Learning Defect Predictors: Lessons from the Trenches

Tim Menzies
LCSEE, WVU

tim@menzies.us

October 28, 2008

Sound Bites

- Static code features can be used to build software quality predictors.
- But they are a shallow well.
  - We can easily get to the bottom.
  - But further effort will not take us deeper.
- Unless we change the rules of the game.

Acknowledgments

National Science Foundation, USA

TÜBİTAK

Scientific and Technological Research Council of Turkey

- WVU
  - Bojan Cukic, Zach Milton, Gregory Gay, Yue Jiang, Justin DiStefano,
  - Supported by NSF grant #CCF-0810879
- Softlab (Boğaziçi University, Turkey)
  - Ayşe Bener, Burak Turhan
  - Supported by Turkish Scientific Research Council (Tubitak EEEAG #108E014).
- PSU: Jeremy Greenwald

Reference herein to any specific commercial product, process, or service by trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the Governments of Turkey or the United States.

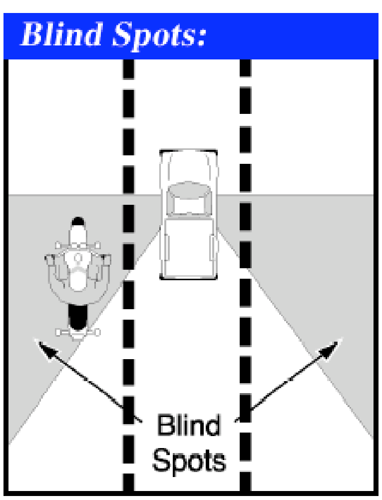
PROMISE '09

- `www.promisedata.org/2009`
- Motto:
  - Repeatable, refutable, improvable
  - Put up or shut up
- Papers ...
  - ... and the data used to generate those papers
  - `www.promisedata.org/data`
- Keynotes:
  - Barry Boehm (USC)
  - Brendan Murphy (Microsoft)

Outline

1 What is “Defect Prediction”?
2 Value of Defect Prediction
3 Variance and Ceiling Effects
4 Changing the Rules
5 Conclusions & Next Steps
6 References

Problem



100% quality assurance (QA) is infinitely expensive

- $Tests(Conf, ErrRate) = \frac{log(1-Conf)}{log(1-ErrRate)}$

Test engineers, skew QA towards what is most critical:

- e.g. model checking restricted to the kernel of the guidance system

But if we only look at {A,B,C}....

- ... what hides in the blind spot of {E,F,G,...}?

Proposal:

- Augment focused expensive QA methods with cheaper faster methods
- But how to sampling methods blind spots?
  - quickly
  - cheaply

One Solution: Data Mining

- Input: rows of data

features			class
name	age	shoeSize	mood
tim	48	11	happy
tim	12	5	sad
tim	28	9	happy
tim	100	11	happy

- Learners:
  - Naive Bayes: statistical feature combinations for class prediction
  - RIPPER : learns rules
  - C4.5 : decision tree learner
  - Random Forests : learn 100s of trees
  - etc
- Output: combinations of features that predict for the class

Solution: the Details

- Using data mining: explore the "modules"
  - smallest functional unit $\in \{function, method, class, ...\}$
- Take logs of modules described as static code modules:
  - Lines of code and comment measures
  - Counts of intra-module symbols (Halstead [1977]);
  - Measures of intra-module call graphs (McCabe [1976]);
- Join the logs to number of defects seen in each module
  - often, discretized to $\{yes, no\}$;
- Find feature combinations that predict for *defective* $\in \{yes, no\}$;

Examples of Static Code Features

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

Why Do That?

Useful: out-performs known industrial baselines at defect detection:

- IEEE Metrics'02 panel: manual software reviews finds ≈60% (Shull et al. [2002]);
- Raffo (pers. comm.): industrial review methods find $pd = TR(35, 50, 65)\%$
- Data mining static code features finds (median) 71% (Menzies et al. [2007]).

Easy to use:

- Automatic, cheap and fast to collect, scales to large systems.
- Manual methods: 8-20 LOC/minute for manual code reviews.

Widely used:

- Hundreds of research papers. Earliest: Porter and Selby [1990]?
- Recommended by numerous texts
- Large government contractors: only review modules that trigger a static code analyzer

Why Use Static Code Features?

