Handbook of Software Engineering and Knowledge Engineering
Volume 1
© World Scientific Publishing Company

Practical Machine Learning for Software Engineering and Knowledge Engineering

Tim Menzies
Department of Electrical & Computer Engineering
University. of British Columbia, Canada
tim@menzies.com; http://tim.menzies.com

Abstract

Machine learning is practical for software engineering problems, even in data-starved domains. When data is scarce, knowledge can be *farmed* from *seeds*; i.e. minimal and partial descriptions of a domain. These seeds can be *grown* into large datasets via Monte Carlo simulations. The datasets can then be *harvested* using machine learning techniques. Examples of this *knowledge farming* approach, and the associated technique of *data-mining*, is given from numerous software engineering domains.

1. Introduction

Machine learning (ML) is not hard. Machine learners automatically generate summaries of data or existing systems in a smaller form. Software engineers can use machine learners to simplify systems development. This chapter explains how to use ML to assist in the construction of systems that support classification, prediction, diagnosis, planning, monitoring, requirements engineering, validation, and maintenance.

This chapter approaches machine learning with three specific biases. First, we will explore machine learning in data-starved domains. Machine learning is typically proposed for domains that contain large datasets. Our experience strongly suggests that many domains lack such large datasets. This lack of data is particularly acute for newer, smaller software companies. Such companies lack the resources to collect and maintain such data. Also, they have not been developing products long enough to collect an appropriately large dataset. When we cannot *mine* data, we show how to *farm* knowledge by growing datasets from domain models.

Second, we will only report mature machine learning methods; i.e. those methods which do not require highly specialized skills to execute. This second bias rules out some of the more exciting work on the leading edge of machine learning research (e.g. horn-clause learning).

Third, in the author's view, it has yet to be shown empirically from realistic examples that a particular learning technique is necessarily better than the others*. When faced with N arguably equivalent techniques, Occam's razor suggests we use the simplest. We hence will explore simple decision tree learners in this chapter. Decision tree learners execute very quickly and are widely used: many of the practical SE applications of machine learning use decision tree learners like C4.5 [33] or the CART

*For evidence of this statement, see the comparisons of different learning methods in [34, 17, 36]

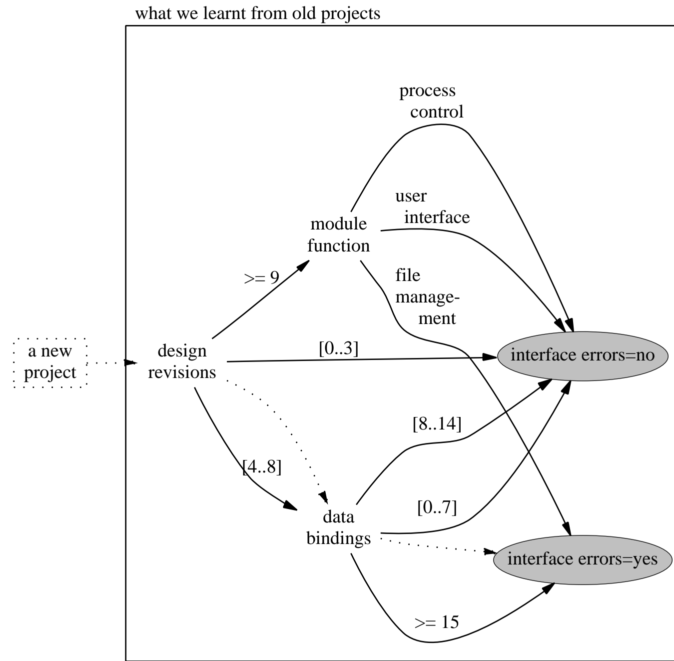


Figure 1: Assessment of a new software project using knowledge learnt from some old projects. In this diagram, "data bindings" is a domain-specific metric for assessing module interrelationships. The experience summarized here declares that the number of design revisions and the large number of data bindings doom some new project into having "interface errors" (see dotted line); i.e. errors arising out of interfacing software modules. This decision tree was automatically learnt using machine learning techniques [30] using the decision tree techniques discussed in this chapter.

system [5]. Decision tree learners are also cheap to buy, are widely available, and are commercially supported (e.g. see `http://www.rulequest.com`).

In order to balance these biases, we offer the following notes:

- For an excellent theoretical discussion on a wide range of machine learning techniques, see [25].
- For examples of machine learning from naturally occurring datasets, see Figures 1,2,3, and 4.
- For reviews on other kinds of learners, see the Reynolds chapter in this volume on evolutionary programming and other work on learning knowledge from data [28]; artificial neural nets [11]; an excellent review on data mining [22]; and a recent special issue of the SEKE journal on different techniques for discovering knowledge [26].
- Looking into the future, we predict that the 2010 version of this handbook will

```
Across whole module:
total operators
total operators

Averages per KSLOC:
assignment statements
cyclomatic complexity
executable statements
decision statements
function calls (FC)
module calls (MC)
FC plus MC
IO statements
IO parameters
origin
operands
operators
comments (C)
source lines (SL)
SL minus C
format statements
```

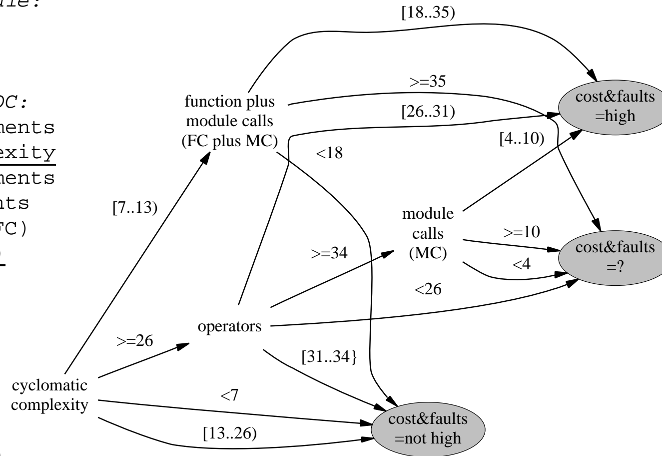


Figure 2: Predicting modules with high cost modules and many faults. Data from 16 NASA ground support software for unmanned spacecraft control [37]. These systems were of size 3,000 to 112,000 lines of FORTRAN and contained 4,700 modules. “Cyclomatic complexity” is a measure of internal program intricacy [21]. In the general case, there any many problems with cyclomatic complexity (see [10, p295] and [35]). Nevertheless, in this specific domain, it was found to be an important predictor of faults. Of the 18 attributes in the dataset (listed on left), only the underlined four were deemed significant by the learner.

contain many entries describing applications of horn-clause learners to the reverse engineering of software specifications from code [2, 7].

This chapter is structured as follows. Firstly, we expand on our distinction between using machine learning for *data mining* and *knowledge farming*. Secondly, we detail how to use a simple decision tree learner. Thirdly, we offer several case studies of knowledge farming. Fourthly, our discussion section describes how the above contributes to the construction of systems that support classification, prediction, diagnosis, planning, monitoring, requirements engineering, validation, and maintenance. Lastly, the appendices describe some of the lower-level technical details.

