge the robustness of a model.
- **Greater credibility for models based on private datasets.** Since most datasets are proprietary, it is difficult to assess the quality of a model built in this context. Using the proposed metrics will make it easier to compare experiment results when the replication is impossible.
- **Greater chances of adoption by the industrial community.** As mentioned earlier, human-based estimation is still the method of choice. By providing a difficulty metric with a set of results, project managers will be able to assess the goodness of an empirical model. This will make it easier for a

project manager to trust an empirical/machine learning-based model. Thus making it easier for the industrial community to more readily adopt empirical software engineering approaches.

FUTURE DIRECTIONS

This work could be extended from a 2-class to an *n*-class problem. For example, the NASA datasets could be divided into four classes, (0, 1, 2, 3+ defects).

Another common type of Software Engineering dataset estimates programming effort. Thus, a likely future direction would examine these types of datasets.

Finally, it would be interesting to see whether accuracy results could be scaled by the "dataset difficulty metrics" in order to make better comparisons over datasets of varying difficulty.

REFERENCES

Albrecht, A.J. (1979). Measuring Application Development Productivity, In *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*.

Abrecht, A. J., & Gaffney, J.E. (1983). Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation, In *IEEE Transactions on Software Engineering*. 9 (2).

Boehm, B.. (1981). Software Engineering Economics. Englewood Cliffs, NJ : Prentice-Hall. ISBN 0-13-822122-7.

Boehm, B., Abts, C., Brown, A., Chulani, S., Clark, B., Horowitz, E., Madachy, R., Reifer, D., & Steece, B. (2000). *Software Cost Estimation with Cocomo II*. Pearson Publishing.

Boehm, B., & Basili, V. (2001). Software Defect Reduction Top 10 List. *IEEE Computer, IEEE Computer Society*, 34 (1), 135-137.

Dalcher D., & Genus, A. (2003). Avoiding IS/IT Implementation Failure, In *Technology Analysis and Strategic Management, TASM*, 15 (4), 403-407.

Heemstra, F. J. & Kusters, R.J. (1991). Function point analysis: Evaluation of a software cost estimation model, In *European Journal of Information Systems* 1(4), 223-237.

Hihn, J. & Habib-Agahi, H. (1991). Cost estimation of software intensive projects: A survey of current practices, In *International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 276-287.

Jones, C. (1998). *Estimating Software Costs*. McGraw Hill.

Jørgensen, M. (1997). An empirical evaluation of the MkII FPA estimation model, In *Norwegian Informatics Conference*, Voss, Norway, Tapir, Oslo, 7-18.

Jørgensen, M. (2004) A review of studies on Expert Estimation of Software Development Effort, In *Journal of Systems and Software*, 70 (1-2), 37-60.

KDD Nuggets Website, Polls: Data Mining Tools You Regularly Use, Knowledge Discovery and Data Mining Poll. Retrieved 2005, from http://www.kdnuggets.com/polls/data_mining_tools_2002_june2.htm

Khoshgoftaar, T.M. (Ed.). (2003), *Computational Intelligence in Software Engineering, Annals of Software Engineering*, Kluwer Academic Publishers, ISBN 1-4020-7427-1.

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

Kish, L. (1965). *Survey Sampling*. New York: John Wiley and Sons, Inc.

Kitchenham, B., Pfleeger, S.L., McColl, B., & Eagan, S. (2002). A case study of maintenance estimation accuracy, To appear in: *Journal of Systems and Software*.

Menzies, T. (2005a). Personal Conversation.

Menzies, T., Raffo D., Setamanit, S., DiStefano, J., & Chapman, R., (2005b). Why Mine Repositories, Submitted to: *Transactions on Software Engineering*.

Paynter, J., (1996). Project estimation using screenflow engineering, In *International Conference on Software Engineering: Education and Practice*, Dunedin, New Zealand, IEEE Computer Society Press, Los Alamitos, CA, 150-159.

Pedrycz, W. (2002). Computational intelligence as an emerging paradigm of software engineering, In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, Ischia, Italy. ACM, 7-14.

Porter, A.A., & Selby, R.W. (1990). Empirically guided software development using metric-based classification trees, In *IEEE Software*, 46–54.

Putnam, L.H. (1978). A General Empirical Solution to the Macro Software Sizing and Estimating Problem, In *IEEE Transactions on Software Engineering*, 345-361.

Quinlin, J.R. (1992). *C4.5: Programs for machine learning*. California: Morgan Kaufmann.

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

Sauer, C., & Cuthbertson, C. (2003), *The State of IT Project Management in the UK*, Templeton College, Oxford.

Shepperd, M., Cartwright, M., & Kadoda, G. (2000). On Building Prediction Systems for Software Engineers, In *Empirical Software Engineering*, Kluwer Academic Publishers, Boston, 175–182.

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

Sonnekus, R., & Labuschagne, L. (2003). *IT Project Management Maturity versus Project Success in South Africa*. RAU Standard Bank Academy for Information Technology, RAU Auckland Park, Johannesburg, South Africa. ISBN: 0-86970-582-2.

Srinivasan, K., & Fisher, D. (1995). Machine learning approaches to estimating software development effort, In *IEEE Transactions on Software Engineering*, 126–137.

The Standish Group (2003). The Chaos Chronicles III.

Tassey, G. (2002). *The Economic Impacts of Inadequate Infrastructure for Software Testing* (Planning Report 02-3). Prepared by RTI for the National Institute of Standards and Technology (NIST). Retrieved 2005, from http://www.nist.gov/director/prog-ofc/report02-3.pdf

Tian, J., & Zelkowitz, M.V. (1995). Complexity measure evaluation and selection, In *IEEE Transaction on Software Engineering*, 21 (8), 641–649.

Witten, I., & Franks, E. (2000). *Data Mining: Practical machine learning tools with Java implementations*. Morgan Kaufmann, San Francisco.

Zhang, D., & Tsai, J.J.P. (2005). *Machine Learning Applications in Software Engineering*, World Scientific Publishing, ISBN: 981-256-094-7.