- Why not use...
  - knowledge of the developers (Nagappan et al. [2008])
  - Or history of runtime defectives (Musa et al. [1987])
  - Or XYZ?
- A: Use whatever is available
  - And that changes from site to site
- I dream of the day that I work with an organization with stable products and practices.
  - Meanwhile, in the real world...

Dealing with Organizational Change

year	#IV&V project	notes	oversight
1988	n/a	Space shuttle begins IV&V	Johnson (Texas)
1991	n/a	New the IV&V facility	Headquarters (east coast)
1994	1	International space station IV&V begins	
1995	1		
1996	2		NASA Ames (west coast)
1996	3	IV&V funded from project budgets	
1997	3		
1998	3		
1999	12	IV&V now considered on all software	
2000	15		GSFC (east coast)
2001	20		
2002	36		
2003	42	IV&V funded from central agency source.	
2004	37		
2005	24	SILAP data collection begins	
2006	26		
2007	24	SILAP data collection ends	

- 2003: Loss of Columbia $\Rightarrow$ "return to flight" reorganization
- 2004: Bush's new vision for space exploration
- Always: layers of contractors; so "oversight", not "insight"

The Real Question

- Not what features "are right";
  - But what features are available "right now".
- Particularly when you can not control data collection
  - Agile
  - Out-source
  - Open source
  - Sub-contractors
  - Your current project?
- Sometimes, all you can access "right now" is source code.

Outline

1 What is "Defect Prediction"?

2 Value of Defect Prediction

3 Variance and Ceiling Effects

4 Changing the Rules

5 Conclusions & Next Steps

6 References

Aren't Static Code Features.... Stupid?

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

- $v(g)$ correlated to LOC
- "For a large class of software $((v(g))$ is no more than a proxy for, and in many cases outperformed by, lines of code"
  – Shepperd & Ince

So What Would Happen if...

We learned defect predictors from NASA aerospace applications

- Then applied them to software developed for Turkish whitegoods?
- (Caveat: in both studies, same data dictionary but different data.)

source	project	language	description
NASA	cm1	C++	Spacecraft instrument
NASA	pc1	C++	Flight software for earth orbiting satellite
NASA	mc2	C++	Video guidance system
NASA	mw1	C++	A zero gravity experiment related to combustion
NASA	kc1	C++	Storage management for ground data
NASA	kc2	C++	Storage management for ground data
NASA	kc3	JAVA	Storage management for ground data
SOFTLAB	ar5	C	Washing machine
SOFTLAB	ar3	C	Dishwasher
SOFTLAB	ar4	C	Refrigerator

Begin digression: how do we measure performance?

Performance Measures

$\{A, B, C, D\}$ = true negatives, false negatives, false positives, and true positives (respectively) found by a binary detector.

$$pd = recall = \frac{D}{B+D}$$
$$pf = \frac{C}{A+C}$$
$$prec = precision = \frac{D}{D+C}$$
$$acc = accuracy = \frac{A+D}{A+B+C+D}$$
$$neg/pos = \frac{A+C}{B+D}$$

For large neg/pos values: can be accurate and still miss most things

		module found in defect logs? no	yes
signal detected?	no	A = 395	B = 67
	yes	C = 19	D = 39

$$Acc = accuracy = 83\%$$
$$pf = Prob.falseAlarm = 5\%$$
$$pd = Prop.detected = 37\%$$

End digression.

Results

For learner=Naive Bayes (why? see later), try "round robin" or "self-" learning

- RR= "Round robin" : train on *them*, test on *me*.
- 'Self" : train and test on *me*

	Data source	Train	Test	Probability Detection	False Alarm
RR	Imported data	all X - X_i	X_i	94 (!)	68
self	Local data	90% *of* X_i	10% *of* X_i	75	29
RR	Filtered imported data	k-nearest of (all X - X_i)	X_i	69	27

- Best data source: local data
- Adequate: using imported data (filtered with nearest neighbor)

Recommendations:

- If no data, start local collection
- Meanwhile, use imported data, filtered with nearest neighbor

Question: how much data is needed to build local detectors?

Outline

1 What is "Defect Prediction"?
2 Value of Defect Prediction
3 Variance and Ceiling Effects
4 Changing the Rules
5 Conclusions & Next Steps
6 References

What is "The" Best Feature?

