Regularities in Learning Defect Predictors

Burak Turhan[1], Ayse Bener[2], and Tim Menzies[3]

[1] Department of Information Processing Science, University of Oulu, Oulu 90014, Finland
burak.turhan@oulu.fi
[2] Department of Computer Engineering, Boğaziçi University, Istanbul 34342, Turkey
bener@boun.edu.tr
[3] Lane Dept. of CS&EE, West Virginia University, Morgantown, WV, USA
tim@menzies.us

Abstract. Collecting large consistent data sets of real world software projects from a single source is problematic. In this study, we show that bug reports need not necessarily come from the local projects in order to learn defect prediction models. We demonstrate that using imported data from different sites can make it suitable for predicting defects at the local site. In addition to our previous work in commercial software, we now explore open source domain with two versions of an open source anti-virus software (Clam AV) and a subset of bugs in two versions of GNU gcc compiler, to mark the regularities in learning predictors for a different domain. Our conclusion is that there are surprisingly uniform assets of software that can be discovered with simple and repeated patterns in local or imported data using just a handful of examples.

Keywords: Defect prediction, Code metrics, Software quality, Cross-company.

1 Introduction

It is surprisingly difficult to find relevant data within a single organization to fully specify the internal parameters inside a complete software process model. For example, after 26 years of trying, Barry Boehm's team of researchers from the University of Southern California collected less than 200 sample projects for their COCOMO effort estimation database [1].

There are many reasons for this, not the least being the business sensitivity associated with the data. Software projects are notoriously difficult to control: recall the 1995 report of the Standish group that described a $250 billion dollar American software industry where 31% of projects were canceled and 53% of projects incurred costs exceeding 189% of the original estimate [2]. Understandably, corporations are reluctant to expose their poor performance to public scrutiny.

Despite this data shortage, remarkably effective predictors for software products have been generated. In previous work we have built:

- *Software effort estimators* using only very high-level knowledge of the code being developed [3] (typically, just some software process details). Yet these estimators offer predictions that are remarkably close to actual development times [4].

M. Ali Babar, M. Vierimaa, and M. Oivo (Eds.): PROFES 2010, LNCS 6156, pp. 116–130, 2010.
© Springer-Verlag Berlin Heidelberg 2010

- *Software defect predictors* using only static code features. Fenton (amongst others) argues persauively that such features are very poor charactizations of the inner complexities of software modules [5]. Yet these seeingly naive defect predictors out-perform current industrial best-practices [6], [7].

The success of such simple models seems highly unlikely. Organizations can work in different domains, have different process, and define/measure defects and other aspects of their product and process in different ways. Worse, all to often, organizations do not precisely define their processes, products, measurements, etc. Nevertheless, it is true that very simple models suffice for generating approximately correct predictions for (say) software development time [4], the location of software defects [6].

One candidate explanation for the strange predictability in software development is that: *Despite all the seemingly random factors influencing software construction, the net result follows very tight statistical patterns.* Other researchers have argued for similar results [8], [9], [10], [11], [12] but here we offer new evidence. The performance of a data miner improves as the size of the training set grows. At some point, the performance *plateaus* and further training data does not improve that performance. Previously, we showed in commercial domain (NASA aerospace software and white-goods controller software) that for defect prediction, plateaus occur remarkably early (after just 30 examples) [7]. Furthermore, we showed that with certain limitations, defect predictors can be learned across projects, i.e. training on project A and testing on project B.

In this paper, we investigate the same effects in open source domain with two different products. Observing an effect in different entities of commercial domain (NASA, white-goods) might be a coincidence. However, observing the same effect in a relatively unrelated domain, we will no longer be able to dismiss the effect as quirks in one domain. Therefore, the goal of this paper is to further investigate the issue of regularities in learning defect predictors by replicating our previous experiments (i.e. [6], [13], [7]) on an additional domain (open-source) in order to strengthen the evidence base for the following assertions:

- The regularities we observe in software are very regular indeed;
- We can depend on those regularities to generate effective defect predictors using *minimal information from projects*[1].

The rest of this paper is structured as follows. We give examples of statistical patterns from general software research and briefly discuss the role of defect predictors in Section 2. Then we focus on defect predictors in detail and introduce observed regularities in defect prediction research (for commercial domain) in Section 3. Section 4 extends the analyses of these regularities to open source domain. Discussions of our observations are given in Section 5 and we conclude our research in Section 6. Please note that the results of Section 3 have appeared previously [6], [13], [7]. The rest of this paper is new work.

[1] This second point is a strong endorsement for our approach since, as discussed above, it can be very difficult to access details and extensive project data from real world software development.

