Autom Softw Eng (2010) 17: 439–468
DOI 10.1007/s10515-010-0072-x

Automatically finding the control variables for complex system behavior

Gregory Gay · Tim Menzies · Misty Davies · Karen Gundy-Burlet

Received: 11 November 2009 / Accepted: 18 May 2010 / Published online: 29 May 2010
© Springer Science+Business Media, LLC 2010

Abstract Testing large-scale systems is expensive in terms of both time and money. Running simulations early in the process is a proven method of finding the design faults likely to lead to critical system failures, but determining the exact cause of those errors is still time-consuming and requires access to a limited number of domain experts. It is desirable to find an automated method that explores the large number of combinations and is able to isolate likely fault points.

Treatment learning is a subset of minimal contrast-set learning that, rather than classifying data into distinct categories, focuses on finding the unique factors that lead to a particular classification. That is, they find the smallest change to the data that causes the largest change in the class distribution. These treatments, when imposed, are able to identify the factors most likely to cause a mission-critical failure. The goal of this research is to comparatively assess treatment learning against state-of-the-art numerical optimization techniques. To achieve this, this paper benchmarks the TAR3 and TAR4.1 treatment learners against optimization techniques across three complex systems, including two projects from the Robust Software Engineering (RSE) group within the National Aeronautics and Space Administration (NASA) Ames Research

G. Gay (✉) · T. Menzies
West Virginia University, Morgantown, WV, USA
e-mail: greg@greggay.com

T. Menzies
e-mail: tim@menzies.us

M. Davies · K. Gundy-Burlet
NASA Ames Research Center, Moffett Field, CA, USA

M. Davies
e-mail: misty.d.davies@nasa.gov

K. Gundy-Burlet
e-mail: karen.gundy-burlet@nasa.gov

Center. The results clearly show that treatment learning is both faster and more accurate than traditional optimization methods.

Keywords Contrast-set learning · Search-based software engineering · Simulation · Optimization · Monte Carlo filtering

1 Introduction

Large-scale industrial systems, often containing both hardware and software components, are expensive to design and expensive to build. The cost of design failures decreases exponentially the earlier that these faults are discovered in the design process (Boehm and Papaccio 1988). When a system is expensive (especially when safety is part of the system cost), it is critical to use the best tools available at each stage of the design life cycle. Early in the design process, lower-level requirements (e.g. the center of gravity shall be placed exactly *here*) are built from higher-level requirements (e.g. the craft shall not spin about any axis above some rate) using simplified models and a factor of safety. As the complexity of the system grows, the compounded errors in the simplified models can erode the factor of safety until the overall system is not as robust as originally intended. As a last line of defense before a complete prototype is constructed, high-fidelity simulations composed of models for the entire system—hardware, software and environment—are used to eliminate potentially costly fault points and estimate the overall system margins to failure.

However, simply simulating the behavior of a system is not enough. Knowing that something fails in simulation is not the same as knowing *why* it failed or whether there are control variables that can be used to eliminate that failure. Once the simulations become high-fidelity enough and complex enough to represent reality, the relationships between the control variables and the eventual system outcomes become obscured. No matter what domains a system falls into, there are a limited number of experts and their time is even further limited. It is cost and risk-prohibitive to ask those experts to sift through gigabytes of simulation settings and outcomes. Therefore, it is desirable to find methods of eliminating that bottleneck: methods that either reduce the amount of data that experts need to examine or methods that can identify the most obvious faults automatically.

For example, consider *Monte Carlo Filtering* as applied at NASA's Robust Software Engineering (RSE) group. The goal of this filtering is to determine which inputs are most likely to determine some portion of the output distributions. The output space is divided into 'good' or 'bad' partitions using some mathematical function of the outputs—for example, a NASA scientist may be most interested in data where the allowable dynamic pressure on the parachutes is exceeded. For this kind of sensitivity analysis, the first step is to run a Monte Carlo experiment sampling the input space, and the second step is to filter the data into two partitions based on the output. In the first step, model inputs are selected at random. In the second step, some sensitivity analysis is then used to predict the variables and ranges in the input space most likely to lead to one of the partitions of the data. A detailed description of Monte Carlo Filtering including a variety of examples and techniques for this type of sensitivity analysis is located in Saltelli et al. (2000).

The field of data mining uses techniques from statistics and artificial intelligence to find small, yet relevant, patterns in large sets of data. The standard practice in this field is to *classify*, to look at an object and make a guess at what category it belongs to. As new evidence is examined, these guesses are refined and improved. When testing a complex hardware system, a scientist might try to classify by assessing whether that particular simulation *succeeded* or *failed*.

Treatment learning (Menzies and Hu 2003) focuses on a different goal. It does not try to determine *what is*, it tries to determine *what could be* (and thus, enables the practice of Monte Carlo Filtering). Classifiers read a collection of data and collect statistics that are used to place unseen data into a series of discrete categories (called classes). Treatment learners work in reverse. They take the classification of a piece of evidence (that is, the category that it belongs to) and try to reverse-engineer the statistical evidence that led a classifier to assign the data to a particular class. If a scientist already knows whether a simulation succeeded or failed, the scientist can use treatment learning to determine why it failed. Treatment learners produce a treatment—a small set of rules that, if imposed, will change the expected classification distribution. By filtering the data for entries that follow the rules set in the treatment, you should be able to identify *why* a particular classification was reached.

Ultimately, classifiers will strive to increase the representational accuracy. They will assess the data and grow a collection of statistical rules with the goal of making more and more accurate categorizations. As a result, if the data is complex, the decision tree output by the classifier will also be complex. Treatment learning instead focuses on minimality: what is the *smallest* rule that can be imposed to cause the *largest* change. Often, these rules may involve only one control variable (e.g. don't launch if the wind speed is greater than 45 knots).

To give a simplified example, consider the case of a rocket intended to place a satellite into orbit. Any number of events could cause this rocket to fail–too long or too short of a burning time, a faulty timing mechanism, or a crack in the outer fairing. A common scenario would be to run a series of simulations prior to any actual launch. Measurements are taken at regular intervals throughout the simulation, noting factors such as the trajectory, external pressure, temperature, etc. Along with these readings comes a classification, whether or not any of the system requirements for the rocket have been violated. After a number of simulation trials, a treatment learner can be focused upon those trials with a "rocket failed" classification. The treatment learner will produce a rule set that states definitively the control variables (temperature too high, fuel level too low) that most often caused a mission-critical failure.

Stated formally, treatment learning is a form of minimal contrast-set association rule learning. The treatments contrast undesirable situations with the desirable ones (represented by weighted classes). Treatment learning, however, is different from other contrast-set methods like STUCCO (Bay and Pazzani 1999) because of its focus on minimal theories. Conceptually, a treatment learner explores all possible subsets of the attribute ranges looking for good treatments. Such a search is infeasible in practice, so the art of treatment learning lies in quickly pruning unpromising attribute ranges (i.e. ignoring those that, when applied, lead to a class distribution where the target class is in the minority).