2. Data Mining and Knowledge Farming

Engineers often build systems without the benefit of large historical datasets which relate to the current project. For example, software development data may be scarce if the development team was not funded to maintain a metrics repository, or the collected data does not relate to the business case, or if contractors prefer to retain control of their own information. Machine learning that supports software engineering or knowl-

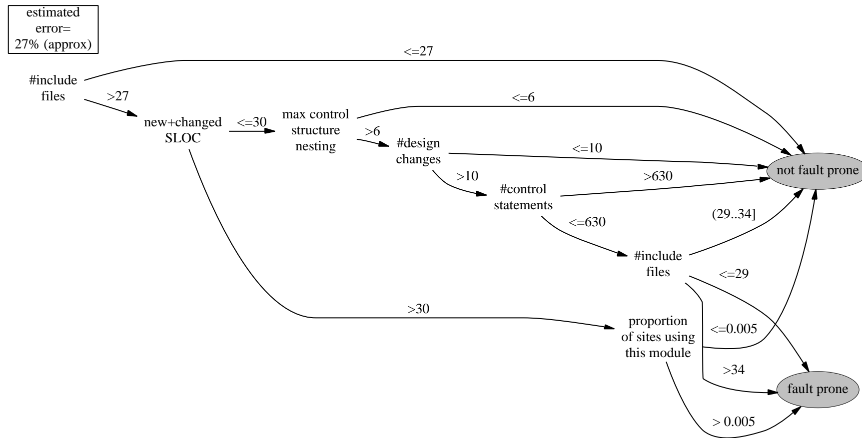


Figure 3: Predicting fault-prone modules [16]. Learnt by the CART tree learner [5] from data collected from a telecommunications system with $>$ 10 million lines of code containing a few thousand modules (exact details are proprietary confidential). Estimated error comes from 10-way cross-validation studies (explained later in this chapter). Of the 42 attributes offered in the dataset, only six were deemed significant by the learner.

edge engineering must therefore work when data is scarce. Hence, this chapter will focus more on *knowledge farming* than *data mining*. In *data mining*, there exists a large library of historical data that we mine to discover patterns. When such libraries exist, ML can (e.g.) infer from past projects the faults expected in new projects (see Figures 1,2,3) or development times (see Figure 4).

The power of data mining is that if the new application is built in exactly the same domain as the historical data, then the resulting predictions are tailored exactly to the local situation. For example, looking through Figures 1,2,3, and 4 we see that very different attributes are available in different domains.

The drawback with data mining is that it needs the data:

> (Data mining) can only be as good as the data one collects. Having good data is the first requirement for good data exploration. There can be no knowledge discovery on bad data [22].

Knowledge farming assumes that we cannot access a large library of historical data. When faced with a data famine, we use domain models as a *seed* to *grow* data sets using exhaustive or monte carlo simulations. We then *harvest* the data sets using ML. The harvested knowledge contains no more knowledge than in the original domain models. However, knowledge in the domain models can be hard to access. It may be expressed verbosely or hidden within redundant or useless parts of the model. Also, using that domain knowledge may be hard since the domain models may be slow to execute. In contrast, the harvested knowledge contains a simple and succinct record of the important knowledge. Further, the harvested knowledge can execute very quickly.

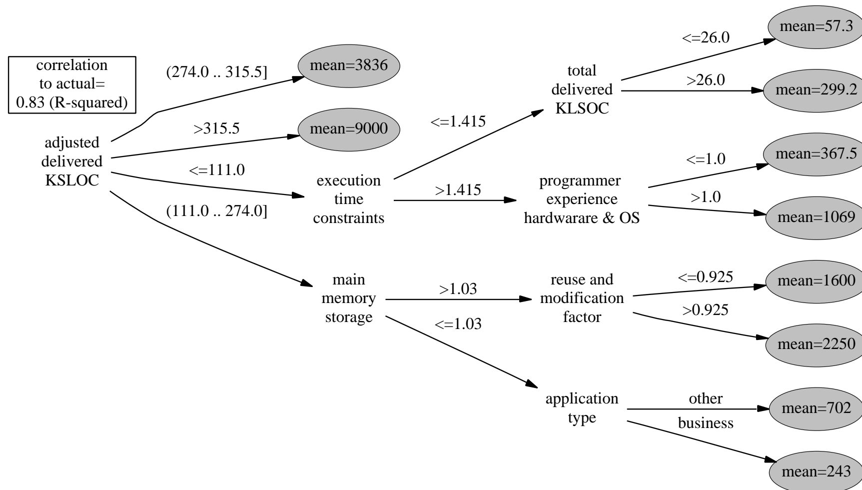


Figure 4: Predicting software development time (in person months) [36]. Learnt by the CART tree learner [5] from the 63 software projects in the COCOMO-1 database. The development times predicted by the learnt trees has a high correlation ($R^2 = 0.83$) with the actual development times in the COCOMO-1 database. For precise definitions of all attributes, see [3]. The tree shown here is just the upper-levels of the actual learnt tree (which is not shown for space reasons). Of the 40 attributes in the dataset, only six were deemed significant by the learner.

Knowledge farming can be much simpler than data mining. Naturally occurring datasets must be "cleansed" before they are useful. Data cleansing is the process of making sense of a large store of data that may be poorly-structured and contain corrupt or out-dated records. Knowledge farmers can control and understand the seed that generates the datasets. Hence, knowledge farmers rarely need to cleanse. Also, suppose a learner uses RAM storage to hold all the frequency counts of all the attribute values. Such RAM-based storage fails when data mining very large datasets; e.g. terabytes of data. However, when processing millions of examples (or less), the RAM-based leaner described here will suffice for generating the decision trees shown in (e.g.) Figures 1,2,3, and 4.

3. Building Decision Trees

Decision tree learners such as C4.5 input classified examples and output decision trees. C4.5 is an international standard in machine learning; most new machine learners are benchmarked against this program. C4.5 uses a heuristic *entropy* measure of information content to build its trees. This measure is discussed in an appendix to this chapter.

As an example of C4.5, suppose we want to decide when to play golf. Figure 5 shows the data we have collected by watching some golfer. The `names` file is a data dictionary describing our dataset. Line one lists our classifications and the other lines

describes the attributes. Attributes are either `continuous` (i.e. numeric) or discrete. In the case of discrete values, each possible value is listed in the `names` file. Users of C4.5 should be aware that the algorithm runs faster for discrete attributes than continuous values. However, the performance on continuous variables is quite respectable. In one experiment with a dataset of 21,600 examples of nine continuous attributes, C4.5 ran in 7 seconds on a 350MHz Linux machine with 64MB of RAM.

The `data` file (bottom left of Figure 5) shows the golf dataset. Each line contains as many comma-separated values as attributes defined in the `names` file, plus the class at the end-of-line. C4.5 supports unknown attribute values: such values may be recorded as a "?" in a `data` file. Of course if the number of unknowns is large, then the learner will have insufficient information to learn adequate theories.

C4.5 is called using the command line:

```
c4.5 -f stem -m minobs
```

where `stem` is the prefix of the `names` and `data` file and `minobs` is the minimum number of examples required before the algorithm forks a sub-tree (default `-m 2`). The command "`c4.5 -f golf -m 2`" generates the tree shown top right of Figure 5 (the tree is generated from the C4.5 text output shown in Appendix C). We see that we should not play golf on high-wind days when it might rain or on sunny days when the humidity is high.

