Reusing Models For Requirements Engineering[1]

Tim Menzies[2], Ying Hu[3]
University of British Columbia[4]
tim@menzies.com, yingh@ece.ubc.ca

September 11, 2001

Abstract

A problem with model-based requirements engineering is that new projects may lack the data required to customize old models. Such data droughts are a common problem in software engineering and are particularly acute in early life cycle activities such as requirements engineering. When specific data relevant to a new project is missing, one technique is to simulate a model across the range of possibilities that might be relevant to a project. This generates voluminous output which can be summarized via a new machine learning technique called *treatment learning*.

KEYWORDS: Software engineering, requirements engineering, machine learning, data mining, knowledge farming, treatment learning, risk assessment, COCOMO.

1 Introduction

At requirements time, much is unknown about a project. Nevertheless, requirements engineers must still make decisions that have far reaching implications for a project. Errors in those decisions is one of the major costs in developing software. Fixing bugs in development or after delivery costs 20 to 200 times (respectively) the cost of repair those same bugs in the requirements phase [3].

Nowhere is the data draught problem more acute than in the model-based requirements engineering (MBRE). Models can be built, or borrowed, to assist in early lifecycle decision making. Often, these models require more data than what is available. For example, suppose a software manager wants to reduce the odds that their project will run over-schedule. The risk of this event can be assessed by the COCOMO risk model [7]. Figure 1 shows the inputs to that model from KC-1: a NASA software project. The column *now1* shows the current state of the project, expressed in the language of COCOMO tool [1]. The $changes1$ column describes some proposed changes to the project. Where precise values are unknown, a range of possibilities has been supplied. $Changes1$ shows 11 proposed changes. That is, there exist $2^{11} = 2048$ combinations of proposed changes to this project. Also, when combined with the ranges in the $now1$ column, there are $6 * 10^6$ options shown in this table. Other uncertainties exist as well:

- COCOMO models need an estimate of source lines of code (SLOC) and this may be hard to determine early in the lifecycle.
- COCOMO needs a set of internal *tunings* which should be calculated from prior projects. Due to the data draught, such tunings are often unavailable.

Hence, our software manager would have great difficulty in using the COCOMO risk model to assess the risk of estimate over-runs.

Data draughts can occur for a variety of reasons. Companies often lack the resources to collect and maintain

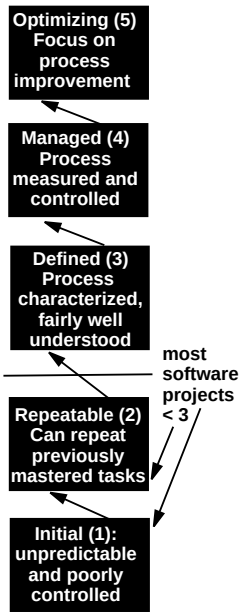
[1] Submitted to the first International Workshop on Model-based Requirements Engineering, November 30, 2001, San Diego, http://www.bfsng.com/mbre01/.

[2] http://tim.menzies.com, phone (604) 822-3381

[3] http://www.ece.ubc.ca/twiki/bin/view/Softeng/YingHu.

[4] Department of Electrical & Computer Engineering; 2356 Main Mall; Vancouver, B.C. Canada V6T1Z4.

