Cross- vs Within-Company Defect Prediction Studies

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

***Abstract*— In a recent May 2007 IEEE TSE article, Kitchenham et.al. explored *effort estimation* and found contradictory evidence about the value of cross- vs within-company data. Those contradictory results may have been the result of effort estimation features, some of which are subjective in nature.**

Static code features are different than effort estimation features. They can be generated in an automatic, rapid, and uniform manner across multiple projects. Therefore, in theory, the conclusions reached from such features may be more uniform. This paper tests that theory by searching for uniform conclusions using cross- or within-company static code features. Whereas Kitchenham et.al. explored effort estimation, this paper explores *defect prediction*.

Cross-company static code features will be found to generate higher false alarm rates than within-company data. Hence, cross-company data is best used for mission critical software where (a) the extra costs associated with high false alarm rates is compensated by (b) an associated increase in the probability of predicting fault modules. For other classes of software, false alarm rates can be decreased using a very small amount of local data (often, just 100 modules). In our experiments, the use of within-company data halved the false alarm rate while decreasing prediction rates by only $\approx 10\%$. Hence, for non-mission-critical software, we strongly recommend using within-company data for defect prediction.

I. Introduction

In the age of open source development, web-accessible XML data, mash-ups, and reusable libraries of SE data, it is quite possible that organizations can access more data on software engineering *outside* their company than *within*. It is hence tempting to use such *cross-company* data since, as Kitchenham et.al. [1] observe:

- 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.

In the case of software effort estimation the value of cross-company data is an open question. Mendes et.al. [2] 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 is 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 [3]. They strongly advocate the use of WC-data.

Other studies are not so positive. MacDonnel & Shepperd tried to find trends in a set of papers relating to project management and effort estimation [4]. 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. [1] 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.* [1].

Effort estimation requires the collection of project data, some of which has ambiguous definitions. For example, one of the features of the COCOMO-family [5] 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.

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, in this paper, we seek conclusions using cross- vs within-company data from static code features taken from the 12 projects of Figure 1. Our hypothesis was confirmed. The relative merits of CC-vs-WC is clear and unambiguous, as revealed by two experiments. The first experiment concerns cross-company data and shows that cross-company data increases the probability of detecting a module (pd) dramatically. However, it does so at the cost of increasing the probability of false alarms (pf). When such large pfs are unacceptable, companies must take the time to collect within-company data.

The second experiment explores how *little* within-company data is required to learn useful predictors. We will show that, often, the predictors learned from a mere one hundred examples perform as well as predictors learned from many more examples. Hence, there are two reasons not to use cross-company data for learning defect predictors:

- Predictors tuned to the particulars of one company can be learned using very little data, collected in a very small amount of time;
- Using within-company data avoids the problem of high false alarm rates.

The rest of this paper discusses the data of Figure 1 and reviews the design and results of our two experiments.

data	language	(# modules) examples	features	%defective
pc5	C++	17,186	38	3.0
mc1	C++	9,466	38	0.71
pc2	C++	5,589	36	0.41
pc3	C++	1,563	37	10.23
pc4	C	1,458	37	12.2
pc1	C++	1,109	21	6.94
kc1	C++	845	21	15.45
kc2	C++	522	21	20.49
cm1	C++	498	21	9.83
kc3	JAVA	458	39	9.38
mw1	C++	403	37	7.69
mc2	C++	61	39	32.29
		39,158		

Fig. 1. Twelve tables of data, sorted in order of number of examples. For details on this data, see the appendix.

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

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

The research described in this paper was supported by Bogazici University research fund under grant number BAP-06HA104 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 October 1, 2007; revised XXX,

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

Note that an important aspect of this work is its *reproducibility*. All the data used in this study is freely available, on-line[2]. Reproducibility is an important methodological principle in other disciplines since it allows a community to confirm, refute, or even improve prior results. In our view, in the field of software engineering, there are all too few examples of reproduced and extended results. We would strongly encourage software engineering researchers to share data, define challenges, and to take the time to rework the results of others.

II. Background

The Figure 1 data comes from 12 NASA systems. These systems were developed in different geographical locations across North America. Within a system, the sub-systems shared some common code base but did not pass personnel or code between sub-systems.

The external validity of generalizing from NASA examples has been discussed elsewhere [6]. 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. [7] have argued that conclusions from NASA data are relevant to the general software engineering industry.

