Learning to Reduce Risks with COCOMO-II

Tim Menzies[1], Erik Sinsel[1], Tim Kurtz[2]

[1] NASA/WVU Software Research Lab, 100 University Drive, Fairmont, USA, 26554, +1-304-367-8447(p), +1-304-367-8211(f), `tim@menzies.com,esinsel@csee.wvu.edu`;
[2] NASA Glenn Research Center, Bldg.501 MS,501/4,21000 Brookpark Rd.bCleveland, Ohio, 44135,USA `Tim.Kurtz@grc.nasa.gov`
`http://tkurtz.grc.nasa/gov/spa`

Abstract. When a lack of data inhibits decision making, large scale what-if queries can be conducted over the uncertain parameter ranges. Such what-if queries can generate an overwhelming amount of data. In the case study explored here, machine learning was used to summarize the output of a Monte Carlo simulation of the COCOMO-II software effort estimation model. Based on that summary, key risk reduction factors were identified. Surprisingly, these factors was the same for both small and large projects suggesting that schedule risk mitigation strategies can be developed, even if SLOC estimates are inaccurate. This method of understanding models is general to many application areas.
Keywords: Machine learning, Monte-Carlo simulations, effort estimation, risk assessment, COCOMO-II, decision support systems.

1 Introduction

Incomplete information can inhibit decision making. For example, suppose a software manager wants to reduce the odds of time over-runs on her project. To perform this task, our manager could use a software effort estimation model like COCOMO-II (see Figure 1). However, to do so, our manager needs to supply values for all the variables listed in Figure 2 as well as an estimate for source lines of code (SLOC). What can our manager do if she is uncertain about some of those values?

In this paper, we describe a case study with using *machine learning* to summarize *Monte Carlo simulations* of COCOMO-II (the terms in *italics* are defined below). In summary, we use an intelligent agent to summarize very large scale "what-if" queries. Surprisingly, it turns out that for the purposes of risk mitigation, we can make definite conclusions despite having uncertain data. For example, we can identify COCOMO factors that are irrelevant to risk reduction. Further, these irrelevant factors are the same for large SLOC and small SLOC programs. That is, even though our estimates for SLOC may be very inaccurate, we still can develop risk mitigation strategies.

This study will make the following assumptions. In this article we focus on process risk and not product risk. That is, when we discuss "risk", we do not mean "time till the next error of severity X" (i.e. the standard view of risk in reliability engineering). Rather, we define risk as "risk of project schedule over-runs". We also make the "coarse-grained" assumption; i.e. risk mitigation strategies should not be based on fine tuning the details of a project's structure. Given the state-of-the-art in software management, we find it improbable that we will have such fine-grained control of a project. Rather, we seek coarse-grained changes that can aggressively push a high-risk project into a low-risk zone. Lastly, our study assumes that the effort-estimation model has been calibrated to the local conditions (for a discussion on calibration, see Figure 1). That is, the conclusions we reach are only relevant to those domains that match the calibrations of our model. Nevertheless, this paper does offer the following general conclusion: given a model relevant to a domain, it is a simple matter to understand that model using machine learning and Monte Carlo simulations. Further, while our case study here relates to COCOMO-II, the methods described below are broadly applicable. Machine learning over Monte Carlo simulations can be used to understand any device where (1) we can generate a large number of outputs very quickly; and (2) we can classify those outputs (for other applications of this technique, see [3, 8, 10]).

This paper is structured as follows. We begin with brief tutorials on Monte Carlo methods and machine learning. We then discuss four key problems in com-

The COCOMO project aims at developing at open-source, public-domain software effort estimation model. COCOMO has been built using the experience of the COCOMO team and the project database. According to [4], the current project database contains information on 161 projects collected from commercial, aerospace, government, and non-profit organizations. 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, the core COCOMO-II effort estimation equations is:

$$months = a * \left(KSLOC^{\left(1.01+\Sigma_{i=1}^{5} SF_i\right)}\right) * \left(\prod_{j=1}^{17} EM_j\right)$$

where a is a domain-specific parameter, and KSLOC is estimated directly or computed from a function point analysis. SF_i are the scale factors (e.g. factors such as "have we built this kind of system before?") and EM_j are the cost drivers (e.g. required level of reliability). Figure 2 lists all the SF_i and EM_j.

Software effort-estimation models like COCOMO-II should be tuned to their local domain. Off-the-shelf "untuned" models have been up to 600% inaccurate in their estimates; e.g. [7, p165] and [5]. However, tuned models can be far more accurate. For example [4] reports a study with a Bayesian tuning algorithm using the COCOMO project database. After Bayesian tuning, a cross-validation study showed that COCOMO-II model produced estimates that are within 30% of the actuals, 69% of the time.

Fig. 1. Some background notes on COCOMO-II. For more details, see [1]

.