C4.5 uses a statistical measure to estimate the classification error on unseen cases. In the case of "`c4.5 -f golf -m 2`", C4.5 estimates that this tree will lead to

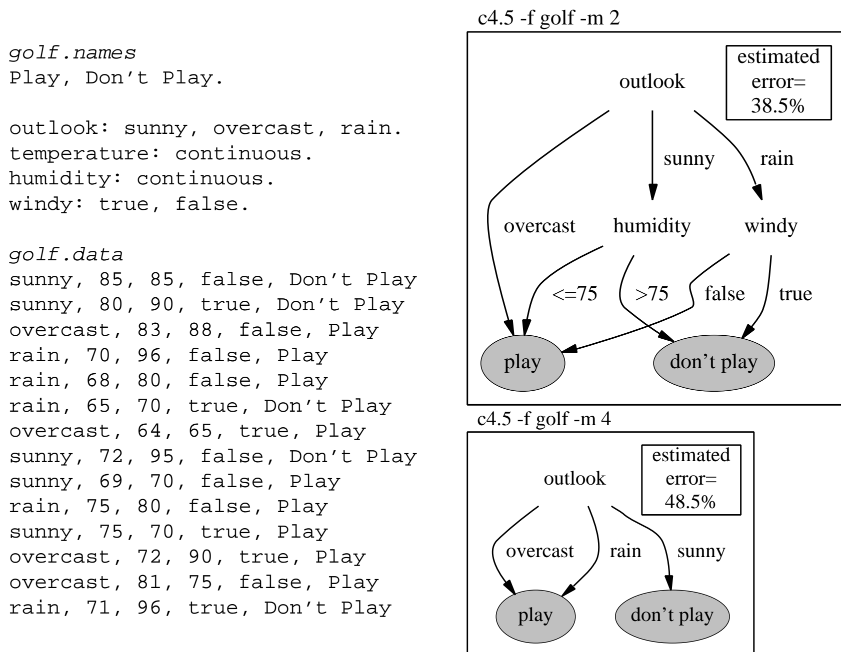


Figure 5: Decision tree learning. Classified examples (bottom left) generate the decision tree (bottom right). Adapted from [32].

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

If humans are to read the learnt tree, analysts must trade-off succinctness (smaller trees) versus classification accuracy. A drawback with decision tree learners is that they can generate incomprehensibly large trees. For example, we describe below cases where 10,000 node trees were generated. In C4.5, the size of the learnt tree is controlled by the `minobs` command-line parameter. Increasing `minobs` produces smaller and more easily understood trees. However, increasing `minobs` also decreases the classification accuracy of the tree since infrequent special cases are ignored. We can observe this effect above: the tree generated using "`c4.5 -f golf -m 4`" (bottom right of Figure 5) is smaller and less accurate than the tree generated using `-m 2`.

4. Case Studies in Knowledge Farming

Having described a simple machine learner, we can now demonstrate knowledge farming. Our examples fall into two groups:

- In the first group, domains lack models and data. In these domain, knowledge farmers must first build models that we can use to grow datasets.
- In the second group, domains lack data but possess quantitative models. In these domains, we can grow datasets from these models.

4.1. *Knowledge Farming From Qualitative Models*

A qualitative model is an under-specified description of a system where the range of continuous attributes are divided into a small number of symbols [13]. We can build and execute such qualitative models quicker than detailed quantitative models since we do not need precise attributes of the system; e.g. the resistance of some bulb.

For example, consider the electrical circuit of Figure 6 (left-hand side). Suppose our goal is to build a diagnosis device that can recognize a "fault" in the circuit. In this example:

- Our "fault" will be that the circuit cannot generating "enough light".
- "Enough light" means that at least two of bulbs are ok.

Ideally, we should be able to recognize this fault using the minimum number of tests on the circuit.

The qualitative model of this circuit is simple to build. We begin by replacing all quantitative numbers x with a qualitative number x' as follows:

$$\begin{aligned} x' &= + \quad if \quad x > 0 \\ x' &= 0 \quad if \quad x = 0 \\ x' &= - \quad if \quad x < 0 \end{aligned}$$

We can now describe the circuit in qualitative terms using the Prolog program `cir-`

cuit shown in Figure 6 (right-hand side). In Prolog, variables start with upper case letters and constants start with lower-case letters or symbols. Circuit's variables conform to the following qualitative relations shown in Figure 7. The sum relation describes our qualitative knowledge of addition. For example, sum(+,+,+) says that the addition of two positive values is a positive value. Some qualitative additions are undefined. For example sum(+,-,Any) says that we cannot be sure what happens when we add a positive and a negative number.

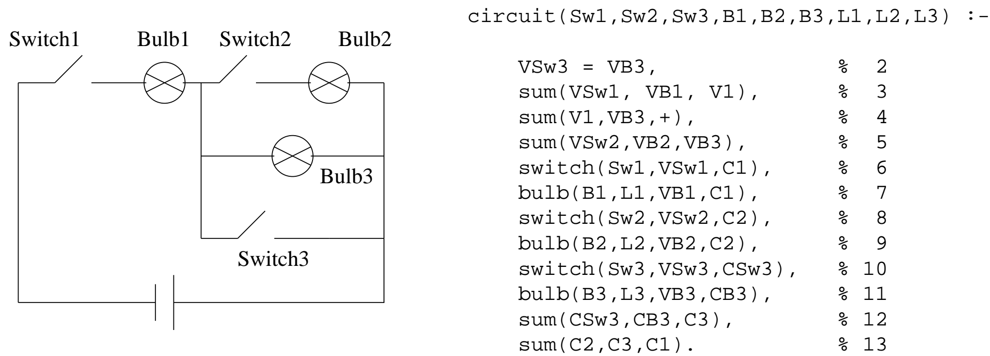
The bulb relation describes our qualitative knowledge of bulb behavior. For example, bulb(blown,dark,Any,0) says that a blown bulb is dark, has zero current across it, and can have any voltage at all.

The switch relation describes our qualitative knowledge of electrical switches. For example switch(on,0,Any) says that that if a switch is on, there is zero voltage drop across it while any current can flow throw it.

The circuit relation of Figure 6 describes our qualitative knowledge of the circuit. This relation just records what we know of circuits wired together in series and in parallel. For example:

- Switch3 and Bulb3 are wired in parallel. Hence, the voltage drop across these components must be the same (see line 2).
- Switch2 and Bulb2 are wired in series so the voltage drop across these two devices is the sum of the voltage drop across each device. Further, this summed voltage drop must be the same as the voltage drop across the parallel component Bulb3 (see line 5).
- Switch1 and Bulb1 are in series so the same current C1 must flow through both (see line 6 and line 7)

Figure 8 shows how we can learnt a decision tree for our circuit. A small Prolog program generates circ.data. In that program, the relation classification describes our recognizer for the faulty circuit: if two out of the tree bulbs are ok, then the classification is good; else, it is bad. Note that the circuit's behavior can be re-classified by changing the classification relation. The names file for this




```
circuit(Sw1,Sw2,Sw3,B1,B2,B3,L1,L2,L3) :-

    VSw3 = VB3,                 %  2
    sum(VSw1, VB1, V1),         %  3
    sum(V1,VB3,+),              %  4
    sum(VSw2,VB2,VB3),          %  5
    switch(Sw1,VSw1,C1),        %  6
    bulb(B1,L1,VB1,C1),         %  7
    switch(Sw2,VSw2,C2),        %  8
    bulb(B2,L2,VB2,C2),         %  9
    switch(Sw3,VSw3,CSw3),      % 10
    bulb(B3,L3,VB3,CB3),        % 11
    sum(CSw3,CB3,C3),           % 12
    sum(C2,C3,C1).              % 13
```

Figure 6: An electrical circuit. From [4].

```
%sum(X,Y,Z).        %blub(Mode,Light,Volts,Amps)  %switch(State,Volts,Amps)
sum(+,+,+).         bulb(blown,dark, Any, 0).     switch(on,    0,    Any).
sum(+,0,+).         bulb(ok,   light,+,  +).      switch(off,   Any,  0).
sum(+,-,Any).       bulb(ok,   light,-,  -).
sum(0,+,+).         bulb(ok,   dark, 0,  0).
sum(0,0,0).
sum(0,-,-).
sum(-,+,Any).
sum(-,0,-).
sum(-,-,-).
```

Figure 7: Some qualitative relations. From [4].

example is shown top right of Figure 8. The learnt tree is shown bottom right. To an accuracy of 85.2%, we can detect when two of our three bulbs are blown by just watching `Bulb1`. A visual inspection of the circuit offers an intuition of why this is so. `Bulb1` is upstream of all the other bulbs. Hence, it is a bottleneck for all the behavior in the rest of the circuit. Based on the learnt tree, we can define a minimal heuristic diagnostic strategy as follows: monitor only `Bulb1` and ignore the other 6 electrical components in the circuit.

