Singleton Attribute Ordering Criteria for Greedy Search in Feature Subset Selection

Daniel Baker,Tim Menzies *Member, IEEE*

***Abstract*— The value of using static code attributes to learn defect predictors has been widely debated. Prior work has explored issues like the merits of "McCabes vs Halstead vs lines of code counts" for generating defect predictors. We show here that such debates are irrelevant since *how* the attributes are used to build predictors is much more important than *which* particular attributes are used. Also, contrary to prior pessimism, we show that such defect predictors are demonstrably useful and, on the data studied here, yield predictors with a mean probability of detection of 71% and mean false alarms rates of 25%. These predictors would be useful for prioritizing a resource-bound exploration of code that has yet to be inspected.**

I. Introduction

IN the 21^{st} century it is now practical to quickly apply artificial intelligence tools to discover previously unknown patterns in historical data. In a software engineering project, such *data miners* can learn predictors for software quality from (e.g.) historical logs of defective and non-defective code modules. When budget does not allow for complete testing of an entire system, software managers can use such predictors to focus the testing on parts of the system that seem defect-prone. These potential defect-prone trouble spots can then be examined in more detail by (e.g.) model checking, intensive testing, etc.

The value of using static code attributes to learn defect predictors has been widely debated. Some researchers endorse them; e.g. [**?**], while others vehemently oppose them [**?**], [**?**]. Recently, sharable public-domain defect data sets have become available from the NASA Metrics Data Program (MDP)[1] and the PROMISE repository of software engineering data[2]. These defect data sets can be used to resolve this debate.

It is possible that prior studies may have reached different conclusions since they were based on different data. This potential conflation can now be removed since it is now possible to define a *baseline experiment* using public-domain data sets which different researchers can use to compare their techniques. This paper *defines* and *motivates* such a baseline.

The baseline *definition* draws from standard practices in the data mining community [**?**], [**?**] and addresses some of the drawbacks with certain prior experiments. For example, prominent results in this area have based their conclusion on just one or two data sets; e.g. [?], [?], [?]. Our preferred experimental method, described below, draws its conclusions after hundreds of thousands of calls to a suite of data miners which examine numerous data sets.

To *motivate* others to use our proposed baseline experiment, we must demonstrate that it can yield interesting results. The baseline experiment of this article shows that the rule-based or decision-tree learning methods used in prior work [?], [?], [**?**], [?], [?] are clearly out-performed by a *Näive Bayes* data miner with a *log-filtering* pre-processor on the numeric data (the terms in italics are defined later in this paper).

Further, the experiment can explain *why* our preferred Bayesian method performs best. That explanation is quite technical and comes from information theory. In this introduction, we need only say that the space of "best" predictors is "brittle"; i.e. minor changes in the data (e.g. a slightly different sample used to learn a predictor) can make different attributes appear most useful for defect prediction.

This brittleness result offers a new insight on prior work. Prior results about defect predictors were so contradictory since they were drawn from a large space of competing conclusions with similar, but distinct properties. Different studies could conclude that (e.g.) lines of code are better/worse predictor for defects than the McCabes complexity attribute, just because of small variations to the data. Bayesian methods smooth over the brittleness problem by polling numerous Gaussian approximations to the numerics distributions. Hence, Bayesian methods don't get confused by minor details about candidate predictors.

Our conclusion is that, contrary to prior pessimism [?], [?], data mining static code attributes to learn defect predictors is useful. Given our new results on Näive Bayes and log-filtering, these predictors are much better than previously demonstrated. Also, prior contradictory results on the merits of defect predictors can be explained in terms of the brittleness of the space of "best" predictors. Further, our baseline experiment clearly shows that it is a mis-directed discussion to debate (e.g.) "lines of code vs McCabe" for predicting defects. As we shall see, *the choice of learning method* is far more important than *which subset of the available data* is used for learning.

II. Background

FOR this study, we learn defect predictors from static code attributes defined by McCabe [?] and Halstead [?]. McCabe (and Halstead) are "module"-based metrics where a

Mr. Baker is with the Lane Department of Computer Science, West Virginia University and can be reach at `danielryanbaker@gmail.com`

Dr. Menzies is with the Lane Department of Computer Science, West Virginia University and can be reached at `tim@menzies.us`

This research was conducted with funds from the NASA Software Assurance Research Program led by the NASA IV&V Facility.

Manuscript received January 1, 2006; revised XXX YY, 200Z.

[1] `http://mpd.ivv.nasa.gov`

[2] `http://promise.site.uottawa.ca/SERepository`

module is the smallest unit of functionality[3]. We study defect predictors learned from static code attributes since they are *useful*, *easy to use*, and *widely-used*.

Useful: This paper finds defect predictors with a probability of detection of 71%. This is markedly higher than other currently-used industrial methods such as manual code reviews:

- A panel at *IEEE Metrics 2002* [**?**] concluded that manual software reviews can find ≈60% of defects[4]
- Raffo found that the defect detection capability of industrial review methods can vary from $pd = TR(35, 50, 65)\%$[5]. for full Fagan inspections [**?**] to $pd = TR(13, 21, 30)\%$ for less-structured inspections.

Easy to use: static code attributes (e.g. lines of code, the McCabe/Halstead attributes) can be automatically and cheaply collected, even for very large systems [?]. By contrast, other methods such as manual code reviews are labor-intensive. Depending on the review methods 8 to 20 LOC/minute can be inspected and this effort repeats for all members of the review team, which can be as large as four or six [**?**].

Widely used: Many researchers use static attributes to guide software quality predictions; e.g. [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?]. Verification and validation (V&V) textbooks (e.g. [**?**]) advise using static code complexity attributes to decide which modules are worthy of manual inspections. For several years, the first author worked on-site at the NASA software Independent Verification and Validation facility and he knows of several large government software contractors that won't review software modules *unless* tools like McCabe predict that they are fault prone.

Nevertheless, static code attributes are hardly a complete characterization of the internals of a function. Fenton offers an insightful example where *the same* functionality is achieved using *different* programming language constructs resulting in *different* static measurements for that module [**?**]. Fenton uses this example to argue the uselessness of static code attributes.

An *alternative interpretation* of Fenton's example is that static attributes can never be a certain indicator of the presence of a fault. Nevertheless, they are useful as as probabilistic statements that the frequency of faults tends to increase in code modules that trigger the predictor.