			KC-1 (very new project)	
	ranges		now1	changes1
Scale drives	prec = 0..5	precedentness	0, 1	
	flex = 0..5	development flexibility	1, 2, 3, 4	1
	resl = 0..5	architectural analysis or risk resolution	0, 1, 2	2
	team = 0..5	team cohesion	1,2	2
	pmat = 0..5	process maturity	0, 1, 2, 3	3
Product attributes	rely = 0..4	required reliability	4	
	data = 1..4	database size	2	
	cplx = 0..5	product complexity	4, 5	
	ruse = 1..5	level of reuse	1, 2, 3	3
	docu = 0..4	documentation requirements	1, 2, 3	3
Platform attributes	time = 2..5	execution time constraints	?	
	stor = 2..5	main memory storage	2, 3, 4	2
	pvol = 1..4	platform volatility	1	
Personnel attributes	acap = 0..4	analyst capability	1, 2	2
	pcap = 0..4	programmer capability	2	
	pcon = 0..4	programmer continuity	1, 2	2
	aexp = 0..4	analyst experience	1, 2	
	pexp = 0..4	platform experience	2	
Project attributes	ltex = 0..4	experience with language and tools	1, 2, 3	3
	tool = 0..4	use of software tools	1, 2	
	site = 0..5	multi-site development	2	
	sced = 0..4	time before delivery	0, 1, 2	2
# of what-ifs (combinations of $now1 \cup changes1$) =			$6 * 10^6$	



Attributes in this figure come from the COCOMO-II software cost estimation model evolved by Boehm, Madachy, et.al. [1]. Attribute values of "2" are nominal. Usually, attribute values lower than "2" denote some undesirable situation while higher values denote some desired situation.

The Software Engineering Institute's capability maturity model [11].

Figure 1: LEFT: The KC-1 NASA software project. RIGHT: The CMM hierarchy.

such data. Alternatively, companies may not been developing their product(s) long enough to collect an appropriately large data set; Also, companies may be basing their development on COTS (commercial off-the-shelf software). While the COTS authors may have much data on the package, this data is rarely distributed. More fundamentally, much of the software industry operates at *less than* CMM-3 (see Figure 1, right); i.e. they neither document their processes nor record data based on those process descriptions (personal communication with SEI researchers).

Whatever the cause, model-based requirements engineering must be practical during data droughts. Hence, we explore an MBRE technique for data-starved domains:

> When data is scarce, we can *grow* data sets via simulations across the space of possibilities within the available models.

That is, if we can access some general model from a previous project, but lack specific data to customize that model, we simply run the model through the *space of possibilities* that may be relevant to the certain project.

The generated data sets may be too large to understand. Hence, after growing the data sets, they must be summarized. This paper proposes a summarization technique based on machine learning. Machine learn-

ing is a summarization technique for extracting the important ideas from examples. Techniques such as pattern recognition, bayesian reasoning, neural networks, association measures, and decision trees have matured dramatically in recent years. These techniques have been successfully applied to software engineering tasks such as cost-estimation (e.g. [5]), or prediction of faulty modules (e.g. [14]). Typically, machine learning is seen as a *data mining* activity; i.e. summarizing large data sets. Data mining is impractical during data droughts. We call our alternative to data mining *knowledge farming*. It has three steps:

1. Using large scale simulations, we quickly *grow* data from some *seed*; i.e. a model describing the options within a domain. This is easy to do using exhaustive or monte-carlo simulations of the model.

2. Once we have grown the data, we use machine learning to *harvest* the data; i.e. build useful summaries.

3. *Validation:* We can validate the summarized knowledge by feeding them back into the model to observe their impact on the model's behavior. This also provides a way for us to gain deeper understanding and to further refine that domain model.

In essence, this is the same idea as Bratko's behavioral cloning but here it is applied to software process models not qualitative models of physical devices [4, 13]. The harvested knowledge contains no more knowledge than in the original domain models. However, knowledge in domain models can be hard to access as it may be expressed verbosely, or it may be spread out over disperse parts of the model. In contrast, the harvested knowledge can be simpler and far more succinct.

This approach is demonstrated by describing how it works on Figure 1. The domain in which we performed our experiments was that of software development risk assessment. Firstly, we disucss our prior research in this area. Secondly, we use the COCOMO risk model to build the space of possible situations for a NASA software project. Thirdly, we describe the TAR2 *treatment learner* (defined below) which summarizes data generated by that model. Fourthly, we present the experimental results and ways to validate those results.

