Software Estimation Models:When is Enough Data Enough?

Tim Menzies
Department of Computer Science, Portland State University, Oregon
tim@menzies.us;http://menzies.us

June 8, 2004

1 Overview

Do you want to be the manager of a canceled software project?[1] Do you know if your budget is adequate to the task at hand? If a project's costs are under-estimated then developers will be forced into many quality-threatening cost-cutting measures as the project develops. Sometimes, "cost-cutting" becomes "project-canceling":

> To gain control over its finances, NASA last week scuttled a new launch control system for the space shuttle. A recent assessment of the Checkout and Launch Control System, which the space agency originally estimated would cost $206 million to field, estimated that costs would swell to between $488 million and $533 million by the time the project was completed.
> –Computer News:Wed June 11, 2003

One reason for poor cost estimation is that, all too often, software managers don't have enough relevant data to make accurate estimations. For example, consider how much work was required to tune the standard COCOMO-II-2000 cost estimation model [2]:

- An initial regression analysis of 83 projects (this generated the COCOMO-I-1981 model [1]);
- Further data collection on 78 more projects;
- A DELPHI panel where experts offered their best judgment on factors controlling software costs;
- A Bayesian tuning phase that integrated the DELPHI results with data from the 83+78 projects.

Most industrial sites lack the resources to repeat the above process. Data collection from industry is notoriously slow and the above process took nearly two decades to complete [1, 2].

To shortcut the development time of an effort estimation model, the COCOMO team offer certain *tuning parameters* a and b in their model. They recommend

> ...having at least 5 data points for local calibrating the multiplicative constant a and at least 10 points for calibrating both the multiplicative constant a and the baseline exponent b.[1, p175]

While the comment comes from a definitive source, the COCOMO teams offers no evidence for the claim that data from 5 to 10 projects is enough. Curiously, the COCOMO team felt the need for 161 projects to generate COCOMO-II-2000. Perhaps there may be some drawback to tuning based on only 5 to 10 projects. To check this speculation, we conducted two studies using data from 62 COCOMO-I-1981 projects and 60 projects

[1] *A submission to the Pacific Northwest Software Quality Conference, Portland, Oregon, Fall, 2004;* http://www.pnsqc.org/

from the Jet Propulsion Laboratory[2]. *Sequence tuning* experiments were performed where some learning device tuned an effort estimation model using more and more data. The learning was then disabled and the estimation model was tested on data not seen during training. Sequence tuning was used since (a) it tests the theory on data not used in training (which is good experimental design); and (b) it emulates standard business practice where managers learn their own estimation models based on the projects seen to date.

In order to study the variance in the effort predictions, this procedure was repeated 30 times, each time with a different randomly selected ordering of the projects. The tuning method was the TUNES tool described later in this paper[3]. The results are shown in Figure 1 (JPL on top, COCOMO-1-1981 below). As sequence tuning proceeds down the x-axis, more data is available for training and performance improves. Performance is measured in terms of PRED(N). For example, at a PRED(30) of 69%, effort estimations for 69% of the projects in the test set are within 30% of the actual value. PRED(30)=69% was the best results achieved by the COCOMO-II team This was a *mean* figure seen in 15 of their test sets and is shown in Figure 1 as solid horizontal lines.