Experiments with Näive Bayes:

- While performance improved, add features.

data	pd	pf	index of selected feature
pc1	48	17	3, 35, 37
mw1	52	15	23, 31, 35
kc3	69	28	16, 24, 26
cm1	71	27	5, 35, 36
pc2	72	14	5, 39
kc4	79	32	3, 13, 31
pc3	80	35	1, 20, 37
pc4	98	29	1, 4, 39
all	71	25	

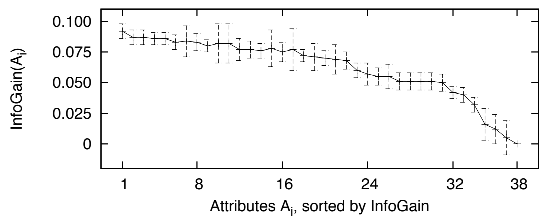
ID	used in	what	type
1	2	loc_blanks	locs
3	2	call_pairs	misc
4	1	loc_code_and_command	locs
5	2	loc_comments	locs
13	1	edge_count	misc
16	1	loc_executable	locs
20	1	I	H'
23	1	B	H'
24	1	L	H'
26	1	T	H'
31	2	node_count	misc
35	3	μ_2	h
36	1	μ_1	h
37	2	number_of_lines	locs
39	2	percent_comments	misc

H' = derived Halstead
h = raw Halstead

Why no single most valuable feature?

Feature Information Variance

10 * { 90% sample, compute "info gain" of each feature}



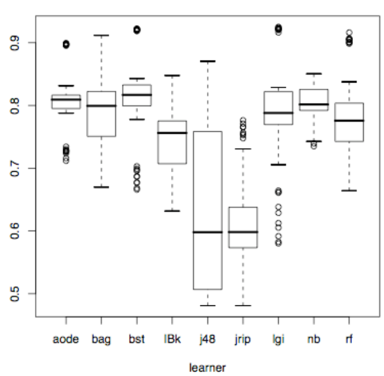
- High left-hand-side plateau (multiple candidate "best" features)
- Low right-hand-side valley (small set of "always worst" features)

Never again: $v(g) > 10$.

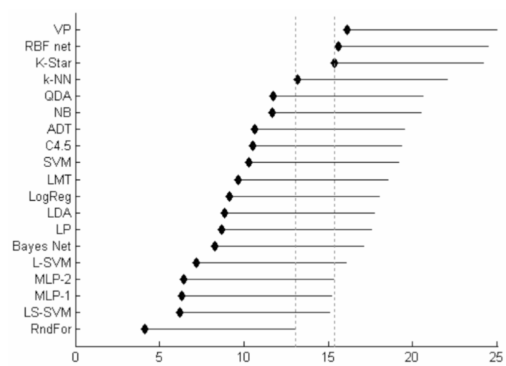
Value of Better Data Miners?

Ceiling effects: an inherent upper bound on the performance of our data miners

- Have yet to improve our mid-2006 defect predictors: Menzies et al. [2007]



6/9 methods are "best"



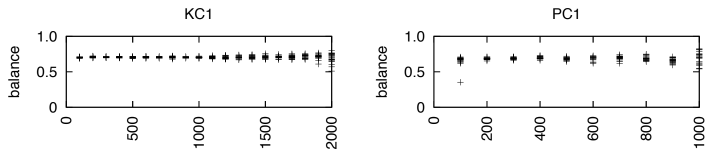
14/19 methods are "best"

Note: evaluation bias- area under the curve of a detection vs false alarm plot

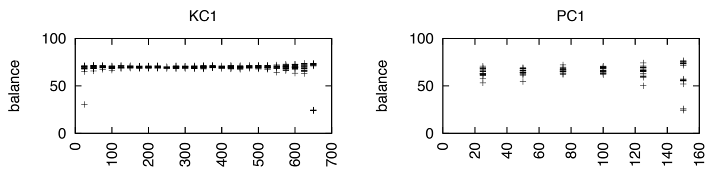
- AUC(PD, PF)

Value of More Data?

Learner = NB Randomized order ; train on "X"; test on next 100



Micro-sampling: N defective + N non-defective, $N \in \{25, 50, 75, \ldots\}$



Statistically, little gain after 100 (random) or 50 (micro-sampling)

End of the Line?