Acronym	Can change	Definition	Low-end	Medium	High-end
acap	yes	analyst capability	worst 15%	55%	best 10%
aexp	maybe	applications experience	2 months	1 year	6 years
cplx	no	product complexity	e.g. simple read/write	e.g. use of simple interface widgets	e.g. critical real-time systems
data	no	DB size (DB bytes/ SLOC)	10	100	1000
docu	no	documentation	many phases undocumented		extensive reports, full life-cycle
flex	yes	development flexibility	rigorously defined	some relaxed guidelines	only general goals
ltex	yes	language, tool-set experience	2 months	1 year	6 years
pcap	yes	programmer capability	worst 15%	55%	best 10%
pcon	no	% personnel change per year	48%	12%	3%
pexp	yes	platform experience	2 months	1 year	6 years
pmat	no	process maturity	CMM level 1	CMM level 3	CMM level 5
prec	yes	precedentedness	never built this software before	somewhat new	thoroughly familiar
pvol	maybe	$\frac{platformchanges}{year}$ $(\frac{major\ changes}{minor\ changes})$	$\frac{12\ months}{1\ month}$	$\frac{6\ months}{2\ weeks}$	$\frac{2\ weeks}{2\ days}$
rely	no	required reliability	errors mean slight inconvenience	errors are easily recoverable	errors can risk human life
resl	yes	architecture or risk resolution	interfaces known, risk removed=few	=most defined/removed	all defined/used
ruse	maybe	required reuse	none	across program	across multiple product lines
sced	yes	required development time	75% of original estimate	no change	160% of original estimate
site	yes	multi-site development	some contact: phone, mail	some email	interactive multimedia
stor	yes	main storage constraints (RAM%)	N/A	50%	95%
team	yes	team cohesion	very difficult interactions	basically co-operative	seamless interactions
time	yes	execution time (CPU%)	N/A	50%	95%
tool	yes	use of software tools	edit,code,debug		well intergrated with lifecycle

Fig. 2. Parameters of the COCOMO-II effort risk model; adapted from `http://sunset.usc.edu/COCOMOII/expert_cocomo/drivers.html`. "Can change" denotes the variables that we and Boehm [2] declare can be changed during a project.

bining these methods: *stability, similarity, satisfactory-ness*, and *summarization.* Finally, we show the factors in our COCOMO-II model which are not usually useful in developing coarse-grained risk mitigation strategies for 100 KSLOC and 20,000 KSLOC projects.

2 An Introduction to Monte Carlo Simulations

Woller [11] defines Monte Carlo methods as follows:

> Monte Carlo (MC) methods are stochastic techniques–meaning they are based on the use of random numbers and probability statistics to investigate problems. You can find MC methods used in everything from economics to nuclear physics to regulating the flow of traffic.

Traditionally, MC methods were used for mathematical integration. For example, to compute the value of π using MC, we might ask a very bad darts player to throw darts at Figure 2. In this approach, our darts player is a stochastic generator of the data. Assuming all the darts land within the square of Figure 2.A, then the ratio of darts hitting the circle will be:

$$\frac{\#\ \mathit{darts\ hitting\ circle}}{\#\ \mathit{throws}} = \frac{\pi r^2}{(2r)^2}$$

which we can rearrange to

$$\pi = \frac{4 * \#\ \mathit{darts\ hitting\ circle}}{\#\ \mathit{throws}} \tag{1}$$

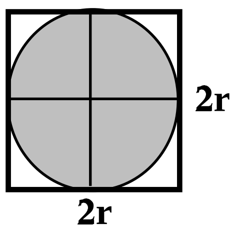

Figure 2.A: A circle of radius r within a square of side $2r$. From [11].

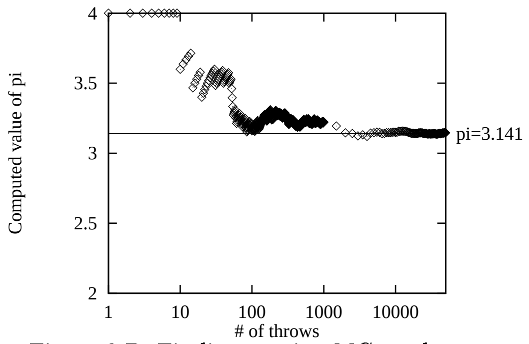

Figure 2.B: Finding π using MC methods; i.e. throwing darts at Figure 2.A and applying Equation 1. Adapted from [11].

Note that the results of an MC study can be skewed by inadequate sampling. If our darts player only makes a few throws, our value for π will be inaccurate. Figure 2.B shows how the value of Equation 1 varied as a computer program simulated the darts player. The value for π seen after 10 "throws" was very different to the value seen after 1000 "throws". However, after 5000 "throws", the value *stabilized*; i.e. more throws did not significantly change the value. The general lesson from this example is that the sample size of an MC study must be extended until the conclusions *stabilize*.