2 Background

2.1 Examples of Regularities in General SE Research

Previous research reports much evidence that software products confirm tightly to simple and regular statistical models, For example, Veldhuizen shows that library reuse characteristics in three unix based systems (i.e. Linux, SunOS and MacOS X), can be explained by Zipf's Law, that is most frequently used library routines [14] are inversely proportional to their frequency ranks. As shown in Figure 1, the distribution is highly regular.

Figure 2 shows a summary of our prior work on effort estimation [4] using the COCOMO features. These features lack detailed knowledge of the system under development. Rather, they are just two dozen ultra-high-level descriptors of (e.g.) develeper

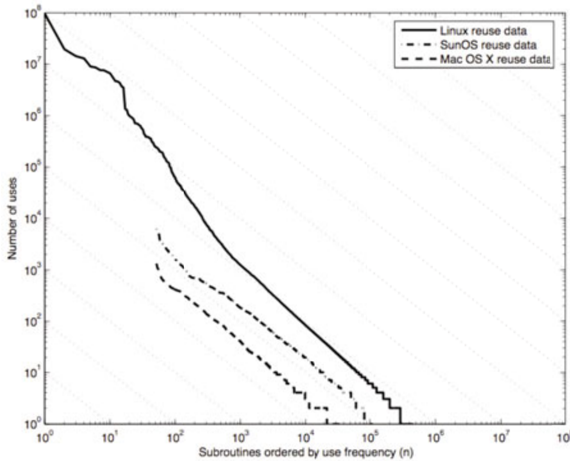


Fig. 1. Distribution of reuse frequencies in three unix based systems [14]

	$100 * (pred - actual)/actual$		
	50% percentile	65% percentile	75% percentile
mode=embedded	-9	26	60
project=X	-6	16	46
all	-4	12	31
year=1975	-3	19	39
mode=semi-detached	-3	10	22
ground.systems	-3	11	29
center=5	-3	20	50
mission.planning	-1	25	50
project=gro	-1	9	19
center=2	0	11	21
year=1980	4	29	58
avionics.monitoring	6	32	56
median	-3	19	39

Fig. 2. 158 effort estimation methods applied to 12 subsets of the COCO81 data

experience, platform violatily, software process maturity, etc. In Figure 2, different effort estimation methods are applied to the COC81 data[2] used by Boehm to develop the original COCOMO effort estimation model [15]. Twenty times, ten test instances were selected at random and effort models were built from the remaining models using 158 different estimation methods[3]. The resulting predictions were compared to the actual estimation times using relative error. COC81's data can be divided into the 12 (overlapping) subsets shown left-hand-side of Figure 2. The right-hand-columns of Figure 2 show MRE at the median, 65%, and 75% percentiles. The median predictions are within within 3% of the actual. Such a close result would be impossible if software did not conform to very tight statistical regularities.

Figure 3 shows Koru and Liu's analysis of two large-scale open source projects (K-office and Mozilla). As shown in those figures, these different systems confirm to nearly an identical Pareto distribution of change-prone classes: 80% of changes occur in 20% of classes [12]. The same 80:20 changes:fault distribution has been observed by Ostrand, Weyuker and Bell in very large scale telecommunication projects from AT&T [8], [9], [10], [11]. Furthermore, the defect trends in Eclipse also follows similar patterns, where they are explained better by a Weibull distribution [17] .

2.2 On Learning Defect Predictors

Figures 1, 2, 3 and 4 show that statistical regularities simplifies predictions of certain kinds of properties. Previously [6], we have exploited those regularities with data miners that learn defect predictors from static code attribtues. Those predictors were learned either from projects previously developed in the same environment or from a continually expanding base of current project's artifacts.

To do so, tables of examples are formed where one column has a boolean value for "defects detected" and the other columns describe software features such as lines of code; number of unique symbols [18]; or max. number of possible execution pathways [19]. Each row in the table holds data from one "module", the smallest unit of functionality. Depending on the language, these may be called "functions", "methods", or "procedures". The data mining task is to find combinations of features that predict for the value in the defects column.

The value of static code features as defect predictors has been widely debated. Some researchers vehemently oppose them [20], [21], while many more endorse their use [22], [23], [6], [24], [25], [26], [27], [28]. Standard verification and validation (V&V) textbooks [29] advise using static code complexity attributes to decide which modules are worthy of manual inspections. The authors are aware of several large government software contractors that won't review software modules *unless* tools like the McCabe static source code analyzer predicts that they exhibit high code complexity measures.

Nevertheless, static code attributes can never be a full characterization of a program module. Fenton offers an insightful example where *the same* functionality is achieved using *different* programming language constructs resulting in *different* static measurements for that module [5]. Fenton uses this example to argue the uselessness of static code attributes for fault prediction.

[2] Available from `http://promisedata.org`

[3] For a description of those methods, see [16].

