Data Mining for Very Busy People

To meet the needs of busy people who only want to know enough to achieve the most benefits, the TAR2 treatment learner generates easy-to-read and immediately useful data mining rules.

Tim Menzies
West Virginia University

Ying Hu
University of British Columbia

For 21st-century businesses, the problem is not accessing data but ignoring irrelevant data. Most modern businesses can electronically access mountains of data such as transactions for the past two years or the state of their assembly line. The trick is effectively using the available data. In practice, this means summarizing large data sets to find the "pearls in the dust"—that is, the data that really matters.

In the data mining community, "learning the least" is an uncommon goal. Most data miners are zealous hunters seeking detailed summaries and generating extensive and lengthy descriptions. The "Data Mining Treatment Learning" sidebar discusses some work in this area. Here, we take a different approach and assume that busy people don't need—or can't use—complex models. Rather, they want only the data they need to achieve the most benefits.

Instead of finding extensive descriptions of things, the TAR2 "treatment learner" is a data mining tool (http://menzies.us/rx.html) that hunts for a minimal difference set between things.[1] A list of essential differences is easier to read and understand than detailed descriptions. Overly elaborate models can complicate, not clarify, a situation. Cognitive scientists and researchers studying human decision making note that humans often use simple models rather than intricate ones.[2] Because it learns much smaller models, TAR2 provides better support for real-world decision making than standard data miners.

TAR2: A SIMPLER, SHORTER RULE

Figure 1 shows a typical decision tree, generated from data on hundreds of houses in the Boston area. Each branch describes identifying factors for houses of high, medium-high, medium-low, and low quality using seven attributes:

- *age:* proportion of houses built before 1940
- *b:* information on the suburb's racial makeup
- *dis:* weighted distances to five employment centers
- *lstat:* living standard
- *nox:* nitric oxides concentration
- *ptratio:* parent-teacher ratio at local schools
- *rm:* number of rooms

To compare Figure 1 to TAR2's output, we first convert the tree to TAR2 rule format. We generated the tree using the Waikato Environment for Knowledge Analysis J4.8 algorithm with the command line J4.8 -C 0.25 -M 10 (www.cs.waikato.ac.nz/~ml/). Although there are many sophisticated

Data mining uses techniques from statistics and artificial intelligence to reduce large sets of examples to small, understandable patterns.

Decision tree learning

Many data mining methods generate *decision trees*—trees whose leaves are classifications and whose branches are conjunctions of features that lead to those classifications. One way to learn a decision tree is to split the example set into subsets based on some attribute value test. The process then repeats recursively on the subsets, with each splitter value becoming a subtree root. Splitting stops when a subset gets so small that further splitting is superfluous or a subset contains examples with only one classification.

A good split decreases the percentage of different classifications in a subset, ensuring that subsequent learning generates smaller subtrees by requiring less further splitting to sort out the subsets. Various schemes for finding good splits exist.[1,2]

Association rule learning

Association rule learners such as Apriori[3] find attributes commonly occurring together in a training set. No attribute can appear on both sides of the association LHS × RHS—that is, LHS × RHS = $\varnothing$.

The rule LHS × RHS holds in the example set with confidence c if c percent of the examples containing LHS also contain RHS: $c = |LHS \cup RHS| \times 100/|LHS|$. The rule LHS × RHS has support s in the example set if s percent of the examples contain LHS ∪ RHS: $s = |LHS \cup RHS| \times 100/|D|$, where $|D|$ is the number of examples. Association rule learners return rules with high confidence (for example, $c > 90$ percent).

Rejecting associations with low support first can cull the search for associations. We can view association rule learners as generalizations of decision tree learning: Decision tree learners restrict the RHS of rules to one class attribute whereas association rule learners can add any number of attributes to the RHS.

Association rule learning variants

Contrast set learning is a variant of association rule learning. Instead of finding rules that describe the current situation, contrast set learners like Stucco[4] find rules that differ meaningfully in their distribution across groups. For example, in Stucco, an analyst could ask, "What are the differences between people with PhD and bachelor's degrees?"