2 Prior Work

Earlier reports of this research [10] used complex combinations of tools called *tree query languages* (TQL) to analyze (e.g.) Figure 1. In summary, our experience is that treatment learners are far simpler to explain, understand, and implement that TQL. Also, in the KC-1 study, treatment learning gives (almost) the same results as TQL. One treatment found by TQL were rejected by treatment learners. This two treatment had a very low utility and ignoring it has minimal impact on the overall worth of the final treatments.

Treatment learners also runs much faster than TQL. Our treatment learner takes minutes to accomplish what TQL took hours to perform. One reason for this is that TQL is a post-processor to a decision tree generation algorithm. That is, to run TQL, a decision tree learner has to be run as well. Treatment learners have no need for this external call to a decision tree learner.

Finally, our prior report on TQL never validated its output. This study offers several validation studies of treatment learning.

3 Case Study

3.1 The COCOMO Risk Model

For our experiments, we used the Madachy COCOMO-based effort-risk model. The COCOMO project aims at developing an open-source, public-domain software effort estimation model. It allows one to estimate the cost, effort, and schedule when planning a new software development activity [1]. The Madachy extension to COCOMO was an experiment in explicating the heuristic nature of effort estimation. The model contains 94 tables, each of which implements a context-dependent modification to internal COCOMO parameters [7]. Two important features of the COCOMO risk model are its classifications and its validation. In the first case, the model generates a numeric effort-risk index which is then mapped into the classifications low, medium, high, very high. In the second, the model has survived at least one validation study (see [7]). Most risk models come with no validation information. The COCOMO risk model is the rare exception.

3.2 Growing Data

The COCOMO risk model was exercised using Monte Carlo simulations. That is, instead of running all 10^6 possible simulations described in Figure 1, we ran a small number N picked at random to see what we could learn. We then ran a larger number (e.g. $2N$,$3N$) of randomly picked simulations. A conclusion was deemed *stable* if it did not change when we used a larger sample size. Our simulation was conducted using randomly selected inputs as follows:

- The outputs of a COCOMO model are dependent on its internal tunings. In practice, users tune COCOMO parameters using historical data in order to generate accurate estimates. For our projects, we lacked the data to calibrate the model. Therefore, we picked our tunings from several published sources. Those sources showed tunings generated via genetic algorithms [6], tunings generated via bayesian learning [5], and the standard tunings found within the COCOMO risk model [1].

- COCOMO estimations are based on SLOC (delivered source lines of uncommented code). SLOC is notoriously hard to estimate. From Boehm's text Software Engineering Economics, we saw that using SLOC=10k, SLOC=100k, SLOC=2000k would cover an interesting range of software systems [2].

- Next, for the three different SLOCs and three tunings, we generated random examples by picking one value at random for each of the parameters from KC-1 in Figure 1. With our machine learner TAR2(see below), we get stabilized conclusion at sample size=30,000.

Part of the data log is shown on Figure 2. Each line contains 24 attributes(SLOC + 23 cost drivers) and a classification. There are total 4 classes indicating the software development risk is "low","medium","high" or "very high", denoted T_LO, T_MD, T_HI, T_VHI (respectively).

Given this data, there are two questions we could ask:

1. Does our KC-1 project belong to low risk project or high risk project?

sloc	*prec*	*flex*	*resl*	...	*site*	*sced*	*class*
100k	*0*	*2*	*1*	...	*2*	*0*	*T_MD*
100k	*1*	*4*	*2*	...	*2*	*1*	*T_LO*
100k	*0*	*1*	*2*	...	*2*	*0*	*T_MD*
10k	*0*	*4*	*0*	...	*2*	*2*	*T_LO*
2000k	*0*	*3*	*0*	...	*2*	*0*	*T_HI*
10k	*1*	*3*	*0*	...	*2*	*1*	*T_HI*
2000k	*1*	*2*	*1*	...	*2*	*2*	*T_MD*
2000k	*1*	*2*	*0*	...	*2*	*0*	*T_LO*
100k	*1*	*1*	*2*	...	*2*	*1*	*T_LO*

Figure 2: A log of risk estimation of kc-1 project.