At first glance, this data seems to come from one company (NASA). However, that "company" is really an umbrella organization used to co-ordinate and fund a large and diverse set of organizations. All the Figure 1 were built by different groups, often at different geographical locations, usually for different tasks. That is, this data satisfies Kitchenham et.al.'s definition of "cross-company" data.

Previously [6], we have explored the drawbacks and advantages of learning module defect predictors from static code features. To learn such predictors, tables of examples are formed (like those in Figure 1) where one column has a boolean value for "defects detected" and the other columns describe static code features such as lines of code, number of unique symbols, or max. number of possible execution pathways. Each row in the table holds data from one "module"; i.e. the smallest unit of functionality. Depending on the language, modules may be called "functions", "methods" or "procedures".

The data mining task is to find combinations of code features that predict for the value in the defects column. The results of such data mining can be baselined against known industrial averages:

- Raffo found that industrial reviews find $pd = TR(35, 50, 65)\%$[3] of a systems errors' (for full Fagan inspections [8]) to $pd = TR(13, 21, 30)\%$ for less-structured inspections.
- Similar conclusions were made at a panel at IEEE Metrics 2002. That panel declined to endorse claims by Fagan [9] and Schull [10] regarding the efficacy of their inspection or directed inspection methods. Rather, it concluded that manual software reviews can find $\approx 60\%$ of defects [11];
- Using NASA data, our new defect predictors [6] have a probability of detection (pd) and probability of false alarm (pf) of

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

It is difficult to compare these new results with those reported by Raffo or those reported at the 2002 IEEE Metrics since they were collected by a variety of different methods. The least we can say is that automatically generated defect predictors do not appear to perform *worse* than industrial averages and might even perform somewhat better. Also, while the manual methods require manual effort, our defect predictors can be automatically and rapidly learned, then quickly and cheaply applied to very large libraries of code.

In any case, according to the terminology of Kitchenham et.al. our prior work was a within-company (WC) study: the new defect predictors were learned on 90% of the rows in one table (selected at random) then tested on the remaining rows in that table. The rest of this paper extends our methodology to the study of the relative merits of within-vs cross-company data for defect prediction.

III. Experiment #1: WC-vs-CC

A. Design

Our first WC-vs-CC experiments repeated the following procedure for all 12 example tables of Figure 1. 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 1 tables comes from systems written in "C/C++" but at least of one of the system 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, is available for download from a certain url. To replicate that scenario, we will make no use of our meta-knowledge about Figure 1. 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 10 times, randomizing the order of the rows in the table each time. In all, we ran 1200 experiments to compare WC-vs-CC:

$$(10*10\% \text{ test sets}) * (10 \text{ randomized orderings}) * (12 \text{ tables})$$

All the numeric distributions in the Figure 1 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 [6].

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 1 data [6]. This theorem offers a relationship between fragments of evidence E_i, a prior probability for a class $P(H)$, and a posteriori probability $P(H|E)$:

$$P(H|E) = P(H)/P(E) \prod_i P(E_i|H)$$

When building defect predictors, the posterior probability of each class ("defective" or "defect-free") is calculated, given the features extracted from a module. The module is assigned to the possibility with the highest probability. For numeric features, a feature's mean μ and standard deviation σ is used in a Gaussian probability function [12]:

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

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

$$pd = \quad recall = \quad D/(B+D) \tag{1}$$

$$pf = \quad C/(A+C) \tag{2}$$

$$bal = \quad balance = \quad 1 - \frac{\sqrt{(0-pf)^2 + (1-pd)^2}}{\sqrt{2}} \tag{3}$$

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

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

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 1, 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 were the target class is so rare (for more on this issue, see [6], [13]).

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:

The Mann-Whitney U test [14] 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 recommended in data mining since many of the performance distributions are non-Gaussian [15].

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 twelve tables (see appendix).

B. Results from Experiment #1

Figure 2 shows the $\{pd, pf\}$ quartile charts for CC vs WC data. 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:

- 25% of the pd values are at 100%.
- 50% of the pd values are above 90%
- 75% of the pd values are at or above 80%;
- And all the pd values are at or over 50%.

By way of comparison, recall from the above that than our previous result had a average pd of 71% [6].

To the best of our knowledge, Figure 2 are the largest pd values ever reported from this data. However, these very high pd values come at some considerable cost. Note in Figure 2 that the median false alarm rate has changed from 26% (with WC) to 52% (with CC) and the maximum pf rate now reaches 100%.