This small example is an example of Bratko's method for understanding qualitative models:

- Build a model (which in knowledge farming is called the seed).
- Exhaustively simulate the model (which we would call growing the seed)

Generator, circ.data:

```
go  :- tell('circ.data'),
       go1,
       told.
go1 :- functor(X,circuit,9),
       forall(X,example(X)).

example(circuit(
     Sw1,Sw2,Sw3,B1,B2,B3,L1,L2,L3)) :-
   classification(B1,B2,B3,Class),
   format('~a,~a,~a,~a,~a,~a~n',
          [Sw1,Sw2,Sw3,L1,L2,L3,Class]).

%classification(B1, B2, B3,Class)
classification( ok, ok, B3,   good):- !.
classification( ok, B2, ok,   good):- !.
classification( B1, ok, ok,   good):- !.
classification( B1, B2, B3,   bad).

:- go, halt.
```

circ.names:

```
good,bad.

switch1: on, off.
switch2: on, off.
switch3: on, off.
bulb1: light, dark.
bulb2: light, dark.
bulb3: light, dark.
```

c4.5 -f circ -m 2

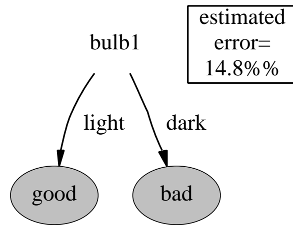


Figure 8: Left: generation of `circ.data`. Right, top: `circ.names`. Right, bottom: learnt tree.

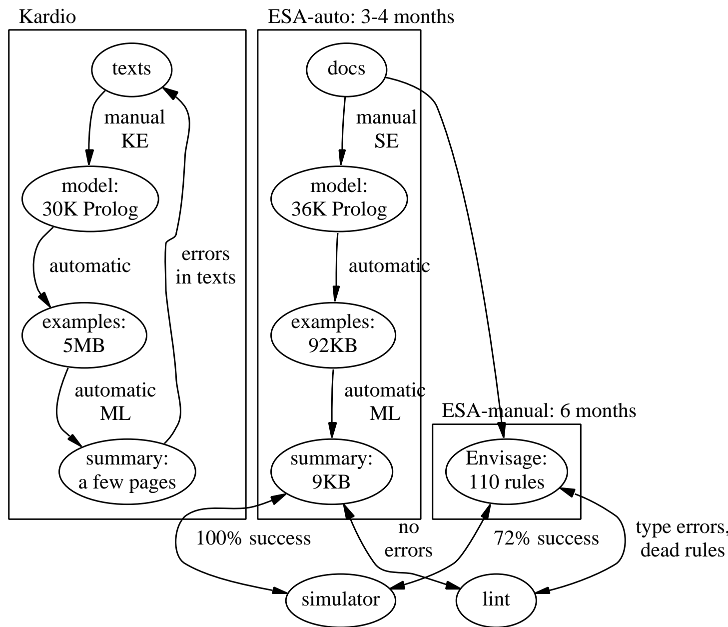


Figure 9: Knowledge farming from Prolog qualitative models. Examples from [4] and [29].

- Summarize the simulation results using machine learning (which we would call harvesting the seed).

Figure 9 illustrates the KARDIO and Pearce application of Bratko's technique. In the KARDIO application [4], a 30K Prolog qualitative model was built from cardiology textbooks. This model was exhaustively simulated to generated 5MB of data files. This data was summarized by a machine learner into a few pages of rules. Medical experts found differences between the learnt knowledge and the knowledge in the textbooks. On reflection, these experts preferred the learnt knowledge, arguing that the texts' knowledge was often too generalized or too specialized [Bratko, pers.comm. 1992].

In the Pearce application [29], a 36KB Prolog qualitative model was built of the electrical circuitry in a European Space Agency satellite. This was exhaustively simulated and summarized using a machine learner. The learnt knowledge was a diagnosis device and was assessed by comparing it to a manually constructed diagnostic expert system constructed by a separate team. Note that:

- The manual system took nearly twice as long to build as the learnt system.
- The manually constructed knowledge base contained more syntactic errors (e.g. type errors and dead-end rules) than the automatically learnt knowledge base.
- When compared to a FORTRAN simulator of the circuitry, the manually constructed knowledge base could reproduce less of simulator's behavior than the

automatically learnt system.

4.2. *Knowledge Framing from Quantitative Knowledge*

4.2.1. *COCOMO*

Our next example of knowledge farming relates to assessing software development risk [23]. This study is a quintessential example of using machine learning in data-starved domains. Given a wide-range of "don't know"s, we:

- Extract variables randomly from those ranges;
- Run a model using those variables as inputs;
- Summarize the resulting behavior using machine learning.

The summaries can be very succinct. In the case study presented here, we will summarize six million "what-if"s in $\frac{1}{3}$ of a page (see Figure 12).

			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 $nowX \cup changesX$) =	$6 * 10^6$	

Figure 10: The KC-1 NASA software project.

Consider the software project shown in Figure 10. The column *now1* in Figure 10 shows the current state of a NASA software project. Attributes in this figure come from the COCOMO-II software cost estimation model evolved by Boehm, Madachy, etl.al. [3, 6, 1]. Attribute values of “2” are nominal. Usually, attribute values lower than “2” denote some undesirable situation while higher values denote some desired situation.

Note the large space of possibilities. The *changes1* column show 11 proposed changes to that project. That is, there exist $2^{11} = 2048$ combinations of proposed changes to this project. Further, when combined with the ranges in the *now1* column, there are $6 * 10^6$ options shown in this table.

Before a machine learner can search this space of $6 * 10^6$ options, we need to somehow classify each option. We will use the Madachy COCOMO-based effort-risk model [20]. The model contains 94 tables of the form of Figure 11. Each such table implements a heuristic adjustment to the internal COCOMO attributes. This model generates a numeric effort-risk index which is then mapped into the classifications *low, medium, high*. Studies with the COCOMO-I project database have shown that the Madachy index correlates well with $\frac{months}{KDSI}$ (where KDSI is thousands of delivered source lines of code) [20]. This validation method reveals a subtle bias in the Madachy model. The model assess the *development risk* of a project; i.e. the risk that the project while take longer than planned to build. This definition is different to the standard view of risk. Standard software risk assessment defines risk as some measure of the runtime performance of the system. That is, in the standard view, risk is defined as an *operational* issue (e.g. [18, 19]). Since we are using the Madachy model, when we must adopt his definitions. Hence, when we say “software risk”, we really mean “development risk”.

There are several sources of uncertainty when using the Madachy model. Firstly, many of its internal attributes are unknown with accuracy. Madachy’s model uses off-the-shelf COCOMO and off-the-shelf COCOMO can be up to 600% inaccurate in its estimates (see [27, p165] and [14]). In practice, users tune COCOMO’s attributes using historical data in order to generate accurate estimates [6]. For the project in Figure 10, we lacked the data to calibrate the model. Therefore we generated three different versions of the model: one for each of the tunings we could find published

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

Figure 11: A Madachy factors table. From [20]. 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 attributes (increments shown top-right). Otherwise, in the non-exceptional case, add nothing to the built-in attributes.

in the literature [8, 6] plus one using the standard tunings found within the Madachy model.

A second source of uncertainty is the SLOC measures in our project (delivered source lines of uncommented code). COCOMO estimations are based on SLOC and SLOC is notoriously hard to estimate. Therefore, we run the Madachy model for a range of SLOCs. 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 [3].