Weighted class learning is another variant. Association rule learners such as Minwal[5] assign weights to classes to focus the learning on issues of interest to a particular audience. TAR2 is a weighted contrast set learner that finds rules associating attribute values with changes to the class distributions. TAR2's design is simpler than many other learners because it assumes the small treatment effect (that is, treatments typically use only a few attributes).

Other machine learning researchers have also discovered that schemes using only a subset of the available attributes can generate effective theories. For example, learners using many attributes performed only moderately better than Robert Holte's 1R machine learner, which was restricted to a single attribute.[6]

TAR2 does not use the 1R technique because our results show that the best treatments can require more than one attribute.

Wrappers

Ron Kohavi and George John wrapped their learners in a preprocessor that used a heuristic search to grow subsets of the available attributes from size 1.[7] At each step, the wrapper called a learner to find the accuracy of the model learned from the current subset. Subset growth stopped when adding new attributes didn't improve accuracy. On average, their experiments showed that up to 83 percent of a domain's attributes could be ignored with only a minimal loss of accuracy.

TAR2 does not use this technique because using wrappers to select relevant features can be prohibitively slow as each step of the heuristic search requires a call to the learning algorithm.

Genetic and simulated annealing algorithms

The genetic[8] and simulated annealing[9] algorithms are two data-mining technologies that perturb current answers to look for better answers.

Genetic algorithms represent answers as bit strings, creating new bit strings by combining parts of old bit strings that scored well on some evaluation function. The bit strings also can mutate randomly.

Simulated annealing restricts perturbation to the single best answer to date. This algorithm uses a probability controlled by a temperature variable to decide whether a new answer is better than an old answer. The process starts "hot" and then "cools down."

While hot, the simulated annealer might randomly jump from an old best answer to a worse new answer. As it cools, however, the jumps revert to standard hill climbing so new best answers must be better than old best answers. Although they seem ill-advised, the hot phase random jumps ensure that the simulated annealer samples more of the answer space and stops it from getting stuck in local maxima.

References

1. R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1992.
2. L. Breiman et al., *Classification and Regression Trees*, tech. report, Wadsworth Int'l, 1984.
3. R. Agrawal, T. Imeilinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases," *Proc. ACM SIGMOD Conf.*, ACM Press, 1993; www.almaden.ibm.com/software/quest/Publications/papers/sigmod93.pdf.
4. S.B. Bay and M.J. Pazzani, "Detecting Change in Categorical Data: Mining Contrast Sets, *Proc. 5th Int'l Conf. Knowledge Discovery and Data Mining*, ACM Press, 1999, pp. 302-306.
5. C.H. Cai et al., "Mining Association Rules with Weighted Items," *Proc. Int'l Database Eng. and Applications Symp.* (IDEAS), 1998; www.cse.cuhk.edu.hk/~kdd/assoc_rule/paper_chcai.pdf.
6. R.C. Holte, "Very Simple Classification Rules Perform Well on Most Commonly Used Data Sets," *Machine Learning*, vol. 11, 1993, pp. 69-91.
7. R. Kohavi and G.H. John, "Wrappers for Feature Subset Selection," *Artificial Intelligence*, vol. 97, no. 1-2, 1997, pp. 273-324.
8. D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.
9. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, 1983, pp. 671-680.

```
stat <= 11.66
| rm <= 6.54
| | lstat <= 7.56 THEN medhigh
| | lstat > 7.56
| | | dis <= 3.9454
| | | | ptratio <= 17.6 THEN medhigh
| | | | ptratio > 17.6
| | | | | age <= 67.6 THEN medhigh
| | | | | age > 67.6 THEN medlow
| | | dis > 3.9454 THEN medlow
| rm > 6.54
| | rm <= 7.061
| | | lstat <= 5.39 THEN high
| | | lstat > 5.39
| | | | nox <= 0.435 THEN medhigh
| | | | nox > 0.435
| | | | | ptratio <= 18.4 THEN high
| | | | | ptratio > 18.4 THEN medhigh
| | rm > 7.061 THEN high
lstat > 11.66
| lstat <= 16.21
| | b <= 378.95
| | | lstat <= 14.27 THEN medlow
| | | lstat > 14.27 THEN low
| | b > 378.95 THEN medlow
| lstat > 16.21
| | nox <= 0.585
| | | ptratio <= 20.9
| | | | b <= 392.92 THEN low
| | | | b > 392.92 THEN medlow
| | | ptratio > 20.9 THEN low
| | nox > 0.585 THEN low
```