Apart from mathematical integration, MC can be used to perform large-scale "what-if" queries. Theoretically, by studying the results from an MC simulation over the uncertain COCOMO-II input values, a software manager manager could:

- Detect what combination of the uncertain values will lead to schedule overruns.
- Determine effort-risk mitigation strategies. For example, our manager might see that if a single parameter (e.g. `time` in Figure 2) is changed, then the estimate of the time to complete the project can be reduced.

Unfortunately, practical considerations may prevent the recognition of schedule over-run factors or the determination of effort-risk mitigation factors:

- COCOMO-II factors are rated on scale with, on average, 5 points: "very low, low, nominal, high, very high".
- Suppose our manager is uncertain on half the 22 factors in Figure 2. A full MC study would require up to $5^{11} \approx 50,000,000$ runs. That is, our manager might be overwhelmed with by a mountain of information.

We now face a dilemma. On the one hand, if we use large samples for our MC studies, our decision makers may be overwhelmed with information. On the other hand, if we don't use large sample sizes, the conclusions may not stabilize (recall that our value for π did not stabilize till after 5000 samples). The rest of this paper tries to resolve this dilemma as follows:

- Use large scale MC runs to collect the data.
- Use machine learning to summarize that data.

3 Machine Learning for Data Reduction

This section introduces decision tree learners using notes from [9].

C4.5 is an algorithm that learns decision trees from examples. It is an international standard in machine learning: most new machine learners are benchmarked against this program. Decision tree learners work from classified examples such as lines 1-15 of Figure 3. C4.5 uses a heuristic entropy measure of information content to build its trees. The parameter range with the most information content is selected as the root of a decision tree. The example set is then divided up according to which examples do/do not satisfy the test in the root. For each divided example set, the process is then repeated recursively. A statistical measure is then used to estimate the classification error on unseen cases.

For example, consider lines 16-24 in Figure 3. In this tree, C4.5 has decided that the weather outlook has the most information content. Hence, it has placed outlook at the root of the learnt tree. If `outlook=rain`, a sub-tree is entered (lines 21-22). In the `outlook=rain` sub-tree, the critical next parameter was `wind`. On line 21 we read that we should not play golf on high-wind days when it might rain. Note the error estimate on line 24. C4.5 estimates that this tree will lead to incorrect classifications 38.5 times out of 100 on future cases. We should expect such a large classification errors when learning from only 15 examples.

INPUT:

```
#outlook, temp, humidity, windy,   class
sunny,    85,   85,       false,   dont_play
sunny,    80,   90,       true,    dont_play
overcast, 83,   88,       false,   play
rain,     70,   96,       false,   play
rain,     68,   80,       false,   play
rain,     65,   70,       true,    dont_play
overcast, 64,   65,       true,    play
sunny,    72,   95,       false,   dont_play
sunny,    69,   70,       false,   play
rain,     75,   80,       false,   play
sunny,    75,   70,       true,    play
overcast, 72,   90,       true,    play
overcast, 81,   75,       false,   play
rain,     71,   96,       true,    dont_play
```

OUTPUT:

```
outlook = overcast: Play
outlook = sunny:
|   humidity <= 75 : Play
|   humidity > 75 : Don't Play
outlook = rain:
|   windy = true: Don't Play
|   windy = false: Play

ESTIMATED ERROR ON UNSEEN CASES: 38.5%
```

Fig. 3. Decision-tree learning. Classified examples (above) generate the decision tree (below).

In general, C4.5 needs hundreds to thousands of examples before it can produce trees with low classification errors.

Recall that C4.5 selects the parameter ranges for the tree using an information theoretic measure. That is, potentially, the algorithm can reduce large example sets. Parameter ranges with high information content appear high in the tree (e.g. `outlook`). Similarly, attribute values with low information content may disappear from the tree all together. For example, note that `temperature` does not appear in the learnt trees of Figure 3.

4 MC and C4.5

This section explores the following idea: if C4.5 can reduce large data sets, perhaps it can assist us with MC studies of COCOMO-II models. In order to stress test this technique, we assumed that our manager knows *none* of the values for the 22 COCOMO-II input parameters; i.e. our MC study may have to execute COCOMO-II up to $5^{22} \approx 10^{15}$ times.

We estimate that to run a COCOMO-II model 10^{15} times would take a century to execute[3]; i.e. it is not practical to explore 10^{15} input combinations to COCOMO-II. Hence, we ran COCOMO-II using randomly selected inputs. Like any MC study, this implied that we had to keep increasing our sample sizes until we detected stability in our conclusions.

