On the Relative Value of Cross-company and Within-Company Data for Defect Prediction

Burak Turhan, *Student Member, IEEE,* Tim Menzies, *Member, IEEE,* Ayse Bener, *Member, IEEE,* and Justin Distefano

***Abstract*— We report experiments where defect predictors from cross-company (CC) data result in dramatically high probabilities of detecting faulty modules (median value of 7 data sets: 97%). Sadly, this CC data also yield very large false alarm rates (median value: 64%).**

We explain this high false alarm rate by hypothesizing that CC data mixes useful data with an excess of *extraneous information*. In support of this hypothesis, we show that simple nearest neighbor (NN) sampling of CC data can filter the extraneous information to yield detectors with performance close to, but not better than, within-company (WC) data. That is, NN-filtered CC data can be a viable stop-gap technique while local data is being collected.

That stop-gap can be quite brief: we demonstrate in this paper that the minimum number of data samples required to build effective defect predictors can be quite small (often, just 100 modules).

Hence, for defect prediction, we recommend a two-phase approach. In phase one, companies should use NN-filtered CC data to initiate defect prediction process and simultaneously start collecting WC (local) data. Once enough WC data is collected, organizations should switch to phase 2 and use predictors learned from WC data.

KEYWORDS: defect prediction;learning; metrics (product metrics); cross-company; within-company

I. Introduction

What is the generality of our theories for common problems in empirical software engineering? Are the lessons learned from *Project-X* in *Company-A* relevant to *Project-Y* in *Company-B*? Theoretically, the answer is "no". Organizations can work in different domains, have different processes, and define/measure defects and other aspects of their products and processes in different ways. Worse, all too often, organizations do not precisely define their processes, products, measurements, etc. Hence, given data from *Project-X* from *Company-A* it is not clear to what extent the domain, process, and measurement varies from that project to other projects.

Mr. Turhan and Dr. Bener are with the Department of Computer Engineering, Bogazici University, Turkey. Emails: turhanb@boun.edu.tr, bener@boun.edu.tr.

Dr. Menzies and Mr. Distefano are with the Lane Department of Computer Science and Electrical Engineering, West Virginia University. Mr. Distefano is also Chief Programmer at Integrated Software Metrics. Emails: tim@menzies.us and jdistefano@ismwv.com

The research described in this paper was supported by Bogazici University research fund under grant number BAP-06HA104, by TUBITAK under grant number EEEAG 108E014 and at West Virginia University under grants with NASA's Software Assurance Research Program. Reference herein to any specific commercial product, process, or service by trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

See See http://menzies.us/pdf/07interra.pdf for an earlier draft of this paper. Manuscript received June, 2008; revised XXX,

Our view is that cross-company data must be used with great care in any software engineering problem. For example, previously [1], we have built elaborate tool sets for converting supposedly general *effort estimation* models to local conditions. That work concluded that data imported from other organizations must be extensively pruned (removal of extraneous features and projects) before it can be tuned to local conditions.

In this paper, we turn the attention to *defect prediction*. Specifically, this paper assesses the relative merits of cross-company (CC) vs within-company (WC) data for defect prediction. To the best of our knowledge, this kind of analysis is a novel one in defect prediction literature. We perform several experiments to answer the following questions:

A. Are the defect predictors learned from CC data beneficial for organizations?

Our first experiment identifies the conditions under which cross-company data are preferred to within-company for the purposes of learning defect predictors. Those conditions will turn out to be quite extreme; so much that they hold in only a small number of organizations (e.g. organizations would have to tolerate extremely high false alarm rates). Hence, except in very rare cases, this first experiment deprecates the use of unfiltered cross-company data for defect prediction.

B. Can we still make use of CC data for defect prediction?

Our explanation for the poor performance of CC data is that it mixes useful information with an excess of extraneous information. To explore, we applied nearest neighbor (NN) filtering to the CC data. This second experiment shows that some of the drawbacks of CC data can be removed. While, WC data still performs best, NN-filtered CC data can be viable alternative, while an organization starts its own local WC collection program.

C. How much data do organizations need for constructing a local model for defect prediction?

Kitchenham *et al.* [2] argue that organizations use cross-company data since within-company data can be so hard to collect:

- The time required to collect enough data on past projects from within a company may be prohibitive.
- Collecting within-company data may take so long that technologies change and older projects do not represent current practice.

We show below that predictors learned from a mere one hundred examples perform as well as predictors learned from many more examples. That is:

- Predictors tuned to the particulars of one company can be learned using very little data, collected in a very small amount of time;
- The above two reasons [2] for avoiding WC-data in *effort estimation* may not neccessarily hold for *defect prediction*.

D. Can our theories and results be generalized?

The above experiments use data collected from NASA projects. In order to check the external validity of our conclusions, we replicated all the above three experiments on data from a company that has no ties with NASA: specifically, a Turkish company writing software controllers for Turkish whitegoods. All the results described above also hold for the Turkish data. While this does not conclusively prove the external validty of our conclusions, it does suggest that these results are not just some fluke of one domain.

II. Related Work

A. Effort Estimation

Before beginning, it is important to question the value of WC vs CC studies. Intuitivaly, it could be argued that there is no issue here since *of course* local data is always better than imported data.

Prior to this article, there was no test of this intuition in the domain of *defect prediction*. However, in the domain of *effort estimation*, it turns out that this intuition has only mixed support:

- Mendes et al. [3] found within-company data performed much better than cross-company data for predicting estimation effort of web-based projects. They only recommend using cross-company data in the special case when that "data are obtained using rigorous quality control procedures".
- A similar conclusion was reached by Abrahansson et al. who discussed learning effort predictors in the context of an agile development process [4]. They strongly advocate the use of WC-data.

However, other studies are not so clear in their conclusions:

- MacDonnel & Shepperd tried to find trends in a set of papers relating to project management and effort estimation [5]. However, the papers studied by MacDonnel & Shepperd used a wide range of data sets so these authors found it hard to offer a definitive combined conclusion.
- In other work, after a review of numerous case studies, Kitchenham et al. [2] concluded that the value of CC vs WC data for effort estimation is unclear:

  > *... some organizations would benefit from using models derived from cross-company benchmarking databases but others would not* [2].

- Premraj and Zimmermann suspects that the reason for the contradictory results are due to heterogeneity in data. Therefore they build business specific cost models to have homogeneity in data. They compare within company, cross company and business specific cost models and report that although cross company models perform slightly worse, neither model is significantly better than others [6].

How can these contradictory results be explained? One possibility is that effort estimation requires the collection of project data, some of which has ambiguous definitions. For example, one of the features of the COCOMO-family [7] of effort predictors is "applications experience" (aexp). According to one on-line source[1], this feature is defined as follows: "the project team's equivalent level of experience with *this type* of application". No guidance is offered regarding how to characterize "this type of application". Hence, there is some degree of ambiguity in this definition. We conjecture that the ambiguity of the effort estimation features is one reason for the variance in the results reported by MacDonnel & Shepperd and Kitchenham et al. [2], [5].

Static code features, on the other hand, are not so ambiguous. Simple toolkits can be used to collect these features in a rapid, automatic, and uniform manner across multiple projects. Therefore, in theory, conclusions reached from these features should be less ambiguous than those reached from effort estimation features.

To test this hypothesis, the rest of this paper assesses the relative merits of WC-vs-CC data for defect prediction.

B. Defect Prediction

This section updates a literature review from a prior publication on defect prediction [8].

For the two experiments in this paper, we learn defect predictors from tables of static code features defined by McCabe [9] and Halstead [10]. McCabe (and Halstead) are "module"-based metrics. Each row in the table stores information from one module; the smallest unit of functionality[2]. To learn such predictors, the data tables are augmented with one column holding boolean values for "defects detected". The data mining task is to find combinations of code features that predict for the value in the defects column.

Defect predictors can be used to bias the ordering of modules to be inspected by verification and validation teams:

- In the case where *there are insufficient resources to inspect all code* (which is a very common situation in industrial developments), defect predictors can be used to increase the odds that the inspected code will have more defects.
- In the case where *all the code is to be inspected*, but that inspection process will take weeks to months to complete, defect predictors can be used to increase the odds that defective modules will be inspected earlier. This is useful since it gives the development team earlier notification of what modules require rework, hence giving them more time to complete that rework prior to delivery.

[1] `http://sunset.usc.edu/research/COCOMOII/expert_cocomo/drivers.html`

[2] In other languages, modules may be called "function" or "method".