methods for translating trees to rules, in this case simply reading each branch as a separate rule works as well as any other method.[3] For example, we can collapse the first three lines of Figure 1 to a rule using two tests:

$$Rule J4.8a \begin{cases} IF: & lstat = 7.56 \\ & AND\ rm = 6.54 \\ THEN: & medhigh \end{cases}$$

To find a house-hunting policy, we can combine some of the decision values at all the branch points in Figure 1—that is, the values that select certain branches—and reject others. Each attribute in Figure 1 corresponds to one of 15 decision values—for example, *lstat* = 11.66 and *rm* = 6.54.

When TAR2 summarizes the same data, it generates a much smaller model with only two decision values (*rm* = 6.6, *ptratio* = 15.9):

$$Rule TAR2a \begin{cases} IF: & rm \geq 6.6 \\ & AND\ ptratio \leq 15.9 \\ THEN: & 97\%\ of\ the\ found \\ & houses\ will\ be\ high \\ & quality \end{cases}$$

Using this rule, a project manager could quickly find high-quality houses while avoiding low-quality houses.

We suspect that busy people would prefer TAR2's simple output to using the complex decision tree in Figure 1.

TREATMENT LEARNING

Three concepts define treatment learning: lift, minimum best support, and the small treatment effect.

Lift

A decision's *lift* is the change some decision makes to a set of examples after imposing that decision. For example, Table 1 is a log showing how much golf an individual played over 14 weekends and the weather conditions for each weekend. The baseline golf-playing behavior is that the golfer played no golf five times, some golf three times, and lots of golf six times: $5/14 \times none + 3/14 \times some + 6/14 \times lots$. If we knew scores for each outcome, such as none = 2, some = 4, and lots = 8, we could sum this baseline to

$$Sum(baseline) =$$
$$\frac{\frac{5}{14} \times 2 + \frac{3}{14} \times 4 + \frac{6}{14} \times 8}{2 + 4 + 8} = 0.357$$

These scores model the domain-specific view of the relative merits of, say, playing lots of golf. The scores indicate that golfers most value playing lots of golf and dislike playing no golf. (The exact scores don't matter too much, as long as we normalize their sum.)

Consider the effects of applying the decision not to play golf on rainy or sunny days. This effectively means *treating* the data by selecting the examples in Table 1 in which *outlook = overcast*. The treatment yields four examples in which the golfer always played lots of golf. We sum this yield as:

$$Sum(outlook = overcast) =$$
$$\frac{\frac{0}{4} \times 2 + \frac{0}{4} \times 4 + \frac{4}{4} \times 8}{2 + 4 + 8} = 0.57$$

We calculate the lift as the ratio of the treatment sum to the baseline sum:

$$Lift = \frac{Sum(outlook = overcast)}{Sum(baseline)}$$
$$= \frac{0.57}{0.357} = 1.6$$