2. What strategy can we take to reduce the development risk ?.

The first question is the question answered by standard machine learners which build classifiers that map examples to classes. In our experience, this question is asked less than the second questions. Managers care less for detailed descriptions of the current situation than for advice on how to improve the current situation. Hence we say that managers want *controllers* (i.e. an answer to question 2) more than mere classifiers (i.e. an answer to question1). To understand the distinction, consider the case of someone reading a map. Classifiers say "you are here" on the map while controllers say "go this way". After much experimentation, our preferred method of learning controllers is *treatment learners*.

3.3 TAR2 Treatment Learner

In our case study, the TAR2 treatment learner is implemented to explore the mass data generated by the COCOMO risk model. Classical machine learners like C4.5 aim at discovering classification rules: i.e. given a classified training set, they output rules that are predictive of the class variable. TAR2 differs from those learners in that:

1. TAR2 assumes the classes are ordered by their *score* (some domain-specific measure). Highly scored classes are preferable to lower scoring classes. Further, one class is more desirable than all others, which is called the *best* class.

2. Rather than finding the classification rules, TAR2 finds rules that predict both increased frequency of

#	outlook	temp(^oF)	humidity	windy?	class
1	sunny	85	86	false	none
2	sunny	80	90	true	none
3	sunny	72	95	false	none
4	rain	65	70	true	none
5	rain	71	96	true	none
6	rain	70	96	false	some
7	rain	68	80	false	some
8	rain	75	80	false	some
9	sunny	69	70	false	lots
10	sunny	75	70	true	lots
11	overcast	83	88	false	lots
12	overcast	64	65	true	lots
13	overcast	72	90	true	lots
14	overcast	81	75	false	lots

Figure 3: A log of some golf-playing behavior.

the best class and decreased frequency of the worst class. That is, TAR2 finds discriminate rules that drive the system away from the worst class to the best class.

3. TAR2 output treatments rather than classifications. A treatment is one or a conjunction of attribute value ranges. It is a constraint on future control inputs of the system.

TAR2 can find the input ranges of KC-1 that decrease the project's risk. Before showing that, we illustrate the TAR2 method using a small example. Figure 3 shows a small training set, in which there are four attributes and 3 classes. Recall that TAR2 assesses a *score* for each class. For a golfer, the classes in Figure 3 could be scored as *none=2* (i.e. worst), *some=4*, *lots=8* (i.e. best). TAR2 then seeks attribute ranges that occur more frequently in the highly scored classes than in the lower scored classes. Let $a = r$ be some attribute range e.g. *outlook=overcast*) and $X(a = r)$ be the number of occurrences of that attribute range in class X (e.g. *lots(outlook=overcast)=4*). $\Delta_{a=r}$ is a measure of the worth of $a = r$ to improve the frequency of the $best$ class. $\Delta_{a=r}$ uses the following definitions:

$best$**:** the highest scoring class; e.g. $best = lots$;

$rest$**:** the non-best class; e.g. $rest = \{none, some\}$;

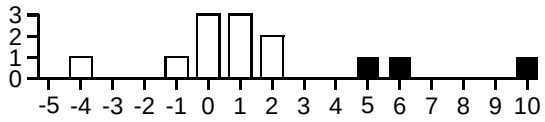


Figure 4: Δ distribution seen in golf data sets. Outstandingly high Δ values shown in black. Y-axis is the number of attribute ranges that have a particular Δ.

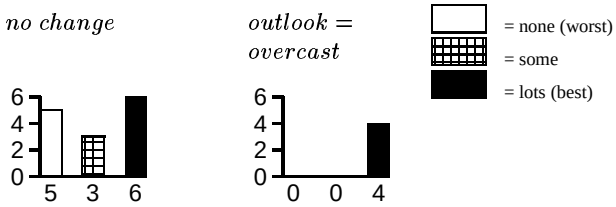


Figure 5: Finding treatments that can improve golf playing behavior. With no treatments, we only play golf lots of times in $\frac{6}{5+3+6} = 57\%$ of cases. With the restriction that *outlook=overcast*, then we play golf lots of times in 100% of cases.