In an industrial setting like NASA, critical mission failures will cost thousands, if not millions, of dollars. Therefore, it is absolutely crucial to identify the conditions

under which a design will fail as early as possible in the design process, so that design changes may be made before any physical hardware is constructed. Standard optimization techniques can be used to relate the control variables to the outcomes, but many such algorithms rely on continuous variables (all of which must be controllable by the simulator). They are ill-equipped to handle many real-world situations where factors are either discrete or stochastic. Treatment learning has no such restrictions.

While treatment learning (specifically the TAR3 algorithm Hu 2002; Gundy-Burlet et al. 2007; Gundy-Burlet et al. 2009; Schumann et al. 2009) has been discussed in prior publications, these algorithms have never been benchmarked against standard or state-of-the-art optimization algorithms in an industrial setting. This study utilizes the TAR3 and TAR4.1 treatment learners, which operate similarly but score treatments in radically different manners, and compares the quality of the produced treatments against the Simulated Annealing (Metropolis et al. 1953; Kirkpatrick et al. 1983) and Quasi-Newton (Gill et al. 1981) optimization methods. Data used in this case study comes from actual simulation trials of projects from NASA's Robust Software Engineering (RSE) group, as well as real-world data from a bicycle ride. Our goal is to show that, in this real-world industrial setting, treatment learning offers a faster, higher-quality identification of the factors likely to cause a failure in a complex system than traditional optimization techniques.

The results of this study clearly demonstrate that:

- Treatment learners are orders of magnitude faster than standard methods.
- TAR3's results are more precise than those from standard techniques.
- TAR4.1's results demonstrate a higher recall, while maintaining a lower false positive rate, than the standard techniques (the Quasi-Newton algorithm also demonstrates a high recall, but at the notable cost of a higher rate of false positives).

2 Data background

NASA often uses high-fidelity physics simulations early in the design process to verify that flight software will meet the mission requirements. The possible inputs to the simulation can be design parameters like lift coefficients and center of gravity positions; they may also be environmental parameters like the average magnitudes and the standard deviations of wind gusts; they may be flags that indicate the failure of a hardware component at some critical time; or they may be any of a plethora of other parameters that specify the bounds on the acceptable flight envelope. Errors in software design are much less expensive to fix early in the design process, but the design space early in the process is very large. To explore all of the parameter combinations exhaustively is infeasible. However, a very large sampling of the configurations increases the chance that a design error will be caught early, and it allows for the possibility of identifying trends or anomalies within the data.

Manual inspection of these large datasets requires domain expertise, and is likely to focus only on absolute compliance with requirements. For systems this complex many different kinds of failures are possible. For aero-braking scenarios, for instance, representative failures include skipping out of the Earth's atmosphere instead of re-entering and parachutes failing to open. One common heuristic for these analyses is

to push the bounds of the flight envelope until between 10 and 30 percent of all of the attempted cases have failed in one way or another—this kind of analysis allows you to find the margins of failure within the system. In the probable event that failures are identified this early in the process, it is a time-consuming task to determine the cause of those failures; the failures may be associated with environmental factors (e.g. strong sustained wind gusts overwhelm the control system), they may be associated with software errors in the unit under test (e.g. the gains on the control system are set incorrectly for the nominal case), or they may be associated with software errors in the simulation used to perform the test (that is, a legacy environmental model in the simulation uses different units than expected). A likely first step towards determining the cause of any of the failures is to find the input parameters that the failures are associated with. Even this first step is non-trivial—there are usually hundreds of input parameters associated with each dataset, and those parameters were chosen from thousands of input parameters to the actual simulation.

Two of the datasets used for this paper were generated using the Advanced NASA Technology ARchitecture for Exploration Studies (ANTARES) (Acevedo et al. 2007) simulation tool. ANTARES is composed of high-fidelity physics models that are used to study Constellation and the International Space Station. The version of ANTARES used for this work contains over 1 million source lines of code in 15 different programming languages. Each individual simulation, however, touches a relatively small subset of this code. The two ANTARES datasets used for this paper represent the Monte Carlo simulation trials done for a re-entry study and for a launch abort study. The collected variables include environmental parameters, internal simulation values (like random seeds), and spacecraft state specifications—including both continuous and modal information. Both of these datasets had many different possible failure types. The third dataset referenced here was collected during a bicycle ride. The data comes from a bicycle power meter that calculates the power output and collects the following data at 60 Hz: distance traveled, heart rate, speed, wind speed, cadence, elevation, hill slope, and temperature. The power calculation for this particular meter could be very noisy and there was an open question about what measured parameters were most associated with this noise. Note that, because the bicycle data is real-world data, the relationships between the measured variables are not explicit.

Failures for the NASA datasets were defined for several phases of flight corresponding to reentry and launch abort scenarios. Some of the specific failure types included missed landings, aerodynamic angles or body rates that exceeded thresholds, excessively high velocity at impact, and dynamic pressures exceeding tolerances for the parachutes. Any of these failures can lead to loss of life or mission, and are considered unacceptable. In general use at NASA, this tool is used to find the margins to failure from nominal launch conditions over all of the mission-critical failures. In essence, the question is—which of the controllable input parameters need to be most-closely monitored in order to prevent any of these unacceptable failures? Since each individual failure type has its own complicated, usually non-smooth, function of the inputs, the composite of all of the failures creates a non-trivial, almost certainly non-convex, hypersurface that must be searched.

To decrease the amount of time necessary to isolate suspicious inputs, the Robust Software Engineering (RSE) group at Ames utilizes a multi-step process (Gundy-Burlet et al. 2009; Gundy-Burlet et al. 2007; Schumann et al. 2008; Schumann et al.

2009) that includes targeting tests with n-factor combinatorial test vectors and sorting the data into clusters with an unsupervised EM algorithm(Fischer and Schumann 2003) in order to find anomalies and to aid visualization. This tool is known as Margins Analysis. A key component in the Margins Analysis is the use of a treatment learner to tie behaviors in the dataset to their associated variables and ranges. The treatment learner is used many times throughout the analysis process. It first is used to find variables associated with the overall performance; a penalty is built for each dataset that accounts for all failures and often includes some continuous metric like a target miss distance. The treatment learner is then used to find the parameters associated with each individual type of failure. In practice there are 10's to 100's of different possible failure types identified for each dataset. Finally, the treatment learner is used within a loop to discover the variables associated with each unsupervised cluster. This analysis aids the researcher in understanding the underlying structure of the dataset and can uncover details in the dataset that lead to new requirements.

3 Data mining and treatment learning

3.1 Data mining

Data mining is a summarization technique that reduces large sets of examples into small understandable patterns using a range of techniques taken from the statistics and artificial intelligence fields (Witten and Frank 1999; Goldberg 1989; Boetticher 2001). One way to learn such patterns is to *split* the whole example set into subsets based on some attribute value test. The process then repeats recursively on the subsets. Each splitter value becomes the root of a sub-tree. Splitting stops when either a subset gets so small that further splitting is superfluous, or a subset is contains examples with only one classification.