The details of our approach are shown in Figure 4. In summary, we built some

```
01. model        ← "cocomoII"
02. sampleSize   ← 1000
03. i            ← 1
04. repeat forever
05. { for j   ← 1 to 5
06.   { repeat sampleSize number of times
07.     { in  ← random selection of model inputs
08.       out  ← call(model,in)
09.     }
10.     tree[j]  ← c45(in,out)
11.   }
12.   for j  ← 1 to 5
13.   { for k  ← j+1 to 5
14.      { delta[i]  ← delta[i]+compare(tree[j],tree[k])
15.   }  }
16.   if    (i > 2                 and
17.         similar(delta[i])       and
18.         satisfactory(tree[1])   and
19.         stable(delta[i], delta[i-1], delta[i-2]))
20    then return summarize(tree[1])
21.   else  { i  ← i + 1
22.           sampleSize  ← 2 * sampleSize.
23.         }
24.   fi
25. }
```

Fig. 4. MC-based machine learning

decision trees using some randomly selected inputs to a COCOMO-II model (lines 5-11). Next, for all pairs of our trees, we collected a list of differences (a.k.a. `deltas`) between them (lines 12-15). At lines 16-19, we check for:

Similarity: Are all our trees telling us roughly the same thing? If so, then we are using enough examples to capture the semantics of the model.

Satisfactory: In Figure 3, we saw that C4.5 can generate trees with large classification errors. That is, when we assess learnt trees, we should ensure that their classifications are sufficiently accurate. Note the use of `tree[1]` at line 18. By line 18, we have demonstrated tree similarity. Hence, testing any one of the learnt trees will let us assess all the trees.

Stability: Are the trees we see at iteration `i` telling us roughly the same thing as iterations `i-1` and `i-2`? If so, then we declare that the conclusions reached

[3] Assuming we can run COCOMO-II one million times per second, then if a year is $\approx 10^7$ seconds, our 10^{15} runs would take $\frac{10^{15}}{10^6 * 10^7} = 10^2 = 100$ years.

by this MC method have stabilized.

If we can't demonstrate that are trees are similar, stable, and satisfactory, then we double the sample size and start over again at line 5. Otherwise, we can use any of the trees learnt at iteration `i` as our output (line 20). Note the `summarize` function at line 20: this function reports decision trees in a format useful to a software manager.

To implement Figure 4, several practical issues had to be addressed:

1. We needed a working version of COCOMO-II which we could run hundreds of thousands of times.
2. C4.5 works using classified examples (recall lines 1-15 of Figure 3). Where are we to get the classifications?
3. How exactly do we test that our trees are similar, stable, and satisfactory?
4. What is the best format for summarizing the trees?

This issues are addressed below.

4.1 Accessing a COCOMO-II model

For our experiments, we used the Madachy COCOMO-based effort-risk model [6] (Dr. Madachy is one of the authors of the COCOMO-II model description [1]). The Madachy model was an experiment in explicating the heuristic nature of effort estimation. The model contains 94 tables of the form of Figure 5. Each such table implements a context-dependent modification to internal COCOMO parameters.

An important feature of the Madachy model is that it generates a numeric effort-risk index. Studies with the COCOMO-I project database have shown that this index correlates well with $\frac{months}{KDSI}$ (where KDSI is thousands of delivered source lines of code) [6].

4.2 Classifying COCOMO-II Outputs

According to Madachy, his risk index can be classified as follows: "low effort-risk" ($index < 5$), "medium effort-risk" ($5 \leq index \leq 15$), or "high effort-risk"

	rely				
	very low	low	nominal	high	very high
sced= very low	0	0	0	1	2
sced= low	0	0	0	0	1
sced= nominal	0	0	0	0	0
sced= high	0	0	0	0	0
sced= very high	0	0	0	0	0

Fig. 5. A Madachy factors table. From [6]. This table reads as follows. In the exceptional case of high reliability systems and very tight schedule pressure (i.e. `sced=low` or very low and `rely= high or very high`), add some increments to the built-in parameters (increments shown top-right). Otherwise, in the non-exceptional case, add nothing to the built-in parameters.

```
sced = 0 :
|   pcon <= 1 : medium
|   pcon > 1 :
|   |   cplx <= 1 : high
|   |   cplx > 1 :
|   |   |   tool <= 3 : medium
|   |   |   tool > 3 : high
sced > 0 :
|   acap <= 1 :
|   |   rely <= 3 :
|   |   |   stor <= 4 :
|   |   |   |   acap = 0 : medium
|   |   |   |   acap > 0 :
|   |   |   |   |   ruse > 2 : medium
|   |   |   |   |   ruse <= 2 :
|   |   |   |   |   |   docu <= 2 : low
|   |   |   |   |   |   docu > 2 : medium
|   |   |   stor = 5 :
|   |   |   |   pcon <= 2 : high
|   |   |   |   pcon > 2 : medium
|   |   rely > 3 :
|   |   |   pcap <= 2 : high
|   |   |   pcap > 2 : medium
|   acap > 1 :
|   |   pcap <= 1 :
|   |   |   pmat > 2 : medium
|   |   |   pmat <= 2 :
|   |   |   |   sced > 3 : medium
|   |   |   |   sced <= 3 :
|   |   |   |   |   cplx <= 1 : medium
|   |   |   |   |   cplx > 1 : low
|   |   pcap > 1 :
|   |   |   ruse <= 2 : low
|   |   |   ruse > 2 :
|   |   |   |   time = 5 : medium
|   |   |   |   time <= 4 :
|   |   |   |   |   pexp <= 1 : medium
|   |   |   |   |   pexp > 1 : low

ESTIMATED ERROR ON UNSEEN CASES = 25.4%
```