Table 1. Log of weather conditions and golf-playing behavior.						
Weekend	Outlook	Temp (°F)	Humidity	Windy	Golf-playing behavior	Outlook = overcast
1	Sunny	85	86	False	None	
2	Sunny	80	90	True	None	
3	Sunny	72	95	False	None	
4	Rain	65	70	True	None	
5	Rain	71	96	True	None	
6	Rain	70	96	False	Some	
7	Rain	68	80	False	Some	
8	Rain	75	80	False	Some	
9	Sunny	69	70	False	Lots	
10	Sunny	75	70	True	Lots	
11	Overcast	83	88	False	Lots	•
12	Overcast	64	65	True	Lots	•
13	Overcast	72	90	True	Lots	•
14	Overcast	81	75	False	Lots	•

In the language of treatment learning, the best treatment is the one that results in the maximum lift greater than one—that is, most improves the outcome distributions compared to the baseline. RuleTAR2a gives the best treatment in the housing example, and *outlook = overcast* is the best treatment in the golf example. The last column of Table 1 shows why this treatment is so effective: *outlook = overcast* always appears when the golfer plays lots of golf and never when the golfer plays no or some golf.

If we apply the lifting notion iteratively to a simulator, treatment learning acts like a traditional sensitivity analysis.[4] In this approach, a simulator runs many times, learning treatments after each run. The simulator constrains each subsequent simulation to the space marked out by the previously learned treatments. In this way, the treatment learner "nudges" the simulator into better behavior.

Minimum best support

In the golfing example, the learned treatment is *outlook = overcast*, which uses only one attribute. Theoretically, treatments can refer to many attributes, potentially capturing more domain details. As treatments increase in size, however, they are harder to read and their benefit decreases. Most effective treatments use fewer than five attributes. Understanding why requires understanding how a treatment learner assesses its output.

Real-world databases contain some *noise*—incorrect values injected by accident or from imperfect data sources. A machine learner who includes every noisy example detail might overfit the model. An overfitted model captures the features of the current examples but will perform badly on future examples. To avoid overfitting, learners need a stopping criterion telling them when the detail is sufficient.

We based TAR2's stopping criterion on the *best support* measure. Recall that treatments select for preferred outcomes and avoid undesired outcomes.

Given a best outcome (outcome with the highest score), the best support is the ratio of the best outcomes a treatment finds. For example, in the golf example, the best outcome is playing lots of golf.

The treatment *outlook = overcast* and this test find four of the six best outcomes. Hence, the best support for this treatment is 4/6 = 0.67. To avoid overfitting, TAR2 rejects all treatments with less than some minimum best support value. The default for this minimum best support is 0.1, and the user can change this default.

Small treatment effect

An interesting side effect of using minimum best support is the *small treatment effect*—that is, treatments rarely use many attributes. Treatment learning rejects examples that fail the best-support test; thus, the more tests, the more rejected examples. As a treatment uses more tests, it becomes more likely that its best support value will become too small. For example, compare the best support of the following two treatments:

$$RuleTAR2b \begin{cases} IF: & outlook = overcast \\ THEN: & 100\% \text{ of the time} \\ & \text{the golfer will} \\ & \text{play lots of golf} \\ SUPPORT: & \dfrac{4}{6} = 0.67 \end{cases}$$

$$RuleTAR2c \begin{cases} IF: & outlook = overcast \\ & AND\, not\, windy \\ THEN: & 100\% \text{ of the time} \\ & \text{the golfer will play} \\ & \text{lots of golf} \\ SUPPORT: & \dfrac{2}{6} = 0.33 \end{cases}$$

Note that RuleTAR2c makes one more test than RuleTAR2b. Hence, the larger treatment selects fewer examples and receives less support. More generally, if every attribute that can takes V equally likely values, each test selects $1/V$ of the examples. A treatment using N tests therefore selects for $(1/V)^N$ of the examples.

TAR2 can convert numeric ranges into three or more discrete values. Assuming $V = 3$, a treatment using five or more tests will select $(1/3)^5 = 0.4$ percent of the examples or less. That is, unless the example set is very large, it is unlikely that large treatments will satisfy the minimum best support criteria.