A *good split* decreases the percentage of different classifications in a subset. Such a good split ensures that smaller subtrees will be generated since less further splitting is required to sort out the subsets. Various schemes have been described in the literature for finding good splits. For example, the C4.5 (Quinlan 1992) and J4.8 (Witten and Frank 1999) decision tree algorithms uses an information theoretic measure (entropy) to find its splits while the CART (Breiman et al. 1984) decision tree learner uses another measure called the GINA index.

Decision trees can be large and complex. The problem of explaining the performance of these learners to end-users has been explored extensively in the literature (see the review in Taylor and Darrah 2005). Often, some *post-processor* is used to convert an opaque model into a more understandable form:

- Towell and Shavlik generate refined rules from the internal data structures of a neural network (Towell and Shavlik 1993).
- Quinlan implemented a post-processor to C4.5 called C45 rules that generates succinct rules from cumbersome decision tree branches via (a) a greedy pruning algorithm followed by (b) duplicate removal then (c) exploring subsets of the rules relating to the same class (Quinlan 1992).

– The first version of our treatment learner (TAR1) was another post-processor to C4.5 that searched for the smallest number of decisions in decision tree branches that (a) pruned the most branches to undesired outcomes while (b) retaining branches leading to desired outcomes (Menzies and Sinsel 2000).

Association rule learners such as APRIORI (Agrawal et al. 1992) find attributes that commonly occur together in a training set. In the association $LHS \rightarrow RHS$, no attribute can appear on both sides of the association; i.e. $LHS \cap RHS = \emptyset$. The rule $LHS \rightarrow RHS$ holds in the example set with *confidence* c if $c\%$ of the examples that contain LHS also contain RHS; i.e. $c = \frac{|LHS \cup RHS| * 100}{|LHS|}$. The rule $LHS \rightarrow RHS$ has *support* s in the example set if $s\%$ of the examples contain $LHS \cup RHS$; i.e. $s = \frac{|LHS \cup RHS| * 100}{|D|}$ where $|D|$ is the number of examples. Association rule learners return rules with high confidence (e.g. $c > 90\%$). The search for associations is often culled via first rejecting associations with low support. Association rule learners can be viewed as generalizations of decision tree learning since the latter restrict the *RHS* of rules to just one special class attribute while the former can add any number of attributes to the *RHS*.

An interesting variant of association rule learning is *contrast set learning*. Instead of finding rules that describe the current situation, contrast set learners like STUCCO (Bay and Pazzani 1999) 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 Ph.D. and bachelor degrees?".

Another interesting variant is *weighted class learning*. Standard classifier algorithms such as C4.5 or CART have no concept of a *good* or *bad* class. Such learners therefore can't filter their learned theories to emphasize the location of the *good* classes or *bad* classes. Association rule learners such as MINWAL (Cai et al. 1998) use weights assigned to each class to focus the learning onto issues that are of particular interest to some audience.

3.2 Treatment learning

In terms of the above, treatment learning is a weighted contrast set learner that finds rules that associate attribute values with changes to the class distributions. Menzies and Sinsel (2000) elaborated the concept of treatment learning while trying to explain the output of data miners to business users. In one domain, he found that users never understood the large theories being generated using any of the above techniques. In an extreme example of this, C4.5 was generating trees with 6000 nodes. The TAR1 prototype (discussed above) achieved some remarkable reductions in that space; specifically, it found constraints on just four variables that pruned away all branches except those leading to the most preferred outcome.

The lesson of TAR1 was that, sometimes, a small minority of constraints can control a much larger space of variables. TAR2 was an experiment in generating tiny theories using this *simplicity assumption*; a small number of factors most influence the outcome. This assumption has two consequences: (1) it implies that the search for an effective model need not be too elaborate; (2) more importantly (in terms of explanation) the generated theory is very small.

The details of treatment learning are discussed below. Before that, it is insightful to ask just how general is this *simplicity assumption* of treatment learning? Empirically, we can state TAR2, TAR3 and TAR4 have been applied to dozens of data sets and in all cases, the small rules generated by this method have been sufficient to select for a large percentage of the preferred outcomes. Other machine learning researchers have also discovered that simple schemes, using only a subset of the available attributes, can generate effective theories. For example, Holte (1993) wrote a machine learner called 1R that was deliberately restricted to learning theories using a single attribute. Surprisingly, he found that learners that use many attributes perform only moderately better than the simpler 1R solution. It should be noted that we don't use the 1R technique—our results show that many of the best treatments will require more than one attribute (though, generally less than four).

In their work on learning simple theories, Kohavi and John (1997) *wrapped* their learners in a pre-processor that used a heuristic search to grow subsets of the available attributes from a size of one. At each step in this growth, a learner was used to assess the accuracy of the model learned from the current subset. Subset growth was stopped when the addition of new attributes failed to improve the accuracy. In their experiments, 83% (on average) of the attributes in a domain could be ignored with only a minimal lose of accuracy. Again, our learners do not use this technique—relevant feature selection with wrappers can be prohibitively slow since *each step* of the heuristic search requires a call to the learning algorithm. Under treatment learning's *simplicity assumption*, such an exhaustive search is needlessly complex.

3.3 Example output

One reason to recommend treatment learning is that its theories are succinct and easy to understand. Figure 1 shows the output of a classifier (the j48 tree learner from the WEKA[1] toolkit) as contrasted with the output of a treatment learner (TAR3, as defined in a following section) in Fig. 2 on housing data from the city of Boston (a nontrivial dataset with over five hundred examples). The tree output by the j48 classifier is detailed, but difficult to understand. A user must follow the branches of the tree in order to derive conclusions. The treatment given by TAR3 is much easier to understand. It only presents three important pieces of information. First, it tells us the baseline class distribution. It then shows the best treatment—the smallest constraint that most changes the class distribution. In this case, the best houses had 6.7 to 9.78 rooms, and the optimal parent-teacher ratio was 12.6 to 16. Finally, it shows the class distribution if that treatment is applied.

These same treatments can be mapped graphically in a way that is even easier to understand. Figure 3 shows a two-variable treatment for a dataset that represents the time series operation of a bicycle. Each variable or combination of variables within the treatment is given a one or two-dimensional plot. All of the possible values for the noted attributes are plotted, with different shapes to indicate the class. The area inside of the rectangle is the data that the treatment learner is trying to isolate. Lines are plotted around the minimum and maximum values of the ranges given by the