Shepperd & Ince and Fenton & Pfleeger might reject the *alternative interpretation*. They present empirical evidence that the McCabe static attributes offer nothing more than uninformative attributes such as lines of code. Fenton & Pfleeger note that the main McCabe's attribute (cyclomatic complexity, or $v(g)$) is highly correlated with lines of code [?]. Also, Shepperd & Ince remarks that "for a large class of software it (cyclomatic complexity) is no more than a proxy for, and in many cases outperformed by, lines of code" [?].

If Shepperd & Ince and Fenton & Pfleeger. are right, then:

	probability of	
data	detection	false alarm
pima diabetes	60	19
sonar	71	29
horse-colic	71	7
heart-statlog	73	21
rangeseg	76	30
credit rating	88	16
sick	88	1
hepatitis	94	56
vote	95	3
ionosphere	96	18
mean	81	20

Fig. 1. Some representative pds and pfs for prediction problems from the UC Irvine machine learning database [?]. These values were generated using the standard settings of a state-of-art decision tree learner (J48). For each data set, ten experiments where conducted where a decision tree was learned on 90% of the data, then testes of the remaining 10%. The numbers shown here are the average results across ten such experiments.

	probability of	
data	detection	false alarm
pc1	24	25
jm1	25	18
cm1	35	10
kc2	45	15
kc1	50	15
mean	36	17

Fig. 2. Prior results of learning defect predictors. From [?].

- The supposedly better static code attributes such as Halstead and McCabes should perform no better than just simple thresholds on lines of code.
- The performance of a predictor learned learned by a data miner should be very poor;

Neither of these are true, at least for the data sets used in this study. Our experimental method seeks the "best" subsets of the available attributes that are most useful for predicting defects. We will show that the best size for the "best" set is larger than one; i.e. predictors based on single lines of code counts do not perform as well as other methods.

Also, the predictors learned from those "best" sets perform surprisingly well. Formally, learning a defect predictor is a *binary prediction problem* where each modules in a database has been labeled "defect-free" or "defective". The learning problem is to build some predictor which guesses the labels for as-yet-unseen modules. Using the methods described below, this paper offers new defect predictors with a probability of detection (pd) and probability of false alarm (pf) of

$$mean(pd, pf) = (71\%, 25\%)$$

Figure 1 lets us compare our new results against standard binary prediction results from the UC Irvine machine learning repository [?]. Our new results of (pd,pf)=(71%,25%) are close to the standard results of (pd,pf)=(81%,20%). which is noteworthy in three ways:

1) It is unexpected. If static code attributes capture so little about source code (as argued by Shepherd, Ince, Fenton and Pfleeger), then we would expect lower probabilities of defection much higher false alarm rates.
2) These new (pd,pf) figures are much larger than any of our prior results. of $mean(pd, pf) = (36\%, 17\%)$ [?]

[3]In C or Smalltalk, "modules" would be called "function" or "method" respectively.

[4]That panel supported neither Fagan claim [**?**] that inspections can find 95% of defects before testing or Shull's claim that specialized directed inspection methods can catch 35% more defects that other methods [**?**].

[5]$TR(a, b, c)$ is a triangular distribution with min/mode/max of a, b, c.

(see Figure 2). Despite much experimentation [?], [?], the only way we could achieve a $pd > 70\%$ was to accept a 50% false alarm rate.

3) There is still considerable room for improvement; e.g. lower pfs and higher pds. We are actively researching better code metrics which, potentially, will yield "better" predictors.

This third point motivates much of this paper. Before we can demonstrate "better", we need to define "better than what?". That is, improvement can only be measured against a well-defined baseline result. That baseline needs to be repeatable and based on public-domain data set. Further, the basis for comparatively assessing different data mining methods should be well-justified and well-specified so that others can repeat, improve, or refute prior results. Hence, much of the rest of this paper is devoted to a meticulous description of our experimental method.

The baseline experiment was selected in response to certain shortcomings in other work. For example, Nagappan and Ball [?, p6] report accuracies of 82.91% for their defect predictor. Accuracy attributes the number of times the predicted class of a module (defect-free or defective) is the same as the actual class. These accuracy values were found in a *self-test*; i.e. the learned predictor was applied to the data used to train it. In our study, we use neither accuracy nor self-tests:

- When the target class (defect-free or defective) is in the minority, accuracy is a poor measure of a learner's performance. For example, a learner could score 90% accuracy on a data set with 10% defective modules, even if it predicts that all defective modules were defect-free.
- Self-tests are deprecated by the data mining community since such self-tests can grossly over-estimate performance [?]. If the goal is to understand how well a defect predictor will work on future projects, it is best to assess the predictor via *hold out* modules not used in to generate that predictor.

Hence, for this study, we use attributes other than accuracy including pd, pf and several others defined below. Also, our learned predictors will be assessed using *hold out* modules.

III. Threats to Validity

Like any empirical data mining paper, our conclusions are biased according to what data was used to generate them. Issues of *sampling bias* threaten any data mining experiment; i.e. what matters *there* may not be true *here*. For example, the sample used here comes from NASA, which works in a unique market niche.

Nevertheless, we argue that results from NASA are relevant to the general software engineering industry. NASA makes extensive use of contractors who are contractually obliged (ISO-9O01) to demonstrate their understanding and usage of current industrial best practices. These contractors service many other industries; e.g. Rockwell-Collins builds systems for many government and commercial organizations. For these reasons, other noted researchers such as Basili, Zelbowitz, et al. [**?**] have argued that conclusions from NASA data are relevant to the general software engineering industry.

All inductive generalization suffers from a sampling bias. The best we can do is define our methods and publicize our data such that other researchers can try to repeat our results and, perhaps, point out a previously unknown bias in our analysis. Hopefully, other researchers will emulate our methods in order to repeat, refute or improve our results. We would encourage such researchers to offer not just their conclusions, but the data used to generate those conclusions. The MDP is a repository for NASA data sets and the PROMISE code repository are places to store and discuss software engineering data sets from other organizations.

Another source of bias in this study is the set of learners explored by this study. Data mining is a large and active field and any single study can only use a small subset of the known data mining algorithms. For example, neural networks [**?**] and genetic algorithms [**?**] were not used for this study as they can be very slow. The experiment described in this paper took weeks to debug and a full day to run once debugged. We were therefore not motivated to explore other, slower, learners but would encourage other researchers with access to super-computers or a large CPU-farm to do so.