A third source of uncertainty are the ranges in Figure 10. There are $2^{11} = 4048$ combinations of possible changes and 10^5 ways to describe the current situation (each "don't know " value, denoted "?", implies that we may have to run one simulation for every point in the don't know ranges). This space of uncertainty can be crippling. If we gave a manager one report for each possibility, they may be buried under a mountain of reports. Clearly, we have to explore this space of options. However, we also have to prune that space as savagely as possible.

We apply knowledge farming to this example as follows. First, we use monte carlo simulations to sample the $6 * 10^6$ options within Figure 10. For every combination of attribute tunings and SLOC, we generated N random examples by picking one value at random for each of the attributes from column 2 of Figure 10. N was increased till the conclusions (generated below) *stabilized*; i.e. conclusions found at sample size N did not disappear at larger sample sizes. In this study, we used $N = 10,000, 20,000, 30,000, 40,000, 50,000$. This generated 45 examples of the system's behavior sorted as follows:

$$3\ SLOCs\ * 3\ tunings\ *\ 5\ samples\ =\ 45\ samples$$

Secondly, we learnt an ensemble of 45 decision trees using C4.5 from each of the 45 samples. This generated 45 trees with tens of thousands of nodes in each tree. Such large trees cannot be manually browsed. With Sinsel, I have developed the TARZAN for polling ensembles of large trees. The premise of TARZAN is that when a human "understands" a decision tree, they can answer the following question:

> Here are some things I am planning to change; tell me the smallest set that will change the classifications of this system.

Note that this question can be answered by hiding the learnt trees and merely showing the operator the significant attribute ranges (SAR) that change the classifications. TARZAN swings through the decision trees looking for the SARs.

TARZAN applied eight pruning steps $P_1, P2, ..., P_8$ to generate a succinct summary of the SARs. For all prunings:

- $Out_i = P_i\,(Out_{i-1})$
- Out_0 is initialized to include all the ranges of all the variables; e.g. all of column 2 of Figure 10.
- The SARs are Out_8.

$Out_1 = P_1\,(Out_0)$: The entropy measure of C4.5 performs the first pruning. Attributes that are too dull to include in the trees are rejected. We saw examples of this rejection process above in Figures 2,3, and 4.

$Out_2 = P_2\,(Out_1)$: P_2 prunes ranges that contradict the domain constraints; e.g. $now1 \cup changes1$ from Figure 10. Also, when applying P_2, we prune any tree branch in the ensemble that use ranges from $Out_2 - Out_1$ (i.e. uses ranges discarded by P_2).

$Out_3 = P_3\,(Out_2)$: P_3 prunes ranges that do not change classifications in the trees. That is, Out_3 only contains the ranges that can change classifications. To compute Out_3, TARZAN finds each pair of branches in a tree that lead to different conclusions. Next, we find attributes that appear in both branches. If the ranges for that attribute in the different branches do not overlap, then we declare that the difference between these two ranges is a change that can alter the classifications in that tree.

$Out_4 = P_4\,(Out_3)$: P_4 prunes ranges that are not interesting to the user, i.e. are not mentioned in the $changes1$ set ($Out_4 = Out_3 \wedge changes1$).

$P_1, ...P_4$ apply to single trees. The remaining pruning methods collects the Out_4s generated from all the single trees, then explores their impacts across the entire ensemble.

$Out_5 = P_5\,(Out_4)$: P_5 prunes ranges that do not change classifications in the majority of the members of the ensemble (our threshold is 66%, or 30 of the 45 trees).

$Out_6 = P_6\,(Out_5)$: P_6 prunes ranges that are not stable; i.e. those ranges that are not always found at the larger samples of the random Monte Carlo simulations. In the case of Figure 10, of the 11 changes found in $changes1$, only 4 survived to Out_6.

$Out_7 = P_7\,(Out_6)$: P_7 explores all subsets of Out_6 to reject the combinations that have low impact on the classifications. The impact of each subset is assessed via how that subset changes the number of branches to the different classifications. For example, Figure 12 shows how some of the subsets of the ranges found in Out_6 affect KC-1. In that figure, bar chart A1 shows the average number of branches to *low, medium*, and *high* risk seen in the 45 trees. For each subset $X \subseteq Out_6$, we make a copy of the trees pruned by P_2, then delete all branches that contradict X. This generates 45 new trees that are consistent with X. The average number of branches to each classification in these new branches is then calculated. P_7 would reject the subsets shown in B1, C1, and B2 since these barely alter the current situation in A1.

$Out_8 = P_8\,(Out_7)$: P_8 compares members of Out_7 and rejects a combination if some smaller combination has a similar effect. For example, in Figure 12, we see in A2 that having moderately talented analysts and no schedule pressure (*acap=[2], sced=[2]*) reduces our risk nearly as much as any larger subset. Exception: C2 applies all four actions from KC-1's Out_6 set to remove all branches to medium and high risk projects. Nevertheless, we still recommend A2, not C2, since A2 seems to achieve most of what C2 can do, with much less effort.

OUT_8 are the operators found by TARZAN which can effect the device. These are the significant attribute ranges (SARs). In this KC-1 study, we have found 2 significant control strategies out of a range of $6 * 10^6$ what-ifs and 11 proposed new changes. TARZAN took 8 minutes to process KC-1 on a 350MHz machine. That time was divided equally between Prolog code that implements the tree processing and some inefficient shell scripts that filter the Prolog output to generate Figure 12. We are currently porting TARZAN to "C" and will use bit manipulations for set processing. We

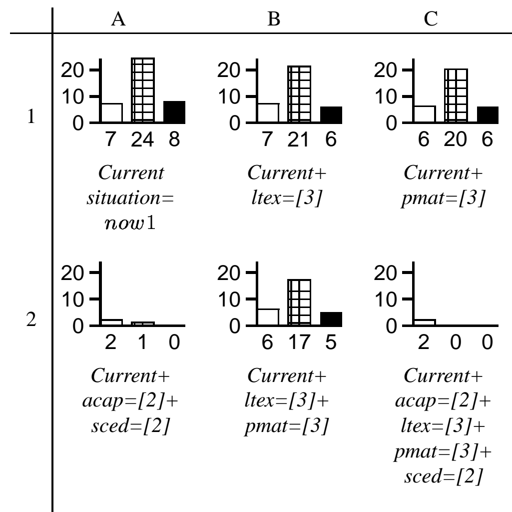


Figure 12: Average number of branches to different classifications in the 45 trees in KC-1, assuming different subsets of the ranges seen in KC-1's Out_6 set. Legend: (white box) =low risk (grid box) =medium risk (black box) =high risk.

are hopeful that this new language and smarter processing will make TARZAN much faster.

4.2.2. *Reachability*

Our other examples related to applications that could be useful for software engineering practitioners. This example describes an application that could be useful for theoretical software engineering researchers.

This example begins with the following theoretical characterization of testing. Testing, we will assume, is the construction of pathways that *reach* from inputs to some

```
byte a=1; byte b=1; bit f=1;
active proctype A(){ do :: f==1 -> if :: a==1 -> a=2;
                                      :: a==2 -> a=3;
                                      :: a==3 -> f=0; a=1;
                                   fi
                     od
}
active proctype B(){ do :: f==0 -> if :: b==1 -> b=2;
                                      :: b==2 -> b=3;
                                      :: b==3 -> f=1; b=1;
                                   fi
                     od
}
```

Figure 13: Two procedural subroutines from the SPIN model checker [12].