$score$**:** The score of a class X is $\$X$;.

$\Delta_{a=r}$ is calculated as follows:

$$\Delta_{a=r} = \sum_{X \in rest} \left(\frac{(\$best - \$X) * (best(a = r) - X(a = r))}{|examples(a = r)|} \right)$$

where $|examples(a = r)|$ is the number of examples in which $a = r$ occurs ; The attribute ranges in our golf example generate the Δ histogram shown in Figure 4.

A *treatment* is a subset of the attribute ranges with an *outstanding* $\Delta_{a=r}$ value. For our golf example, such attributes can be seen in Figure 4: they are the outliers with outstandingly large Δs on the right-hand-side.

To *apply* a treatment, TAR2 rejects all example entries that contradict the conjunction of the attribute rages in the treatment. The ratio of classes in the remaining examples is compared to the ratio of classes in the original example set. The *best treatment* is the one that most increases the relative percentage of preferred classes. In our golf example, the best treatment is *outlook=overcast*; Figure 5 shows the class distribution before and after that treatment. i.e. if we bribe disc jockeys to always forecast overcast weather, then in 100% of cases, we should be playing lots of golf, all the time.

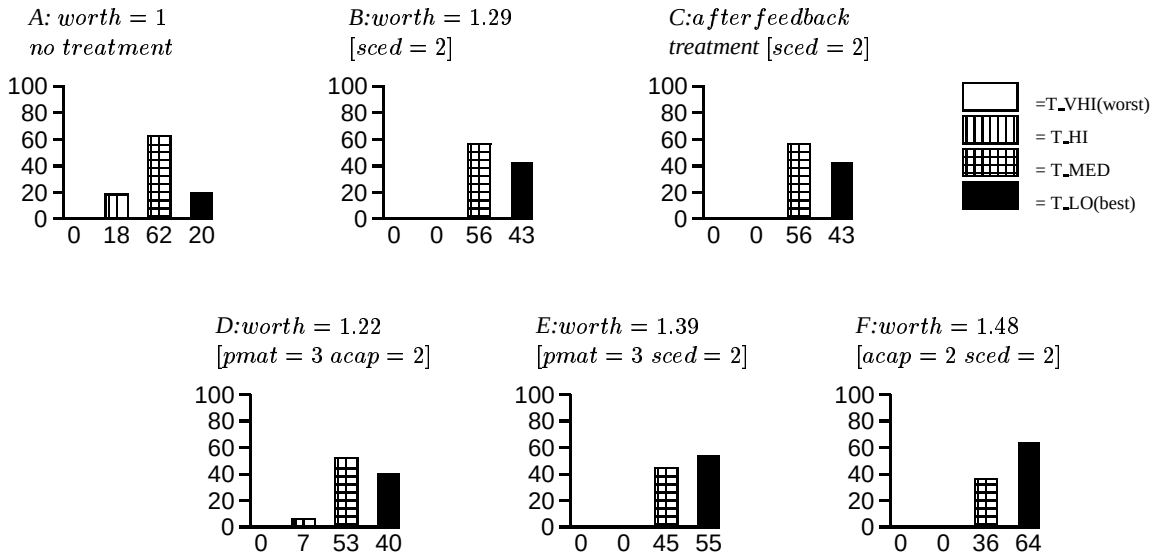


Figure 6: Improvements in class distributions generated by TAR2 from kc-1 data.