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 80%, or higher. This is higher than currently-used industrial methods such as manual code reviews:

- A *IEEE Metrics 2002* [11] panel concluded that manual software reviews can find ≈60% of defects[3]
- In 2004, Raffo (personnel communication) reports that the defect detection capability of industrial review methods can vary from probability of detection: $pd = TR(35, 50, 65)\%$[4]. for full Fagan inspections [14] to $pd = TR(13, 21, 30)\%$ for less-structured inspections.

Easy to use: static code attributes like lines of code and the McCabe/Halstead attributes can be automatically and cheaply collected, even for very large systems [15]. By contrast:

- Other methods such as manual code reviews are labor-intensive; e.g. 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 [16].
- Other features (e.g. number of developers, the software development practices used to develop the code) may be unavailable or hard to characterize.

Widely used: Many researchers use static attributes to guide software quality predictions (see [9], [10], [15], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33]). Verification and validation (V&V) textbooks ([34]) advise using static code complexity attributes to select modules 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. For example, in Feburary 2008 the first author attended two briefings by NASA and European Space Agency test engineers who both cited McCabe's $v(g) \geq 10$ as one of their triggers for modules requiring closer inspections.

Nevertheless, there are many reasons to doubt the value of static code attributes for defect prediction. Descriptions of software modules *only* in terms of static code attributes can overlook some important aspects of software including the type of application domain; the skill level of the individual programmers involved in system development; contractor development practices; the variation in measurement practices; and the validation of the measurements and instruments used to collect the data. For this reason some researchers augment, or replace static code measures with other information such as the history of past faults or changes to code or number of developers who have worked on the code [35].

Also, 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 [37]. Fenton uses this example to argue the uselessness of static code attributes. Further, Fenton & Pfleeger note that the main McCabe's attribute (cyclomatic complexity, or $v(g)$) is highly correlated with lines of code [37]. Shepperd & Ince repeated that result, commenting 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" [38].

[3]That panel supported neither Fagan's claim [12] 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 [13].

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

	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 [36]. These values were generated using the standard settings of a state-of-art decision tree learner (J48). For each data set, ten experiments were conducted, where a decision tree was learned on 90% of the data, then tests are done of the remaining 10%. The numbers shown here are the average results across ten such experiments.

If the above criticisms are correct then we would predict that, in general, the performance of a predictor learned by a data miner should be very poor. More specifically, the supposedly better static code attributes such as Halstead and Mccabe should perform no better than just simple thresholds on lines of code.

Neither of these predictions are true, at least for the data sets used in this study. The defect predictors learned from static code attributes 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. In this paper we find predictors with a probability of detection (pd) and probability of false alarm (pf) of

$$median(pd) \geq 80; median(pf) \leq 26$$

Figure 1 lets us compare our new results against standard binary prediction results from the UC Irvine machine learning repository of standard test sets for data miners [36]. Our (pd, pf) are very close to the standard results of (pd,pf)=(81%,20%) which is noteworthy in two 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 detection and much higher false alarm rates.
2) These (pd, pf) results are better than currently used industrial methods such as the pd≈60% reported at the 2002 IEEE Metrics panel or the $median(pd) = 21..50$

reported by Raffo[5].

While Figure 1 shows that our defect detectors work nearly as well as standard data mining methods, it does not demonstrate that false alarm rates of around $\frac{1}{4}$ are useful in an industrial context. Assessing such detectors in an industrial context depends on the industry. But one of our clients (the company from which SOFTLAB data in Figure 2 are collected) is keen to use our detectors, arguing that they operate in a highly competitive market segment where profit margins are very tight. Therefore reducing the cost of the product even by 1% can make a major difference both in market share and profits. Their applications are embedded systems where, over the last decade, the software components have taken precedence over the hardware. Hence their problem is a software engineering problem. According to Brooks [39], half the cost of software development is in unit and systems testing. The company also believes that their main challange is the testing phase and they seek predictors that indicate where the defects might exist *before* they start testing. This allows them to efficiently use their scarce resources.

III. DATA

This paper focuses solely on predictors learned from static code features (These features are shown in Figure 3 and explained in Figure 4 and Figure 5). It is therefore prudent to digress and discuss the value of other types of features.

Defect predictors have been successfully learned from wide variety of measures such as:

- static code features [8];
- churn metrics (rates of code change) [21];
- personnel details about the development team [40];
- features extracted from requirements documents [41].

This list is hardly exhaustive and can be extended be either adding new feature types of combining existing ones. For example, Jiang, Cukic & Menzies [41] found that defect predictors learned from requirements *and* code measures can create predictors that out-perform predictors learned from just code *or* just requirements metrics.

Nevertheless, we hesitate to demand defect prediction should *always* be based on churn, requirements, code, or personnel features. We have some theoretical evidence suggesting that the "best" set of features may change from project to project (see the feature subset selection experiments of [42]). More pragmatically, we note that different projects use different tools and build software using different processes. Hence, different projects have access to different sets of features which may be useful for predicting where faults hide; for example:

[5]But note that we can only relatively compare the defect *detection* properties of automatic vs manual methods. Unlike automatic defect prediction via data mining, the above manual inspection methods don't just report "true,false" on a module, Rather, the manual methods also provide specific debugging information. Hence, a complete comparison of automatic vs manual defect prediction would have to include both an analysis of the time to *detect* potential defects *and* the time required to *fix* them. Manual methods might score higher to automatic methods since they can offer more clues back to the developer about what is wrong with the method. However, such an analysis is beyond the scope of this paper. Here, we focus only on the relative merits of different methods for *detecting* errors.

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

Fig. 3. Features used in this study. The Halstead features are explained in Figure 4 and the Mccabe features are explained in Figure 5.

- An open source software (OSS) or agile process may have no access to the detailed requirement documents used to build model by Jian, Cukic, & Menzies;
- The data used in this study (see Figure 2) comes from NASA sub-contractors who do not report the personnel information used by Nagappan et.al. in their defect predictions [40].

Debates over the value of one kind of feature over another is orthogonal to the main point of this paper. In this work, we identify a particular type of feature (i.e. static code features) which we can collect from multiple projects. With that data in hand, we now explore the value of CC vs WC learning. For future work, we would repeat this analysis with those other kinds of features, if and when they become available from multiple projects.

An advantage of static code features is that they can be quickly and automatically collected from the source code, even if no other information is available. The experiments of this paper use the static code data of Figure 2, which are downloaded from the PROMISE repository[6]. We considered using static code features collected from OSS and examined the defect logs of the Mozilla project. This experiment was abandoned when we realized that, at least for the Mozilla project, that those issue reports did not accurately map between defect reports and modules; e.g. a developer may checking many modules as part of a bug fix, some of which may actually be enhancements generated as a side-effect of the bug fix. Other researchers have had more success with OSS defect prediction [43] and in future work we will try again to apply our methods to OSS.

Our data are taken from software developed in different geographical locations across North America (NASA) and Turkey (SOFTLAB). Within a system, the sub-systems shared some common code base but did not pass personnel or code between sub-systems. While NASA and SOFTLAB are one

[6]http://promisedata.org/repository

The Halstead features were proposed by Maurice Halstead in 1977. He argued that modules that are hard to read are more likely to be fault prone [10]. Halstead estimates reading complexity by counting the number of operators and operands in a module: see the h features of Figure 3. These three raw h Halstead features were then used to compute the H: the eight derived Halstead features using the equations shown in Figure 3. In between the raw and derived Halstead features are certain intermediaries:

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

Fig. 4. Notes on the Halstead features

An alternative to the Halstead features of Figure 4 are the complexity features 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 [9].
The first three lines of Figure 3 shows McCabe three main features 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 [37].
- The *essential complexity*, $(ev(G))$ of 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" [37]). $ev(G) = v(G) - m$ where m is the number of subflowgraphs of G that are D-structured primes [37].
- Finally, the *design complexity* $(iv(G))$ of a module is the cyclomatic complexity of a module's reduced flow graph.

Fig. 5. Notes on the McCabe features

single *source* of data, there are several projects within each source. For example, NASA is really an umbrella organization used to co-ordinate and fund a large and diverse set of projects:

- The NASA data was collected from across the United States over a a period of five years from numerous NASA contractors working at different geographical centers.
- These projects represent a wide array of projects, including satellite instrumentation, ground control systems and partial flight control modules (i.e. Attitude Control).
- The data sets also represent a wide range of code reuse: some of the projects are 100% new, and some are modifications to previously deployed code.

That is, even if we explore just (say) the NASA data sets, we can still examine issues of cross- vs within- company data use.