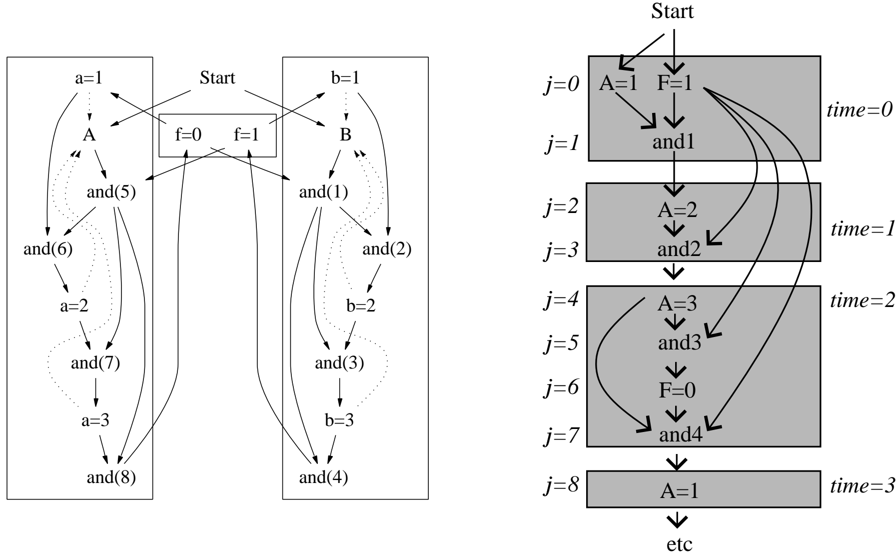


Figure 14: Left: And-or links found in the SPIN model from Figure 13. `Proctype A` and `B` are shown in the boxes on the left-hand side and right-hand side respectively. These `proctypes` communicate via the shared variable `f`. Dotted lines represent indeterminism; e.g. after `proctype A` sets `a=2`, then we can go to any other statement in the proctype. Right: Example reachability network built from the left-had-side network showing how to violate property `always not A=3` at height $j = 4$.

interesting zone of a program. This zone could be a bug or a desired feature. In this *reachability* view, the goal of testing is to show that a test set uncovers no bugs while reaching all desired features.

This theoretical view of testing can be modeling mathematically. The *reachability model* is a set of mathematical equations that computes the average case probability of finding a bug in a program using random inputs [24]. Reachability represents programs as directed graphs indicating dependencies between variables in a global symbol table; see Figure 14 (left). Such graphs have V nodes which are divided into $andf$% and-nodes and $(orf = 1 - andf)$% or-nodes. These nodes are connected by yes-edges, which denote valid inferences, and no-edges, which denote invalid inferences. For the sake of clarity, we only show the yes-edges in Figure 14 (left). To add the no-edges, we find every impossible pair and connect them; e.g. we link `A=1` to `A=2` with a no-edge since we cannot believe in two different assignments for the same variable at the same time. We say that, on average, each node is touched by no_μ no-edges. When inferencing across these graphs, variables can be assigned, at most T different values; i.e. a different assignment for each time tick T in the execution (classic propositional systems assume $T = 1$; simulation systems assume $T \geq 1$). Nodes are reached across the dependency graph using a path of height j where the inputs are at height $j = 0$. For example, Figure 14 (left) shows a the network of paths of height $j = 8$ that reaches

$A = 1$, then $A = 2$, then $A = 3$. Note that since four values are assigned to A, then time T must be 4.

If we use In inputs, then the odds of reaching a node at height $j = 0$ is $P[0] = \frac{In}{V}$. The odds of reaching a node at height $j > 0$ depends on the node type:

- We can't reach an and-node unless we reach all its parents at some height $i < j$.
- We can't miss an or-node unless we miss all its parents at some height $i < j$.

Based on these rules, we can compute the odds $P[j]$ of reaching a node at height j (see [24] for the details). We convert these odds to the required number of tests as follows. After N tests, we are C certain that we will see an event of odds $P[j]$ with odds:

$$C = 1 - \left((1 - P[j])^N\right)$$

Assuming we want to be 99% sure of reaching that node, then this equation re-arranges to:

$$N = \frac{log(1 - 0.99)}{log(1 - P[j])}$$

which we can use to classify each simulation run:

$$Class = \begin{cases} fast\,and\,cheap & if\ N < 10^2, \\ fast\,and\,moderately\,expensive & if\ N < 10^4, \\ slow\,and\,expensive & if\ N < 10^6, \\ impossible & otherwise. \end{cases}$$

Figure 15 shows a decision tree learnt by C4.5 using the above classifications. For reasons of readability, this tree is truncated using a large "`minobs`" value. From the tree, we can make make many inferences about the impact of various factors on testability. For example, minor changes to the structure of a program can have massive and undesirable changes to a program's reachability and hence testability:

- We see in Figure 15 that there are three ways to conclude that less than 100 tests are required to reach most parts of a program. The mean percentage of and-nodes appearing in all three paths is:

  $$(andf_\mu \leq 0.33) \wedge (andf_\mu \leq 0.4) \wedge (andf_\mu \leq 0.6) = andf_\mu \leq 0.6$$

  That is, if $andf_\mu$ ever rises above 0.6, there is no way that 100 random tests will be enough to test that program.

- Figure 15 also tells us that there is only one way to show that more than 1,000,000 tests are required to reach most parts of a program. The and-node frequencies in that path are:

  $$(andf_\mu > 0.6) \wedge (andf_\mu > 0.67) = andf_\mu > 0.67$$

  That is, if $andf_\mu$ rises from 0.6 to 0.67, then a system with easy reachability could suddenly transform into a system with hard reachability, hence hard testability.

Not only can we use machine learning to understand systems like the reachability model, we can also use machine learning to control the modeling process. Using a machine learner, we can recognize what portions of a model are not critical to the success of the model. Hence, machine learning can stop analysts wasting time exploring pointless issues. For example, when developing the reachability model with Singh and Cukic, we were unsure of the importance of two attributes:

$Andf_\mu$: the frequency of conjunctions in a program

$isTree?$: If this boolean is true/false, the equations of reachability returns the lower/upper estimates (respectively) on number of upstream variables used to reach a bug.

To determine the importance of these variables, we ran the reachability model by randomly selecting input values for all attributes including $andf_\mu$ and $isTree?$. We then asked C4.5 to learn a predictor of reachability from 150, 1,500, or 150,000 randomly selected outputs from the model. Three predictors were built:

1. The first predictor was built from examples that held values for all attributes in the reachability model. This provided the baseline measure shown in All plot of Figure 16.

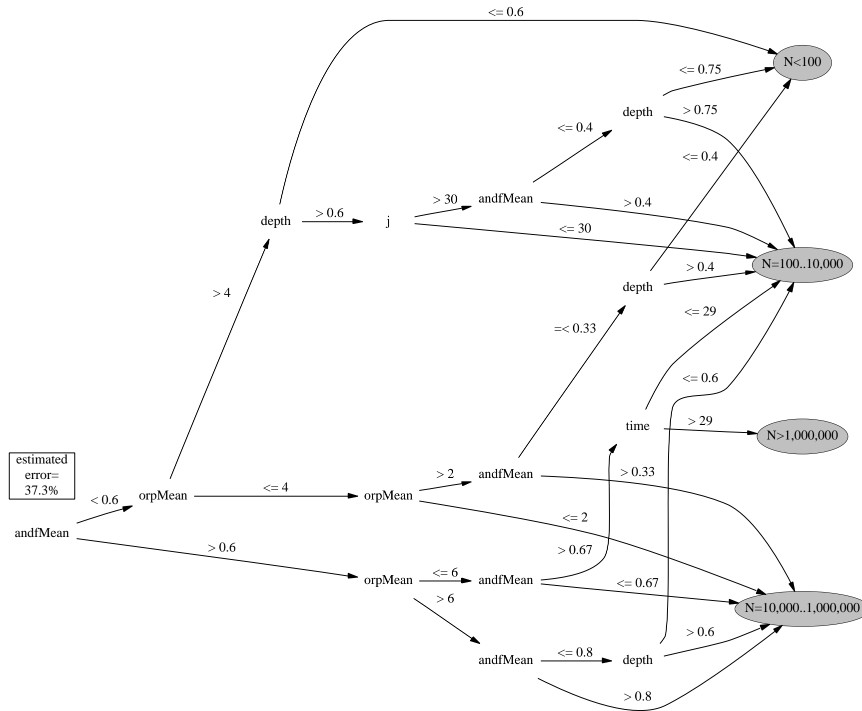


Figure 15: Predicting reachability from simulations of reachability model using 1500 randomly chosen inputs. The tree was learnt using `c4.5 -m 45 -f runs1500`.