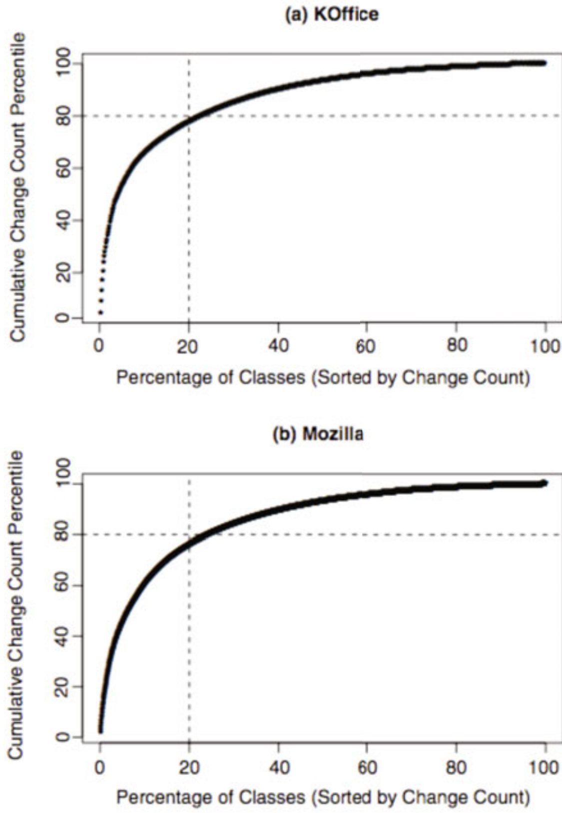


Fig. 3. Distribution of change prone class percentages in two open source projects [12]

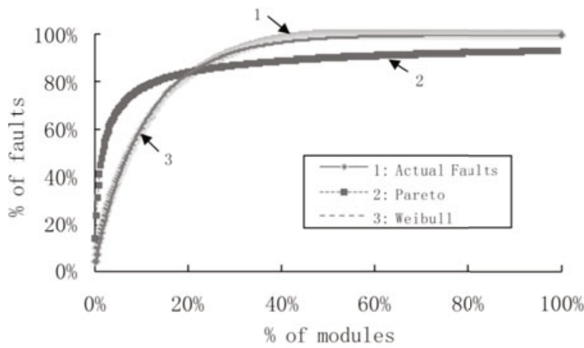


Fig. 4. Distribution of faulty modules in Eclipse [17]

Using NASA data, our fault prediction models find defect predictors [6] with a probability of detection (pd) and probability of false alarm (pf) of $mean(pd, pf) = (71\%, 25\%)$. These values should be compared to baselines in data mining and industrial practice. Raffo (personnel communication) found that industrial reviews find $pd = TR(35, 50, 65)\%$[4] of a systems errors' (for full Fagan inspections [30]) to $pd = TR(13, 21, 30)\%$ for less-structured inspections. Similar values were reported at a IEEE Metrics 2002 panel. That panel declined to endorse claims by Fagan [31] and Schull [32] regarding the efficacy of their inspection or directed inspection methods. Rather, it concluded that manual software reviews can find $\approx$60% of defects [33];

That is, contrary to the pessimism of Fenton, our $(pd, pf) = (71, 25)\%$ results are better than currently used industrial methods such as the $pd{\approx}60\%$ reported at the 2002 IEEE Metrics panel or the $median(pd) = 21..50$ reported by Raffo. Better yet, automated defect predictors can be generated with a fraction of the effort of alternative methods, even for very large systems [24]. Other methods such as formal methods or manual code reviews may be more labor-intensive. Depending on the review method, 8 to 20 lines of code (LOC) per minute can be inspected. This effort repeats for all members of the review team (typically, four or six [34]).

3 Regularities in Defect Predictors in Commercial Domain

In prior work [13], [7], we have used the NASA and SOFTLAB data of Figure 5 to explore learning defect predictors using data miners. To learn defect predictors we use a Naive Bayes data miner since prior work [6] could not find a better data miner for learning defect predictors. In all our experiments, the data was pre-processed as follows:

- Since the number of features in each data table is not consistent, we restricted our data to only the features shared by all data sets.
- Previously [6], we have observed that all the numeric distributions in the Figure 5 data are exponential in nature. It is therefore useful to apply a "log-filter" to 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 $ln(0.000001)$).

Inspired by the recent systematic review of within vs cross company effort estimation studies by Kitchenham et al. [35], we have done extensive experiments on Promise data tables to analyze predictor behavior using a) local data (within the same company) b) imported data (cross company). For each NASA and SOFTLAB table of Figure 5, we built test sets from 10% of the rows, selected at random. Then we learned defect predictors from:

- the other 90% rows of the corresponding table (i.e. local data).
- 90% rows of the other tables combined (i.e. imported data).