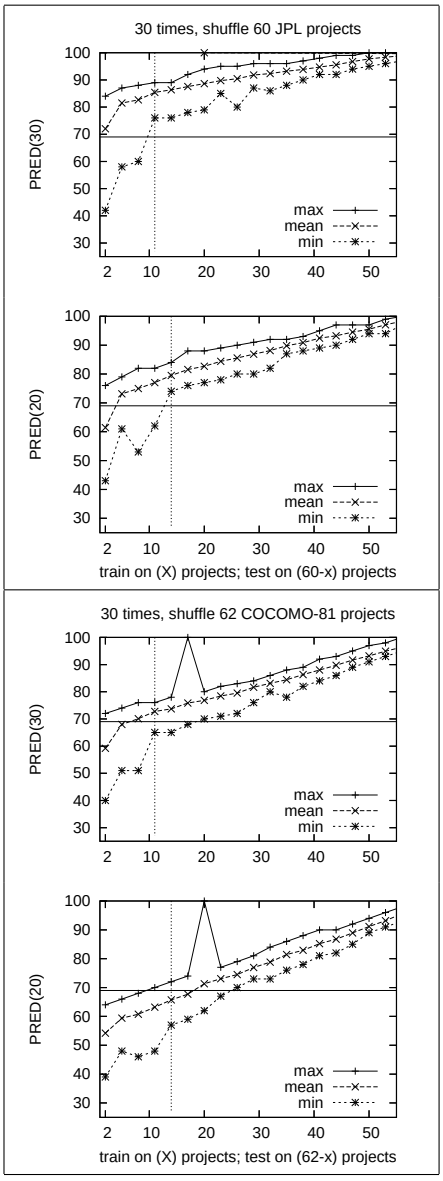

Figure 1: Sequence tuning.

From Figure 1, several conclusions follow. Firstly, and most importantly, in all our studies we agree with the recommendations of the COCOMO team. Eleven projects are enough to achieve COCOMO-II-2000 levels of performance of $PRED(30) = 69\%$. Secondly, the TUNES method can achieve *better than* COCOMO-II-2000 performance where *better* is defined in several ways:

- TUNES's data requirements are *better* than COCOMO-II-2000. COCOMO-II-2000 reached PRED(30)=69% after (i) adding five new variables to the COCOMO-I-1983 model; (ii) analyzing 78 new projects; (iii) convening a DELPHI panel of experts; and (iv) using Bayesian tuning to combine the results of the DELPHI panel with data from the 78 new projects. In contrast, TUNES needed only 11 new projects, no new variables, no panel of experts, and no Bayesian tuning.
- TUNES can achieve *better* PRED(N) levels. PRED(20) is a stricter criteria than PRED(30) since PRED(20) requires project estimates falling *closer* to the actual. After seeing 5 JPL projects and 20 COCOMO-I projects, TUNES can achieve a mean PRED(20) of 69% (see the PRED(20) plots of Figure 1).

Thirdly, it may not be enough to report *mean* PRED(N) if the variance is large. Note that after tuning on just a few projects the variance in the PRED(N) figures can be quite large: see the difference between maximum and minimum PRED(30) at low X

[2]COCOMO data from [6], and downloaded from `http://www.vuse.vanderbilt.edu/~dfisher/tech-reports/raw-TSE-95`. JPL data courtesy of Dr. Jairus Hihn. Both data sets are available from `http://menzies.us/data.html`

[3]Available from `http://scant.org/tunes/tunes.html`

values in top plot of Figure 1. This is to be expected: when the training set is small, many mistakes are made during testing. Therefore, we say that it is important to assess an effort estimation model not only on its *mean* result, but also on its *variance* when tested on unseen data.

The vertical dashed lines of Figure 1 show where the variance on the mean and minimum PRED(N) curves found by TUNES seem to stabilize; i.e. after that point they seem to develop smooth trends. Such stability was achieved at $x = \{11, 14\}$ for PRED(30) and PRED(20) respectively (these points are somewhat subjective and the reader might care to check our assertion that the mean and minimum variance to the right of the dashed lines of Figure 1 is larger and more changeable than on the right). Once the variance stabalizes, then enough project data should be collected to achieve *good* PRED(N) levels. If we use the COOCMO-II benchmark of PRED(N)=69% to define *good*, then the points in Figure 1 where PRED(N) was *good* and stable were $x = 11$ for PRED(30) and $x = \{14, 20\}$ for PRED(20).

The rest of this paper described exactly how Figure 1 was generated. That description starts with more details on COCOMO, then some notes on the JPL and COCOMO-I data used in these experiments. This is followed by a description of the TUNES tool.

rely: required software reliability
data: data base size
cplx: process complexity
time: time constraint for cpu
stor: main memory constraint
virt: machine volatility
turn: turnaround time
acap: analysts capability
aexp: application experience
pcap: programmers capability
vexp: virtual machine experience
lexp: language experience
modp: modern programing practices
tool: use of software tools
sced: schedule constraint

Figure 2: COCOMO-I-1981 effort multipliers.

2 COCOMO

The COCOMO project aims at developing an open-source, public-domain software effort estimation model. The project has collected information on 161 projects from commercial, aerospace, government, and non-profit organizations[3]. As of 1998, the projects represented in the database were of size 20 to 2000 KSLOC (thousands of lines of code) and took between 100 to 10000 person months to build.