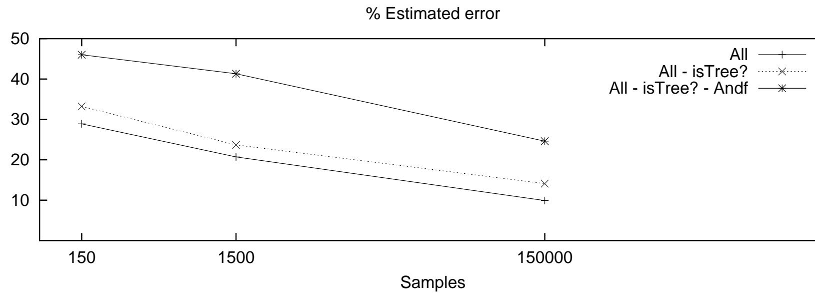


Figure 16: Relative importance of $andf_\mu$ and $isTree?$ in the reachability model.

2. The second predictor was built from examples that held all values except the $isTree?$ attribute. As we would expect, since the learner was given less information, the learnt tree was not as good as the All tree: loss of accuracy≈5% (see the $All - isTree?$ plot of Figure 16).

3. The third predictor was built from examples that held all values except the $isTree?$ and $andf$ attributes. Again, since we are giving the learner less information, the learnt tree is less accurate (see the $All - isTree? - Andf_\mu$ plot of Figure 16).

Note that this third plot is much less accurate than the others. That is, while knowledge of $isTree?$ is not so vital, $andf_\mu$ is clearly a crucial attribute. Therefore, if we could accept a 5% loss in the accuracy of our analysis, we could stop exploring the problems relating to the $isTree?$ issue and focus more on issues relating to $andf_\mu$.

5. Discussion

When we lack sufficient data for mining, we can go farming. We can seed our knowledge with domain models, then grow and harvest decision trees. The models need not be too detailed (e.g. `circuit` in Figure 6). Further, if we are unsure of parts of those models, we can use machine learning to identify which areas to explore and which to ignore. For example, with the reachability model, we could show that equations using the $isTree?$ parameter were not crucial to the model.

We have seen in this chapter that machine learning can be used for many applications:

Prediction and Classification: In Figures 1,2 and 3 we saw decision trees learnt to predict faults in software modules. Figure 4 showed a decision tree learnt to predict software development effort. These predictors can assess new examples, or classify old examples.

Diagnosis: Figure 8 showed a diagnosis tree learnt from a toy-example using qualitative modeling. Figure 9 showed how the same style of modeling could be used to build diagnosis trees for larger applications such as cardiac disease and satellite electrical malfunctions.

Planning: Figure 12 showed the implications of several plans for changing a software project. We can use that figure to select which plan we wish to apply.

Monitoring: Machine learning can rationalize metrics collection and monitoring. Figure 15 found an attribute range ($andf_\mu = 0.6$ to $andf_\mu = 0.67$) where some desired property of a system (testability) could degrade very sharply. Clearly, there is a strong case for monitoring at least that attribute.

Requirements engineering: When conflicts arise between stakeholders, machine learning can find which decisions are crucial and which arguments do not impact the system. Debates could then be shortened to just the crucial issues. For example, in the COCOMO KC-1 study, stakeholders need not debate the 2048 possible changes to a project. Instead, machine learning found the handful of changes that really matter (recall Figure 12).

Machine learning can also support some crucial software processes:

Validation: We can validate the models used in knowledge farming by studying their behavior. For example, using TARZAN, test engineers can inspect Out_8 to find the effects of changing key attributes in the system. This validation scheme could fault the model used as the seed if the test engineers find that system behavior changes inappropriately when the key control attributes change. We envision that this style of validation will become very important to organizations like NASA in the near future. NASA already has hundreds of simulators of flight systems. Such simulators are used to explore alternatives in system design and flight profiles. Tools like TARZAN can be used to check if those simulators are generating sensible output.

Maintenance: When domain knowledge changes, we must manually change the seed. However, once that change has been made, we can then automatically generate classifiers, predictors, diagnosis engines, requirement engineering tools, planners, validation tools, and monitors.

Note that, in contrast to data mining, all the above are possible in domains that lack large datasets. For example, the Figure 12 plans were generated in a domain were many crucial attributes were unknown.

In conclusion, we note that knowledge farming offers a natural integration of software engineering and knowledge engineering:

- Software engineers or requirements analysts develop seeds;
- Knowledge engineers grow and harvest the seeds using machine learning and tools like TARZAN

6. 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. F. Bergadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering*. The MIT Press, 1995.
3. B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
4. I. Bratko, I. Mozetic, and N. Lavrac. *KARDIO: a Study in Deep and Qualitative Knowledge for Expert Systems*. MIT Press, 1989.
5. L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. Technical report, Wadsworth International, Monterey, CA, 1984.
6. 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.
7. W. W. Cohen. Inductive specification recovery: Understanding software by learning from example behaviors. *Automated Software Engineering*, 2:107–129, 1995.
8. R. Cordero, M. Costamagna, and E. Paschetta. A genetic algorithm approach for the calibration of cocomo-like models. In *12th COCOMO Forum*, 1997.
9. T.G. Dietterich. Machine learning research: Four current directions. *AI Magazine*, 18(4):97–136, 1997.
10. N. E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thompson Press, 1997.
11. G.E. Hinton. How neural networks learn from experience. *Scientific American*, pages 144–151, September 1992.
12. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
13. Y. Iwasaki. Qualitative physics. In P.R. Cohen A. Barr and E.A. Feigenbaum, editors, *The Handbook of Artificial Intelligence*, volume 4, pages 323–413. Addison Wesley, 1989.
14. C.F. Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, May 1987.
15. B.W. Kerningham and C.J. Van Wyk. Timing trials, or, the trials of timing: Experiments with scripting and user-interface languages, 1998. From Lucent Technologies Inc. Available from `http://netlib.bell-labs.com/cm/cs/who/bwk/interps/pap.html`.
16. T.M. Khoshgoftaar and E.B. Allen. Model software quality with classification trees. In H. Pham, editor, *Recent Advances in Reliability and Quality Engineering*. World Scientific, 1999.
17. F. Lanubile and G. Visaggio. Evaluating predictive quality models derived from software measures: Lessons learned. *The Journal of Software and Systems*, 38:225–234, 1997.
18. N. Leveson. *Safeware System Safety And Computers*. Addison-Wesley, 1995.
19. M.R. Lyu. *The Handbook of Software Reliability Engineering*. McGraw-Hill, 1996.
20. R. Madachy. Heuristic risk assessment using cost factors. *IEEE Software*, 14(3):51–59, May 1997.
21. T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.
22. M. Mendonca and N.L. Sunderhaft. Mining software engineering data: A survey, September 1999. A DACS State-of-the-Art Report. Available from `http://www.dacs.dtic.mil/techs/datamining/`.
23. 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`.
24. Tim Menzies, Bojan Cukic, Harhsinder Singh, and John Powell. Testing nondeter-