We repeated this procedure 100 times, each time randomizing the order of the rows in each table, in order to control *order effects* (where the learned theory is unduly affected

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

source	data	(# modules) examples	features	%defective
NASA	pc1	1,109	22	6.94
NASA	kc1	845	22	15.45
NASA	kc2	522	22	20.49
NASA	cm1	498	22	9.83
NASA	kc3	458	22	9.38
NASA	mw1	403	22	7.69
NASA	mc2	61	22	32.29
SOFTLAB	ar3	63	29	12.70
SOFTLAB	ar4	107	29	18.69
SOFTLAB	ar5	36	29	22.22
OPEN SOURCE	cav90	1184	26	0.40
OPEN SOURCE	cav91	1169	26	0.20
OPEN SOURCE	gcc	116	26	100.0

Fig. 5. Datasets used in this study

by the order of the examples). We measured the performance of predictor 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:

$$pd = \quad recall = \quad D/(B + D) \tag{1}$$
$$pf = \qquad C/(A + C) \tag{2}$$
$$bal = balance = 1 - \frac{\sqrt{(0 - pf)^2 + (1 - pd)^2}}{\sqrt{2}} \tag{3}$$

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 ($pd = 1$). We then used the Mann-Whitney U test [36] in order to test for statistical difference.

Our results are visualized in Figure 6 as *quartile chaorts*. Quartile charts are generated from sorted sets of results, divided into four subsets of (approx) same cardinality. For example these numbers have four quartiles:

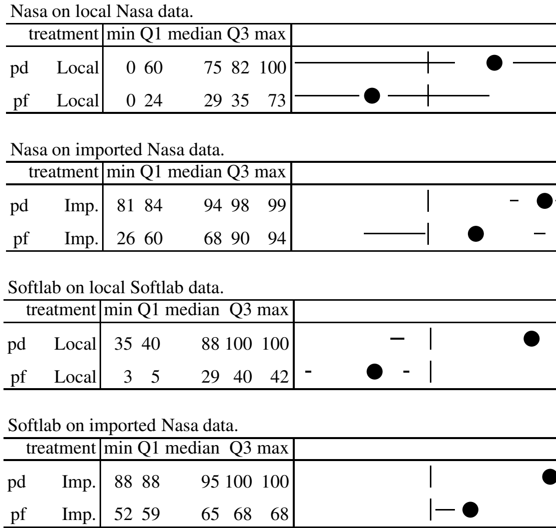
$$\{\overbrace{4, 7, 15}^{q1}, 20, 31, \overbrace{40}^{median}, 52, 64, \overbrace{70, 81, 90}^{q4}\}$$

These quartiles can be drawn as follows: 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. For example, the above numbers can be drawn as:

0% — ● | —— 100%

In a finding consistent with our general thesis (that software artifacts conform to very regular statistical patterns), Figure 6 shows the same stable and useful regularity occuring in both seven NASA data sets and three SOFTLAB data sets [7].:

- Using imported data dramatically increased the probability of detecting defective modules (for NASA: 74% to 94% median pd; for SOFTLAB: 88% to 95% median pd);



Nasa on local Nasa data.

	treatment	min	Q1	median	Q3	max
pd	Local	0	60	75	82	100
pf	Local	0	24	29	35	73

Nasa on imported Nasa data.

	treatment	min	Q1	median	Q3	max
pd	Imp.	81	84	94	98	99
pf	Imp.	26	60	68	90	94

Softlab on local Softlab data.

	treatment	min	Q1	median	Q3	max
pd	Local	35	40	88	100	100
pf	Local	3	5	29	40	42

Softlab on imported Nasa data.

	treatment	min	Q1	median	Q3	max
pd	Imp.	88	88	95	100	100
pf	Imp.	52	59	65	68	68

Fig. 6. Quartile charts for NASA and SOFTLAB data [7]. Numeric results on left; quartile charts on right. "Q1" and "Q3" denote the 25% and 75% percentile points respectively.

– But imported data also dramatically increased the false alarm rate (for NASA: 29% to 68% median pf; for SOFTLAB: 29% to 65% pd) . Our suspicion is that the reason for the high false alarm rates was the irrelevancies introduced by imported data.

We have then designed a set of experiments for NASA data tables to see the effects of sampling strategies on the performance and to determine the lower-limit on the number of samples for learning defect predictors, i.e. the point where a plateau is observed in the performance measure. In those experiments we applied over/under and micro-sampling (see [13]) to the data tables and observed that:

– the performance of predictors does not improve with over sampling.
– under-sampling improves the performance of certain predictors, i.e. decision trees, but not of Naive Bayes [13].
– with micro-sampling, the performance of predictors stabilize after a mere number of defective and defect-free examples, i.e. 50 to 100 samples.