The external validity of generalizing from NASA examples has been discussed elsewhere [8]. In summary, NASA uses contractors who are contractually obliged (ISO-9O01) to demonstrate their understanding and usage of current industrial best practices. These contractors service many other industries; for example, Rockwell-Collins builds systems for many government and commercial organizations. For these reasons, other noted researchers such as Basili, Zelkowitz, et al. [44] have argued that conclusions from NASA data are relevant to the general software engineering industry.

Nevertheless, it is always wise to test claims of external validity. Hence:

- The SOFTLAB data are, initially, in reserve. Our first three experiments will be based solely on the aerospace applications found in the NASA data.
- Our last experiment will check if the SOFTLAB data exhibit the same pattern as the NASA data.

The results are shown below.

IV. Experiment #1: WC-vs-CC

A. Design

Our first WC-vs-CC experiments repeated the following procedure for all 7 NASA tables of Figure 2. For each table, test sets were built from 10% of the rows, selected at random. Defect predictors were then learned from:

- Treatment 1 (CC): all rows from the other 11 tables.
- Treatment 2 (WC): just the other 90% rows of this table;

Most of the Figure 2 tables come from systems written in "C/C++" but at least one of the systems was written in JAVA. For cross-company data, an industrial practitioner may not have access to detailed meta-knowledge (e.g. whether it was developed in "C" or JAVA). They may only be aware that data, from an unknown source, are available for download from a certain url. To replicate that scenario, we will make no use of our meta-knowledge about Figure 2. As we shall see, a clear stable effect will occur across all the tables, regardless of (say) the implementation language.

In order to control for *order effects* (where the learned theory is unduly affected by the order of the examples) our procedure was repeated 20 times, randomizing the order of the rows in the table each time. Therefore, randomly selected training and test sets did not stay constant throughout the repetitions, In all, we ran 280 experiments to compare WC-vs-CC:

$$(2 \text{ treatments}) * (20 \text{ randomized orderings}) * (7 \text{ tables})$$

All the numeric distributions in the Figure 2 data are exponential. A "log-filter" replaces all numerics N with $log(N)$. This spreads out exponential curves more evenly across the space from the minimum to maximum values (to avoid numerical errors with $ln(0)$, all numbers under 0.000001 are replaced with $ln(0.000001)$). This "spreading" can significantly improve the effectiveness of data mining [8].

In prior work we have explored a range of data mining methods for defect prediction and found that classifiers based on Bayes theorem work best for the Figure 2 data [8] (for notes on Bayes classifiers, see Figure 7). Since that study, we have tried to find better data mining algorithms for defect prediction. To date, we have failed. Our recent (as yet, unpublished) experiments have found no additional statistically

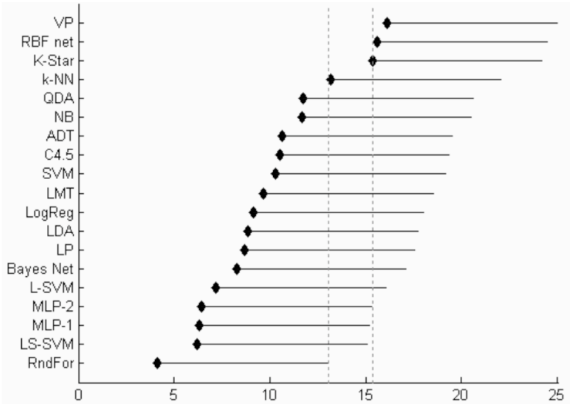

Fig. 6. Range of ranks seen in 19 learners building defects predictors when, 10 times, a random 66% selection of the data are used for training and the remaining data are used for testing. In ranked data, values from one method are replaced by their rank in space of all sorted values (so smaller ranks mean better performance). In this case, the performance value was area under the false positive vs true positive curve (and larger values are better). Vertical lines divide the results into regions where the results are statistically similar. For example, all the methods whose top ranks are 4 to 12 are statistically insignificantly different. From [51].

significant improvement from the application of the following data mining methods: logistic regression; average one-dependence estimators; under- or over-sampling [45], random forests, RIPPER [46], decision tree learning with J48 [47], rule learning with OneR [48] and bagging [49]. Only boosting [50] on discretized data offers a statistically better result than a Bayes classifier. However, we cannot recommend boosting: the median improvement is quite negligible and boosting is orders of magnitude slower than our simple Bayes classifier.

Other researchers have also failed to improve our results. For example, Lessmann et al. investigate the statistical difference of the results between 19 learners, including naive Bayes, on the same datasets [51]. Figure 6 shows that the simple Bayesian method discussed above ties in first place along with 15 other methods.

Data mining effectiveness was measured using pd, pf and $balance$ [8], [55]. If $\{A, B, C, D\}$ are the true negatives, false negatives, false positives, and true positives (respectively) found by a defect predictor, then:

$$pd = \quad recall = \quad D/(B+D) \tag{2}$$

$$pf = \quad C/(A+C) \tag{3}$$

$$bal = \quad balance = \quad 1 - \frac{\sqrt{(0-pf)^2 + (1-pd)^2}}{\sqrt{2}} \tag{4}$$

All these values range zero to one. Better and *larger* balances fall *closer* to the desired zone of no false alarms ($pf = 0$) and 100% detection ($pd = 1$).

Other measures such as $accuracy$ and $precision$ were not used since, as shown in Figure 2, the percent of defective examples in our tables was usually very small (median value around 8%). Accuracy and precision are poor indicators of performance for data where the target class is so rare (for more on this issue, see [8], [55]).

The WC and CC results were visualized using *quartile charts*. To generate these charts, the performance deltas for some treatment are sorted to isolate the median and the lower and upper quartile of numbers. For example:

$$\{\overbrace{4, 7, 15}^{q1}, 20, 31, \overbrace{40}^{median}, 52, 64, \overbrace{70, 81, 90}^{q4}\}$$

In our quartile charts, 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 the 50% percentile value. The above numbers would therefore be drawn as follows:

$$0\% \quad — \quad \bullet \; | \quad —— \quad 100\%$$

The Mann-Whitney U test [56] was used to test for statistical difference between treatments. This non-parametric test replaces (e.g.) pd values with their rank inside the population of all sorted pd values. Such non-parametric tests are

Bayesian classifiers offers a relationship between fragments of evidence E_i, a prior probability for a class $P(H)$, and a posteriori probability $P(H|E)$. Note that the likelihood $P(H|E)$ is approximated by the product term due to the i.i.d. (independent and identically distributed) assumption:

$$P(H|E) = \left(\prod_i P(E_i|H)\right) \frac{P(H)}{P(E)} \tag{1}$$

For example, in our data sets, there are two hypotheses: modules are either defective or not; i.e. $H \in \{defective, nonDefective\}$. Also, if a particular module has $numberofSymbols = 27$ and $LOC = 40$ and was previously classified as "defective" then

$$\begin{array}{lcl} E_1 & : & numberOfSymbols = 27 \\ E_2 & : & LOC = 40 \\ H & : & defective \end{array}$$

When building defect predictors, the posterior probability of each class ("defective" or "defect-free") is calculated, given the features extracted from a module. So, if a data set has 100 modules and 25 of them are faulty, then:

$$P(defective) = 0.25$$

When testing new data, a module is assigned to the class with the higher probability, calculated from Equation 1.
For numeric features, a feature's mean μ and standard deviation σ is used in a Gaussian probability function [52]:

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

Simple Bayes classifiers are often called "naive" since they assume independence of each feature. While this assumption simplifies the implementation (frequency counts are required only for each feature), it is possible that correlated events are missed by this "naive" approach. Potentially, this is a significant problem for our kinds of data sets where the static code measures are highly correlated (e.g. the number of symbols in a module increases linearly with module lines of code).
In a previous work, we have investigated variants of Bayesian models considering this issue and observed that the independence assumption of naive Bayes is safe for the Nasa datasets [53]. In another work, we have also performed extensive feature subset selection (FSS) experiments, where similar prediction performances were obtained using small subsets of features selected by Information Gain [8]. However, we did not observe an increase in the prediction performances when we used whole available data. Furthermore, Domingos and Pazzani show theoretically that the independence assumption is a problem in a vanishingly small percent of cases [54]. This explains the repeated empirical result that, on average, seemingly naive Bayes classifiers perform as well as other seemingly more sophisticated schemes (e.g. see Table 1 in [54]).
The Domingos and Pazzani result also explains our prior experiments where naive Bayes did not perform worse than other learners that continually re-sample the data for dependent instances (e.g. decision-tree learners that recurse on each "split" of the data [47]).