We explain these increases in pd, pf with the following speculation. Using a large training set (e.g. eleven of the tables in Figure 1) 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).

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

treatment		min	Q1	median	Q3	max
pd	CC	50	80	91	100	100
	WC	17	67	80	88	100
pf	CC	4	26	52	73	100
	WC	0	17	26	33	73

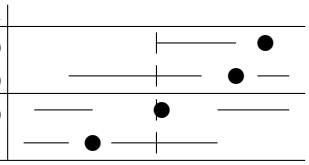

Fig. 2. Experiment #1 results. Numeric results on left; quartile charts on right. "Q1" and "Q3" denote the 25% and 75% percentile points (respectively). The upper quartile of the first row is not visible since it runs from 100% to 100%; i.e. it has zero length.

group	pd $WC \rightarrow CC$	pf $WC \rightarrow CC$	tables	$\|tables\|$
a	increased	increased	CM1 KC1 KC2 MC2 MW1 PC1 PC3 PC4	8
b	same	same	KC3	1
c	same	increased	MC1 PC2	2
d	decreased	decreased	PC5	1

Fig. 3. U test results (95% confidence): moving from WC to CC

to that pattern. Figure 3 shows a summary of results when U tests were applied to test results from each of the 12 tables, *in isolation*:

- Usually ($\frac{8}{12}$), 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.
- In another case, two tables of data (see group c) saw no change to pd but the false alarm grew worse.
- In only one case was there a clear contradiction to the majority case. In group d, containing $pc5$, using cross-company data decreased pf at the cost of also decreasing pd.

Overall, the *general result* holds in the majority of cases and is only clearly contradicted in the $pc5$ table.

$Pc5$'s anomalous behavior prompted the following investigation. Observe in Figure 1 that the two largest data sets ($pc5$ and $mc1$) contain more modules than the remaining 10 tables of data. Therefore, it seemed prudent to repeat our experiment with and without these two data sets. Figure 4 repeats the cross-company parts of experiment #1 without $pc1$, then without $pc1$ and $mc1$. Note how that the general pattern reported above (of large pd and pf values) is not altered. In fact, Figure 4 changes very little from Figure 2. It turns out that since these larger files have the lowest defect rates, they tend to introduce far more non-defective modules than defective modules. When we remove the largest file (which contains nearly half of all modules), we reduce this bias and increase the median pd (from 91% in Figure 2 to 97% in Figure 4). If we further remove the next largest file ($mc1$), then pd does not change much. However, the first pf quartile jumps from 29% to 45% since now we have removed a very large portion of non-defective samples so the model is more frequently saying "defective".

In summary, $pc5$ has enough unusual features to let us safely discount its anomalous results in Figure 3.

D. Discussion of Experiment #1

When practitioners use defect predictors with high false alarm rates, they must allocate a large portion of their debugging budget to the unfruitful exploration of erroneous alarms. Hence:

- For most software applications, very high pf rates like the CC results of Figure 2 make the predictors impractical to use.
- Therefore, we can only recommend cross-company learning for mission critical software where the extra costs associated with high false alarm rates are compensated by the associated increase in software assurance.

Without largest data set ($pc5$)

treatment		min	Q1	median	Q3	max
pd	CC	33	83	97	100	100
pf	CC	14	29	54	73	98

Without twp largest data set ($pc5 + mc1$)

treatment		min	Q1	median	Q3	max
pd	CC	33	83	97	100	100
pf	CC	12	45	55	76	98

Fig. 4. Variants on the experiment #1 results.

When budgetary considerations restrict the exploration of numerous false alarms, projects seek predictors with lower pf values and adequate pd values. Our next experiment addresses this concern.

IV. Experiment #2: Incremental WC

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 test set will only contain $1109 * 0.9 * 6.94/100 = 69$ defective modules. Despite this, $pc1$ yields an adequate median $\{pd, pf\}$ results of $\{88, 34.5\}\%$.

Experiment #2 was therefore designed to check how *little* data is required to learn defect predictors. Experiment #2 was essentially the same as the last experiment, but without treatment #1 (the cross-company study). Instead, experiment #2 took the 12 example tables of Figure 1 and learned predictors using:

- Treatment 3 (reduced WC): a randomly selected subset of the 90% rows of this table;

Specially, 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 100 rows not used in training (selected at random).