The last observation suggests that the number of cases that must be reviewed in order to arrive at the performance ceiling of a defect predictor is very small: as low as 50 randomly selected modules (25 defective and 25 non-defective). In Figure 7 for Nasa tables, we visualize number of training examples (increments of 25) vs. balance performance of Naive Bayes predictor. It is clear that the performance does not improve with more training examples, indeed it may deteriorate with more examples.

We now report the results of the same experiment for SOFTLAB data. Note that SOFTLAB tables *ar3, ar4* and *ar5* have {36,63,107} modules respectively, with a total

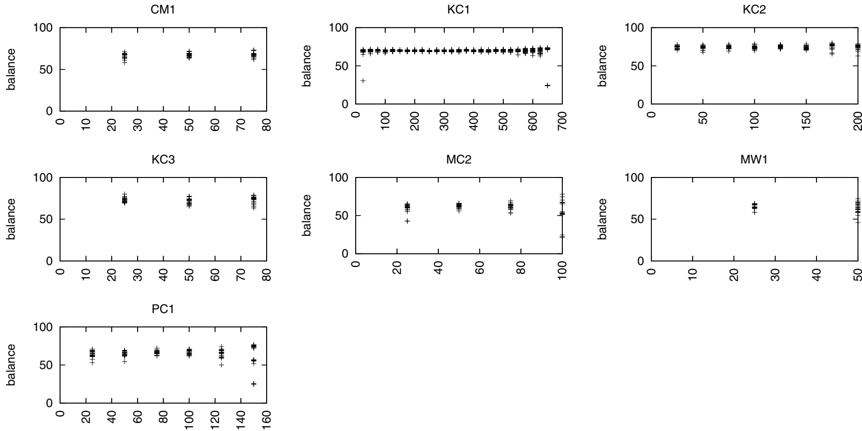


Fig. 7. Micro-sampling results for NASA data [7]

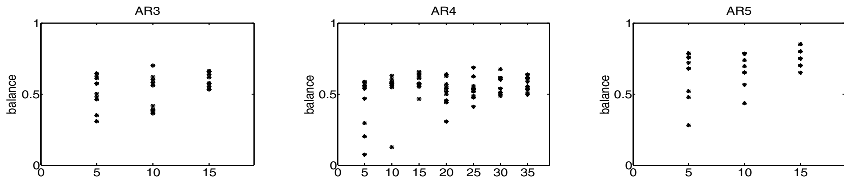


Fig. 8. Micro-sampling results for SOFTLAB data

of 206 modules of which only 36 are defective. Thus, local results in Figure 6 are achieved using a minimum of $(36 + 63) * 0.90 = 90$ and a maximum of $(107 + 63) * 0.90 = 153$ examples. Nevertheless we repeat the micro-sampling experiment for SOFTLAB data tables, with increments of 5 due to relatively lower number of defects. In Figure 8, we plot the results for SOFTLAB data tables. We observe the same pattern as in Figure 6: performance tends to stabilize after a small number of training examples.

Results for NASA and SOFTLAB data tables suggest that practical defect predictors can be learned using only a handful of examples. In order to allow for generalization, it is appropriate to question the external validity of the above two results: The data used for those results come from very different sources:

- The SOFTLAB data were collected from a Turkish white-goods manufacturer (see the the datasets ($\{$ar3, ar4, ar5$\}$) from Figure 5) building controller software for a washing machine, a dishwasher and a refrigerator.
- On the other hand, NASA software are ground and flight control projects for aerospace applications, each developed by different teams at different locations .

The development practices from these two organizations are very different:

- The SOFTLAB software were built in a profit- and revenue-driven commercial organization, whereas NASA is a cost-driven government entity.
- The SOFTLAB software were are developed by very small teams (2-3 people) working in the same physical location while the NASA software was built by much larger team spread around the United States.
- The SOFTLAB development was carried out in an ad-hoc, informal way rather than formal, process oriented approach used at NASA.

The fact that the same defect detection patterns hold for such radically different kinds of organization is a strong argument for the external validity of our results. However, an even stronger argument would be that the patterns we first saw at NASA/ SOFTLAB are also found in software developed at other sites. The rest of this paper collects evidence for that stronger argument.

4 Validity in Open Source Domain

This section checks for the above patterns in two open source projects:

- two versions of an anti-virus project: Clam AV v0.90 and v0.91;
- a subset of defective modules of the GNU gcc compiler.

In Figure 5, these are denoted as cav90, cav91 and gcc, respectively. Note that these are very different projects, build by different developers with very different purposes. Also note that the development processes for the open source projects are very different to the NASA and SOFTLAB projects studied above. Whereas NASA and SOFTLAB were developed by centrally-controlled top-down management teams, cav90, cav91 and gcc were build in a highly distributed manner. Further, unlike our other software, gcc has been under extensive usage and maintenance for over a decade.