Fig. 7. About Bayes classifiers.

	treatment	min	Q1	median	Q3	max
pd	CC	50	83	97	100	100
	WC	17	63	75	82	100
pf	CC	14	53	64	91	100
	WC	0	24	29	36	73

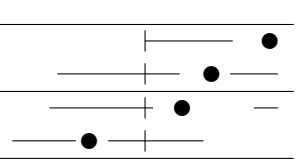

Fig. 8. Experiment #1 results averaged over seven NASA tables. Numeric results on left; quartile charts on right. "Q1" and "Q3" denote the 25% and 75% percentile points (respectively). The upper quartile of the first row is not visible since it runs from 100% to 100%; i.e. it has zero length.

group	pd $WC \rightarrow CC$	pf $WC \rightarrow CC$	tables	$\|tables\|$
a	increased	increased	CM1 KC1 KC2 MC2 MW1 PC1	6
b	same	same	KC3	1

Fig. 9. Summary of U-test results (95% confidence): moving from WC to CC. For full results, see Figure 10.

recommended in data mining since many of the performance distributions are non-Gaussian [57].

We report one deviation from our prior procedure. The tables of data come from different sources and, hence, have different features. For this study, all the tables were pruned such that they only contained features that appear in all the seven tables.

B. Results from Experiment #1

Figure 8 shows the $\{pd, pf\}$ quartile charts for CC vs WC data averaged over seven NASA datasets. The trend is very clear: CC data dramatically increases *both* the probability of detection and the probability of false alarms. The pd results are particularly striking.

For cross-company data:

- 50% of the pd values are at or above 97%
- 75% of the pd values are at or above 83%;
- And all the pd values are at or over 50%.

By way of comparison, recall from the above that our previous result had an average pd of 71% [8].

To the best of our knowledge, Figure 8 are the largest pd values ever reported from these data. However, these very high pd values come at some considerable cost. Note in Figure 8 that the median false alarm rate has changed from 29% (with WC) to 64% (with CC) and the maximum pf rate now reaches 100%.

We explain these increases in pd, pf with the following *extraneous hypothesis*: Using a large training set (e.g. seven of the tables in Figure 2) informs not only all the causes of errors, but also of numerous irrelevancies (e.g. applying statistics gathered from JAVA programs to "C" programs). Hence, large training sets increase the probability of detection (since there are more known sources of errors) as well as the probability of false alarms (since there are more *extraneous factors* introduced to the analysis). We test the extraneous hypothesis in the next experiment.

C. Sanity Checks on Experiment #1

This section explores threats to the external validity of the Experiment#1 conclusions. It can be skipped at first reading of this paper.

Once a *general result* is defined (e.g. CC dramatically increases both pf and pd), it is good practice to check for specific exceptions to that pattern. Figure 9 shows a summary of results when U tests were applied to test results from each of the 7 tables, *in isolation* and Figure 10 shows the $\{pd, pf\}$ quartile charts for Experiment #1 for each NASA table:

- Usually ($\frac{6}{7}$), the general pattern still holds (see group a).
- In one case (see group b), there was no difference in the results of the different treatments.

Overall, the *general result* holds in the majority of cases (i.e. $\frac{6}{7}$), which is not a 100% internally consistent, however it is still a very clear effect.

D. Discussion of Experiment #1

When practitioners use defect predictors with high false alarm rates (e.g.the 64% reported above), they must allocate a large portion of their debugging budget to the unfruitful exploration of erroneous alarms.

In our industrial work, we have sometimes seen several situations where detectors with high false alarms are useful:

- *When the cost of missing the target is prohibitively expensive.* In mission critical or security applications, the goal of 100% detection may be demanded in all situations, regardless of the cost of chasing false alarms.
- *When only a small fraction the data is returned.* Hayes, Dekhtyar, & Sundaram call this fraction $selectivity$ and offer an extensive discussion of the merits of this measure [58].
- *When there is little or no cost in checking false alarms.* For example, a detector we have found useful in industrial settings is to check modules where

$$\frac{lines\ of\ comments}{lines\ of\ code} > 0.25$$

This detector triggers on complex functions that programmers comment extensively, instead of splitting up into smaller, more maintainable, functions. This detector suffers from high false alarms- it often triggers on well-written functions with detailed comments. On the other hand, it may be fast and simple for a human agent to inspect the identified modules and discern which ones were well-written and which were over-commented to compensate for being badly coded.

Having said that, we observe that most sites do *not* accept false alarm rates as high as 64%. Therefore, the conditions under which the benefits of CC data (high probabilities of detection) outweigh their high costs (high false alarm rates) are quite rare.

In summary, for most software applications, very high pf rates like the CC results of Figure 8 make the predictors impractical to use.

V. Experiment #2: NN-Filtered CC

The results of the first experiment limits the use of CC data to a limited domain (i.e. mission critical) and may look discouraging at first glance. In the first experiment we

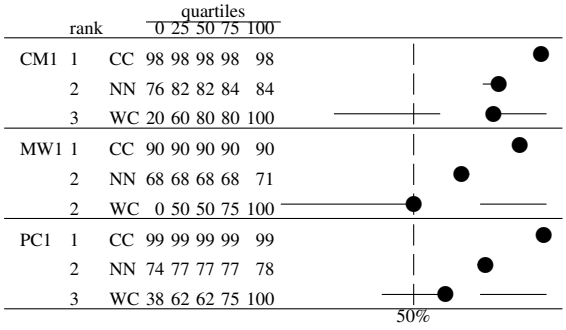

Fig. 11. Experiment #2 PD results where $NN_{pd} \geq WC_{pd}$. Rankings computed via Mann-Whitney (95% confidence) comparing each row to all other rows.

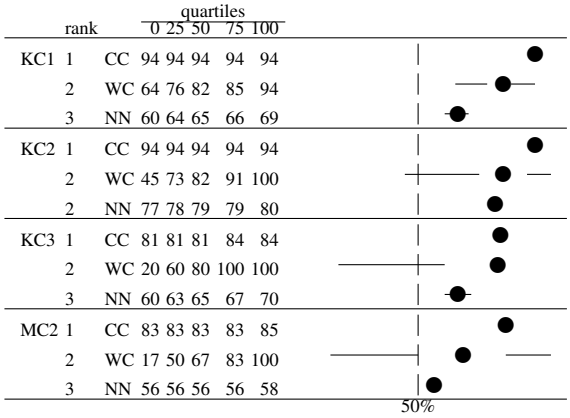

Fig. 12. Experiment #2 PD results where $NN_{pd} < WC_{pd}$.

explained our observations with the *extraneous hypothesis*, that the results are affected by the irrelevancies in CC data. In effort estimation literature, Kitchenham *et al.* argues the same concept as the heterogeneity of CC data [2]. Further, Premraj and Zimmermann builds business specific cost models to avoid such heterogeneity [6]. However, the effects of more homogeneous data for cost estimation are observed in a range of patterns that vary in different companies. (Recall that we explain this with the ambiguity in cost estimation features). Here, we will perform similar analysis for defect prediction to test the *extraneous hypothesis*.

A. Design

In this experiment, we try to construct more homogeneous *defect* datasets from CC data. For this purpose we use a simple sampling method (i.e. NN). Our idea behind sampling is to collect similar instances together in order to construct a learning set that is homogeneous with the validation set. We simply use the k-Nearest Neighbor (k-NN) method to measure the similarity between the validation set and the training candidates. The expected outcome of the sampling part is to obtain a subset of available CC data that shows similar characteristics to the local code culture.

We first calculate the pairwise distances between the validation set and the *candidate* training set samples (i.e. all CC data). Let N be the number of validation set size. For each validation sample, we pick its $k = 10$ nearest neighbors from candidate training set. Then we come up with a total of $10 \times N$ similar samples. Note that these $10 \times N$ samples may not be unique (i.e. a single data sample can be a nearest neighbor of many data samples in the validation set). Using only unique ones, we form the training set and use a random 90% of it for training a predictor. We repeat the last step 100 times.

B. Results

The ideal result would be that NN can be used as a surrogate for local WC data. This, in turn, would mean that developers could avoid the tedium and expense of local data collection. If NN out-performed WC then two observations would appear:

- *Observation1:* NN would have pd values above or equal to WC's pd. The examples displaying $Observation1$ are shown in Figure 11.