Fig. 6. A decision tree of 39 nodes generated from 125 randomly selected inputs to COCOMO-II with KSLOC=100. Parameters are ranked on the scale 0=very low; 1=low; 2=nominal; 3=high; 4=very high; 5=extremely high.

($index > 15$). The Madachy tool is a CGI-based tool (see `http://sunset.usc.edu/COCOMOII/expert_cocomo/`). After downloading the C-source code, we removed the HTML components and added some shell scripts. The result was a command-line interface version that accepted COCOMO-II input parameters and outputed a classification: "low", "medium", "high". With this tool, we could generate millions of classified COCOMO-II outputs in under an hour. From these outputs, we could learn decision trees such as Figure 6.

4.3 Detecting Satisfactory Trees

Figure 7 shows the effects of increasing sample size on tree size and classification error. As sample size grows, the estimated error on unseen cases decreases exponentially while the tree size grows linearly. We conclude that there is some benefit in using sample sizes of tens of thousands, but probably little benefit in moving above 20,000 samples.

These plots also serve to introduce why we need the `summarize` function of Figure 4, line 20. If we sample 20,000 times, we will be generating trees 100 times bigger than Figure 6 (recall that Figure 6 was generated from 125 samples). Such large trees need to be condensed in order to make them human readable.

4.4 Summarizing Trees

This section describes heuristics for summarizing large decision trees. To illustrate these heuristics, we use Figure 8 which was manually created for demonstration purposes.

Focus on risk mitigation strategies: One goal of our research is the generation of risk mitigation strategies. Hence, we need only report those parts of the trees that show us how to change a projects risk from $RISK_1$ to $RISK_2$. For example, suppose our machine learner had generated Figure 8. If a project arrives at line 9 of Figure 8, then it is a high-risk project and $RISK_1 = high$. One risk mitigation strategy might be to reduce the CPU time requirements (i.e. make $\texttt{time} \leq 4$). This strategy would convert $RISK_1 = high$ at line 9 to $RISK_1 = medium$ at line 8.

Focus on "powerful" risk mitigation strategies: We can further condense our trees if we focus only the strategies that can drive a high risk project to a low risk project; i.e. $RISK_1 = high$ and $RISK_2 = low$. Adopting this heuristic means we would ignore the "reduce CPU time requirements" described above since this merely converts us from $RISK_1 = high$ to $RISK_1 = medium$. A more powerful risk mitigation strategy would be to target $RISK_1 = low$ at line 19. Figure 9 discusses the implications of this strategy.

Ignore impossible changes: Column 2 of Figure 2 reports which COCOMO-II factors can not be changed within the lifetime of one software project: `cplx, data, docu, pcon, pmat, rely`. We will exclude all risk mitigation strategies that require us to change these unchangeable parameters. Continuing our example, we note that none of the changes in Figure 9 are impossible.

Focus only on "coarse-grained" strategies: Most software managers only have a very coarse-level of control of their projects. Hence, some of the fine-grained

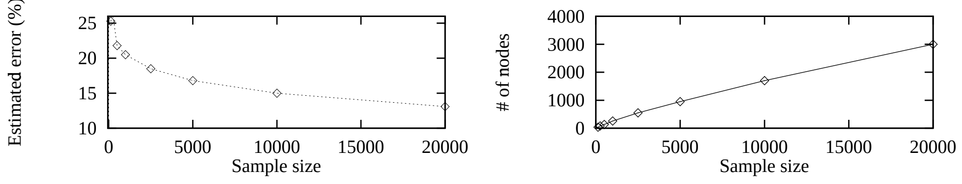

Fig. 7. Changes to sample size, estimated error, and tree size.

details required by Figure 9 are improbable. For example, the strategy reported in Figure 9 requires us to hold the project schedule at exactly 0.75; i.e. `sced=0`. This strategy was based on the Figure 8 sub-tree starting at line 11. Any increase to our schedule (i.e. `sced>0`), throws us into the Figure 8 sub-tree starting at line 20 and invalidates the Figure 9 strategy. Hence, we will not report such risk mitigation strategies and only display coarse-grained pivotal changes. For our purposes, we declare that a pivotal change is coarse if it includes one end value and at least two range values. Such pivotal changes serve to divide the total range of a parameter into two sets: useful and useless Hence, (e.g.) $\texttt{acap} \leq 3$ is coarse but $\texttt{acap} = 2$ is not. Note that if we restrict ourselves to coarse-grained strategies, then we must discard Figure 9. Figure 10 shows the only coarse-grained powerful risk mitigation strategy of Figure 8. Note the good news: we can reduce risk while using less experienced programmers (see the changes made to `pcap`).