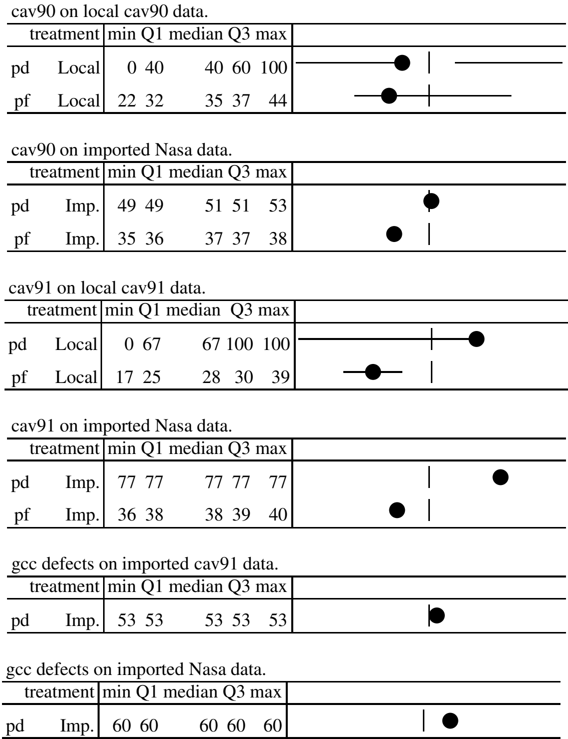
Local/ imported data experiments are once more applied on cav90, cav91 and gcc data tables and the results are visualized in Figure 9. We again used Nasa tables as imported data, since they provide a large basis with 5000+ samples. Note that partial gcc data includes only a sample of defects, thus we are not able to make a 'local data' analysis for gcc. Rather, we report the detection rates of predictors built on imported data from Nasa and cav91. These predictors can correctly detect up to median 60% of the subset of bugs that we were able to manually match to functional modules.

Recall that cav91 has a defect rate of 0.20%. The probability of detection rates for cav91 are median 67% and 77% for local and imported data respectively, which is another evidence on the usefulness of statistical predictors.

At the first glance, the patterns in Figure 9 seem a bit different than those in Figure 6. There are still increases in probability of detection and false alarms rates, though not dramatically. However, this is not a counter example of our claim. We explain this behavior with the following assertions:

For our experiments:

- in commercial software analysis, local data corresponds to single projects developed by a relatively small team of people in the same company and with certain business knowledge.



cav90 on local cav90 data.

	treatment	min	Q1	median	Q3	max
pd	Local	0	40	40	60	100
pf	Local	22	32	35	37	44

cav90 on imported Nasa data.

	treatment	min	Q1	median	Q3	max
pd	Imp.	49	49	51	51	53
pf	Imp.	35	36	37	37	38

cav91 on local cav91 data.

	treatment	min	Q1	median	Q3	max
pd	Local	0	67	67	100	100
pf	Local	17	25	28	30	39

cav91 on imported Nasa data.

	treatment	min	Q1	median	Q3	max
pd	Imp.	77	77	77	77	77
pf	Imp.	36	38	38	39	40

gcc defects on imported cav91 data.

	treatment	min	Q1	median	Q3	max
pd	Imp.	53	53	53	53	53

gcc defects on imported Nasa data.

	treatment	min	Q1	median	Q3	max
pd	Imp.	60	60	60	60	60

Fig. 9. Quartile charts for OPENSOURCE data

– in commercial software analysis, imported data corresponds to a variety of projects developed by various people in different companies and spans a larger business knowledge.
– in open source software analysis, local data corresponds to single projects developed by a larger team of people at different geographical locations with various backgrounds.

We argue that, the above assertions differentiate the meaning of local data for open source projects from commercial projects. The nature of open source development allows the definition of local data to be closer to commercial imported data, since both are developed by people at different sites with different background. That's the reason why adding commercial imported data does not add as much detection capability as it does for commercial local data. Furthermore, the false alarms are not that high since there are less irrelevancies in local open source data than commercial imported data, which is the cause of high false alarms. That's because open source local data consist of a single project and commercial imported data consist of several projects, which introduce irrelevancies.