COCOMO measures effort in calendar months where one month is 152 hours (and includes development and management hours). The core intuition behind COCOMO-based estimation is that as systems grow in size, the effort required to create them grows exponentially, i.e. $effort \propto KSLOC^x$. More precisely:

$$months = a * \left(KSLOC^b\right) * \left(\prod_j EM_j\right) \quad (1)$$

where a and b are domain-specific parameters; KSLOC is estimated directly or computed from a function point analysis; and EM_j is one of a set of *effort multipliers*. In COCOMO-I, the exponent on KSLOC was a single value ranging from 1.05 to 1.2. In COCOMO-II, the exponent was divided into a constant, plus the sum of five *scale factors* which modeled issues such as "have we built this kind of system before?". This TUNES study used the COCOMO-I model since the data available to this study did not contain scale factors

The effort multipliers in COCOMO-I are listed in Figure 2. The COCOMO-II effort multipliers are similar but COCOMO-II dropped one of the Figure 2 parameters; renamed some others; and added a few more (for "required level of reuse"", "multiple-site development", and "schedule pressure").

The numeric values of the effort multipliers are shown in Figure 3. These were learnt by Boehm after a regression analysis of the projects in the COCOMO-I data set [1]. The last column shows $\frac{EM_{max}}{EM_{min}}$ and shows the overall

effect of a single effort multiplier. For example, *acap* (analyst experience) is most important and *lexp* (language experience) is least important.

There is much more to COCOMO that the above description. Equation 1 is only the effort model and COCOMO comes with a schedule model as well. The COCOMO-II text [2] is over 500 pages long and offers all the details needed to implement data capture and analysis of COCOMO in an industrial context. For example, that text includes the templates, definitions and training material needed to deploy COCOMO in an industrial setting. That text also describes numerous details such as how to apply COCOMO very early in the life cycle; and how to handle multi-component systems or systems with significant amounts of automatic code generation. Further, Chapter Five of that text describes numerous emerging extensions of COCOMO including models for rapid prototyping, COTS integration, quality and risk assessment.

	very low	low	nominal	high	very high	extremely high	productivity range
acap	1.46	1.19	1.00	0.86	0.71		2.06
rely	0.75	0.88	1.00	1.15	1.40		1.87
cplx	0.70	0.85	1.00	1.15	1.30	1.65	1.86
pcap	1.42.	1.17	1.00	0.86	0.70		1.67
aexp	1.29	1.13	1.00	0.91	0.82		1.57
tool	1.24	1.10	1.00	0.91	0.83		1.49
virt		0.87	1.00	1.15	1.30		1.49
vexp	1.21	1.10	1.00	0.90			1.34
modp	1.24.	1.10	1.00	0.91	0.82		1.34
turn		0.87	1.00	1.07	1.15		1.32
time			1.00	1.11	1.30	1.66	1.30
data		0.94	1.00	1.08	1.16		1.23
sced	1.23	1.08	1.00	1.04	1.10		1.23
stor			1.00	1.06	1.21	1.56	1.21
lexp	1.14	1.07	1.00	0.95			1.20

Figure 3: COCOMO-I-1981 effort multipliers.

This current study is very relevant to those emerging models. When those COCOMO extensions are validated and calibrated to local sites, it will be vital to know how much data is required for that validation and calibration.

3 The Data

A curious feature of Figure 1 is that TUNES worked faster on the JPL data than the COCOMO-I data. The reason for this is simple: the JPL project data comes from one organization and the COCOMO-I project data comes from many organizations. Figure 4 plots source lines of code vs development effort (in months) for our two data sets. Note that the *intra-organizational* JPL data has much less variance than the *inter-organizational* data from the COCOMO-I data set (e.g. at KSLOC=10, the COCOMO-I efforts vary by an order of magnitude which is much more than the KSLOC=10 JPL efforts). TUNES works better on intra-organizational data since it is less of a struggle to tune a and b to data with less variance.

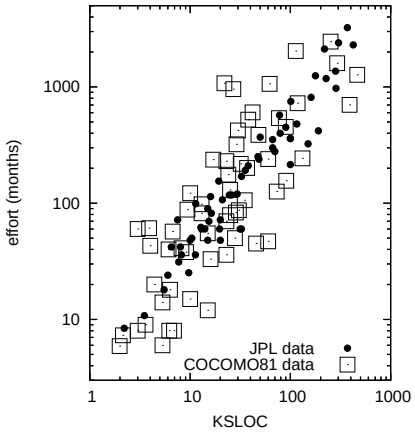

Figure 4: LOC,effort for JPL & COCOMO81

4 The TUNES Tool

Our previous experiments with COCOMO [4, 5] allowed for variation in all the effort multipliers of Figure 3, and the scale factors. This is a large space of options.