"...the importance of the (learner) is less than generally assumed and practitioners are free to choose from a broad set of candidate models when building defect predictors."
– Lessmann et al.

No value in new algorithms for defect prediction?

- Not unless we change the rules of the game

Outline

1 What is "Defect Prediction"?

2 Value of Defect Prediction

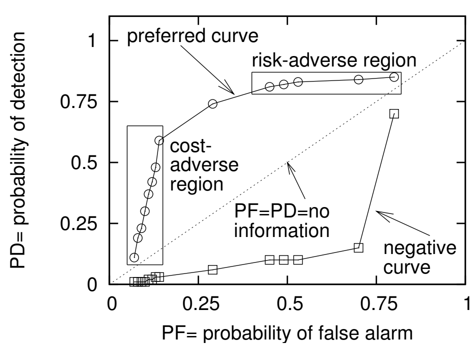
3 Variance and Ceiling Effects

4 Changing the Rules

5 Conclusions & Next Steps

6 References

Generalized Evaluation Bias



	pd	pf	effort#
Risk adverse (e.g. airport bomb detection, morning sickness)	hi	hi	
Cost adverse (e.g. budget conscious)	med	lo	
Arisholm and Briand [2006]			$< pd$

#effort = LOC in the modules predicted to be faulty

Evaluation-aware Learners

All learners have an search bias S and an evaluation bias E. e.g. C4.5:

- S = infogain
- E = pd,pf,accuracy, etc
- Note: usually, $S \neq E$

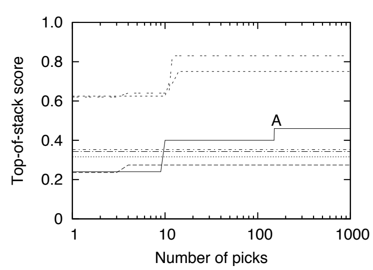
Question: What if we make $S = E$?

- Answer: Milton [2008]

Evaluation-aware Learning with "WHICH"

1. Discretize all numeric features.
2. Sort all ranges using E onto a stack
3. Pick any 2 items near top-of-stack
4. Combine items, score them with E, insert them into the sorted stack.
5. Goto 3

Note: no S; E is customizable.



Top of stack stabilizes quickly (UCI data).

Other methods can bias learner predictions:

- Apply E during decision tree splitting
- Elkan [2001]: cost-sensitive learning (*)
- Fawcett [2001]: ROC ensemble combinations (*)

(*) But what work if search criteria is orthogonal to the evaluation criteria?

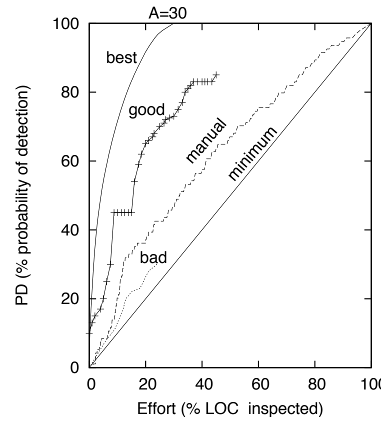
Experiments with "WHICH"

Arisholm and Briand [2006]

- For a budget-conscious team,
- if X% of modules predicted to be faulty
- but they contain $\leq X\%$ of the detects,
- then that defect predictor is not useful
- i.e. they prefer $pd > effort$

Operationalizing their bias:

- Find modules triggered by the learner
- Sort them in ascending order of size
- Assume human inspectors find Δ of the defects in the triggered modules
- Score learner as ratio of "best" effort-vs-pd curve
  - "best" only triggers on defective modules
  - Note : Δ cancels out



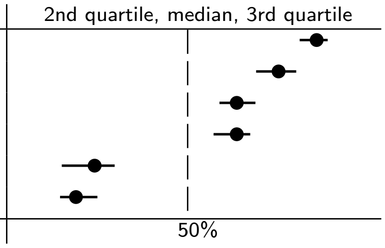
"bad" : worse than manual
"good" : beats manual

Learning Defect Predictors:Lessons from the Trenches
└Changing the Rules
└Milton [2008]

Experiments with "WHICH"

- 10 random orderings * 3-way cross-val
- 10 sets of static code features from NASA, Turkish whitegoods
- "Rank" computed using Mann-Whitney U test (95%)
- Micro20: training on 20 defective + 20 non-defective

rank	treatment	median "best"%	2nd quartile, median, 3rd quartile
1	WHICH	87.3	
2	micro20	76.3	
3	NB	64.2	
3	manual	64.2	
4	C4.5	23.1	
4	jRip	17.7	



Shallow well: we do not need much data to do it (40 examples).