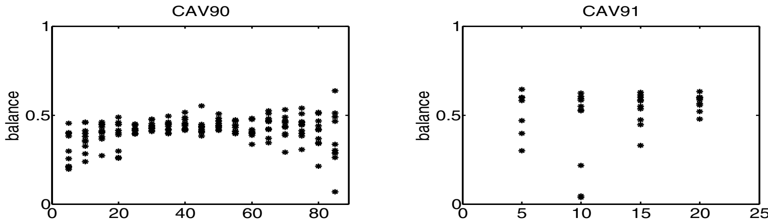


Fig. 10. Micro-sampling results for OPEN-SOURCE data

We also repeat the 10*10 way cross-validation micro-sampling experiment for cav90 and cav91 data tables, this time with increments of 5 due to limited defect data (Figure 7 used increments of 25). In Figure 10 we see a similar pattern:

- Balance values tend to converge using only 20-30 samples of defective and non-defective modules.
- Using less examples produce unstable results, since there are not enough samples to learn a theory.
- For cav90, using more than 60 training examples affects the stability of the results.

The importance of Figure 10 results is that our results in two commercial domains, considering the minimum number of examples to train a predictor, are once more validated in a third, open-source domain.

5 Summary of Results and Limitations

We can summarize our results on commercial and open-source data as follows:

- Using imported data increases the detection capability of predictors.
- Predictors can be learned using only a handful of data samples. In both commercial and open-source domain the performance of predictors converge after a small number of training examples.
- These results are generalizable through different cultural and organizational entities, since same patterns are observed in NASA, SOFTLAB and OPEN-SOURCE data tables.

Yet, these results have associated limitations. For instance imported data increase false alarm rates as well. We tried to avoid this by applying a relevancy filtering. This effect is less visible in open source domain for the reasons discussed above. We also observed that using larger training instances may cause variations in predictor performances. This should be controlled through incremental experiments on the training set size. Finally, though the data analyzed in this paper spans a large space of software products and provide strong evidence in favor of regularity, it is not possible to claim a formal generalization of our results due to their empirical nature. Nevertheless, please note that

these result are observed: in a variety of projects (i.e. 7 NASA, 3 SOFTLAB, 2 OPEN-SOURCE) from different domains (i.e. commercial and open-source) spanning a wide range time interval (i.e. from 1980's to 2007). Therefore, we assert that:

There exists repeated patterns in software that can be discovered and explained by simple models with minimal information no matter what the underlying seemingly random and complex processes or phenomena are.

This assertion should be processed carefully. We do not mean that *everything* about software can be controlled through patterns. Rather, we argue that these patterns exist and are easy to discover. It is practical and cost effective to use them as guidelines in order to understand the behavior of software and to take corrective actions. In this context we will propose two directions for further research in our conclusions.

6 Conclusions

Building defect predictor models is easy, fast and effective in guiding manual test effort to correct locations. What is more important is that these automated analysis methods are applicable in different domains. We have previously shown their validity in two commercial domains and in this paper we observe similar patterns in two projects of the open source domain. We have also shown that although these models are sensitive to the information level in the training data (i.e. local/ imported), they are not affected by the organizational differences that generate them.

Based on our results, we argue that no matter how complicated the underlying processes may seem, software has a statistically predictable nature. Going one step further, we claim that the patterns in software are not limited to individual projects or domains, rather they are generalizable through different projects and domains. Therefore, we suggest two directions for further research:

- One direction should explore software analysis using rigorous formalisms that offer ironclad gaureentees of the corrections of a code (e.g. interactive theorem proving, model checking, or the correctness preserving transoftations discussed by Doug Smith[5]). This approach is required for the analysis of mission-critical software that must always function correctly.
- Another direction should explore automatic methods with a stronger focus on maximizing the effectiveness of the analysis while minimizing the associated cost.

There has been some exploration of the second approach using lightweight formal methods (e.g. [37]) or formal methods that emphasis the usability and ease of use of the tool (e.g. [38], [39]). However, our results also show that there exists an under-explored space of extrememly cost-effective automated analysis methods.

Acknowledgements

This research is partially supported: by Tubitak under grant number EEEAG 108E014 at Bogazici University, by Tekes under Cloud Software Program at University of Oulu, and

[5] Keynote address, ASE'07.

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.