[1] http://www.cs.waikato.ac.nz/ml/weka/.

```
LSTAT <= 14.98
|   RM <= 6.54
|   |   DIS <= 1.6102
|   |   |   DIS <= 1.358: high (4.0/1.0)
|   |   |   DIS > 1.358
|   |   |   |   LSTAT <= 12.67: low (2.0)
|   |   |   |   LSTAT > 12.67: medlow (2.0)
|   |   DIS > 1.6102
|   |   |   TAX <= 222
|   |   |   |   CRIM <= 0.06888: medhigh (3.0)
|   |   |   |   CRIM > 0.06888: medlow (4.0)
|   |   |   TAX > 222: medlow (199.0/9.0)
|   RM > 6.54
|   |   RM <= 7.42
|   |   |   DIS <= 1.8773: high (4.0/1.0)
|   |   |   DIS > 1.8773
|   |   |   |   PTRATIO <= 19.2
|   |   |   |   |   RM <= 7.007
|   |   |   |   |   |   LSTAT <= 5.39
|   |   |   |   |   |   |   INDUS <= 6.41: medhigh (25.0/1.0)
|   |   |   |   |   |   |   INDUS > 6.41: medlow (2.0)
|   |   |   |   |   |   LSTAT > 5.39
|   |   |   |   |   |   |   DIS <= 3.9454
|   |   |   |   |   |   |   |   RM <= 6.861
|   |   |   |   |   |   |   |   |   INDUS <= 7.87: medhigh (9.0)
|   |   |   |   |   |   |   |   |   INDUS > 7.87: medlow (3.0/1.0)
|   |   |   |   |   |   |   |   RM > 6.861: medlow (3.0)
|   |   |   |   |   |   |   DIS > 3.9454: medlow (14.0/1.0)
|   |   |   |   |   RM > 7.007: medhigh (29.0)
|   |   |   |   PTRATIO > 19.2: medlow (11.0/1.0)
|   |   RM > 7.42
|   |   |   PTRATIO <= 17.9: high (25.0/1.0)
|   |   |   PTRATIO > 17.9
|   |   |   |   AGE <= 43.7: high (2.0)
|   |   |   |   AGE > 43.7: medhigh (3.0/1.0)
LSTAT > 14.98
|   CRIM <= 0.63796
|   |   INDUS <= 25.65
|   |   |   DIS <= 1.7984: low (5.0/1.0)
|   |   |   DIS > 1.7984: medlow (37.0/2.0)
|   |   INDUS > 25.65: low (4.0)
|   CRIM > 0.63796
|   |   RAD <= 4: low (13.0)
|   |   RAD > 4
|   |   |   NOX <= 0.655
|   |   |   |   AGE <= 97.5
|   |   |   |   |   DIS <= 2.2222: low (8.0)
|   |   |   |   |   DIS > 2.2222: medlow (6.0/1.0)
|   |   |   |   AGE > 97.5: medlow (5.0)
|   |   |   NOX > 0.655
|   |   |   |   CHAS = 0: low (80.0/8.0)
|   |   |   |   CHAS = 1
|   |   |   |   |   DIS <= 1.7455: low (2.0)
|   |   |   |   |   DIS > 1.7455: medlow (2.0)
```

Fig. 1 A decision tree generated by the WEKA's j48 classifier

treatment. A single glance informs the user that the area contained within these bold lines is the area of interest. For example, the plot in Fig. 3 shows the values for the attributes "Hill Slope" and "Cadence" The treatment suggests limiting the values for these variables to the area enclosed by these lines (about –14% to –1% for Hill Slope and 1.3 to 2.2 for Cadence). This output is easy to read and understand, especially when compared to the complex tree output by a classifier (see Fig. 1).

```
Baseline class distribution:
low:~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~       [   133 - 29%]
medlow:~~~~~~~~~~~~~~~~~~~~~~~~~~~~~     [   131 - 29%]
medhigh:~~~~~~~~~~~~~~~~~~~~~            [    97 - 21%]
high:~~~~~~~~~~~~~~~~~~~~~               [    94 - 21%]

 Treatment:[PTRATIO=[12.6..16)
           RM=[6.7..9.78)]

New class distribution:
low:                                     [     0 -  0%]
medlow:                                  [     0 -  0%]
medhigh:                                 [     1 -  3%]
high:~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~      [    38 - 97%]
```

Fig. 2 A textual treatment generated by the TAR3 learner

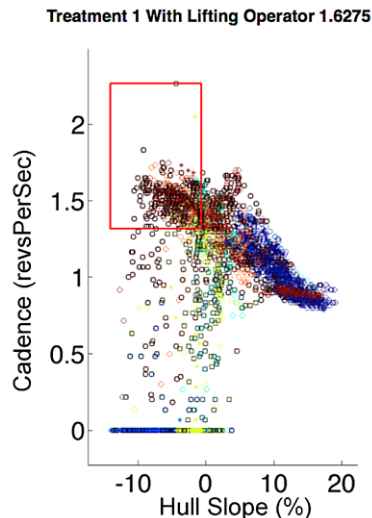

Fig. 3 A graphical representation of a treatment generated by the TAR3 learner

Cognitive scientists have demonstrated that humans are far more likely to use simple models over more complex ones when making decisions (Gigerenzer and Goldstein 1996). The simpler the output, the easier it is to implement and the more likely that designers will make use of it. This is a key point. For treatment learning to make an effective difference during the design phase of a project, it must produce output that can be understood in a single glance.

3.4 BORE classification

The raw datasets commonly produced by NASA simulations are not stamped with a basic classification (such as "failed" or "succeeded"). Instead, each simulation trial is assigned a score from a continuous distribution (as assigned by a penalty function unique to each system being simulated). Therefore, before they can be used by the treatment learner, these scores must be sorted into discrete classes. We assign these

classes using a process called *BORE*, short for *b*est *o*r *re*st. BORE is a general classification scheme that—given data and a mathematical function involving one or more attributes—categorizes a piece of data as "best" or "rest" according to this function. Commonly, this is not a strict binary split. For example, the scores might be sorted into four quartiles. The top quartile ($0.75 * MAX$ to MAX) will be classified as "best," while the other three quartiles will be classified as *rest*1, *rest*2, *rest*3.

BORE maps the individual factors into a hypercube, which has one dimension for each scored utility. It then normalizes instances scored on the N dimensions from 0 for "worst" to 1 for "best." The corner of the hypercube at $1, 1, \ldots$ is the *apex* of the cube and represents the desired goal for the system. All of the examples are scored by their normalized Euclidean distance to the apex.

For the purposes of this study, outputs were scored on only one dimension- the scores assigned to each data instance by that system's penalty function. For each run i of the simulator, the n outputs X_i are normalized to the range 0..1 as follows:

$$N_i = \frac{X_i - \min(X)}{\max(X) - \min(X)}. \tag{1}$$

The Euclidean distance of $N_1, N_2, \ldots$ to the ideal position of $N_1 = 1, N_2 = 2, \ldots$ is then computed and normalized to the range 0..1 as

$$W_i = 1 - \frac{\sqrt{N_1^2 + N_2^2 + \cdots}}{\sqrt{n}}, \tag{2}$$

where higher W_i $(0 \leq W_i \leq 1)$ correspond to better runs. This means that the W_i can only be improved by increasing all of the utilities. To determine the "best" and "rest" values, all of the W_i scores were sorted according to a given threshold. Those that fall above this threshold are classified as "best" and the remainder as "rest" (or, in some cases, multiple divisions of "rest").