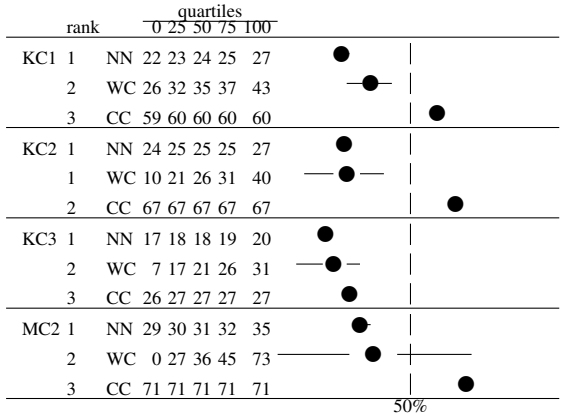

Fig. 13. Experiments #2 PF results where $NN_{pf} \leq WC_{pf}$.

- *Observation2:* NN would have pf values below or equal to WC's pf. The examples displaying $Observation2$ are shown in Figure 13.

Note that the conjunction of $Observation1$ and $Observation2$ is uncommon. In fact, our results suggest that $Observation1$ and $Observation2$ are somewhat mutually exclusive:

- As shown in Figures 11 and 14, the examples where NN increases the probability of detection are also those where it increases the probability of false alarms.

Hence, we cannot recommend NN as a replacement for WC.

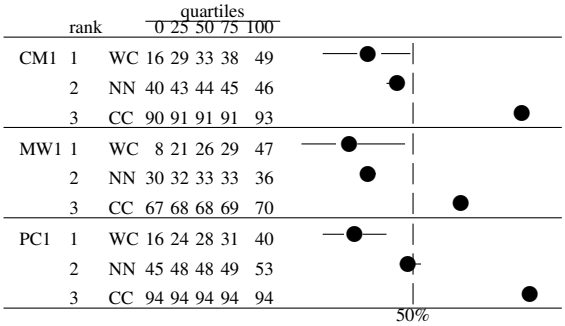

Fig. 14. Experiment #2 PF results where $NN_{pf} > WC_{pf}$.

Nevertheless, if local WC data is unavailable, then we would recommend processing foreign CC data with NN.

In all cases, NN dramatically reduces the high false alarm rates associated with the use of cross-company data. Often that reduction halves the false alarm range. For example, in MW1, the false alarm rate drops between CC to NN from 68% to 33%.

C. Discussions

In raw cross company models we see that the false alarm rates substantially increase compared to the within company models. Our new experiment shows that NN sampling removes the increased false alarm rates. Now we argue that, using NN sampling instead of random sampling, helps choosing training examples that are similar to problem at hand. Thus, the irrelevant information in non-similar examples are avoided. However, this also removes the rich sample base and yields a slight decrease in detection rates. Mann-Whitney tests reveal that NN sampling is

- far better than random sampling cross company data,
- and still worse than using within company data.

NN Sampling picks training examples that are similar to the examples about which we want to make predictions. Without introducing irrelevant information, NN sampling populates a set that has the same characteristics with the problem. That means, the likelihood of implementing similar software modules and having similar defects in these modules is high. Thus, NN sampling simulates that the selected cross company data are coming from within the company. Considering the *extraneous hypothesis*, NN sampling introduces homogeneity whereas random sampling introduces heterogeneity to cross company data.

Combining these results, if a company lacks local data, we would suggest a two-phase approach. In phase one, that organization uses imported CC data, filtered via NN. Also, during phase one, the organization should start a data collection program to collect static code attributes. Phase two commences when there is *enough* local WC data to learn defect predictors. During phase two, the organization would switch to new defect predictors learned from the WC data.

Our next experiment is, therefore, designed to determine the number of examples required to build defect predictors from WC data.

VI. Experiment #3: Incremental WC

In the introduction, the case was made that CC data are attractive to organizations if it avoids a time-consuming or expensive local data collection program. Also, our results of Experiment #1 and #2 reveals the necessity for WC data. In this section, we will show that adequate defect predictors can be learned from very small samples of WC data.

A. Design

A curious aspect of the above results is that defect predictors were learned using only a handful of defective modules. For example, consider a 90%/10% train/test split on $pc1$ with 1,109 modules, only 6.94% of which are defective. On average, the training set will only contain $1109 * 0.9 * 6.94/100 = 69$ defective modules. Despite this, $pc1$ yields an adequate median $\{pd, pf\}$ results of $\{63, 27\}\%$.

Experiment #3 was therefore designed to check how *little* data are required to learn defect predictors. Experiment #3 was essentially the same as the first experiment, but without treatment #1 (the cross-company study). Instead, experiment #3 took the 7 example tables of Figure 2 and learned predictors using:

- Treatment 3 (reduced WC): a randomly selected subset of up to 90% of the rows.

After randomizing the order of the rows, training sets were built using just the first 100,200,300,... rows in the tables. After training, the learned theory was applied to the remaining rows not used in training.

Experiment #1 only used the features found in all tables of data. For this experiment, we imposed no such restrictions and used whatever features were available in each data set.

B. Results from Experiment #3

Recall that Equation 4 defined "balance" to be a combination of $\{pd, pf\}$ that decreases if pd decreases or pf increases. As shown in Figure 15, there was very little change in balanced performance after learning from 100,200,300,... examples. Indeed, there is some evidence that learning from larger training sets had detrimental effects: the more training data, the larger the variance in the performance of the learned predictor. Observe how, in $kc1$ and $pc1$, as the training set size increases (moving right along the x-axis) the dots showing the balance performance start spreading out.

The Mann-Whitney U test was applied to check the visual trends seen in Figure 15. For each table, all results from training sets of size 100,200,300... were compared to all other results from the same table. The issue was "how much data are enough?" i.e. what is the minimum training set size that never lost to other training set of a larger size. Usually, that min value was quite small:

- In five tables $\{cm1, kc2, kc3, mc2, mw1\}$, $min = 100$;
- In $\{kc1, pc1\}$, $min = \{200, 300\}$ instances, respectively.

We explain the experiment #3 results as follows. These experiments used simplistic static code features such as lines of code, number of unique symbols in the module, etc. Such simplistic static code features are hardly a complete characterization of the internals of a function. Recall Fenton's example where *the same* functionality is achieved using *different* programming language constructs resulting in *different* static measurements for that module [59]. We would characterize such static code features as having *limited information content* [60]. Limited content is soon exhausted by repeated sampling. Hence, such simple features reveal all they can reveal after a small sample.

C. Sanity Checks on Experiment #3

This section checks for precedents on the Experiment #3 results and can be skipped at first reading of this paper.

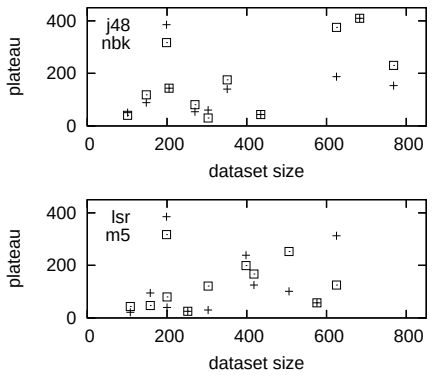

Fig. 16. Y-axis shows plateau point after learning from data sets that have up to X examples (from [61]). The left plot shows results from using Naive Bayes (nbk) or a decision tree learner (j48) [47] to predict for discrete classes. Right plot shows results from using linear regression (lsr) or model trees (m5) [62] to learn predictors for continuous classes. In this study, data sets were drawn from the UC Irvine data repository [36].

There is also some evidence that the results of Experiment 3 (that performance improvements stop after a few hundred examples) have been seen previously in the data mining literature (caveat: to the best of our knowledge, this is the first report of this effect in the defect prediction literature):

- In their discussion on how to best handle numeric features, Langley and John offers plots of the accuracy of Naive Bayes classifiers after learning on 10,20,40,..200 examples. In those plots, there is little change in performance after 100 instances [63].
- Orrego [61] applied four data miners (including Naive Bayes) to 20 data sets to find the *plateau point*: i.e. the point after which there was little net change in the performance of the data miner. To find the plateau point, Oreggo used t-tests to compare the results of learning from Y or $Y + \Delta$ examples. If, in a 10-way cross-validation, there was no statistical difference between Y and $Y + \Delta$, the plateau point was set to Y. As shown in Figure 16, many of those plateaus were found at $Y \leq 100$ and most were found at $Y \leq 200$. Note that these plateau sizes are consistent with the results of Experiment 3.