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 #2

Recall that Equation 3 defined "balance" to be a combination of $\{pd, pf\}$ that decreases if pd decreases or pf increases. As shown in Figure 5, 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 $pc2$, as the training set size increases (moving right along the x-axis) the dots showing the on balance performance start spreading out. A similar, but smaller, *spread effect* can be see in $kc2$ and $mc1$.

The Mann-Whitney U test was applied to check the visual trends seen in Figure 5. 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 is 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 seven tables $\{cm1, kc2, kc3, mw1, pc3, pc4\}$, $min = 100$;
- In $\{kc1, pc1\}$, $min = \{200, 300\}$ instances, respectively.

In other tables of data, min was much larger. In $\{pc2, mc1, pc5\}$ the min values were found at $\{4900, 8300, 11000\}$, respectively. However, much smaller training set sizes performed nearly as well as these larger training sets:

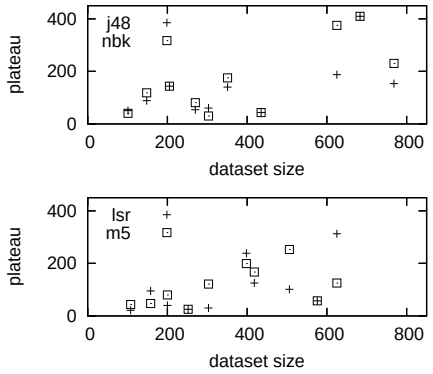

Fig. 6. Y-axis shows plateau point after learning from data sets that have up to X examples (from [17]). The top plot shows results from using Naive Bayes (nbk) or a decision tree learner (j48) [18] to predict for discrete classes. Bottom plot shows results from using linear regression (lsr) or model trees (m5) [19] to learn predictors for continuous classes. In this study, data sets were drawn from the UC Irvine data repository [20].

- In $pc5$, predictors learned from 300 examples only lost to other sizes twice in 169 trials;
- In $mc1$, predictors learned from 400 examples only lost to other sizes once out of 92 trials);
- In $pc2$, predictors learned from 800 examples only lost to other size twice out of 53 trials.

We explain the experiment #2 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. Fenton offers an insightful example where *the same* functionality is achieved using *different* programming language constructs resulting in *different* static measurements for that module [16]. We would characterize such static code features as having *limited information content*. 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 #2

This section checks for precedents on the Experiment #2 results and can be skipped at first reading of this paper.

There is also some evidence that the results of Experiment 2 (that performance improvements stop after a few hundred examples) has been seen previously in the data mining literature (caveat: to the best of our knowledge, this is 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 [21].
- Orrego [17] 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 6, many of those plateaus where 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 2.