3.5 TAR3

TAR3 (and its predecessor TAR2 (Menzies and Hu 2003)) are based on two fundamental concepts—lift and support. The *lift* of a treatment is the change that some decision makes to a set of examples after imposing that decision. TAR3 is given a set of training examples E. Each example $e \in E$ contains a set of attributes, each with a specific value (which have commonly been discretized into a series of ranges). These attributes (and the range their values fall within are directly mapped to a specific classification (stated formally—$R_i, R_j, \ldots \rightarrow C$). The individual class symbols $C_1, C_2, \ldots$ are ranked and sorted based on a utility score ($U_1 < U_2 < \cdots < U_C$, where U_C is the target class). Within dataset E, these classes occur at certain frequencies ($F_1, F_2, \ldots, F_C$) where $\sum F_i = 1$ (that is, each class occupies a fraction of the overall dataset). A treatment T of size M is a conjunction of attribute value ranges $R_1 \wedge R_2 \wedge \cdots \wedge R_M$ (these ranges are obtained by discretizing and combining several of the original continuous attribute values). Some subset of the dataset ($e \subseteq E$) is contained within the treatment; that is, if the treatment is used to filter E, $e \subseteq E$ is what will remain. In that subset, the classes occur at frequencies $f_1, f_2, \ldots, f_C$. TAR3

seeks the smallest treatment T which induces the biggest changes in the weighted sum of the utilities multiplied by the frequencies of the classes. This score, the score of $e \subseteq E$ where T has been imposed, is divided by the score of the baseline (dataset E when no treatment has been applied). Formally, the lift is defined as

$$lift = \frac{\sum_c U_c f_c}{\sum_c U_c F_c}. \quad (3)$$

The classes used for treatment learning are assigned a score $U_1 < U_2 < \cdots < U_C$ and the learner uses this to assess the class frequencies resulting from applying a treatment by finding the subset of the inputs that falls within the reduced treatment space. In normal operation, a treatment learner conducts *controller learning*; that is, it finds a treatment which selects for better classes and rejects worse classes. By reversing the scoring function, treatment learning can also select for the worst classes and reject the better classes. This mode is called *monitor learning* because it locates the one thing we should most watch for.

Real-world datasets, especially those from hardware systems, contain some *noise*—incorrect or misleading data caused by accidents and miscalculations. If these noisy examples are perfectly correlated with failing examples, the treatment may become overfitted. An overfitted model may come with a massive lift score, but it does not accurately reflect the general conditions of the search space. To avoid overfitting, learners need to adopt a threshold and reject all treatments that fall on the wrong side of this threshold. We define this threshold as the *minimum best support*.

Given the desired class, the best support is the ratio of the frequency of that class within the treatment subset to the frequency of that class in the overall dataset. To avoid overfitting, TAR3 rejects all treatments with best support lower than a user-defined minimum (usually 0.2). As a result, the only treatments returned by TAR3 will have both a high *lift* and a high *best support*. This is also the reason that TAR3 prefers smaller treatments. The fewer rules adopted, the more evidence that will exist supporting those rules.

TAR3's lift and support calculations can assess the effectiveness of a treatment, but they are not what generates the treatments themselves. A naive treatment learner might attempt to test all subsets of all ranges of all of the attributes. Because a dataset of size N has 2^N possible subsets, this type of brute force attempt is inefficient. The art of a good treatment learner is in finding good heuristics for generating candidate treatments.

The algorithm begins by discretizing every continuous attribute into smaller ranges by sorting their values and dividing them into a set of equally-sized bins. It then assumes the small-treatment effect; that is, it only builds treatments up to a user-defined size. Past research (Gundy-Burlet et al. 2007; Gundy-Burlet et al. 2009) has shown that this threshold should be no higher than four attributes. Note that this is not hard scientific fact, more of a *rule of thumb*—larger treatments are harder for humans to quickly comprehend.

TAR3 will only build treatments from the discretized ranges with a high heuristic value. It determines which ranges to use by first determining the lift score of each attribute's value ranges (that is, the score of the class distribution obtained by filtering for the data instances that contain a value in that particular range for that particular

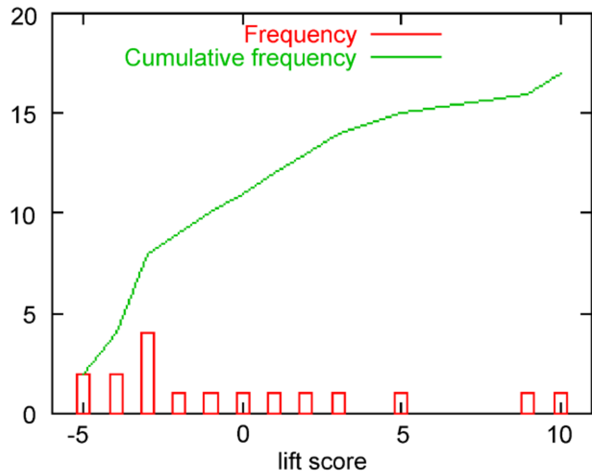

Fig. 4 Probability distribution of individual attribute scores

attribute). These individual scores are then sorted and converted into a cumulative probability distribution, as seen in Fig. 4. TAR3 randomly selects values from this distribution, meaning that low-scoring ranges are unlikely to be selected. To build a treatment, n (random from $1 \ldots$ max treatment size) ranges are selected and combined. These treatments are then scored and sorted. If no improvement is seen after a certain number of rounds, TAR3 terminates and returns the top treatments.

3.6 TAR4.1

TAR3, while effective at generating informative treatments, is not a very efficient algorithm. It stores all examples from the dataset in RAM and requires three scans of the data in order to discretize, build theories, and rank the generated treatments. The TAR4.1 treatment learner was designed to address these inefficiencies. Modeled after the SAWTOOTH (Orrego 2004) incremental Naive Bayes classifier, TAR4.1's scoring heuristic allows for an improved runtime, lower memory usage, and a better ability to scale to large datasets.

Naive Bayes classifiers offer a relationship between fragments of evidence E_i, a prior probability for a class $P(H)$, and a posteriori probability $P(H|E)$ defined by

$$P(H|E) = \prod_i P(E_i|H) \frac{P(H)}{P(E)}. \tag{4}$$

For numeric features, a features mean μ and standard deviation σ are used in a Gaussian probability function (Witten and Frank 2005):

$$f(x) = 1/(\sqrt{2\pi}\sigma)e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{5}$$

TAR4.1 still requires two passes through the data, for discretization and for building treatments. These two steps function in exactly the same manner as the corresponding steps in the TAR3 learner. TAR4.1, however, eliminates the final pass by

building a scoring cache during the BORE classification stage. As explained previously, examples are placed in a U-dimensional hypercube during classification, with one dimension for each utility. Each example $e \in E$ has a normalized distance $0 \leq D_i \leq 1$ from an *apex*, an area where the best examples reside. When BORE classifies examples into *best* and *rest*, that normalized distance is added as a score, called D_i (the Euclidean distance from 0), to the *down* table and a separate score, $1 - D_i$ (or, the distance from the best), is entered into the *up* table.