IV. Data

An experiment needs three things:

- Data to be processed;
- A processing method;
- A reporting method;

This section discusses the data used in this study. Processing via data miners and our reporting methods are discussed later.

All our data comes from the MDP. At the time of this writing, ten data sets are available in that repository. Two of those data sets have a different format to the rest and were not used in this study. This left eight, shown in Figure 3. Each module of each data sets describes the attributes of that module, plus the number of defects known for that module. This data comes from eight sub-systems taken from four systems. These systems were developed in different geographical locations across North America. Within a system, the sub-systems shared some a common code base but did not pass personnel or code between sub-systems. Figure 4 shows the module sizes of our data; e.g. there are 126 modules in the $kc4$ data set, most of them are under 100 lines of code, but a few of them are over 1000 lines of code long.

Each data set was pre-processed by removing the module identifier attribute (which is different for each row). Also, the $error\_count$ column was converted into a boolean attribute called *defective?* as follows: $defective? = (error\_count \geq 1)$. Finally, the $error\_density$ column was removed (since it can be derived from line counts and $error\_count$). The pre-processed data sets had 38 attributes, plus one target attribute (*defective?*) shown in Figure 5 and include Halstead, McCabe, lines of code, and other miscellaneous attributes.

The Halstead attributes were derived by Maurice Halstead in 1977. He argued that modules that are hard to read are more likely to be fault prone [?]. Halstead estimates reading complexity by counting the number of operators and operands in a module: see the h attributes of Figure 5. These three raw

system	language	sub-system data set	total LOC	# modules	% defective
spacecraft instrument	C	cm1-05	17K	506	9
storage management for ground data	Java	kc3	8K	459	9
		kc4	25K	126	49
Db	C	mw1	8K	404	7
Flight software for earth orbiting satellite	C	pc1-05	26K	1,108	6
		pc2	25K	5,590	0.4
		pc3	36K	1,564	10
		pc4	30K	1,458	12

Fig. 3. Data sets used in this study. The datasets cm1-05 and pc1-05 updates data sets $cm1$ and $pc1$ processed previously by the author; e.g. in [?].

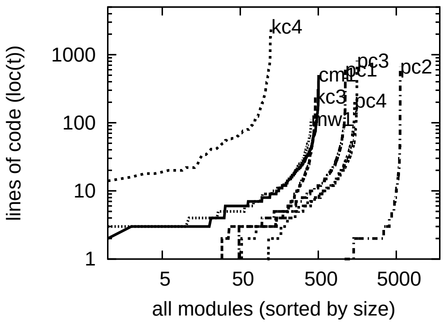


Fig. 4. Log-log plot of module sizes in the Figure 3 data.

most	m = Mccabe		$v(g)$	cyclomatic_complexity
			$iv(G)$	design_complexity
			$ev(G)$	essential_complexity
	locs	loc		loc_total (one line = one count
		loc(other)		loc_blank
				loc_code_and_comment
				loc_comments
				loc_executable
				number_of_lines (opening to closing brackets)
	Halstead	h	N_1	num_operators
			N_2	num_operands
			μ_1	num_unique_operators
			μ_2	num_unique_operands
		H	N	length: $N = N_1 + N_2$
			V	volume: $V = N * log_2 \mu$
			L	level: $L = V^*/V$ where $V^* = (2 + \mu_2{}^*) log_2 (2 + \mu_2{}^*)$
			D	difficulty: $D = 1/L$
			I	content: $I = \hat{L} * V$ where $\hat{L} = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$
			E	effort: $E = V/\hat{L}$
			B	error_est
			T	prog_time: $T = E/18$ seconds
	misc = Miscellaneous			branch_count
				call_pairs
				condition_count
				decision_count
				decision_density
				design_density
				edge_count
				global_data_complexity
				global_data_density
				maintenance_severity
				modified_condition_count
				multiple_condition_count
				node_count
				normalized_cyclomatic_complexity
				parameter_count
				pathological_complexity
				percent_comments

Fig. 5. Attributes used in this study.

h Halstead attributes were then used to compute the H: the eight derived Halstead attributes using the equations shown in Figure 5. In between the raw and derived Halstead attributes are certain intermediaries (which don't appear in the MDP data sets);

- $\mu = \mu_1 + \mu_2$;
- minimum operator count: $\mu_1^* = 2$;
- μ_2^* is the minimum operand count and equals the number of module parameters.

An alternative to the Halstead attributes are the complexity attributes proposed by Thomas McCabe in 1976. Unlike Halstead, McCabe argued that the complexity of pathways *between* module symbols are more insightful than just a count of the symbols [?]. The first three lines of Figure 5 shows McCabe three main attributes for this pathway complexity. These are defined as follows. A module is said to have a *flow graph*; i.e. a directed graph where each node corresponds to a program statement, and each arc indicates the flow of control from one statement to another. The *cyclomatic complexity* of a module is $v(G) = e - n + 2$ where G is a program's flow graph, e is the number of arcs in the flow graph, and n is the number of nodes in the flow graph [**?**]. The *essential complexity*, ($ev(G)$) or a module is the extent to which a flow graph can be "reduced" by decomposing all the subflowgraphs of G that are *D-structured primes* (also sometimes referred to as "proper one-entry one-exit subflowgraphs" [?]). $ev(G) = v(G) - m$ where m is the number of subflowgraphs of G that are D-structured primes [?]. Finally, the *design complexity* ($iv(G)$) of a module is the cyclomatic complexity of a module's reduced flow graph.

At the end of Figure 5 are a set of *misc* attributes that are less well-defined than lines of code attributes or the Halstead and McCabe attributes. The meaning of these attributes is poorly documented in the MDP database. Indeed, they seem to be values generated from some unknown tool set that was available at the time of uploading the data into the MDP. Since there are difficulties in reproducing these attributes at other sites, an argument could be made for removing them from this study. A counter-argument is that if static code attributes are as weak as suggested by Shepherd, Ince, Fenton and Pfleeger then we should use all possible attributes in order to make maximum use of the available information. This study took a middle ground: all these attributes were be passed to the learners and they determined which ones had most information.