D. Discussion of Experiment #3

In the majority case, predictors learned from as little as one hundred examples perform as well as predictors learned from many more examples. This suggests that the effort associated with learning defect predictors from within-company data may not be overly large. For example, Figure 17 estimates that the effort required to build and test 100 modules may be as little as 2.4 to 3.7 months. Considering the proposed two-phase approach, these are the optimistic waiting times before switching to phase two (i.e. local predictors from WC data).

VII. Experiment Replication

Experiments # 1 to # 3 were based on NASA data. In order to search evidence for the external validity of the conclusions of those experiments, we obtained new data sets. Those data sets were deliberately chosen to be as far removed as possible from American aerospace software applications. Using our connections with the Turkish software industry, we collected three data sets in the format of Figure 3 from a Turkish white- goods manufacturer. The new datasets ({ar3, ar4, ar5}), marked as "SOFTLAB" in Figure 2, are the controller software for:

- A washing machine;
- A dishwasher;
- And a refrigerator.

Note that this software was developed via methods that are both culturally and organizationally different to NASA aerospace software. Turkish domestic appliances company software are developed by a small team of 2-3 people. The development is carried out in an ad-hoc, informal way rather than formal, process oriented approach in NASA. Furthermore, the company is a profit and revenue driven commercial organization, whereas NASA is a cost driven government entity.

For each SOFTLAB data set, we follow the same procedure as in Experiments # 1 and #2; i.e. 10% of the rows of each data set are selected at random for constructing test sets and then defect predictors are learned from:

- Treatment 1 (CC): all rows from 7 NASA tables.
- Treatment 2 (WC): random 90% rows of remaining SOFTLAB tables[7];
- Treatment 3 (NN): *similar* rows from 7 NASA tables.

The SOFTLAB tables include 29 static code features, 19 of which common with the NASA tables. In order to simplify the comparison between these new data and Experiment #1 and #2, we used only these shared attributes in our CC experiments. On the other hand we use all available attributes in WC experiments for SOFTLAB tables. In the following external validity experiment, we treated each NASA table as cross- company data for SOFTLAB tables.

Figure 18 shows the results:

- The *pd* values for CC treatment increases compared to WC treatment with the cost of increased *pf*.

100 modules may take as little as two to four months to construct. This estimate was generated as follows:

- In the $cm1$ data base, the median module size is 17 lines. 100 randomly selected modules would therefore use 1700 LOC.
- To generate an effort estimate for these modules, we used the on-line COCOMO [7] effort estimator (`http://sunset.usc.edu/research/COCOMOII/expert_cocomo/expert_cocomo2000.html`). Estimates were generated assuming 1700 LOC and the required reliability varying from very low to very high.
- The resulting estimates ranged from between 2.4 and 3.7 person months to build and test those modules.

Fig. 17. An estimate of the effort required to build and test 100 modules.

[7]In order to reflect the use in practice, we do not use the remaining 90% of the same project for training, we rather use a random 90% of data from other projects. Note that all WC experiments in this paper reflects within-company, not within project simulations. Since SOFTLAB data are collected from a single company, learning a predictor on some projects and to test it on a different one does not violate within company simulation.

Average results on SOFTLAB data

treatment		min	Q1	median	Q3	max
pd	CC	88	88	95	100	100
	WC	35	40	88	100	100
pf	CC	52	59	65	68	68
	WC	3	5	29	40	42

Table ($ar3$)

treatment		min	Q1	median	Q3	max
pd	CC	88	88	88	88	88
	WC	88	88	88	88	88
pf	CC	62	65	65	65	65
	WC	40	40	40	40	42

Table ($ar4$)

treatment		min	Q1	median	Q3	max
pd	CC	95	95	95	95	95
	WC	35	40	40	40	40
pf	CC	52	55	56	59	60
	WC	3	3	3	5	5

Table ($ar5$)

treatment		min	Q1	median	Q3	max
pd	CC	100	100	100	100	100
	WC	88	100	100	100	100
pf	CC	57	68	68	68	68
	WC	29	29	29	29	29

Fig. 18. Experiment #1 results for the SOFTLAB tables. Averaged and individual results are shown respectively.

	rank		quartiles 0	25	50	75	100
AR4	1	CC	35	90	100	100	100
	2	NN	65	65	70	70	70
	3	WC	35	40	40	40	45
AR3	1	CC	75	88	88	88	88
	1	NN	88	88	88	88	88
	1	WC	88	88	88	88	88
AR5	1	CC	88	100	100	100	100
	1	NN	100	100	100	100	100
	1	WC	88	100	100	100	100

50%

Fig. 19. Experiment #2 PD results for the SOFTLAB tables where $NN_{pd} \geq WC_{pd}$. Rankings computed via Mann-Whitney (95% confidence) comparing each row to all other rows.

	rank		quartiles 0	25	50	75	100
AR3	1	NN	38	38	38	38	40
	2	WC	40	40	40	40	42
	3	CC	38	55	56	60	80
AR5	1	NN	21	25	25	25	32
	2	WC	29	29	29	29	29
	3	CC	29	46	50	50	79

50%

Fig. 20. Experiments #2 PF results for the SOFTLAB tables where $NN_{pf} \leq WC_{pf}$.

- The CC treatment shifts {Q1, median} of *pf* from {5, 29} to {59, 65}.
- For the CC treatment:
  - 25% of the pd values are at 100%.
  - 50% of the pd values are above 95%
  - And all the pd values are at or over 88%.

	rank		quartiles 0	25	50	75	100
AR4	1	WC	3	3	3	5	7
	2	NN	24	25	25	25	28
	3	CC	6	47	62	67	78

50%

Fig. 21. Experiment #2 PF results for the SOFTLAB tables where $NN_{pf} > WC_{pf}$.

These results also provide suggestive evidence for the generality of our conclusions for Experiment # 3 (where we discuss that the minimum number of instances for training a predictor is merely $100 - 200$ examples). Note that SOFTLAB tables *ar3, ar4* and *ar5* have {63,107, 36} modules respectively, with a total of 206 modules. Thus, WC results in Figure 18 are achieved using a minimum of $(63 + 36) * 0.90 = 90$ examples (i.e. learn a predictor on *(ar3 + ar5)* and test it on *ar4*) and a maximum of $(63 + 107) * 0.90 = 153$ examples (i.e. learn a predictor on *(ar3 + ar4)* and test it on *ar5*).

Figure 19 to Figure 21 shows *Observation1* (i.e. $NN_{pd} \geq WC_{pd}$) and *Observation2* (i.e. $NN_{pf} \leq WC_{pf}$) for SOFTLAB data. Recall that these observations were mutually exclusive for NASA data. The pattern is similar in SOFTLAB data:

- for *ar4* mutual exclusiveness hold: $NN_{pd} \geq WC_{pd}$ and $NN_{pf} > WC_{pf}$
- for *ar3* and *ar5*: $NN_{pf} \leq WC_{pf}$. If the observations were mutually exclusive, we would expect $NN_{pd} < WC_{pd}$. Yet, for *pd* $NN_{pd} \geq WC_{pd}$, however this inequality holds with the equity (see Figure 19) and $NN_{pd} \not> WC_{pd}$

In summary, the WC, CC and NN patterns found in American NASA rocket software are also observed in software controllers of Turkish domestic appliances. While this is not the definitive proof of the external validity of our results, we find it a very compelling result that provides suggestive evidence for the generality for our observations.

VIII. Conclusion

In this study, we have found a clear and unambiguous conclusion for defect prediction amongst our static code features:

- CC-data dramatically increase the probability of detecting defective modules;
- But CC-data also dramatically increase the false alarm rate.
- This can be explained by the *extraneous hypothesis*;
- NN-filtering CC data avoids the high false alarm rates by removing irrelevancies in CC data;
- Yet WC-data models are still the best and they can be constructed with small amounts of data (i.e. 100 examples).

Interpreting these in terms of the posed questions in the introduction, we conclude that:

- Companies can benefit from raw CC data in extreme cases such as mission critical projects, where the cost of false alarms can be afforded.
- Pruning CC data with NN-filter allows the use of CC data for constructing practical defect predictors in other

domains. NN-filtered CC data yields much better results than raw CC data, yet closer but worse results than WC data.
- The best option of using WC data requires the collection of a mere hundred examples from within a company and can be done in a short time (i.e. a few months).
- We observe the same patterns not only in aerospace software from NASA, but also in software from a completely different company round the globe. While this is a strong evidence of generality, we take great care *not to* interpret it as a proof of external validity.