Of course, this is merely an argument that building large treatments is useless. Experimentation with TAR2 reveals that small treatments are useful—that is, a treatment learner can learn adequate controllers using a small number of attributes.[1,5,6]

Heuristics

Although the lift calculation can assess candidate treatments, it doesn't help generate them. The art of treatment learning is finding good heuristics for generating candidate treatments.

A treatment is a conjunction of attribute ranges. TAR2 aims to find treatments that generate a large lift. A naive treatment learner might compute the lift for all subsets of all ranges of all attributes. However, because a set of size N has 2^N subsets, an exponential time search is inefficient. TAR2 hence uses three heuristic tricks to cull the search space:

- It chunks (or discretizes) continuous attributes into a small range by sorting their values and dividing the resulting array into a small number of equal-size bins.
- It assumes the small treatment effect and only builds candidate treatments for small treatments. By default, TAR2 only uses two chunks and builds treatments of size three. Although users can change the default, experience suggests that using more than five chunks or treatments with more than five attributes is rarely necessary.
- It only searches ranges with a high heuristic value.

TAR2 computes a range's heuristic value as follows. Given scores $\$O_i$ assigned to each outcome O_i, a valuable range occurs more frequently in desired outcomes (those with larger scores) than in undesired outcomes (those with lower scores).

TAR2 weights these frequency counts according to the difference in the scores between the desired and undesired outcomes and normalizes them by the sum of the frequency counts:

$$value(x) =$$
$$\frac{\sum_{r \in rest} (\$best - \$r) \times (|x \wedge best| - |x \wedge r|)}{|x|}$$

Here, *best* is the best outcome (playing lots of golf) and *rest* are the nonbest classes (none or some golf). In Table 1, for example, outlook = overcast appears 4, 0, 0 times when playing golf lots, some, and none, respectively. Also in the golf example, $\$lots = 8$ $\$some = 4$, and $\$none = 2$. Outlook = overcast scores the highest value of any range in Table 1:

$$\frac{\overbrace{((8-2) \times (4-0))}^{lots \to none} + \overbrace{((8-4) \times (4-0))}^{lots \to some}}{4 + 0 + 0} = 10$$

To assist analysts, TAR2 first prints a histogram of all the values of all the ranges in the data. Analysts then choose a threshold that selects only the most valuable attribute ranges.

CASE STUDIES

In the case studies discussed here, researchers applied standard data-mining methods and treatment learning to the same problems.[1,5] In all cases, standard methods generated decision trees with thousands of nodes whereas treatment learning generated useful models of fewer than a dozen lines.

Software risk estimation

In one of the earliest successful treatment-learning applications, researchers explored a space of 54 million options to find the two key control variables that most reduced a software engineering project's development risk.[7]

One case study used a Cocomo-based tool to evaluate the risk that a NASA software project would suffer from develop-time overrun.[8,9] Cocomo, a public-domain software cost estimation tool, requires a guesstimate of the system's source lines of code (SLOC) and certain internal tuning parameters ideally available in historical data. Lacking such data, the study used three guesses for SLOC and three sets of tunings from the literature.

In the study, feuding stakeholders proposed 11 changes to a project. Some of the project features were unknown at the time of this analysis (for

example, the expected CPU requirements of software that had not been built yet). For project features that were unknown, project managers could only offer ranges for the required inputs to the Cocomo-based tool. These ranges offered 2,930 possible input combinations. When combined with other uncertainties, this generated a space of 54 million possibilities: $2,930 \times 2^{11} \times$ three guesses for SLOC $\times$ three tunings $= 54 \times 10^6$. Faced with this overdose of possibilities, the researchers used data-mining techniques to build a system behavior log by performing 50,000 Monte Carlo simulations using inputs from the 54 million possibilities.