An interesting repeated pattern in our data sets are *exponential distributions* in the numeric attributes. For example,

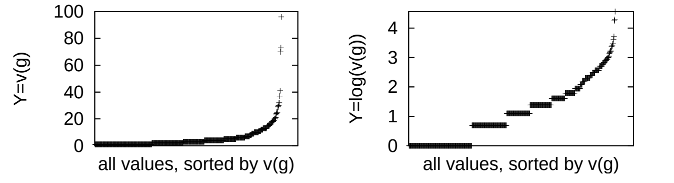


Fig. 6. A McCabe's metric from $cm1$: raw values on left, log-filtered on right.

Figure 6, on the left-hand-side, shows the sorted McCabe $v(g)$ attributes from *cm1*. These values form an exponential distribution with many small values and a few much larger values. Elsewhere, we have conducted limited experiments suggesting that a *logarithmic filter* on all numeric values might improve predictor performance [?]. Such a filter replaces all numerics n were their logarithm. $ln(n)$. The effects of such a filter are shown on the right-hand-side of Figure 6: the log-filtered values are now more evenly spread across the y-range, making it easier to reason about them. To test the value of log-filtering, all the data was passed through one of two filters:

1) *none*; i.e. no change;
2) *logNums*; i.e. logarithmic filtering. To avoid numerical errors with $ln(0)$, all numbers under 0.000001 are replaced with $ln(0.000001)$.

V. Learners

The above data was passed to three learners from the WEKA data mining toolkit [?]: OneR, J48, and Näive Bayes[6]. This section describes those learners.

Holte's OneR algorithm [**?**]. builds prediction rules using one or more values from a single attribute attribute. For example, OneR executing on the *kc4* data set can return:

```
EDGE_COUNT:
< 2.99 -> defect-free
>= 2.99 -> defective
```

which may be read as follows: "a module is defect-free if its edge count is less than 2.99".

OneR was chosen to test the value of predictors based on *simple thresholds on single attributes*. For an example of such a simple threshold, recall that McCabe recommends inspected modules that satisfy: $v(g) > 10$ or $iv(g) > 4$. Several other example thresholds exist in defect prediction literature:

- Chapman and Solomon advocate predicting defects using $v(g) > 20$ or $ev(g) > 8$ [?];
- In early work we advocated $ev(g) > 7$ or lines of code $>$ 118 [?] (based on this study, we now reject that old advice).

OneR can only return simple thresholds on single attributes. If predictors built by OneR were as good as any other, then that would support the use of simple thresholds such as those advocated by McCabe, Chapman, *et al.*.

One way to view OneR's defect predictions rules is a decision tree of maximum depth one whose leaves are either the label *defective* or *defect-free*. The J48 learner builds decision trees of any depth. For example, J48 executing on the *kc4* data set can return:

```
CALL_PAIRS <= 0: defect-free
CALL_PAIRS > 0
|  NUMBER_OF_LINES <= 3.12: defect-free
|  NUMBER_OF_LINES > 3.12
|  |  NORMALIZED_CYLOMATIC_COMPLEXITY <= 0.02
|  |  |   NODE_COUNT <= 3.47: defective
|  |  |   NODE_COUNT > 3.47: defect-free
|  |  NORMALIZED_CYLOMATIC_COMPLEXITY>0.02: defective
```

which may be read as follows: "a module is defective if it has non-zero call-pairs and has more than 3.12 lines and hasn't a low normalized cyclomatic complexity (0.02) or it has a low normalized cyclomatic complexity and a low node-count (up to 3.47)".

Note that J48 predictors can be more complex and explore more special cases than OneR. J48's predictors would outperform OneR if defect prediction required such elaboration.

J48 is a JAVA implementation of Quinlan's C4.5 (version 8) algorithm [**?**]. The algorithm recursively *splits* a data set according to tests on attribute values in order to separate the possible predictions (although attribute tests are chosen one at a time in a greedy manner, they are dependent on results of previous tests). C4.5/J48 uses information theory to assess candidate splits: the *best split* is the one that *most simplifies* the target concept. Concept simplicity is measured using information theory. Suppose a data set has 80% defect-free modules and 20% defective modules. Then that data set has a class distribution C_0 with classes $c(1) = defect-free$ and $c(2) = defective$ with frequencies $n(1) = 0.8$ and $n(2) = 0.2$. The number of bits required to encode an arbitrary class distribution C_0 is $H(C_0)$ defined as follows:

$$\left.\begin{array}{rcl} N &=& \sum_{c \in C} n(c) \\ p(c) &=& n(c)/N \\ H(C) &=& -\sum_{c \in C} p(c) log_2 p(c) \end{array}\right\} \quad (1)$$

A split divides C_0 (before the split) into C_1 and C_2 (after the split). The best split leads to the simplest concepts; i.e. maximize $H(C_0) - (H(C_1) + H(C_2))$.

Another way to build defect predictors is to use a Näive Bayes data miner. Such data miners employ a simplified version of Bayes theorem to predict if a module is defective or not. The posterior probability of each class ("defective" or "defect-free") is calculated, given the attributes extracted from a module; e.g. lines of code, the McCabe attributes, the Halstead attributes, etc. The module is assigned to the possibility with the highest probability. This is straightforward processing and involves simply estimating the probability of attribute measurements within the historical modules. Simple frequency counts are used to estimate the probability of discrete attribute attributes. For numeric attributes it is common practice to use the probability density function for a normal distribution [?]:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where $\{\mu, \sigma\}$ are the attribute {mean,standard deviation}. To be precise, the probability of a continuous attribute being a

[6]Available from `http://www.cs.waikato.ac.nz/~ml/weka/index_downloading.html`.

particular continuous value x is zero, but the probability that it lies within a small region, say $x \pm \epsilon/2$, is $\epsilon \times f(x)$. Since ϵ is a constant that weighs across all possibilities, it cancels out and needs not to be computed.

The above learning technology can be used to generate defect predictors from data *or* to assess the value of different portions of the data. Various *attribute subset selection* algorithms [?] (hereafter, *subsetting*) find what attributes can be deleted, without damaging the performance of the learned predictor. Subsetting can be used independently of the learning technique of choice, as a general method for data reduction.