An important issue worth more mentioning is the concern about the time required for setting up a metric program (i.e. in order to collect data for building actual defect predictors). Our incremental WC results suggest that, in the case of defect prediction, this concern may be less than previously believed. In most of our experiments, as few as 100 modules may be enough to learn adequate defect predictors. When so few examples are enough, it is possible that projects can learn local defect predictors that are relevant to their current technology in just a few months.

Further, our experiences with our industry partners show that data collection is not necessarily a major concern. Static code attributes can be automatically and quickly collected with relatively little effort. We have found that when there is high level management commitment, it becomes a relatively simple process. For example, in an extreme case, the three projects of SOFTLAB data were collected in less than a week's time. Neither the static code attributes, nor the mapping of defects to software modules were available when the authors attempted to collect these data. Since these were smaller scale projects, it was sufficient to spend some time with the developers and going through defect reports. Although not all projects have 100 modules individually, the company has a growing repository from several projects and enough data to perform defect prediction.

We also have experience with a large scale telecommunication company, where a long-term metric program for monitoring complex projects (around 750.000 lines of code) requires introducing automated processes. Again with high level management support, it was possible to employ appropriate tool support and these new processes were introduced easily and invisible to the staff. For that project, we have now a growing repository of defects mapped with source code (around 25 defects per month). Note that the software in that project are being developed for more than 10 years and have very low defect rates. We have obtained the first results in the 8^{th} month of the project and it is planned to be completed in 12 months. In summary setting up a metric program for defect prediction can be done more quickly than it is perceived.

We conclude our findings by proposing a two-phase approach for initiating defect prediction process:

- *Phase1:* Use NN filtered CC data to make local predictions and start to collect WC data.
- *Phase2:* After a few hundred examples are available in the local repository (usually a few months), discard the predictor learned on CC data and switch to those learned from WC data.

REFERENCES

[1] T. Menzies, Z. Chen, J. Hihn, and K. Lum, "Selecting best practices for effort estimation," *IEEE Transactions on Software Engineering*, November 2006, available from http://menzies.us/pdf/06coseekmo.pdf.

[2] B. A. Kitchenham, E. Mendes, and G. H. Travassos, "Cross- vs. within-company cost estimation studies: A systematic review," *IEEE Transactions on Software Engineering*, pp. 316–329, May 2007.

[3] E. Mendes, G. Dinakaran, and N. Mosley, "How valuable is it for a web company to use a cross-company cost model, compared to using its own single-company model?" in *16th International World Wide Web Conference, Banff, Canada, May 8-12*, 2007, available from http://www2007.org/paper326.php.

[4] P. Abrahamsson, R. Moser, W. Pedrycz, A. Sillitti, and G. Succi, "Effort prediction in iterative software development processes – incremental versus global prediction models," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007, pp. 344–353.

[5] S. MacDonell and M. Shepperd, "Comparing local and global software effort estimation models – reflections on a systematic review," in *Empirical Software Engineering and Measurement, ESEM 2007*, 2007, pp. 401–409.

[6] R. Premraj and T. Zimmermann, "Building software cost estimation models using homogenous data," *Empirical Software Engineering and Measurement*, Jan 2007. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4343767

[7] B. Boehm, "Safe and simple software cost analysis," *IEEE Software*, pp. 14–17, September/October 2000, available from http://www.computer.org/certification/beta/Boehm_Safe.pdf.

[8] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, January 2007, available from http://menzies.us/pdf/06learnPredict.pdf.

[9] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, Dec. 1976.

[10] M. Halstead, *Elements of Software Science*. Elsevier, 1977.

[11] F. Shull, V. Basili, B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What we have learned about fighting defects," in *Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada*, 2002, pp. 249–258, available from http://fc-md.umd.edu/fcmd/Papers/shull_defects.ps.

[12] M. Fagan, "Advances in software inspections," *IEEE Trans. on Software Engineering*, pp. 744–751, July 1986.

[13] F. Shull, I. Rus, and V. Basili, "How perspective-based reading can improve requirements inspections," *IEEE Computer*, vol. 33, no. 7, pp. 73–79, 2000, available from http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.77.pdf.

[14] M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, 1976.

[15] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *ICSE*, 2005, pp. 580–586. [Online]. Available: http://doi.acm.org/10.1145/1062558

[16] T. Menzies, D. Raffo, S. on Setamanit, Y. Hu, and S. Tootoonian, "Model-based tests of truisms," in *Proceedings of IEEE ASE 2002*, 2002, available from http://menzies.us/pdf/02truisms.pdf.

[17] M. Chapman and D. Solomon, "The relationship of cyclomatic complexity, essential complexity and error rates," 2002, proceedings of the NASA Software Assurance Symposium, Coolfont Resort and Conference Center in Berkley Springs, West Virginia. Available from http://www.ivv.nasa.gov/business/research/osmasas/conclusion2002/Mike_C%hapman_The_Relationship_of_Cyclomatic_Complexity_Essential_Complexity_and_Erro%r_Rates.ppt.

[18] T. Menzies, J. DiStefano, A. Orrego, and R. Chapman, "Assessing predictors of software defects," in *Proceedings, workshop on Predictive Software Models, Chicago*, 2004, available from http://menzies.us/pdf/04psm.pdf.

[19] "Polyspace verifier®," 2005, available from http://www.di.ens.fr/~cousot/projects/DAEDALUS/synthetic_summary/POLYSP%ACE/polyspace-daedalus.htm.

[20] G. Hall and J. Munson, "Software evolution: code delta and code churn," *Journal of Systems and Software*, pp. 111 – 118, 2000.

[21] A. Nikora and J. Munson, "Developing fault predictors for evolving software systems," in *Ninth International Software Metrics Symposium (METRICS'03)*, 2003.

[22] T. Khoshgoftaar, "An application of zero-inflated poisson regression for software fault prediction," in *Proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong*, Nov 2001, pp. 66–73.

[23] T. Khoshgoftaar and N. Seliya, "Comparative assessment of software quality classification techniques: An empirical case study," *Empirical Software Engineering*, vol. 9, no. 3, pp. 229–257, 2004.

[24] W. Tang and T. M. Khoshgoftaar, "Noise identification with the k-means algorithm," in *ICTAI*, 2004, pp. 373–378. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICTAI.2004.93

[25] T. M. Khoshgoftaar and N. Seliya, "Fault prediction modeling for software quality estimation: Comparing commonly used techniques," *Empirical Software Engineering*, vol. 8, no. 3, pp. 255–283, 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1024424811345

[26] T. Menzies, J. D. Stefano, K. Ammar, K. McGill, P. Callis, R. Chapman, and D. J, "When can we test less?" in *IEEE Metrics'03*, 2003, available from http://menzies.us/pdf/03metrics.pdf.

[27] T. Menzies, J. S. DiStefeno, M. Chapman, and K. Mcgill, "Metrics that matter," in *27th NASA SEL workshop on Software Engineering*, 2002, available from http://menzies.us/pdf/02metrics.pdf.

[28] T. Menzies, J. D. Stefano, and M. Chapman, "Learning early lifecycle IVV quality indicators," in *IEEE Metrics '03*, 2003, available from http://menzies.us/pdf/03early.pdf.

[29] T. Menzies and J. S. D. Stefano, "How good is your blind spot sampling policy?" in *2004 IEEE Conference on High Assurance Software Engineering*, 2003, available from http://menzies.us/pdf/03blind.pdf.

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

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

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

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

[34] S. Rakitin, *Software Verification and Validation for Practitioners and Managers, Second Edition*. Artech House, 2001.

[35] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Software Eng.*, vol. 26, no. 7, pp. 653–661, 2000, available on-line at www.niss.org/technicalreports/tr80.pdf.

[36] C. Blake and C. Merz, "UCI repository of machine learning databases," 1998, uRL: http://www.ics.uci.edu/~mlearn/MLRepository.html.

[37] N. E. Fenton and S. Pfleeger, *Software Metrics: A Rigorous & Practical Approach (second edition)*. International Thompson Press, 1995.

[38] M. Shepperd and D. Ince, "A critique of three metrics," *The Journal of Systems and Software*, vol. 26, no. 3, pp. 197–210, September 1994.

[39] F. P. Brooks, *The Mythical Man-Month, Anniversary edition*. Addison-Wesley, 1995.

[40] V. B. Nachiappan Nagappan, Brendan Murphy, "The influence of organizational structure on software quality: An empirical case study," in *ICSE'08*, 2008.