Initial experiments with data mining from this data set were not promising: Decision tree learners generated trees that were far too big to understand (some 6,000 nodes). Faced with this output overload, the researchers rethought their goals. They realized that a software project manager reading the trees might care only about the key decisions that most favored low-risk projects. This line of reasoning led to treatment learning, which succeeded in a domain in which conventional data-mining techniques had failed. Figure 2 shows the results.
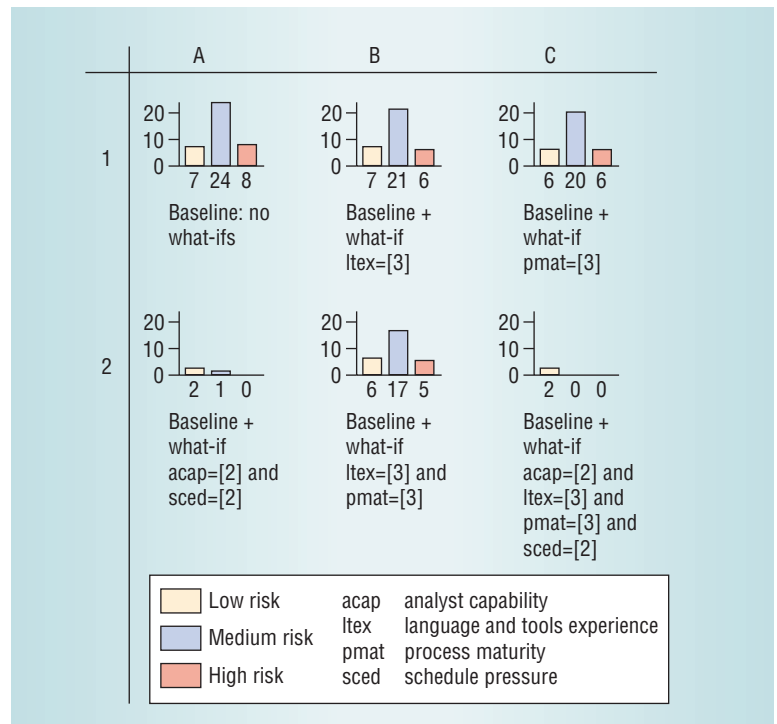
Cell A1 of Figure 2 gives the baseline risk profile of the current project seen in the 50,000 generated examples. Prior to learning, the ratio of risk types in the 50,000 examples is 7:24:8 for low-, medium-, and high-risk projects, respectively.

After treatment learning, the pattern is different. Seven of the proposed changes had little impact on the baseline. Two of the remaining four proposed changes are clearly superior. Cell A2 shows that having moderately talented analysts and no schedule pressure (*acap* = [2], *sced* = [2]) reduced the risk in this project nearly as much as any other, larger subset. One exception is cell B2, which applies actions to remove all branches to medium- and high-risk projects. Nevertheless, A2, not B2, is the recommended option because A2 seemed to achieve most of what B2 did, with much less effort.

Note that Figure 2 requires only 1/6th of a page to display the key factors controlling the classifications of 54,000,000 possibilities, proof of treatment learning's utility.

Software inspection policies

Software process modeling is a technique for understanding interactions within a software development. This study used a two-part software process model—a state-based simulation built using i-Logix's Statemate Magnum tool and a discrete event model using the Extend Simulation Language—to find the best software inspection policy for a particular software development organization.[10]