The simplest and fastest subsetting method is to rank attributes from the most informative to least informative. After discretizing numeric data[7] then if A is a set of attributes, the number of bits required to encode a class after observing an attribute is:

$$H(C|A) = -\sum_{a \in A} p(a) \sum_{c \in C} p(c|a) log_2(p(c|a)$$

The highest ranked attribute A_i is the one with the largest *information gain*; i.e the one that most reduces the encoding required for the data *after* using that attribute; i.e.

$$InfoGain(A_i) = H(C) - H(C|A_i) \qquad (2)$$

where $H(C)$ comes from Equation 1. In *iterative InfoGain subsetting*, predictors are learned using the $i = 1, 2..., N$-th top-ranked attributes. Subsetting terminates when $i + 1$ attributes perform no better than i. In *exhaustive InfoGain subsetting*, the attributes are first ranked using iterative subsetting. Next, predictors are built using all subsets of the top j ranked attributes. For both iterative and exhaustive subsetting, the process is repeated 10 times using 90% of the data (randomly selected). Iterative subsetting takes time linear on the number of attributes N while exhaustive subsetting takes time 2^j (so it is only practical for small $j \leq N$).

VI. Experimental Design

This study used the *(M=10)*(N=10)-way cross-evaluation iterative attribute subset selection* shown in Figure 7. The study is nearly the same as the procedure defined in Hall and Holmes' subsetting experiments [?] (but we have added a data filtering step). The data set is divided into N buckets. For each bucket in a 10-way cross-evaluation, a predictor is learned on the nine of the buckets, then tested on the remaining bucket.

Hall and Holmes advise repeating an N-way study M times, randomizing the order each time. Many algorithms exhibit *order effects* where certain orderings dramatically improve or degrade performance [**?**] (e.g. insertion sort runs slowest if the inputs are already sorted in reverse order). Randomizing the order of the inputs defends against order effects.

These M*N studies implements a *hold out* study which, as argued above, is necessary to properly assess the value of learned predictor. Hold out studies are the referred evaluation method when the goal is to produce predictors intended to predict future events [?].

[7]E.g. given an attribute's minimum and maximum values, replace a particular value n with $(n - min)/((max - min)/10)$. For more on discretization, see [**?**].

```
M    = 10
N    = 10
All  = 38  # all the attributes
DATAS=(cm1 kc3 kc4 mw1 pc1 pc2 pc3 pc4) # data set list
FILTERS= (none logNums)                 # filter list
LEARNERS= (oneR j48 nb)                 # learner list

 for data  in DATAS
   for filter  in FILTERS
      data' = filter(data)
      rank data' attributes via InfoGain # Equation 2
       for i = 1,2,3, All
         attribute' = the i-th highest ranked attributes
         data''     = select attributes' from data'
        repeat M times
            randomized order from data''
            generate N bins from data''
           for i  in 1  to N
              tests        = bin[i]
              trainingData = data'' - tests
             for learner  in LEARNERS
                METHOD         = (filter attributes' learner)
                predictor      = learner(trainingData)
                RESULT[METHOD] = apply predictor to tests
```

Fig. 7. This study. Data is filtered and the attributes are ranked using InfoGain. The data is then shuffled into a random order and divided into 10 bins. A learner is then applied to a *training set* built from nine of the bins. The learned predictor is test on the remaining bin.

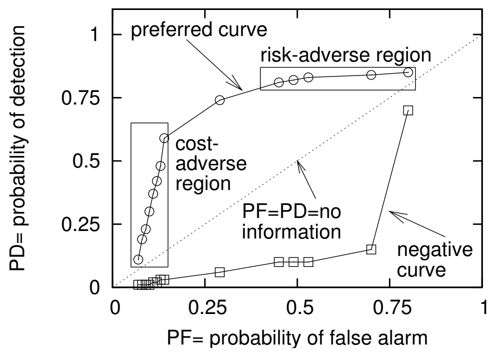


Fig. 8. Regions of a typical ROC curve.

The 10*10-way study was wrapped inside scripts that explored different subsets of the attributes in the order suggested by *InfoGain* (Equation 2). In the inner-most loop of the study, some *method* was applied to some data set. As shown in the third last line of Figure 7, these *methods* were some combination of *filter, attributes', learner*.

VII. Assessing Performance

The performance of the learners on the MDP data was assessed using *receiver-operator* (ROC) curves. Formally, a defect *predictor* hunts for a *signal* that a software module is defect prone. Signal detection theory [**?**] offers ROC curves as an analysis method for assessing different predictors. A typical ROC curve is shown in Figure 8. The y-axis shows probability of detection (pd) and the x-axis shows probability of false alarms (pf). By definition, the ROC curve must pass through the points $pf = pd = 0$ and $pf = pd = 1$ (a predictor that never triggers never makes false alarms; a predictor that always triggers always generates false alarms). Three interesting trajectories connect these points:

1) A straight line from (0,0) to (1,1) is of little interest

		module found in defect logs?	
		no	yes
signal detected?	no (i.e. $v(g) < 10$)	A = 395	B = 67
	yes (i.e. $v(g) \geq 10$)	C = 19	D = 39

$pd =$	$Prop.detected =$	37%
$pf =$	$Prob.falseAlarm =$	5%
$notPf =$	$1 - pf =$	95%
$bal =$	$Balance =$	45%
$Acc =$	$accuracy =$	83%

Fig. 9. A ROC sheet assessing the predictor $v(g) \geq 10$. Each cell {A,B,C,D} shows the number of modules that fall into each cell of this ROC sheet. The bal (or balance) variable is defined below.

since it offers *no information*: i.e. the probability of a predictor firing is the same as it being silent.

2) Another trajectory is the *negative curve* that bends away from the ideal point. Elsewhere [?], we have found that if predictors negate their tests, the negative curve will transpose into a *preferred curve*.
3) The point ($pf = 0$, $pd = 1$) is the ideal position (a.k.a. "sweet spot") on a ROC curve. This is where we recognize all errors and never make mistakes. *Preferred curves* bend up towards this ideal point.

In the ideal case, a predictor has a high probability of detecting a genuine fault (pd) and a very low probability of false alarm (pf). This ideal case is very rare. The only way to achieve high probabilities of detection is to trigger the predictor more often. This, in turn, incurs the cost of more false alarms.

Pf and pd can be calculated using the ROC sheet of Figure 9. Consider a predictor which, when presented with some signal, either triggers or is silent. If some oracle knows whether or not the signal is actually present, then Figure 9 shows four interesting situations. The predictor may be silent when the signal is absent (cell A) or present (cell B). Alternatively, if the predictor registers a signal, sometimes the signal is actually absent (cell C) and sometimes it is present (cell D).