4 Experiment Results

Figure 6 shows the results from the KC-1 risk estimate data generated by TAR2. Since the classes are ordered, we introduce a variable "worth" to measure the class distribution of a data set. "worth" is calculated as follows:

$$worth = \sum_{X \in classes} (\frac{\$X * |X|}{|examples|})$$

where $|X|$ is the number of examples that belong to class X. TAR2 reports the worth's of the treated data sets as the ratio of the untreated raw data set. From Figure 6 we could see that 3 attribute values are particularly important in improving the class distribution; i.e.,$sced = 2$, $acap = 2$ and $pmat = 3$. When combined, the best treatment found by TAR2 was Figure 6F:

Acap=2 Use adequate analysts (near the 55% percentile).

Sced=2 Allow the team 100% of the estimated time to finish the project

Ignorables: One of the benefits of TAR2 is that it tells an analysts what factors are ignorable. In this case, of the 11 proposed modifications seen in Figure 1's *changes1* column, most of the modifications (i.e. *flex*, *resl*, *team*, *ruse*, *docu*, *stor*, *pcon*, *ltex*) were comparatively less effective that just changing *acap* and *sced*.

Figure 6F claims that in the case of *acap=2* and *sced=2*, our KC-1 project should have a 64% possibility to be low risk project(class=T_LO) and a 0% possibility of being a high risk project (class=T_HI). This is a considerable improvement over Figure 6A where the majority of projects were medium risk and there was a possibility (18%) of a high risk project.

TAR2 yields much information that can assist in software management. For example, suppose that in KC-1, schedule pressure is hard to avoid; i.e. using *sced=2* may be impractical. Fortunately, Figure 6D offers an alternative. In the case of *acap=2 AND pmat=3* (i.e. moving to an SEI CMM level3 style project [11]), an improvement over Figure 6A can be achieved. Note that the improvement is not as good as Figure 6E. That is, managers now can assess the merits of one treatment over another.

5 Validation

The treatments learnt from KC-1 and the COCOMO expert must be carefully assessed for their *stability*, *external validity* and *generality*. Stability and external validity are discussed below. Utility can be assessed using at least the following methods:

N-way cross-validation In this procedure, the available data is divided into N blocks so as to make each block's number of cases and class distribution as similar as possible. Next, N times, learning is performed on $\frac{N-1}{N}$ of the data and tested on the remaining $\frac{1}{N}$th of the data.

Model Feedback Ideally, the results can be applied to the model that generated the data. This is a more robust validation because the treatments are assessed by an outside device, avoided the effect of the training data.

5.1 Assessing Utility via Model Feedback Studies

In our case study, we are able to apply our treatments to the COCOMO risk model. For example, in order to test the treatment $sced = 2$, we did the following:

1. With other attribute remain randomized according to their available values seen in the *now1* column of Figure 1, we restrict $sced = 2$ (i.e. used only $sced = 2$ as the input value of $sced$), and generate a new data set of 30,000 cases.
2. If the treatment is valid, it should influence the model's behavior to generate data sets that have class distribution similar to the distribution seen in Figure 6B
3. Our validation results are quite satisfying: the new data set has exactly the same class distribution as Figure 6B predicted.(see Figure 6C.)

5.2 Stability

The essence of knowledge farming is:

- Identify ranges of possible values.
- Many times, pull input parameters from those ranges and run a model.
- Summarize the generated data using treatment learning.

Knowledge farming is a failure if emergent stable conclusions can't be found amongst the simulated data.

In this study, a model was run through the 10^6 possibilities shown in Figure 1, through three possible tunings and 2 possible SLOCs. Emergent stable properties were detected which, when tested, provided adequate control of KC-1 (recall the discussion in the last section).

5.3 External Validity

Are the treatments see in Figure 6 specific to KC-1 or general to other software projects? To test this, we did a control experiment:

- Without all the constrains on the KC-1 project, we ran the COCOMO risk model across all the possibilities. This simulation generated another data set of 30,00 examples. We called this data set the *generic-COCOMO* data set.
- TAR2 was applied to the generic-COCOMO data.