When treatments are scored by TAR4.1, the algorithm does a linear-time table lookup instead of scanning the entire dataset. Each range $R_j \in example_i$ adds scores $down_i$ and up_i to counters $F(R_j|rest)$ and $F(R_j|best)$. These counters are a summation of scores for a range R_j across the dataset, and represent how often data examples containing that range appear in the *best and rest*. These summations are then used to compute the following probability and likelihood equations:

$$P(best) = \frac{\sum_i up_i}{\sum_i up_i + \sum_i down_i}, \tag{6}$$

$$P(rest) = \frac{\sum_i down_i}{\sum_i up_i + \sum_i down_i}, \tag{7}$$

$$P(R_j|best) = \frac{F(R_j|best)}{\sum_i up_i}, \tag{8}$$

$$P(R_j|rest) = \frac{F(R_j|rest)}{\sum_i down_i}, \tag{9}$$

$$L(best|R_k \wedge R_l \wedge \cdots) = \prod_x P(R_x|best) * P(best), \tag{10}$$

$$L(rest|R_k \wedge R_l \wedge \cdots) = \prod_x P(R_x|rest) * P(rest). \tag{11}$$

TAR4.1 finds the *smallest* treatment T that *maximizes*

$$P(best|T) = \frac{L(best|T)^2}{L(best|T) + L(rest|T)}. \tag{12}$$

Note the squared term in the top of the equation, $L(best|T)^2$. The standard Naive Bayes design assumes independence between all attributes and keeps singleton counts. By not squaring that term, TAR4.1 adds redundant information, which alters the generated probabilities. In effect, it produced treatments with high scores, but without the *support* required by the TAR3 algorithm. By squaring that term, the likelihood that a range appears in an area of top scores, those treatments that lack support are pruned in favor of those that have both a good score and support.

4 Optimization techniques

Treatment learning is a relatively unexplored field, limiting the number of algorithms that TAR3 and TAR4.1 can be benchmarked against. However, the treatment prob-

lem is fundamentally an *optimization* (Dechter 2003) problem. The scoring methods used are simply mathematical objective functions. Therefore, it becomes possible to compare the treatment learning tools against the state-of-the-art techniques used to address optimization problems.

When presented as an optimization problem, the objective function F for TAR3 looks like:

$$\text{maximize } F(x) = \frac{\sum_c U_c f_c(x)}{\sum_c U_c F_c}, \tag{13}$$

where U_c, f_c and F_c are defined as for (3), and x is the attributes and ranges for a suggested treatment. Similarly, the objective function for TAR4.1 is defined as

$$\text{maximize } F(x) = \frac{L(apex|x)^2}{L(apex|x) + L(base|x)}, \tag{14}$$

where the likelihood functions L are the same as those given in (12). For both (13) and (14), x is the treatment—the attributes (discrete values) and their ranges (both continuous and discrete values)—and the size of x will vary based on the number of attributes that the algorithms choose for that particular treatment. Note that since the algorithms used by TAR3 and TAR4.1 do not use gradients there is no requirement for either of the above $F(x)$ to be smooth, and, in practice, both objective functions are highly non-smooth.

Numerous researchers have warned of the difficulties associated with comparing radically different algorithms (Uribe and Stickel 1994; Holzmann 1997; Gu et al. 1997). Both of these optimization techniques—Simulated Annealing and a Quasi-Newton method—were chosen because they are well-studied, powerful, and ubiquitous approaches that could easily use the same objective functions as the TAR3 and TAR4.1 treatment learners (thus rendering the results comparable). Furthermore, the algorithms that we use are *unconstrained* (constrained algorithms work towards a pre-determined number of possible solutions while unconstrained methods are allowed to adjust to the goal space). Simulated Annealing has been used to optimize similar design models in our own previous work (Gay et al. 2010).

Technically, any gradient-based approach (including the Quasi-Newton method used in our experiments) is at a disadvantage when addressing these problems; both the problem and the search space are both a mixture of discrete and continuous variables and the solution space is often locally discontinuous or highly non-linear. Still, researchers often choose to use gradient-based optimization for these problems because, while there is no expectation that they should work, they often (against expectation) do work. When gradient-based methods perform well, they usually do so at a lower computational cost than standard sampling methods (which require heuristics in order to avoid becoming stuck within local minima). Quasi-Newton methods make local approximations to the function, and as a result, they don't require that the function is locally smooth in order to make their next guess. Furthermore, as they are constantly rebuilding the Hessian matrix, they do not have the same tendency towards searching a subspace of a high-dimensional space that Conjugate Gradient methods tend to have. The particular Quasi-Newton method implemented here utilizes heuristics for jumping away from discontinuities in the solution when those discontinuities

are discovered. Thus, of all gradient-based methods, this is the most applicable for solving the optimization problem presented by these design simulations.

4.1 Simulated annealing

Simulated Annealing (SA) is a classic stochastic search algorithm. It was first described in 1953 (Metropolis et al. 1953) and refined in 1983 (Kirkpatrick et al. 1983). Fundamentally, SA is a hill climber—it starts in a random location and travels to higher-scoring locations in the immediate neighborhood. Standard hill climbers are prone to becoming stuck in local maxima. To avoid this, Simulated Annealing borrows a heuristic from its namesake, the metallurgy technique "annealing." In real-world annealing, a material is rapidly heated, then cooled. The heat causes the atoms in the material to rapidly jump around. However, as it cools, the atoms stabilize and solidify—they transition from large jumps to small wiggles. Similarly, Simulated Annealing will jump to sub-optimal solutions at a probability determined by the current state of the temperature function. At first, it will rapidly jump around the search space before finding stability.

The Simulated Annealing algorithm used in this experiment begins by making an initial guess. This guess is a twelve number vector that approximates a four variable treatment. The exact structure of this vector is {ATTRIBUTE, MIN, MAX} repeated four times. The algorithm then tries to solve objective functions corresponding to (13) and (14), except that, in this case x is limited by the algorithm to the structure of the initial guess. Note that Simulated Annealing only requests that the objective functions be smooth in a limited region near the final solution. We have no such guarantees for our problem, but in practice this is a workable approximation.

The algorithm will continue to operate until (a) the number of tries is exhausted, (b) improvement has not been seen for several rounds, or (c) a certain temperature threshold (a function of the time) is met. The current worth will then be compared to a minimum worth threshold and, if that value is not met, the algorithm will reset. The number of possible resets can be limited, but was not for this experiment.

4.2 Quasi-Newton optimization

The data mining problem posed in this study requests some subset of the variables and then requests a range for those variables. The best solution to this problem necessarily consists of a combination of integer and (likely nonlinear) continuous values. What's more, the search space is large: there are on the order of hundreds of attributes and on the order of thousands of individual runs. To complicate the problem further, the possible values for each attribute may themselves be continuous or discrete. As a final barrier to traditional optimization techniques, the input variables may not be directly correlated with the output class that is being chosen—either because the appropriate input variables were not selected out of the thousands or (sometimes) tens of thousands of possible input variables or because of some non-determinism in the solution. These factors cause the objective function to be highly non-smooth—there are likely to be many discontinuities and there are no guarantees that the neighborhood of the global solution will be discontinuity-free. Classically, this is the sort of problem

that must be solved by direct search optimization methods. However, on a practical basis, optimization techniques derived by assuming that the optimization function is smooth tend to be much more efficient, and often work better-than-expected by utilizing a wide array of numerical "tricks" that work to make the objective function act as if it were more smooth.