Developers at one software development firm built and debugged Extend and Statemate models for their processes over many months. The resulting models accurately predicted the impact of process changes. For example, the model predicted that development of one complex subsystem would take approximately twice the normal development time. Although management initially ignored this result as too long, the company's experience corresponded quite accurately with the model's predictions.
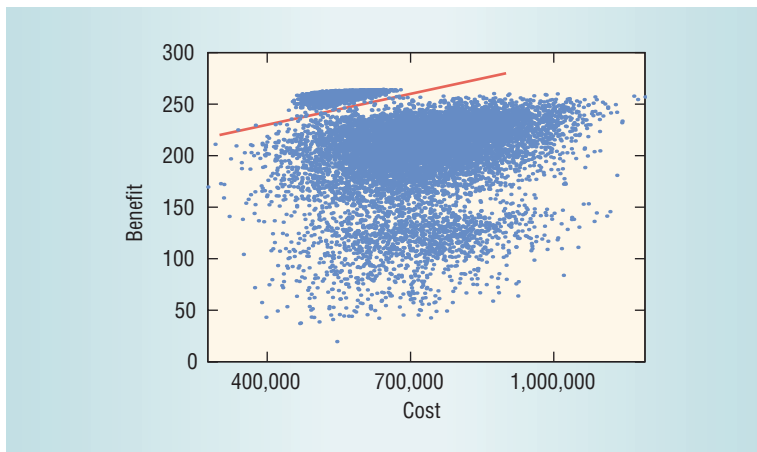
The project primarily used manual inspection methods for quality assurance. In the model, the number of staff (drawn from random distributions known to the model) involved in an inspection characterized that inspection. The model used four inspection policies:

- Do nothing.
- Conduct the company's current informal inspection method.
- Perform a somewhat more structured inspection.
- Perform a full formal inspection.[11]

Developers conducted inspections at various stages of the project life cycle, such as

- during initial functional specification,
- after high-level design,
- after low-level design, and
- after writing the code.

Hence, $4^4 = 256$ inspection configurations existed.

Figure 2. Branches to different risk classifications. The histograms detail the decision tree pathways to different outcomes under a variety of treatments.

Figure 3. Results from the satellite domain. The dots below the line show the model's initial output. The dots above the line show the final outputs of the model after five iterations of TAR2 learning.

Running the model required determining the number of staff involved in the inspections and the inspection policy at each stage.

Next, developers executed the model and assessed the output according to a domain-specific utility function. The utility function used in this study modeled local concerns about tradeoffs between the software process's cost and duration as well as the quality of the generated software, measured in the estimated number of software defects.

The problem with the model is that it generated too much output. For example, sampling the range of possible staff allocations to each inspection required executing each of the 356 inspection configurations 30 times. Each execution's output contained details on nine possible process options, including the inspection policies. Worse still, the interrelationships between those $30 \times 256 \times 9 = 69,120$ numbers were complex: A decision tree learner working on this example generated a tree with 7,209 nodes, far too large to understand.

In contrast, TAR2 learned a preferred inspection policy that increased the mean utility values seen in the simulator by a factor of 1.35 while reducing the standard deviation of the utilities by a factor of 2.5. TAR2's analysis also showed that the reported inspection policy was valid, despite large-scale variations in other process options. Finally, TAR2's output was much smaller than the decision tree learner. Instead of generating a tree with thousands of nodes, TAR2 generated a single rule that mentioned only the best inspection policy.

Requirements engineering

Analysts at NASA's Jet Propulsion Laboratory (JPL) sometimes debate satellite design by building a semantic network connecting design decisions to satellite requirements.[12] The network links faults and risk mitigation actions that affect a stakeholder-written requirements tree. Stakeholders model potential faults within a project as influences on the edges between requirements; they model potential fixes as influences on the edges between faults and requirements edges.

This kind of requirements analysis seeks to maximize requirements coverage while maximizing how the actions reduce the impact of the faults and minimizing their costs. The model's interior interactions complicate optimizing all criteria.

The JPL analysts execute the semantic net by selecting actions and observing the resulting benefits. One such network included 99 possible actions, or $2^{99} \times 10^{30}$ combinations of actions. In Figure 3, 10,000 random selections of the decisions and the collection of their associated costs and benefits generated the dots below the black line at the top left. All the dots above this line represent high benefit, low-cost projects found by iterative applications of TAR2.

At each iteration, researchers gave the simulator's output to TAR2 to find the settings that most improved cost and reduced benefits. Researchers then imposed the treatment that TAR2 found on the simulator for subsequent iterations.