If the predictor registers a signal, there are three cases of interest. In one case, the predictor has correctly recognized the signal. This probability of this detection is the ratio of detected signals, true positives, to all signals:

$$\text{probability detection} = pd = recall = D/(B+D) \tag{3}$$

(Note that pd is also called $recall$.) In another case, the probability of a false alarm is the ratio of detections when no signal was present to all non-signals:

$$\text{probability false alarm} = pf = C/(A+C) \tag{4}$$

For convenience, we say that $notPf$ is the complement of pf:

$$\text{notPf} = 1 - C/(A+C) \tag{5}$$

Figure 9 also lets us define the *accuracy*, or acc, of a predictor as the percentage of true negatives and true positives:

$$\text{accuracy} = acc = (A+D)/(A+B+C+D) \tag{6}$$

If reported as percentages, these attributes have the range:

$$0 \leq\ acc\%,\ pd\%,\ ,notPf\% \leq 100$$

Ideally, we seek predictors that maximize *acc%, pd%, notPf%*.

Note that maximizing any one of these does not imply high values for the others. For example Figure 9 shows an example with a high accuracy (83%) but a low probability of detection (37%). Accuracy is a good measure of a learner's performance when the possible outcomes occur with similar frequencies. The data sets used in this study, however, have very uneven class distributions (see Figure 3). Therefore this paper will assess its learned predictors using $bal, pd, notPf$ and not acc.

In practice, engineers *balance* between pf and pd. To operationalize this notion of *balance*, we define bal to be the Euclidean distance from the sweet spot $pf = 0, pd = 1$ to a pair of $< pf, pd >$. For convenience, we (a) normalize bal by the maximum possible distance across the ROC square ($\sqrt{2}$); (b) subtract this from 1; and (c) express it as a percentage; i.e.

$$balance = bal = 1 - \frac{\sqrt{(0-pf)^2 + (1-pd)^2}}{\sqrt{2}} \tag{7}$$

Hence, better and *higher* balances fall *closer* to the desired sweet spot of $pf = 0, pd = 1$.

VIII. Quartile Charts of Performance Deltas

Recall from Figure 7 that a *method* is some combination of $filter, attributes', learner$. This experiment generated nearly 800,000 *performance deltas* (defined below) comparing $pd, notPf, bal$ values from different *methods* applied to the same $test$ data.

The performance deltas were computed using simple subtraction, defined as follows. A *positive performance delta* for method X means that method X has out-performed some other method in *one* comparison. Using performance deltas, we say that the best method is the one that generates the largest performance deltas over *all* comparisons.

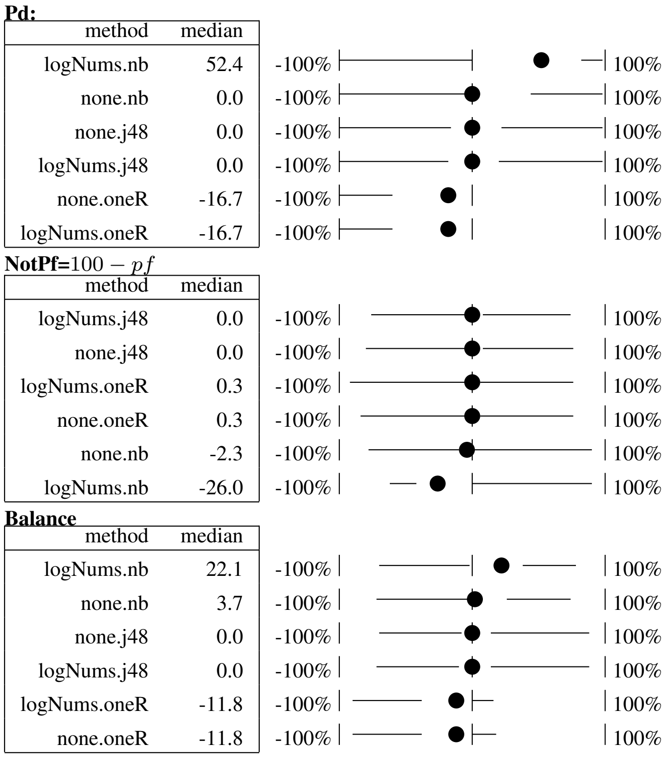
The performance deltas for each method were sorted and displayed as *quartile charts*. To generate these charts, the performance deltas for some $method$ were sorted to find the lowest and highest quartile as well as the median value; e.g.

$$\overbrace{\underbrace{-59}_{min},-19,-19}^{lowest\ quartile}, -16, -14, -10, \underbrace{-10}_{median}, 5, 14, 39, \overbrace{42, 62, \underbrace{69}_{max}}^{highest\ quartile}$$

In a quartile chart, the upper and lower quartiles are marked with black lines; the median is marked with a black dot; and vertical bars are added to mark (i) the zero point and (ii) the minimum possible value and (iii) the maximum possible value (in our case, -100% and 100%). The above numbers would therefore be drawn as follows:

$$-100\% \mid \quad — \quad \bullet\mid \qquad —— \mid 100\%$$

We prefer quartile charts of performance deltas to other summarization methods for M*N studies. Firstly, they offer a very succinct summary of a large number of experiments. For example, Figure 10 display 200,000 performance deltas in $\frac{1}{4}$ of a page. Secondly, they are a *non-parametric* display; i.e. they make no assumptions about the underling distribution. Standard practice in data mining is to compare the mean



Pd:

method	median
logNums.nb	52.4
none.nb	0.0
none.j48	0.0
logNums.j48	0.0
none.oneR	-16.7
logNums.oneR	-16.7

NotPf=$100 - pf$

method	median
logNums.j48	0.0
none.j48	0.0
logNums.oneR	0.3
none.oneR	0.3
none.nb	-2.3
logNums.nb	-26.0

Balance

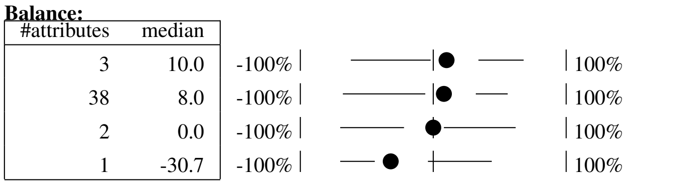
method	median
logNums.nb	22.1
none.nb	3.7
none.j48	0.0
logNums.j48	0.0
logNums.oneR	-11.8
none.oneR	-11.8

Fig. 10. Performance deltas for $pd, notPf, bal$ using all 38 attributes.
.

performance of different methods using t-test [?]. T-tests are a *parametric method* that assume that the underling population distribution is a Gaussian. Recent results suggest that there are many statistical issues left to explore regarding how to best to apply those t-tests for summarizing M*N-way studies [**?**]. Such t-tests assume Gaussian distributions and some of our results are clearly non-Gaussian:

- The Näive Bayes performance delta pd results (using $logNums$) of Figure 10 exhibits an extreme skewness (a median point at 52.4 with a quarter of the performance deltas pushed up towards the maximum figure of 100%).
- All the OneR performance delta pd results of Figure 10 are highly skewed. OneR's pd performance delta was *never* higher than 16.7 and over half the performance deltas for that method had that value (hence, the missing upper arms in the OneR results of Figure 10).