The particular optimization technique implemented for this work is a Quasi-Newton method with a BFGS update (Sims 1999). This $O(n^2)$ method builds up Hessian information as the iterations proceed, and the approximate Hessian is updated with a rank-one matrix. This constant building of the curvature information tends to avoid the subspace-searching problem that Conjugate Gradient methods can fall into for large problems like those solved within the RSE group. It also avoids the uncertainty that comes with parameter tuning for trust-region methods like Levenberg-Marquardt. However, like all descent methods, the algorithm assumes that the function it is optimizing is essentially continuous.

Mixed integer-nonlinear problems can be solved in several ways (Gill et al. 1981). The first such way is a combinatorial approach—solving all possible combinations and choosing the combination that gives the best answer to the objective function. For the types of problems solved in practice within the RSE group at NASA Ames, the combinatorial approach is computationally intractable. One recent example looked for the best 4 attribute treatment out of 128 attributes; the solution of this problem would have required almost 11 million separate optimizations. Even limiting the problem to choose the one best treatment would still require 128 different optimizations. Instead, as suggested by Gill et al. (1981), we introduced a new set of variables n_i into the TAR3 objective function where each variable was the percentage likelihood that the attribute i should be included in the treatment. We then introduced the term $\sum_i {n_i}^{2^{-1}}$ into the objective function to further increase the likelihood of a single discrete choice being made by the optimizer. The final form of this objective function is

$$\text{minimize } F(x) = \sum_i \frac{1}{n_i^2} \frac{-\sum_c U_c f_c(x, n_i, \delta_x(n_i))}{\sum_c U_c F_c}, \tag{15}$$

where U_c and F_c are defined as for (3). The vector x now takes the form $\{n_i$, MIN, MAX, $\ldots\}$ where all of the components are continuous and is 3 times the number of attributes N in size. The variable f_c is still the frequency of the class within each subset, but each sum in f_c is now modified by n_i, the percentage likelihood that the attribute i should be included in the treatment. The threshold function δ_x determines whether a particular attribute has enough of a percentage likelihood of being included in the treatment by comparing the value of n_i to a predetermined threshold. The initial values of n_i are set to the inverse of the number of attributes, $1/N$. As the optimizer shrinks some of the values of n_i to maximize the $\sum_i {n_i}^{2^{-1}}$ term, some of the values of n_i become less than $1/(N+1)$. When n_i crosses this threshold, the δ_x function removes the attribute from the rule. The optimizer must then choose between increasing the $\sum_i {n_i}^{2^{-1}}$ term by choosing discrete values (this will also increase the support term) at the cost of reducing the overall worth when attributes are dropped from the rule. Note that Quasi-Newton BFGS methods have some small advantage over some other gradient-based methods in this case because they are making local

smooth approximations to the curvature of the function. However, the problem itself, as mentioned before, is inherently non-smooth. In fact, in this case, the objective function can have cliffs even with respect to the continuous ranges because the input to the objective function, the data, is also inherently discrete. As a result, it is very likely that the optimizer will become stuck in local equilibria and that the final solution will be highly dependent on the initial conditions.

While it is possible that we could have improved the performance of any gradient-based method by recasting the objective function to one that was more smooth or (perhaps) by recasting the problem as a constrained minimization problem, no such solution presented itself after a good-faith effort to discover it. This is a common limitation of gradient-based methods. While this limitation exists, it does not prevent researchers from trying to use gradient-based optimization methods for non-smooth problems—when the problem is smooth enough and the initial guess is close enough to the global minimum there can be a significant performance increase over direct search methods like polytope or simulated annealing. Treatment learning is, in essence, a direct search method.

5 Related work

The work being performed here—choosing the inputs and ranges most likely to lead to some output—can be thought of as a type of sensitivity analysis known as Monte Carlo Filtering (Rose et al. 1991). The heavy lifting in most sensitivity analyses of this type is currently being done either with purely linear correlation between the output class and the inputs, a method known as regional sensitivity analysis (RSA) which relies heavily on standard statistical tests such as the Smirnov two-sample test, or it is being done using some sort of regression analysis in which the relationships between the inputs and the outputs are derived from the data (Oakley and O'Hagan 2004; Austin et al. 2007; Saltelli et al. 2008). Linear correlations fail in models which are not smooth. For large models, RSA tends to have a very low success rate (Spear et al. 1994). Regression analysis builds a polynomial relationship by finding the correlation coefficients between the inputs and outputs. These correlation coefficients are then used to solve the original question—which inputs and their ranges most affect the output—by looking at the magnitudes of the correlation coefficients across the entire range. Regression analysis is computationally expensive and tends to be limited to relatively small numbers of theoretically independent inputs. RSA also tends to assume that relationships between the inputs and outputs are smooth (Oakley and O'Hagan 2004; Saltelli et al. 2008).

The types of problems being solved in this paper are non-smooth and of high dimensionality. To overcome the complications involved in finding the correlation coefficients for this sort of problem, we choose in practice to ignore the correlation coefficients altogether and use machine learning techniques that sample the space and solve the original question directly. One example of another existing sensitivity analysis that uses machine learning is the identification of tool faults in the semiconductor industry. Intel uses a technique similar in spirit to the analysis used by NASA's Robust Software Engineering (RSE) group to find spatial fault patterns on silicon

wafers (Jing et al. 2007). While the overall goal and appearance of Intel's method is comparable to RSE's, the details for every step of the analysis are very different and they do not use treatment learning for their analysis (Torkkola and Tuv 2006; Tuv et al. 2006; Eruhimov et al. 2007). The fact that parametric testing is being used across widespread applications demonstrates its promise; the massive divergence in the individual components of the technique is evidence that there is still significant research to be done to streamline its use for real-world data.

Gay and Menzies recently conducted a similar treatment learning exercise on NASA Jet Propulsion Lab projects (Gay et al. 2010). These projects were encoded in the Defect Detection & Prevention format (Cornford et al. 2001; Feather et al. 2008), which is a compiled model representing the requirements of a module, the risks that could compromise those requirements, and mitigations that can allay these risks. Their candidate solution, KEYS2, is based on the theory that a small number of important ("key") variables control the overall search space. The algorithm generates a large population of treatments and uses a Bayesian ranking mechanism similar to that of the TAR4.1 algorithm (presented later in this paper) to score these treatments. Each round, the top-scoring treatments are used to fix model attributes to specific values. They benchmarked KEYS2 against Simulated Annealing, MaxWalkSat, and an A* search and found that their treatment learner proposed solutions that completed a higher number of requirements on a lower budget than the other optimization techniques. Additionally, KEYS2 executed the largest models in a fraction of the time that it took for other algorithms.