This large problem can be converted to a much simpler problem as follows:

- Ignore the scale factors; i.e. use COCOMO-I and not COCOMO-II);
- Freeze the effort multipliers;
- Just seek values of a and b that minimize the difference between the $actual$ the $estimated$ development effort; i.e. $(actual - estimated)/actual$.

This simpler problem is small enough to enable a search over the entire range of a and b. Given N projects, the `test` function of the TUNES tool shown in Figure 5 inputs a and b parameters and calls COCOMO-I for the projects numbered $start$ to $stop$ ($start \leq stop \leq N$). The `learn` function calls `test` for projects 1 to i ($i \leq N$) to find the a and b parameters that minimizes the `test` failure count. These best a and b values are then `test`ed on the projects numbered $i + 1$ to N. All the plots in this paper and reports of the test run, repeated 30 times (with the project numbering randomized before each repeat).

TUNES is a very simple system and could be improved. For example, Equation 1 shows that a and b effect $effort$ monotonically and continuously. Hence, the exhaustive search on lines 5 and 6 of `learn` might be replaced with a binary search. Secondly, TUNES is implemented in an interpreted C-like language (awk) and a reimplementation in "C" would make it run faster. Nevertheless, given that this sub-optimal exhaustive search implemented in an interpreted language takes just a few minutes to generate Figure 1, we are not motivated to explore optimizations. Further, the exhaustive nature of the search makes it hard to argue that some other method might do better than TUNES.

Figure 6 show how TUNES changed a and b for the PRED(20) runs on the JPL and COCOMO-I data. For all the experiments reported here, the a values were explored from 2 to 5 in increments of 0.2; i.e. $a \in \{2, 2.2, ..., 4.8, 5.0\}$. The b values were explored from 0.9 to 1.2 in increments of 0.02; i.e. $b \in \{0.9, 0.92, ..., 1.18, 1.2\}$. The whiskers in Figure 1 are 2 standard deviations wide, centered on the mean value. As seen before in Figure 1, the variances tend to decrease as more training data is used. Note that the JPL runs converge to significantly much higher a and b values that offered by standard COCOMO-I and COCOMO-II (those standard values are

```
   function test(start,stop,a,b,pred) {
1.   failures=0
2.   for(i=start;i<=stop;i++) {
3.     est   = cocomo(i,a,b)
4.     act   = actual(i)
5.     delta = (act-est)/act
6.     if (abs(delta)>pred) failures++}
7.   return failures}
```

```
   function learn() {
1.   Repeats=30
2.   while(Repeats--) {
3.     randomizeOrderOfProjects()
4.     for(i=2;i<=N;i += 3) {

         #Training stage
5.       for(a=AMin;   a <=AMax; a += AInc) {
6.         for(b=BMin; b<=BMax;  b += BInc) {
7.           sum=test(1,i,a,b,Pred)
8.           if (sum < least) { least=sum
9.                              BestA=a
10.                             BestB=b }}}

         #Testing stage
11.      failures = test(i+1,N,BestA,BestB,Pred)
12.      print Repeats " " i " " failures}}}
```

Figure 5: The TUNES tool: pseudo-code.