In a result consistent with the small treatment effect, the learner could search a space of 10^{30} decisions to find 30 (out of 99) that crucially affected the satellite's cost/benefit ratio. This means TAR2 also found $99 - 30 = 69$ arbitrary decisions that could be made with minimal software impact.

Applying genetic and simulated annealing algorithms to the Figure 3 domain revealed decisions that generated high-benefit, low-cost projects.[13] Further, the comparison revealed that the benefits and costs were about the same as those TAR2 found. However, these algorithms generated solutions that commented on every possible decision, and there was no apparent way to ascertain which decisions were most critical. The TAR2 solution required just 30 actions.

Lotus founder Mitchell Kapor once said, "Getting information off the Internet is like drinking from a fire hydrant." We should take Kapor's observation seriously. Unless we can process the mountain of information surrounding us, we must either ignore it or let it bury us. ∎

ufacturer, or otherwise does not constitute or imply its endorsement by the United States government.

References

1. T. Menzies et al., "Condensing Uncertainty via Incremental Treatment Learning," *Ann. Software Eng.*, 2002; http://menzies.us/pdf/02itar2.pdf.

2. G. Gigerenzer and D.G. Goldstein, "Reasoning the Fast and Frugal Way: Models of Bounded Rationality," *Psychological Rev.*, vol. 103, 1996, pp. 650-669.

3. I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann, 1999.

4. J.P.C. Kliijnen, "Sensitivity Analysis and Related Analyses: A Survey of Statistical Techniques," *J. Statistical Computation and Simulation*, vo1. 57, no. 1-4, 1987, pp. 111-142.

5. T. Menzies and Y. Hu, *Just Enough Learning (of Association Rules): The TAR2 Treatment Learner*, tech. report, Dept. Computer Science and Electrical Eng., West Virginia Univ., 2002.

6. T. Menzies and H. Singh, "Many Maybes Mean (Mostly) the Same Thing," *Proc. 2nd Int'l Workshop Soft Computing Applied to Software Eng.*; http://menzies.us/pdf/00maybe.pdf.

7. T. Menzies and E. Sinsel, "Practical Large-Scale What-If Queries: Case Studies with Software Risk Assessment," *Proc. 15th IEEE Int'l Conf. Automated Software Eng.* (ASE 2000), IEEE CS Press, 2000, pp. 165-173.

8. R. Madachy, "Heuristic Risk Assessment Using Cost Factors," *IEEE Software*, May 1997, pp. 51-59.

9. B.W. Boehm et al., *Software Cost Estimation with Cocomo II*, Prentice-Hall, 2000.

10. T. Menzies et al., "Model-Based Tests of Truisms," *Proc. 16th IEEE Int'l Conf. Automated Software Eng.* (ASE 2002), IEEE CS Press, 2002, pp. 183-191.

11. M. Fagan, "Advances in Software Inspections," *IEEE Trans. Software Eng.*, July 1986, pp. 744-751.

12. M.S. Feather, S.L. Cornford, and T.W. Larson, "Combining the Best Attributes of Qualitative and Quantitative Risk Management Tool Support," *Proc. 15th IEEE Int'l Conf. Automated Software Eng. (ASE 2000)*, IEEE CS Press, 2000, pp. 309-312.

13. M.S. Feather and T. Menzies, "Converging on the Optimal Attainment of Requirements," *Proc. IEEE Joint Conf. Requirements Eng.* (RE 2002), IEEE CS Press, 2002, pp. 263-272.

Tim Menzies is a research associate professor in the Lane Department of Computer Science and Electrical Engineering at West Virginia University. His research interests include software engineering and data mining. Menzies received a PhD in artificial intelligence from the University of New South Wales, Australia. He is a member of the ACM and the IEEE. Contact him at tim@menzies.us or visit his Web site at http://menzies.us.

Ying Hu received a master's of applied science in electrical and computer engineering from the University of British Columbia, Vancouver, Canada. Her research interests include machine learning and uncertain reasoning. Contact her at yingh@ece. ubc.ca.