minate systems. In *ISSRE 2000*, 2000. Available from `http://tim.menzies.com/pdf/00issre.pdf`.
25. R.S. Michalski. Toward a unified theory of learning. In B.G. Buchanan and D.C. Wilkins, editors, *Reading in Knowledge Acquisition and Learning*, pages 7–38. Morgan Kaufmann, 1993.
26. S. Morasca and Gunther Ruhe. Guest editors' introduction of the special issue on knowledge discovery from software engineering data. *International Journal of Software Engineering and Knowledge Engineering*, October 1999.
27. 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.
28. M.J. Pazzani. Knowledge discovery from data? *IEEE Intelligent Systems*, pages 10–13, 2000.
29. D. Pearce. The induction of fault diagnosis systems from qualitative models. In *Proc. AAAI-88*, 1988.
30. A.A. Porter and R.W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, pages 46–54, March 1990.
31. J. R. Quinlan. Boosting first-order learning. In Setsuo Arikawa and Arun K. Sharma, editors, *Proceedings of the 7th International Workshop on Algorithmic Learning Theory*, volume 1160 of *LNAI*, pages 143–155, Berlin, October 23–25 1996. Springer.
32. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.
33. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1992. ISBN: 1558602380.
34. J.W. Shavlik, R.L. Mooney, and G.G. Towell. Symbolic and neural learning algorithms: An experimental comparison. *Machine Learning*, 6:111–143, 1991.
35. M. Shepped and D.C. Ince. A critique of three metrics. *The Journal of Systems and Software*, 26(3):197–210, September 1994.
36. K. Srinivasan and D. Fisher. Machine learning approaches to estiamting software development effort. *IEEE Trans. Soft. Eng.*, pages 126–137, Feburary 1995.
37. J. Tian and M.V. Zelkowitz. Complexity measure evaluation and selection. *IEEE Transaction on Software Engineering*, 21(8):641–649, August 1995.

Appendix A: Inside a Learner

The learner described here uses a heuristic *entropy* measure of information content to build its trees. The attribute that offers the largest *information gain* 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.

The information gain of each attribute is calculated as follows. A tree C contain p examples of some class and n examples of other classes. The *information required* for the tree C is as follows [32]:

$$I(p,n) = -\left(\frac{p}{p+n}\right) log_2 \left(\frac{p}{p+n}\right) - \left(\frac{n}{p+n}\right) log_2 \left(\frac{n}{p+n}\right)$$

Say that some attribute A has values $A_1, A2, ...A_v$. If we select A_i as the root of a new sub-tree within C, this will add a sub-tree C_i containing those objects in C that have

A_i. We can then define the expected value of the information required for that tree as the weighted average:

$$E(A) = \sum_{i=1}^{v} \left(\frac{p_i + n_i}{p + n} \right) I(p_i, n_i)$$

The information gain of branching on A is therefore:

$$gain(A) = I(p, n) - E(A)$$

For example, consider the decision tree learnt by C4.5 in Figure 5. In that tree, C4.5 has decided that the weather `outlook` has the most information gain. Hence, it has placed `outlook` near the root of the learnt tree. If `outlook=rain`, a sub-tree is entered whose next-most informative attribute is `wind`. Attributes with little information content are not included in the final tree. Examples of numerous absent attributes can be found in Figures 2,3, and Figure 4.

Appendix B: Testing a Learner

Before using the learnt trees, we should test them. When testing the trees, it is good practice to use examples not seen during learning. A standard technique for this is *N-way cross validation*. The example set is divided into N buckets, where each bucket has the same frequency distribution of classifications as the full set of examples (the `xval` script in the standard C4.5 distribution automates this task). For each bucket we place that bucket aside then learn a theory from the other buckets. The learnt tree is then assessed by making it classify the examples in the bucket set aside. The average classification accuracy of these N trials is then the final reported classification accuracy.

A side-effect of cross-validation is that we learn N trees. Since each tree is learnt from a different sample set, each tree may be different. This is the "many oracle problem" which has been summarized as follows: a man with one watch knows the time but a man with two watches is never sure. That is, if we learn one oracle, we get one set of conclusions. However, if we learn an ensemble of N oracles, they may all offer different conclusions.

The many oracle problem can be solved as follows: if 19 of your 25 watches all say that it is morning, then you could reasonably decide that it is time to get out of bed. That is, when using multiple oracles, the entire ensemble should be polled to find the conclusion(s) offered by the majority of oracles. Surprisingly, the conclusions from ensembles can be more accurate than those from any single oracle [9]. To see why, recall that cross-validation builds trees from different samples of the domain. Each such sample may explore a different part of a model. Hence, a poll of multiple samples may tell us more than a poll from a single sample.

Various systems support the polling of an ensemble of oracles. For example:

- Refinements of C4.5 offer automatic support for *boosting* (see `http://www.rulequest.com`); i.e. datasets are generated with greater emphasis for examples mis-classified in previous learning trials [9, 31].

- The TARZAN package (described above) searches ensembles of decision trees for attribute value changes that usually alter classifications [23].

Appendix C: Output from C4.5

Many of the decision trees shown in this chapter are pretty-printed from the C4.5 text output using some awk scripts [15] and the `dot` package (`http://www.research.att.com/sw/tools/graphviz/examples/`) C4.5's text output look like this:

```
C4.5 [release 8] decision tree generator
Sun Jun 25 17:33:55 2000
----------------------------------------

    Options:
File stem <circ>

Read 378 cases (6 attributes) from circ.data

Decision Tree:

light3 = light: good (8.0)
light3 = dark:
|   light1 = light: good (13.0/4.0)
|   light1 = dark: bad (357.0/45.0)

Simplified Decision Tree:

light1 = light: good (21.0/5.9)
light1 = dark: bad (357.0/50.0)

Tree saved

Evaluation on training data (378 items):

 Before Pruning           After Pruning
----------------   ---------------------------
Size      Errors   Size      Errors   Estimate

   5   49(13.0%)      3   49(13.0%)    (14.8%)   <<
```

Note that two trees may be generated. Sometimes C4.5 will prune subtrees if that pruning has little impact on the the classification accuracy.

Within a tree, sub-trees are denoted by indentation. Every level of indentation is marked with a "|" symbol. Each final classifications is followed by two numbers in brackets: the number of cases that fell into this branch and the number of cases that fell incorrectly into this branch ("incorrect" means that the case's classification was different to the case shown at the end of this branch).

C4.5 shows classification errors on the last line of the report. Three numbers are generated: one for the unpruned tree, one for the pruned tree, and one that is an estimate of the tree's classification accuracy on future cases. All the estimated errors reported in this chapter come from this third figure (shown bottom right- in this example, it is 14.8%).

It is a simple matter to write awk or perl scripts to convert C4.5's decision trees into

`dot` format. In writing such a conversion tool, the following tips may be useful:

- Re-compile C4.5 after setting the `#define Width` value in the file `trees.c` to a large number (e.g. 10000). This will stop C4.5 breaking up the tree into 80-character wide subtrees which can complicate the conversion process.
- The shell script shown below will automatically strip away the header and footer of the C4.5 output, leaving just the decision tree.
- In the stripped-out tree, it is easy to recognize a line with a classification. Each such line contains a "`(`" character.

```
#report2tree- extract tree text from C4.5 output file
#author: tim@menzies.com, June 2000
#usage: c4.5 -f stem -m M > c45.out; report2tree c45.out > c45.tree

getTree() {
    gawk 'BEGIN{flag=0}
        NR> 3 && $0~/Decision tree:/       {flag=1; next}
        $0~/Tree saved/                    {exit}
        flag==1                            {print $0}' $1
}
getSimpleTree() {
     gawk 'BEGIN{flag=0}
        NR > 3 && $0~/Simplified Decision Tree:/ {flag=1; next}
        $0~/Tree saved/                          {exit}
        flag==1                                  {print $0}' $1
}
if   grep "Simplified Decision Tree:" $1 >> /dev/null
then getSimpleTree $1
else getTree       $1
fi
```