The treatments learnt from generic-COCOMO were different to the treatments learnt from the KC-1 data set. In generic-COCOMO, the best found treatment was:

Pmat=4: i.e. use a very mature software process;

Sced=4: i.e. stretch the development schedule by 160%;

These results are hardly surprising: given an overabundance of development time and a highly mature software process, of course we get low risk projects! However, there is a more serious lesson from this study:

- Treatments that are "best" in generic data sets may not be relevant to particular projects.
- Project specifics should always be used to constraint the simulations in order to generate treatments relevant to a particular project.

While the knowledge source used in this study (the COCOMO expert mode) has passed international peer review, it is hardly universally accepted. Proponents of some other knowledge source could reject the specifics of the above conclusions, but still use the general technique; i.e. they could encode their preferred knowledge source as a model, execute it, then summarize it using treatment learners. For example, Menzies and Kipers [9] explore the use of treatment learning to study a model of CMM level2 [12]. Elsewhere, we have discussed general principles for rapidly building models in early lifecycle [8].

6 Conclusion

Data droughts are an acute problem impeding which impede model-based requirements engineering. This paper explores a three-part solution to data droughts. An initial *knowledge farming* stage simulates the available models across the space of possibilities relevant to a particular project. This is followed by a second *harvesting* stage that uses the TAR2 treatment learner to summarize the logs of the simulations. If insights are found during the summarization stage, then these must be *validated* in a third and final stage using techniques such as N-way cross-validation or model feedback.

This three-part solution was explored here via an example based on a COCOMO-based risk model. For this case study, the proposed solution is both a *practical* (i.e. fast to run and simple to organize) and *effective* (i.e. as determined by our validation studies) method for reusing models during requirements engineering.

TAR2's summaries are *controllers* to the model. Such treatments, when applied to the model, actually change the model's behavior and drive it towards a preferred mode of operation. We are encouraged by the above results since it is possible that TAR2 can generate controllers even in the presence of data droughts.

References

[1] C. Abts, B. Clark, S. Devnani-Chulani, E. Horowitz, R. Madachy, D. Reifer, R. Selby, and B. Steece. COCOMO II model definition manual. Technical report, Center for Software Engineering, USC,, 1998. `http://sunset.usc.edu/COCOMOII/cocomox.html#downloads`.

[2] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1991.

[3] B. Boehm and P. Papaccio. Understanding and controlling software costs. *IEEE Trans. on Software Engineering*, 14(10):1462–1477, October 1988.

[4] I. Bratko, I. Mozetic, and N. Lavrac. *KARDIO: a Study in Deep and Qualitative Knowledge for Expert Systems*. MIT Press, 1989.

[5] 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.

[6] R. Cordero, M. Costamagna, and E. Paschetta. A genetic algorithm approach for the calibration of cocomo-like models. In *12th COCOMO Forum*, 1997.

[7] R. Madachy. Heuristic risk assessment using cost factors. *IEEE Software*, 14(3):51–59, May 1997.

[8] T. Menzies and Y. Hu. Building models for requirements engineering. In *Submitted to the first International Workshop on Model-based Requirements Engineering*, 2001. Available from `http://tim.menzies.com/pdf/01buildre.pdf`.

[9] T. Menzies and J. Kiper. Better reasoning about software engineering activities. In *ASE-2001*, 2001. Available from `http://tim.menzies.com/pdf/01ml4re.pdf`.

[10] 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://tim.menzies.com/pdf/00ase.pdf`.

[11] M. Paulk, B. Curtis, M. Chrissis, and C. Weber. Capability maturity model, version 1.1. *IEEE Software*, 10(4):18–27, July 1993.

[12] M. Paulk, C. Weber, B. Curtis, and M. Chriss. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, 1995.

[13] D. Pearce. The induction of fault diagnosis systems from qualitative models. In *Proc. AAAI-88*, 1988.

[14] J. Tian and M. Zelkowitz. Complexity measure evaluation and selection. *IEEE Transaction on Software Engineering*, 21(8):641–649, Aug. 1995.