Report only "pivotal values": A *pivotal value* is some parameter change that can drive a high risk project to a low risk project. Figure 10 shows the following pivotal values: `acap`>0, `sced`>1, `pcap`> 0. If we report only pivotal values, we are reporting all the changes that are effective and powerful risk mitigation strategies.

```
01. acap = 0 :
02. |   sced = 0:high
03. |   sced > 0
04. |   |   pcap <= 1:high
05. |   |   pcap > 1
06. |   |   |   cplx > 3:high
07. |   |   |   cplx <= 3
08. |   |   |   |   time <= 4:medium
09. |   |   |   |   time = 5:high
10. acap > 0
11. |   sced = 0
12. |   |   pcap = 0:medium
13. |   |   pcap > 0
14. |   |   |   time = 5: medium
15. |   |   |   time <= 4
16. |   |   |   |   aexp = 0:medium
17. |   |   |   |   aexp > 0
18. |   |   |   |   |   ltex = 0:medium
19. |   |   |   |   |   ltex > 0:low
20. |   sced > 0 :
21. |   |   pcap > 0
22. |   |   |   ltex < 1 :medium
23. |   |   |   ltex >= 1
24. |   |   |   |   sced <= 1:medium
25. |   |   |   |   sced > 1
26. |   |   |   |   |   ruse <= 4:low
27. |   |   |   |   |   ruse > 4:medium
...
```

Fig. 8. An artificially generated decision tree, built for demonstrated purposes. Parameters are ranked as per Figure 6.

	old	line	new	line	comments
acap	$= 0$	1	> 0	10	Small increase required
sced	> 0	3	$= 0$	11	Can reduce risk while delivering earlier.
pcap	> 1	5	> 0	13	Can reduce risk while using less-skilled programmers.
cplx	≤ 3	7	-	-	Irrelevant to changing risk
time	$= 5$	9	≤ 4	15	Have to decrease CPU requirements
aexp	-	-	> 0	17	your analysts can't be novices
ltex	-	-	> 0	19	Schedule training sessions for your tools

Fig. 9. Comparing a $RISK_1 = high$ project at line 9 of Figure 8 to a $RISK_2 = low$ project at line 19 of Figure 8. Line numbers refer to line numbers of Figure 8.

	old	line	new	line	comments
acap	$= 0$	1	> 0	10	Small increase required
sced	> 0	3	> 1	25	To reduce risk, need more development time.
pcap	> 1	5	> 0	13	Can reduce risk while using less-skilled programmers.
cplx	≤ 3	7	-	-	Irrelevant to changing risk
time	$= 5$	9	-	-	No need to decrease CPU requirements
ltex	-	-	≥ 1	23	Schedule training sessions for your tools, and hire in one guru in your toolset.
reuse	-	-	≤ 4	26	Don't try to develop reusable components across product lines.

Fig. 10. Comparing a $RISK_1 = high$ project at line 9 of Figure 8 to a $RISK_2 = low$ project at line 26 of Figure 8. Line numbers refer to line numbers of Figure 8.

4.5 Measuring Stability and Similarity

The above discussion has dealt with classification and how to summarize a tree. This section deals with the remaining technical issues of Figure 4: stability and similarity.

We begin by observing that each pathway in a decision tree from the tree root to a leaf is a logical rule. To compute stability and similarity, we annotate every pair of such rules with one of four terms, described below: *clashes*, *same*, *shrinks-X*, and *grows-Y*.

"Sames" detects logical equivalence. Equivalence can be reported if the same pathway exists in two trees.

"Shrinks-X" detects a special case of logical subsumption. Consider `rule1` which came from some learnt decision tree `treeA`:

```
rule  1 from  treeA
if    analyst capability is very low
      and schedule pressure is very tight
then  risk=high
```

Suppose another tree called `treeB` contained the pathway:

```
rule 2 from treeB
if   analyst capability is nominal or low or very low
     and schedule pressure is very tight
then risk=high
```

We say that treeA "shrinks-2" from treeB; i.e. when we compare rule2 in treeB to rule1 in treeA, we see that two range points for analyst capability has been removed (the references to low and nominal).

"Grows-Y" detects another special case of logical subsumption. Whereas Shrinks-X detects parameter ranges shrinking, grows-Y detects the number of parameters increasing. For example, suppose a tree called treeC contained the pathway:

```
rule 3 from treeC
if   analyst capability is nominal or low or very low
     and schedule pressure is very tight
     and language and tool experience is very limited
then risk=high
```

Note that rule3 is the same as rule2, but mentions one extra parameter; i.e. language and tool experience. From this comparison of rule3 in treeC and rule2 in treeB, we say that treeC grows-1 from treeB.