Outline

1 What is "Defect Prediction"?

2 Value of Defect Prediction

3 Variance and Ceiling Effects

4 Changing the Rules

5 Conclusions & Next Steps

6 References

But Why does it Work so Well?

Related question:

- Why does IR work so well?
- Same answer for both questions?

Is our technical work constrained by:

- compilers,
- languages,
- target domains,
- human short-term memory,
- etc

What is this invisible hand?

F.A.Q.

Are defect predictors useful?

- As a method to reduce the cost of more expensive QA, yes.

Are defect predictors general?

- Yes: after NN, NASA & SOFTLAB's predictors work on each other's site

But learning from local data is best.

- How much data is needed?
  - Dozens to under 100 for micro-sampling to random sampling

F.A.Q. (more)

Which learner is best for building defect predictors?

- When maximizing PD-vs-PF, there are many choices.
- Otherwise, tune learner to local evaluation bias (e.g. WHICH)

What is the "best" feature?

- Wrong question. "Best" is domain-specific.
- Collect everything that is readily and cheaply available

Research Directions: NSF CCF funding

We have seen above that the “best” features are data set dependent.

- Due to feature information variance

So lets look harder at exploiting local knowledge.

- Microsoft example

Given the small samples needed for learning detectors (dozens to 100)

- Augment (? replace) data mining
- ... with human-in-the-loop case-based-reasoning

Do you think that with your domain expertise you can do better than stupid static features?

- Then lets talk some.

And Finally..

- Static code features can be used to build software quality predictors.
- But they are a shallow well.
  - Easily get to the bottom
    With $\leq$ 100 examples.
  - Further effort will not take us deeper
    Ceiling effects on AUC(Pd,Pf).
- Unless we change the rules of the game:
  - Using evaluation aware learning
  - Augment data mining with human-in-the-loop

Outline

1 What is "Defect Prediction"?

2 Value of Defect Prediction

3 Variance and Ceiling Effects

4 Changing the Rules

5 Conclusions & Next Steps

6 References

References

E. Arisholm and L. Briand. Predicting fault-prone components in a java legacy system. In *5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE), Rio de Janeiro, Brazil, September 21-22*, 2006. Available from `http://simula.no/research/engineering/publications/Arisholm.2006.4`.

Charles Elkan. The foundations of cost-sensitive learning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI01)*, 2001. Available from `http://www-cse.ucsd.edu/users/elkan/rescale.pdf`.

Tom Fawcett. Using rule sets to maximize roc performance. In *2001 IEEE International Conference on Data Mining (ICDM-01)*, 2001. Available from `http://home.comcast.net/~tom.fawcett/public_html/papers/ICDM-final.pdf`.

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

M.H. Halstead. *Elements of Software Science*. Elsevier, 1977.

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`.

S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 2008.

T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976. ISSN 0098-5589.

References (more)

T. Menzies, M. Benson, K. Costello, C. Moats, M. Northey, and J. Richarson. Learning better IV&V practices. *Innovations in Systems and Software Engineering*, March 2008a. Available from http://menzies.us/pdf/07ivv.pdf.

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)*, 2008b. Available from http://menzies.us/pdf/08ceiling.pdf.

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.

Tim Menzies, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. Problems with precision. *IEEE Transactions on Software Engineering*, September 2007a. http://menzies.us/pdf/07precision.pdf.

Z.A. Milton. The WHICH Stochastic Rule learner. Master's thesis, Lane Department of Computer Science and Electrical Engineering, WVU 2008.

References (yet more)

J. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw Hill, 1987.

N. Nagappan, B. Murphy, and Basili V. The influence of organizational structure on software quality: An empirical case study. In *ICSE'08*, 2008.

A.A. Porter and R.W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, pages 46–54, March 1990.

M. Shepperd and D.C. Ince. A critique of three metrics. *The Journal of Systems and Software*, 26(3):197–210, September 1994.

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

B. Turhan, A. Bener, and T. Menzies. Nearest neighbor sampling for cross company defect predictors. In *Proceedings, DEFECTS 2008*, 2008.