[41] Y. Jiang, B. Cukic, and T. Menzies, "Fault prediction using early lifecycle data," in *ISSRE'07*, 2007, available from http://menzies.us/pdf/07issre.pdf.

[42] G. Boetticher, T. Menzies, and T. Ostrand, "The PROMISE Repository of Empirical Software Engineering Data," 2007, http://promisedata.org/repository.

[43] S. Kim, J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE TSE*, pp. 181–196, March/April 2008.

[44] V. Basili, F. McGarry, R. Pajerski, and M. Zelkowitz, "Lessons learned from 25 years of process improvement: The rise and fall of the NASA software engineering laboratory," in *Proceedings of the 24th International Conference on Software Engineering (ICSE) 2002, Orlando, Florida*, 2002, available from http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/83.88.pdf.

[45] C. Drummond and R. C. Holte, "C4.5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling," in *Workshop on Learning from Imbalanced Datasets II*, 2003.

[46] W. Cohen, "Fast effective rule induction," in *ICML'95*, 1995, pp. 115–123, available on-line from http://www.cs.cmu.edu/~wcohen/postscript/ml-95-ripper.ps.

[47] R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1992, iSBN: 1558602380.

[48] R. Holte, "Very simple classification rules perform well on most commonly used datasets," *Machine Learning*, vol. 11, p. 63, 1993.

[49] L. Brieman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[50] Y. Freund and R. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *JCSS: Journal of Computer and System Sciences*, vol. 55, 1997.

[51] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *accepted for publication IEEE Transactions on Software Engineering*, 2009.

[52] I. H. Witten and E. Frank, *Data mining. 2nd edition*. Los Altos, US: Morgan Kaufmann, 2005.

[53] B. Turhan and A. Bener, "A multivariate analysis of static code attributes for defect prediction," in *Proceedings of the Seventh International Conference on Quality Software*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 231–237.

[54] P. Domingos and M. J. Pazzani, "On the optimality of the simple bayesian classifier under zero-one loss," *Machine Learning*, vol. 29, no. 2-3, pp. 103–130, 1997. [Online]. Available: citeseer.ist.psu.edu/domingos97optimality.html

[55] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with precision," *IEEE Transactions on Software Engineering*, September 2007, http://menzies.us/pdf/07precision.pdf.

[56] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, 1947, available on-line at http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&hand%le=euclid.aoms/1177730491.

[57] J. Demsar, "Statistical comparisons of clasifiers over multiple data sets," *Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006, avaliable from http://jmlr.csail.mit.edu/papers/v7/demsar06a.html.

[58] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods," *IEEE Trans. Software Eng*, vol. 32, no. 1, pp. 4–19, 2006. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2006.3

[59] N. E. Fenton and S. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*. International Thompson Press, 1997.

[60] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of ceiling effects in defect predictors," in *Proceedings of PROMISE 2008 Workshop (ICSE)*, 2008, available from http://menzies.us/pdf/08ceiling.pdf.

[61] A. Orrego, "Sawtooth: Learning from huge amounts of data," Master's thesis, Computer Science, West Virginia University, 2004.

[62] J. R. Quinlan, "Learning with Continuous Classes," in *5th Australian Joint Conference on Artificial Intelligence*, 1992, pp. 343–348, available from http://citeseer.nj.nec.com/quinlan92learning.html.

[63] G. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence Montreal, Quebec: Morgan Kaufmann*, 1995, pp. 338–345, available from http://citeseer.ist.psu.edu/john95estimating.html.

source	project	language	description	(# modules) examples	features	loc	%defective
NASA	pc1	C++	Flight software for earth orbiting satellite	1,109	21	25,924	6.94
NASA	kc1	C++	Storage management for ground data	845	21	42,965	15.45
NASA	kc2	C++	Storage management for ground data	522	21	19,259	20.49
NASA	cm1	C++	Spacecraft instrument	498	21	14,763	9.83
NASA	kc3	JAVA	Storage management for ground data	458	39	7749	9.38
NASA	mw1	C++	A zero gravity experiment related to combustion	403	37	8341	7.69
SOFTLAB	ar4	C	Embedded controller for white-goods	107	30	9196	18.69
SOFTLAB	ar3	C	Embedded controller for white-goods	63	30	5624	12.70
NASA	mc2	C++	Video guidance system	61	39	6134	32.29
SOFTLAB	ar5	C	Embedded controller for white-goods	36	30	2732	22.22
				4,102			

Fig. 2. Ten tables of data, sorted in order of number of examples. The rows labeled "NASA" come from NASA aerospace projects while the rows labeled "SOFTLAB" come from a Turkish software company writing applications for domestic appliances. For details on the features used in each data set, see Figure 3.

Table ($cm1$)

treatment		min	Q1	median	Q3	max
pd	CC	80	100	100	100	100
	WC	40	60	80	100	100
pf	CC	87	91	96	96	98
	WC	24	27	33	38	47

Table ($kc2$)

treatment		min	Q1	median	Q3	max
pd	CC	82	91	91	100	100
	WC	55	73	82	91	100
pf	CC	57	62	64	74	81
	WC	14	24	31	33	45

Table ($mc2$)

treatment		min	Q1	median	Q3	max
pd	CC	50	67	83	100	100
	WC	17	33	67	67	83
pf	CC	55	64	73	73	100
	WC	0	27	36	45	73

Table ($pc1$)

treatment		min	Q1	median	Q3	max
pd	CC	88	100	100	100	100
	WC	38	63	63	75	88
pf	CC	89	92	93	95	99
	WC	17	25	27	30	34

Table ($kc1$)

treatment		min	Q1	median	Q3	max
pd	CC	82	88	94	94	100
	WC	64	73	82	85	97
pf	CC	47	49	51	53	57
	WC	27	34	36	38	40

Table ($kc3$)

treatment		min	Q1	median	Q3	max
pd	CC	60	80	80	80	100
	WC	40	60	80	80	100
pf	CC	14	19	24	31	38
	WC	10	17	21	26	36

Table ($mw1$)

treatment		min	Q1	median	Q3	max
pd	CC	75	75	100	100	100
	WC	25	50	75	75	100
pf	CC	50	55	63	66	82
	WC	13	18	21	29	37

Fig. 10. Project-wise Experiment #1 results for NASA tables.

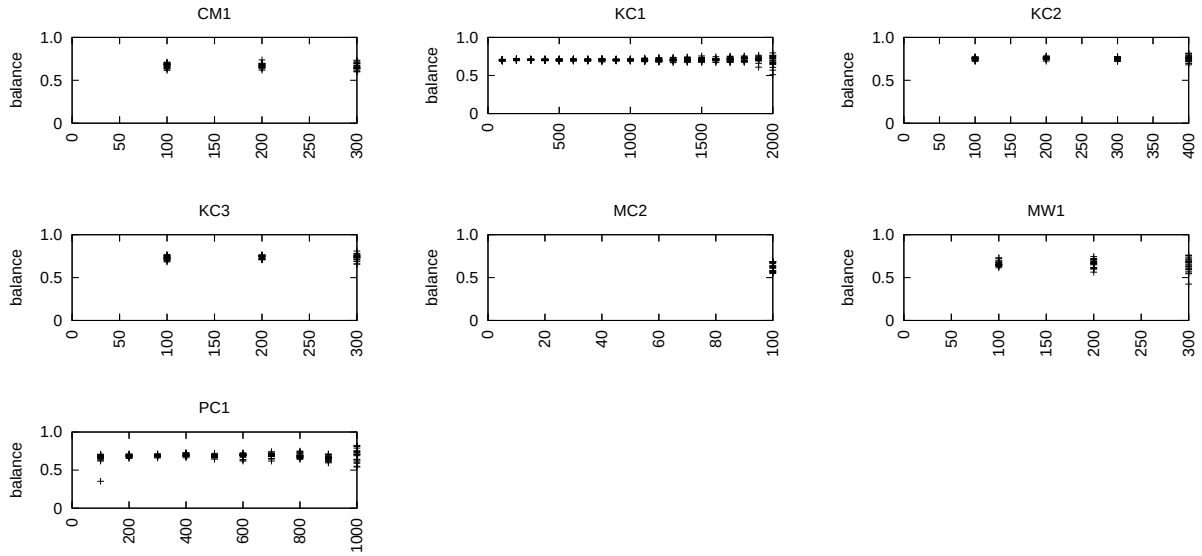

Fig. 15. Results from experiment #3. Training set size grows in units of 100 examples, moving left to right over the x-axis. The MC2 results only appear at the maximum x-value since MC2 has less than 200 examples.