"Clashes" detects logical contradictions. Consider rule4:

```
rule4 from  treeD
if   analyst capability is very low
     and schedule pressure is  nominal
then risk=high
```

We say that treeD "clashes" with treeA since there exists one rule in treeD and one rule in treeA with the same conclusion but conflicting pre-conditions (nominal schedule pressure in rule4 versus very tight schedule pressure in rule1). If two rules clash, we do not explore shrinks-X or grows-Y.

With these definitions in hand, we can now describe the delta[i] set computed at line 14 of Figure 4. For each pair of pathways from each tree being compared, all the clashes, sames, shrinks-X, and grows-Y are found and added to the delta[i] set. Internally, these are stored as frequency counts. For example, if the pathway comparisons generated:

```
{grows-2, grows-2
 shrinks-1, shrinks-1, shrinks-1,
 grows-3,
 clashes, clashes
 same, same, same}
```

then this is stored internally as:

```
delta[i]= {grows-2   = 2
           shrinks-1 = 3
           grows-3   = 1
           clash     = 2
           same      = 3}
```

Figure 11 shows how delta[i] changed as we increased the sample size. In terms of detecting stability, Figure 11 shows that our comparison measures all plateau at sample size ≥ 5000 (execption: shrink-1). That is, our conclusions are stable above that point. In terms of detecting similarity, clearly clashes and sames are most important and shrinks-X and grows-Y are least important. However,

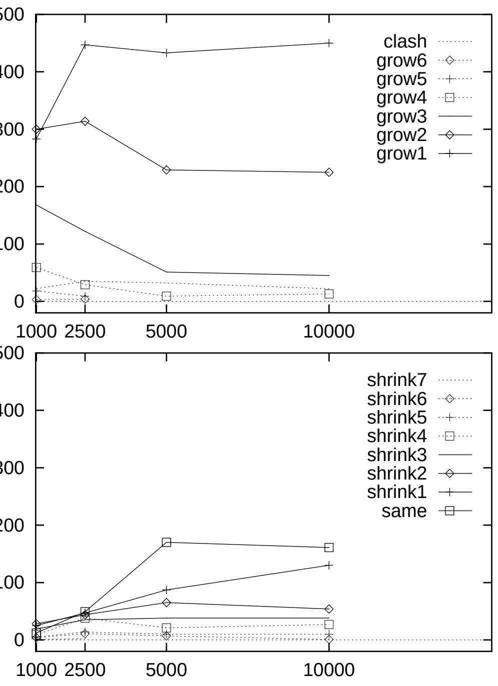

Fig. 11. Frequency counts of members of `delta[i]` (y-axis) vs sample size (x-axis).

the shrinks and grows measures let us assess the degree to which our trees subsume each other. Note that at sample size ≥ 5000, the clashes are rare and the sames are common. Further, the shrinks-X and grows-Y values are very low; i.e. $\{X, Y\} \in \{1, 2\}$. That is, at sample size ≥ 5000, the trees only slightly subsume each other.

In summary, at sample size ≥ 5000, we can confirm similarity and stability. Nevertheless, we will use sample sizes of $20,000$ since we argued above that this generated satisfactory trees.

5 Results

Having determined the best sample size, and having implemented tools to report the possible pivotal changes of powerful risk mitigation strategies, we next performed two studies:

Effort risk for programming-in-the-small: Summarize an MC simulation where KSLOC=100; i.e. small programs.

Effort risk for programming-in-the-large: Summarize an MC simulation where KSLOC=20,000; i.e. large programs.

For each study, we counted the number of times a COCOMO-II parameter appeared in a possible pivotal change in a powerful risk mitigation strategies (i.e. can drive a high-risk project to a low-risk project). This generated (e.g.) $X_{\texttt{sced}}$ number of reports where some parameter (e.g.) `sced` was pivotal. The frequency

	KSLOC=20,000	KSLOC=100
sced	16.3 %	18.0 %
acap	15.9 %	16.0 %
time	15.4 %	13.0 %
pcap	15.4 %	16.8 %
tool	10.2 %	9.8 %
reuse	7.8 %	9.2 %
stor	5.8 %	4.4 %
aexp	5.7 %	5.8 %
ltex	5.2 %	4.1 %
pexp	1.2 %	1.7 %
pvol	0.4 %	0.2 %
team	0.2 %	0.2 %
site	0.1 %	0.1 %
prec	0.1 %	0.2 %
flex	0.1 %	0.1 %

Fig. 12. Percentage of times a parameter appears as a pivotal change in a powerful risk mitigation strategy. Sorted by column 2.