For the sake of completeness, we applied t-tests when sorting quartile charts: one quartile chart appears above its neighbor if it was statistically different (at the 95% confidence level) and has a larger mean. However, given the skews we are seeing in the data, we base our conclusions on *stand-out* effects seen in the non-parametric quartile diagrams. A *stand-out* effect is a large and positive median with a highest quartile bunched up towards the maximum figure. The pd results for Näive Bayes (with $logNums$) are an example of such a stand-out effect. On the other hand, OneR's pd results are a *negative stand-out*: those performance deltas tend to bunch down towards -100%; i.e. in terms of pd, OneR usually performs much worse than anything else.



Balance:

#attributes	median
3	10.0
38	8.0
2	0.0
1	-30.7

Fig. 11. On balance performance deltas of Näive Bayes (with logNums) using just the best 1,2, or 3 attributes, or all 38 attributes.

IX. Results

Näive Bayes with a log-transform has both a positive stand-out result for pd and a negative stand-out result for $notPf$. This result, of winning on pd, but losing on pf, is to be expected. Figure 8 showed that the cost of high pds are higher pfs. The other learning methods cannot emulate the high pds of Näive Bayes (with log-transforms) since they take less chances (hence, have lower false alarm rates).

The *balance* results of Figure 10 combines the pd and pf results, using Equation 7. On balance, with 38 attributes:

- OneR loses more often than it wins: observe that OneR has a negative median $balance$.
- The best method, on balance, is clearly Näive Bayes with log-transforms since it has a minority of negative balance performance deltas (only 25%); and it beats other methods by 22.1% (or more) half the time.

A review of the J48 and OneR quartile charts in the Figure 10 shows that J48 out-performs OneR in terms of pd and $notPf$ and bal. That is, for these data sets, predictors that use simple threshold comparisons (e.g. OneR) perform worse that predictors built from more elaborate decision trees (e.g. J48).

Since, on balance *logNums.nb* performs best, the rest of this article only presents the subsetting results for that method. Initial experiments with iterative InfoGain subsetting showed that all but one of the data sets (pc1) could be reduced from 38 to three attributes without degrading the on-balance performance. However, iterative subsetting selected seven attributes for $PC1$. Therefore, for that data set only, exhaustive subsetting was performed on 2^7 subsets to find the three best attributes.

These InfoGain results were then compared to various other subsetting methods: CFS [**?**]; Relief [**?**], [**?**]; and CBS [**?**]. Measured in terms of pd or $nofPf$ or $balance$, or number of selected attributes, there was no apparent advantage in using these other subsetting methods instead of InfoGain

Figure 11 shows the InfoGain results for Näive Bayes with logNums. On balance, large reductions in the number of attributes are possible, without compromising the performance of the learned predictor. Using two or three attributes worked as well as using 38 attributes. However, using only one attribute resulted in inferior performance.

All the results up to this point have been *comparisons between* different methods. Having determined that Näive Bayes (with logNums) is our preferred method, the next question is how well does that method perform in *absolute* terms. To test that, in Figure 12, a standard 10*10-way experiment with attribute subset-selection was performed (hence, each line in

		%		selected attributes	selection
data	N	pd	pf	(seeFigure 13)	method
pc1	100	48	17	3, 35, 37	exhaustive subsetting
mw1	100	52	15	23, 31, 35	iterative subsetting
kc3	100	69	28	16, 24, 26	iterative subsetting
cm1	100	71	27	5, 35, 36	iterative subsetting
pc2	100	72	14	5, 39	iterative subsetting
kc4	100	79	32	3, 13, 31	iterative subsetting
pc3	100	80	35	1, 20, 37	iterative subsetting
pc4	100	98	29	1, 4, 39	iterative subsetting
all	800	71	25		

Fig. 12. Best defect predictors learned in this study. Mean results from Näive Bayes after a 10 repeats of (i) randomize the order of the data; (ii) divide that data into ten 90%:10% splits for training:test. Prior to learning, all numerics where replaced with logarithms. InfoGain was then used to select the best two or three attributes shown in the right-hand column (and if "three" performed as well as "two", then this table shows the results using "two").

ID	frequency in Figure 12	what	type
1	2	loc_blanks	locs
3	2	call_pairs	misc
4	1	loc_code_and_command	locs
5	2	loc_comments	locs
13	1	edge_count	misc
16	1	loc_executable	locs
20	1	I	H (derived Halstead)
23	1	B	H (derived Halstead)
24	1	L	H (derived Halstead)
26	1	T	H (derived Halstead)
31	2	node_count	misc
35	3	μ_2	h (raw Halstead)
36	1	μ_1	h (raw Halstead)
37	2	number_of_lines	locs
39	2	percent_comments	misc

Fig. 13. Attributes used in Figure 12, sorted into the groups of Figure 5.

Figure 12 shows the results of 10*10=100 experiments on the smallest useful attribute subset). On average, Näive Bayes (with logNums) built predictors with mean $pd = 71\%$, and mean $pf = 25\%$.

The Figure 12 results are better than they first appear:

- Recall from Figure 3 that the number of defective modules may be very small: the most extreme example of this is $PC2$ with only 0.4% defective modules. It is somewhat of an achievement that, for $PC2$, our methods yielded $\{pd = 72\%, pf = 14\%\}$ for such a tiny target.
- The best we have achieved in the past with cross-validation was a mean pd was under 50% [**?**] (recall Figure 2). In those experiments, the only way to achieve a $pd > 70\%$ was to accept aore50% false alarm rate [?], [?] The results of Figure 11 results have much higher pds and lower pfs.