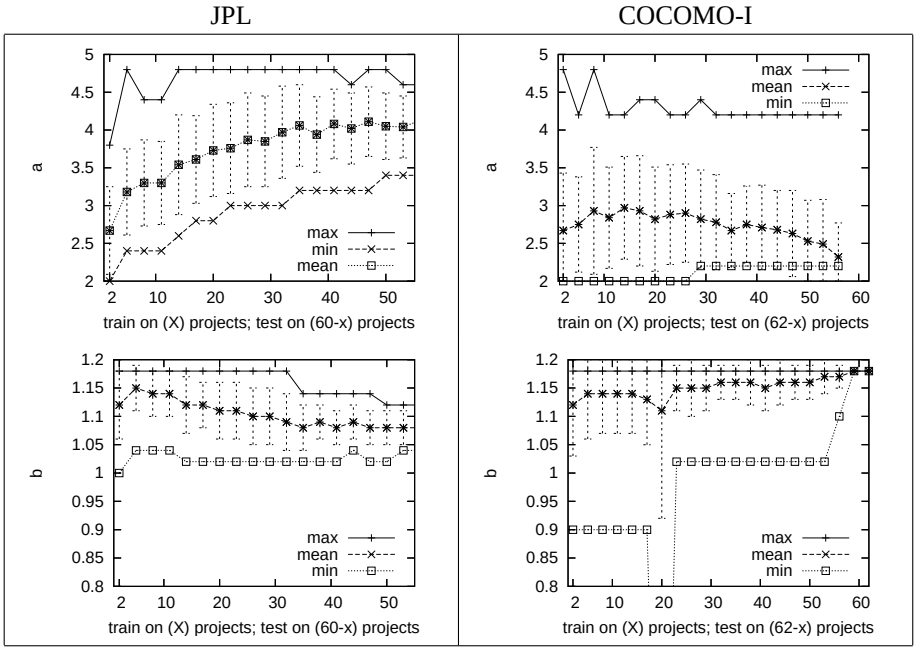

Figure 6: Tuning a and b in a PRED(20) run on JPL data (top) and COCOMO-I data (bottom).

$\langle a, b \rangle = \langle 2.94, 0.91 \rangle$ respectively). Hence, JPL would be advised not to use un-tuned COCOMO-I for its effort estimations.

5 Conclusions

In summary, and in answer the question in the title of this paper, Figure 1 shows that the TUNES method can:

- Find stable PRED(30) predictions with low variances after 11 new projects.
- Find stable PRED(20) predictions with low variances after 14 to 20 new projects.

It would be an error to summarize this study as (say) "here are new a,b parameters for COCOMO-I". The large variances in the *inter-organizational data* (COCOMO-I) of Figure 6 show that we can't just offer a single point estimate for COCOMO-I tunings (at least, if those tunings are learnt by TUNES). The conclusions of this paper have to be interpreted within the context of TUNES. Figure 1 is the preferred format and such plots can be summarized as follows:

> Based on sequence tuning experiments on N past projects, we predict that we can estimate effort on future projects within such-and-such intervals.

(those intervals being read off the min-max curves of Figure 1). Further, a sequence tuning experiment might also conclude that:

Based on the history of projects seen to date, we predict that the variance in our effort estimations will reduce by *this much* if we can collect data from *these many* more projects.

Lest this reports appears too critical of COCOMO, it is important to note that TUNES is an extension to COCOMO-I-1981 and could not work without it. As to COCOMO-II-2000, the TUNES results offer a simpler method of obtaining the same, or even better, results:

- One of the main motivations for the Bayesian analysis of COCOMO-II was the regression results from the 83+78 projects had slopes that contradicted certain expert intuitions. For example regression of the COCOMO data concluded that building reusable components *decreased* development costs. Most experts believe that the extra effort required to generalize a design actually *increases* the cost of building such components. This anomaly was explained as follows: the 83+78 projects did not contain enough samples of projects that make heavy use of reuse. To DELPHI panel and the subsequent Bayesian tuning was used to fill in the gaps in the project data with expert knowledge. This combination of DELPHI+Bayesian methods proved successful: COCOMO-II-2000 had much higher PRED(N) levels than COCOMO-I.
- Given the TUNES results, a much simpler method for incorporating expert knowledge is possible. Recall that TUNES can find reach good PRED(20) after 23 projects. This number of projects could be artificially generated from, say, 4 experts each asked to describe 5 exemplar projects showing the range of projects typically done at their company. If those descriptions were made in terms of the COCOMO-I parameters, then TUNES could then tune an effort model to that expert option using those 20 expert-generated examples.

Acknowledgments

This research was conducted at Portland State University, partially sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

References cited

[1] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[2] B. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A. W. Brown, S. Chulani, and C. Abts. *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.

[3] S. Chulani, B. Boehm, and B. Steece. Bayesian analysis of empirical software engineering cost models. *IEEE Transaction on Software Engineerining*, 25(4), July/August 1999.

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

[5] T. Menzies and E. Sinsel. Practical large scale what-if queries: Case studies with software risk assessment. In *Proceedings ASE 2000*, 2000. Available from `http://menzies.us/pdf/00ase.pdf`.

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