counts for each parameter was then expressed as percentage of the total reports i.e.

$$percentage(X_i) = \frac{X_i * 100}{\sum_{j=1}^{16} X_j}$$

where X_j was one of the 16 parameters that we can change within a project (see the parameters marked "yes" or "maybe" in the 'changeable?" column of Figure 2). The results are shown in Figure 12. The parameters that appear high in Figure 12 were often pivotal in converting high-risk projects into low-risk projects. That is, for projects that match the calibrations of Madachy's COCOMO-II risk model, the factors that often change high-risk projects to low-risk are:

- schedule pressure (`sced`)- most critical
- analyst capability (`acap`)- next most critical
- CPU requirements (`time`)
- programmer capability (`pcap`)
- the integration of tools with the software life-cycle (`tool`)
- the required level of reuse (`reuse`)
- RAM requirements (`stor`)
- analyst experience (`aexp`)
- experience with the language and toolset (`ltex`).

Hence, when designing risk mitigation strategies, we would tend to focus on these factors. Further, we would tend to avoid the parameters at the bottom of Figure 12 such as programmer experience `pexp`, platform volatility `pvol`, team cohesion `team`, multi-site development `site`, precedentedness `prec`, and process flexibility `flex`.

Note that factors like process maturity (`pmat`) do not appear in Figure 12 since they can never be pivotal (recall that `pmat` is one of the parameters marked "no" in the "changeable?" column of Figure 2).

Comparing columns 2 and 3 of Figure 12, we see that approximately the ranking of parameters appears for small projects (KSLOC=100) as for large projects (KSLOC=20,000). This is an exciting result since it suggests we can assess effort-risk even when we lack accurate estimates for KSLOC.

6 Conclusion

When we are unsure of a particular value, we should execute what-if queries over the range of our uncertainty. However, such a what-if query can overwhelm a user with information. For example, a full simulation of COCOMO-II would require 10^{15} runs. We have shown here that small random samples from a Monte Carlo simulation can generate satisfying, stable, and similar decision trees. However, these trees can still be very big: the trees we learnt from 20,000 COCOMO-II samples was 3000 nodes; i.e. 100 times bigger than Figure 6. While a computer program could execute such a structure, human beings have a hard time processing that much information. Hence, we need to summarize the summary generated by decision tree learners. The following heuristics proved useful in reducing two 3000 node decision trees down to Figure 12:

- Only report the pivotal parameters that ...
- ... appear in coarse-grained risk mitigation strategies ...
- ... that can drive a project from high-risk to low-risk ...
- ... without requiring some impossible change to a project feature.

We stress again that the rankings shown in Figure 12 apply to those projects which match the COCOMO-II parameter settings within the Madachy model. However, given other parameter settings, the above analysis could easily be repeated to generate other powerful risk mitigation strategies. We view this as a strength of our technique: mitigation strategies can be customized to the particulars of different domains.

ACKNOWLEDGEMENTS

Barry Boehm and Ray Madachy were kind enough to grant us access to the source code of the Madachy COCOMO-based risk assessment model. John R. Josephson first proposed that we run a small number of randomly selected inputs to the system. Dr. Josephson also first proposed the "powerful" mitigation strategy heuristic for summarizing our results. Naresh Iyesr enthusiastically supported our earlier experiments with summarizing hundreds of thousands of calls to COCOMO-II. Siamak Yassini released the first author from some administration tasks to allow for the creation of this article. This work was partially supported by NASA through cooperative agreement #NCC 2-979.

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, 1981.
3. I. Bratko, I. Mozetic, and N. Lavrac. *KARDIO: a Study in Deep and Qualitative Knowledge for Expert Systems.* MIT Press, 1989.
4. 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.
5. C.F. Kemerer. An Empirical Validation of Software Cost Estimation Models. *Communications of the ACM*, 30(5):416–429, May 1987.
6. R. Madachy. Heuristic Risk Assessment Using Cost Factors. *IEEE Software*, 14(3):51–59, May 1997.
7. T. Mukhopadhyay, S.S. Vicinanza, and M.J. Prietula. Examining the Feasibility of a Case-based Reasoning Tool for Software Effort Estimation. *MIS Quarterly*, pages 155–171, June 1992.
8. D. Pearce. The Induction of Fault Diagnosis Systems From Qualitative Models. In *Proc. AAAI-88*, 1988.
9. J.R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1:81–106, 1986.
10. C. Sammut, S. Hurst, D. Kedzier, and D. Michie. Learning to Fly. In D. Sleeman, editor, *Ninth International Conference on Machine Learning*, pages 385–393. Morgan Kaufmann, 1992.
11. J. Woller. The Basics of Monte Carlo Simulations, 1996. University of Nebraska-Lincoln Physical Chemistry Lab. Available at `http://wwitch.unl.edu/zeng/joy/mclab/mcintro.html`.