One interesting aspect of Figure 12 is that different data sets selected very different "best" attributes (see the *selected attribute* column). This aspect can be explained by Figure 14 which shows the InfoGain of all the attributes in an MDP data set (KC3). Note how the highest ranked attributes (those on the left-hand-side) offer very similar information. That is, there are no clear winners so minor changes in the training sample (e.g. the 90% sub-sampling used in subsetting or a cross-validation study) can result in the selection of very different "best" attributes.

The pattern of InfoGain values of Figure 14 (where there

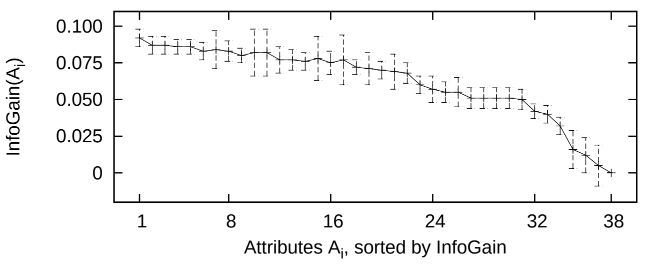


Fig. 14. InfoGain for KC3 attributes. Calculated from Equation 2. Lines show means and t-bars show standard deviations after 10 trials on 90% of the training data (randomly selected).

are many alternative "best" attributes) repeats in all the MDP data sets. This pattern explains a prior observation of Shepperd & Ince who found 18 publications where an equal number of studies reporting that the McCabe cyclomatic complexity is the same; is better; or is worse than lines of code in predicting defects [?]. Figure 14 motivates the following principles:

- Don't seek "best" subsets of static code attributes.
- Rather, seek instead for learning methods that can combine multiple partial defect indicators (e.g. the statistical methods of Näive Bayes).

X. Conclusion

These results strongly endorse building defect predictors using Näive Bayes (with logNums). The combination of learner+filter generated predictors with average results of $pd = 71\%$ and $pf = 25\%$ (see Figure 12). This is an interesting result since, as mentioned above, if static code attributes capture so little about source code (as argued by Shepherd, Ince, Fenton and Pfleeger), then we would expect much lower probabilities of defection and much higher false alarm rates.

Our results also comment on the relative merits of certain learners. Based on these experiments, we would reject the use of simple thresholds for defect prediction. If simple thresholds such as $v(g) > 10 \vee iv(g) > 4$ were the best defect predictors, then two results would be predicted. Firstly, the single attribute tests of OneR would perform as well as the multiple tests of J48. Secondly, the subsetting methods would select attribute sets of size one. Neither of these results were seen in Figure 10 and Figure 11.

This experiment was also negative regarding the merits of building intricate decision trees to predict defects. Recalling Figure 10, Näive Bayes (with logNums) out-performed J48. We offer two explanations why Näive Bayes with logNums out-performs our prior work:

- Recalling Figure 6, it is possible that code defects are actually associated in some log-normal way to static code attributes. Of all the methods studied here, only Näive Bayes (with $logNums$) was able to directly exploit this association.
- Recalling Figure 14, many of the static code attributes have similar information content. Perhaps defect detection is best implemented as some kind of thresholding systems; i.e. by summing the signal from several partial

indicators. Of all the learners used in this study, only the statistical approach of Näive Bayes can sum information from multiple attributes.

The best attributes to use for defect prediction vary from data set to data set. Hence, rather than advocating a particular subset of possible attributes as being the *best attributes*, these experiments suggest that defect predictors should be built using *all available* attributes, followed by subsetting to find the most appropriate particular subset for a particular domain.

In summary we endorse the use of static code attributes from predicting detects with the following caveat. Those predictors should be treated as probabilistic, not categorical, indicators. While our best methods have a non-zero false alarm, they also have a usefully high probability of detection (over $\frac{2}{3}$rds). Just as long as users treat these predictors as *indicators* and not definite *oracles*, then the predictors learned here would be pragmatically useful for (e.g.) focusing limited verification and validation budgets on portions of the code base that are predicted to be problematic.

Since we are optimistic about using static code attributes, we need to explain prior pessimism about such attributes (e.g. [?], [?]):

- Prior work would not have found good predictors if that work had focused on attribute subsets, rather than the learning methods. Figure 12 shows that the best attribute subsets for defects predictors can change dramatically from data set to data set. Hence, conclusions regarding the *best attribute(s)* are very brittle; i.e. may not still apply when we change data sets.
- Also, prior work would not have found good predictors if that work had not explored a large space of learning methods. It took much searching for this study to find a data mining method with a performance better than random noise. Figure 10 shows that, of the six methods explored here, only *one* (Näive Bayes with logNums) had a median performance that was both large and positive.

More generally, our high-level conclusion is that it is no longer adequate to assess defect learning methods using only one data set and only one learner. Further research should assess the merits of their proposed techniques via extensive experimentation.

XI. Future Work

The role of any baseline experiment, such as the one offered here, is to be superseded by subsequent results. Paradoxically, this paper will be a success if it is quickly superseded. Our hope is that numerous researchers repeat our experiments and discover learning methods that are superior to the one proposed here.

There are many ways to design learning methods that could out-perform the results of this paper. Here, we list just two:

- With regard to pre-processing the numerics, we might be able to do even better than our current results. Dougherty et.al. [?] report spectacular improvements in the performance of Näive Bayes via the use of better numeric preprocessing than just simple log-filtering.
- The Halstead and McCabe attributes were defined in the 1970s and complier technology has evolved considerably since then. Halstead and McCabe are *intra-module* metrics and with modern intra-procedural data flow analysis, it should be possible to define a new set of 21^{st} century *inter-module* metrics that yield better defect predictors.

References

Dan Baker is a student.

Jeremy Greenwald is a graduate student in the Computer Science Department at Portland State University. He received his BS in Physics and Astronomy from the University of Pittsburgh in 2001. He has over six years of research experience in numeric methods and data mining. His master thesis focuses on comparative study of data mining techniques and equivalences with numeric optimization techniques. He also has interned at a software development firm in Beaverton, Oregon. His expected graduation date from PSU is late 2006.

Art Frank Art Frank is an undergraduate student pursuing a BS in Computer Science at Portland State University. He is currently working as an IT Manager and Database Administrator at a large non-profit organization in Portland, Oregon.