References

1. Menzies, T., Elrawas, O., Barry, B., Madachy, R., Hihn, J., Baker, D., Lum, K.: Accurate estimates without calibration. In: International Conference on Software Process (2008)
2. The Standish Group Report: Chaos (1995)
3. Menzies, T., Port, D., Chen, Z., Hihn, J., Stukes, S.: Specialization and extrapolation of induced domain models: Case studies in software effort estimation. In: IEEE ASE 2005 (2005)
4. Menzies, T., Chen, Z., Hihn, J., Lum, K.: Selecting best practices for effort estimation. IEEE Transactions on Software Engineering (2006)
5. Fenton, N.E., Pfleeger, S.: Software Metrics: A Rigorous & Practical Approach. International Thompson Press (1997)
6. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering (2007)
7. Turhan, B., Menzies, T., Bener, A.B., Di Stefano, J.: On the relative value of cross-company and within-company data for defect prediction. Empirical Softw. Engg. 14(5), 540–578 (2009)
8. Bell, R., Ostrand, T., Weyuker, E.: Looking for bugs in all the right places. In: ISSTA 2006: Proceedings of the 2006 international symposium on Software testing and analysis (2006)
9. Ostrand, T., Weyuker, E., Bell, R.: Where the bugs are. ACM SIGSOFT Software Engineering Notes 29(4) (2004)
10. Ostrand, T., Weyuker, E.: The distribution of faults in a large industrial software system. In: ISSTA 2002: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (2002)
11. Ostrand, T., Weyuker, E., Bell, R.: Automating algorithms for the identification of fault-prone files. In: ISSTA 2007: Proceedings of the 2007 international symposium on Software testing and analysis (2007)
12. Koru, A.G., Liu, H.: Identifying and characterizing change-prone classes in two large-scale open-source products. JSS (2007)
13. Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., Jiang, Y.: Implications of ceiling effects in defect predictors. In: Proceedings of PROMISE 2008 Workshop, ICSE (2008)
14. Veldhuizen, T.L.: Software libraries and their reuse: Entropy, kolmogorov complexity, and zipf's law. arXiv cs.SE (2005)
15. Boehm, B.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs (1981)
16. Jalali, O.: Evaluation bias in effort estimation. Master's thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia University (2007)
17. Zhang, H.: On the distribution of software faults. IEEE Transactions on Software Engineering 34(2), 301–302 (2008)
18. Halstead, M.: Elements of Software Science. Elsevier, Amsterdam (1977)
19. McCabe, T.: A complexity measure. IEEE Transactions on Software Engineering 2(4), 308–320 (1976)
20. Fenton, N., Ohlsson, N.: Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software Engineering, 797–814 (2000)

21. Shepperd, M., Ince, D.: A critique of three metrics. The Journal of Systems and Software 26(3), 197–210 (1994)
22. Khoshgoftaar, T.M., Seliya, N.: Fault prediction modeling for software quality estimation: Comparing commonly used techniques. Empirical Software Engineering 8(3), 255–283 (2003)
23. Tang, W., Khoshgoftaar, T.M.: Noise identification with the k-means algorithm. In: ICTAI, pp. 373–378 (2004)
24. Nagappan, N., Ball, T.: Static analysis tools as early indicators of pre-release defect density. In: ICSE 2005, St. Louis (2005)
25. Nikora, A., Munson, J.: Developing fault predictors for evolving software systems. In: Ninth International Software Metrics Symposium, METRICS 2003 (2003)
26. Porter, A., Selby, R.: Empirically guided software development using metric-based classification trees. IEEE Software, 46–54 (1990)
27. Srinivasan, K., Fisher, D.: Machine learning approaches to estimating software development effort. IEEE Trans. Soft. Eng., 126–137 (1995)
28. Tian, J., Zelkowitz, M.: Complexity measure evaluation and selection. IEEE Transaction on Software Engineering 21(8), 641–649 (1995)
29. Rakitin, S.: Software Verification and Validation for Practitioners and Managers, 2nd edn. Artech House (2001)
30. Fagan, M.: Design and code inspections to reduce errors in program development. IBM Systems Journal 15(3) (1976)
31. Fagan, M.: Advances in software inspections. IEEE Trans. on Software Engineering, 744–751 (1986)
32. Shull, F., Rus, I., Basili, V.: How perspective-based reading can improve requirements inspections. IEEE Computer 33(7), 73–79 (2000)
33. Shull, F., Basili, V., Boehm, B., Brown, A., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., Zelkowitz, M.: What we have learned about fighting defects. In: Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada, pp. 249–258 (2002)
34. Menzies, T., Raffo, D., Setamanit, S., Hu, Y., Tootoonian, S.: Model-based tests of truisms. In: Proceedings of IEEE ASE 2002 (2002)
35. Kitchenham, B.A., Mendes, E., Travassos, G.H.: Cross- vs. within-company cost estimation studies: A systematic review. IEEE Transactions on Software Engineering, 316–329 (2007)
36. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Statist. 18(1), 50–60 (1947)
37. Easterbrook, S., Lutz, R.R., Covington, R., Kelly, J., Ampo, Y., Hamilton, D.: Experiences using lightweight formal methods for requirements modeling. IEEE Transactions on Software Engineering, 4–14 (1998)
38. Heimdahl, M., Leveson, N.: Completeness and consistency analysis of state-based requirements. IEEE Transactions on Software Engineering (1996)
39. Heitmeyer, C., Jeffords, R., Labaw, B.: Automated consistency checking of requirements specifications. ACM Transactions on Software Engineering and Methodology 5(3), 231–261 (1996)