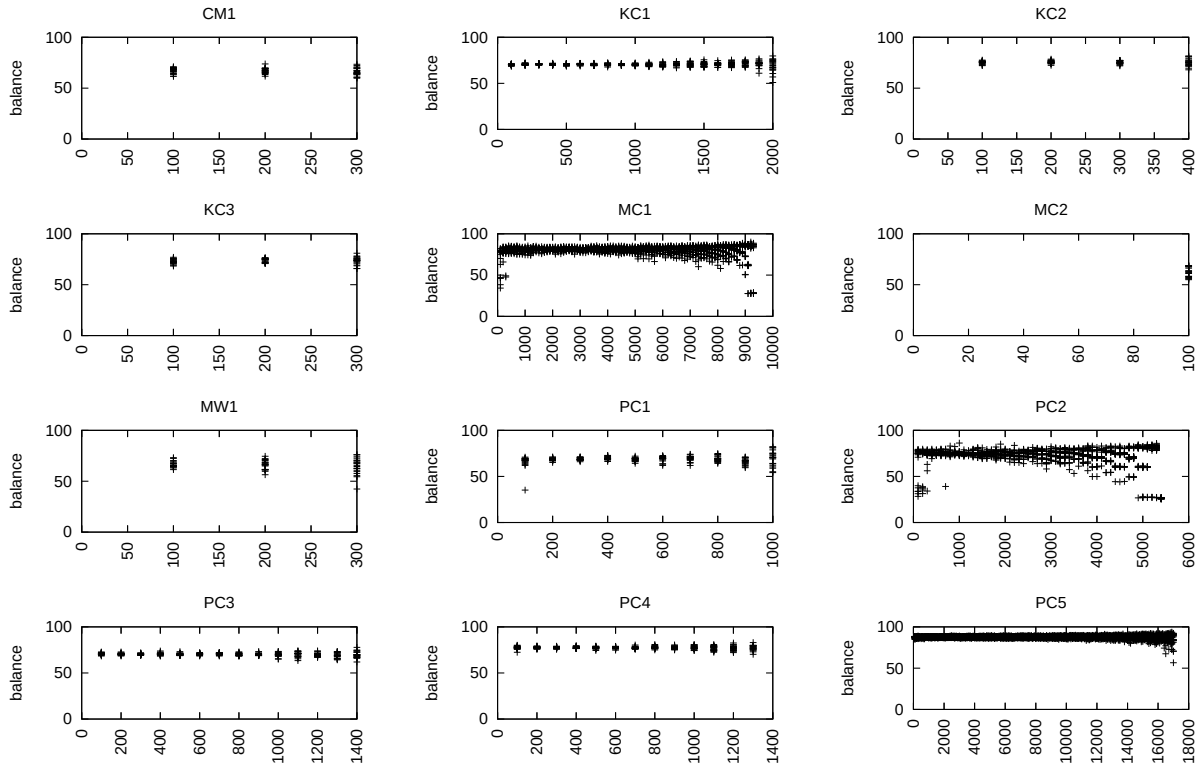

Fig. 5. Results from experiment #2. Training set size grows in units of 100 examples, moving left to right over the x-axis. The MC2 results only appear at the maximum x-value since MC2 has less than 200 examples.

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 [5] 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. 7. An estimate of the effort required to build and test 100 modules.

D. Discussion of Experiment #2

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 7 estimates that the effort required to build and test 100 modules may be as little as 2.4 to 3.7 months.

V. Conclusion

The value of cross-vs-within-company data (CC vs WC) for effort estimation is an open question. Three of the papers reviewed in the introduction comment that it can be hard to use CC data due to different data collection practices at different organizations [1], [2], [22].

Our hypothesis was that the uncertainties seen in studies of CC-vs WC-data may be due to ambiguity in the definitions of the effort estimation features. Static code features, on the other hand, can be collected in a uniform, rapid and automatic manner. For example, in this study, we have found a clear and unambiguous conclusion amongst our static code features:

- CC-data dramatically increases the probability of detecting defective modules;
- But CC-data also dramatically increases the false alarm rate.
- Therefore, we can only recommend cross-company learning for mission critical software where the extra costs associated with high false alarm rates are compensated by the associated increase in software assurance.

Another clear and unambiguous conclusion from static code features is that, for the purposes of defect prediction, the need for CC data may be less than previously believed. Kitchenham et.al. list many reasons to use CC-data including these two points:

- 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.

Our incremental WC results suggest that, in the case of defect prediction, these two points may not be compelling arguments. In most of our experiments, as little 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.

References

[1] 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.

[2] M. E, 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] F. Shull, V. B. ad 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] I. H. Witten and E. Frank, *Data mining. 2nd edition*. Los Altos, US: Morgan Kaufmann, 2005.

[13] 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.

[14] 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 online at http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&hand%le=euclid.aoms/1177730491.

[15] 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.

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

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

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

[19] 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.

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

[21] 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.

[22] M. Shepperd, "Software project economics: A roadmap," in *International Conference on Software Engineering 2007: Future of Software Engineering*, 2007.

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

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

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

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. 8. Features used in this study.

APPENDIX

The data tables used in this study can be downloaded from the PROMISE repository[4]. The static code features of our 12 tables of data are shown in Figure 8. These features divide into lines of code features, Halstead features, and McCabe features.

The Halstead features were derived by Maurice Halstead in 1977. He argued that modules that are hard to read are more likely to be fault prone [23]. Halstead estimates reading complexity by counting the number of operators and operands in a module: see the h features of Figure 8. These three raw h Halstead features were then used to compute the H: the eight derived Halstead features using the equations shown in Figure 8. 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.

An alternative to the Halstead features 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 [24]. The first three lines of Figure 8 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 [25]. 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" [25]). $ev(G) = v(G) - m$ where m is the number of subflowgraphs of G that are D-structured primes [25]. Finally, the *design complexity* $(iv(G))$ of a module is the cyclomatic complexity of a module's reduced flow graph.

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