6 Experiment

Ten Monte Carlo Filtering analyses were run for three different datasets, using five different methods: TAR3, TAR4.1, Simulated Annealing with the TAR3 objective function, Simulated Annealing with the TAR4.1 objective function, and a Quasi-Newton BFGS method with a modified version of TAR3's scoring function, as shown in (15). As discussed in Sect. 2, two of the datasets come from actual analyses performed within the RSE group at NASA Ames. The data in these two sets were gathered during Monte Carlo runs using a high-fidelity physics simulation. One of these datasets represents reentry simulations while the other dataset represents launch abort simulations. The first dataset contains 191 runs worth of 52 different attributes. The second dataset contains 1000 runs worth of 249 attributes. The data from these two projects had complicated penalty functions based on all of the flight requirements—these requirements include metrics like bounds on miss distances, fuel consumption, and the stress on the parachutes. Data with the highest penalty function values are said to have 'failed' and data with the lowest penalty function values are considered to be the 'best' data. Note that, because of the complicated penalty function, the 'failed' data items are likely to have exceeded the allowed values on more than one requirement. An analysis like this gives an overall view of the safest flight conditions given all of the different possible individual mission-critical failures. While the RSE group will often go on to look at individual failure types, the purpose of this experiment was to search for the factors leading to any type of mission-critical failure.

To demonstrate the broad applicability of the technique, we also ran a Monte Carlo Filtering analysis on some data obtained during a bicycle ride. The software that generated the data gave a particularly noisy power measurement. The penalty function used in this dataset evaluated each point in real-time as an individual trial and penalized each run by the noise in the power measurement. The goal was to see which of the other measured parameters was most likely to correspond with the noisy power measurement. This dataset contained 4435 runs over 11 attributes.

7 Results

During each trial, several statistics were collected in order to assess the treatments output by that algorithm. Let $\{A, B, C, \text{and } D\}$ denote the true negatives, false negatives, false positives, and true positives. From these measures, we can compute certain standard formulas.

$$recall = probability\ of\ detection = \frac{D}{B+D} \tag{16}$$

$$probability\ of\ false\ alarm = \frac{C}{A+C} \tag{17}$$

$$precision = \frac{D}{D+C} \tag{18}$$

For recall and precision, higher values are better. For the probability of false alarm, lower values are desired. Those performance measures, along with the runtime (which should be minimized) were collected for each individual run of each algorithm, then the averages, medians, and standard deviations were saved for each trial. After ten repeats, those statistics were combined and used to create quartile charts. The results for each of our algorithms were combined and ranked using the Mann-Whitney rank-sum test (Mann and Whitney 1947).

These quartile charts, sorted by the Mann-Whitney ranks, can be seen in Figs. 5, 6, and 7. Note that SA-T3 and SA-T4 refer to the two variants of Simulated Annealing, using the TAR3 and TAR4.1 objective functions respectively. Where used, QN refers to the Quasi-Newton gradient-based method. Also note that only the median values from each individual run were used in the combined results. In each quartile chart, the horizontal lines show the 25 to 75 percentile range, and the black dot represents the median point. The ranks come from the Mann-Whitney rank-sum test at a 95% confidence level. Each rank, from one to five, is statistically different and better than the following rank. If two algorithms have the same rank, their results are not statistically different.

After looking at the results from all three datasets, a clear ranking emerges for each of the collected performance statistics (assessed by the Mann-Whitney test). These rankings are (from best to worst, with parentheses denoting a tie):

- Runtimes: TAR4.1, TAR3, QN, (SA-T4, SA-T3)
- Recall: (TAR4.1, QN), SA-T4, TAR3, SA-T3
- Probability of False Alarm: TAR3, SA-T3, TAR4.1, SA-T4, QN
- Precision: TAR3, SA-T3, TAR4.1, (SA-T4, QN)

Metric	Project 1			
Runtime (seconds)	Rank	Program	Median	
	1	TAR4.1	0.13	
	2	TAR3	0.31	
	3	QN	6	
	4	SA-T4	15	
	4	SA-T3	16	
Recall	Rank	Program	Median	Quartiles
	1	TAR4.1	59	
	1	QN	36	
	2	SA-T4	25	
	3	TAR3	22	
	4	SA-T3	20	
				0 50 100
P(False Alarm)	Rank	Program	Median	Quartiles
	1	TAR3	1	
	2	SA-T3	9	
	3	TAR4.1	25	
	4	QN	34	
	4	SA-T4	71	
				0 50 100
Precision	Rank	Program	Median	Quartiles
	1	TAR3	90	
	2	TAR4.1	44	
	2	SA-T3	42	
	3	QN	24	
	3	SA-T4	11	
				0 50 100

Fig. 5 Results on several criterion, sorted by Mann-Whitney rank, for the RSE Project 1. In each quartile chart, the horizontal lines (if any) show the 25 to 75 percentile range, and the black dot represents the median point. Quartiles are obtained from summarizing data over ten repeats. Row i is *ranked* higher than row $i-1$ if their value distributions are statistically different (Mann-Whitney 95% confidence level) and the median of row i is better than row $i+1$. For recall and precision, higher values are better. For the probability of false alarm, lower values are desired

While these rankings do not show a single "winner," they do present a clear victory for the two treatment learning techniques. Either TAR3 or TAR4.1 is ranked at the top in every category, and neither of them are ranked worst in any category. TAR4.1 ties with the Quasi-Newton method in the recall category when one considers the statistical ranking; however, TAR4.1 does tend to show a higher median value.

The Simulated Annealer acted in accordance with its objective function. When using the TAR3 objective function, it tends towards high precision and low recall. Likewise, when using the TAR4.1 objective function, Simulated Annealing returns treatments with higher recall and low precision. In both cases, it performed more

Metric	Project 2			
Runtime (seconds)	Rank	Program	Median	
	1	TAR4.1	0.1	
	2	TAR3	0.2	
	3	QN	3.3	
	4	SA-T3	2.5	
	5	SA-T4	3.1	
Recall	Rank	Program	Median	Quartiles (0, 50, 100)
	1	TAR4.1	48	
	1	QN	44	
	2	SA-T4	27	
	3	TAR3	23	
	4	SA-T3	19	
P(False Alarm)	Rank	Program	Median	Quartiles (0, 50, 100)
	1	SA-T3	3	
	2	TAR3	6	
	3	SA-T4	23	
	3	TAR4.1	28	
	4	QN	40	
Precision	Rank	Program	Median	Quartiles (0, 50, 100)
	1	TAR3	58	
	1	SA-T3	53	
	2	TAR4	37	
	3	SA-T4	25	
	3	QN	25	

Fig. 6 Results on several criterion, sorted by Mann-Whitney rank, for RSE Project 2. In each quartile chart, the horizontal lines (if any) show the 25 to 75 percentile range, and the black dot represents the median point. Quartiles are obtained by summarizing data over ten repeats. Row i is *ranked* higher than row $i-1$ if their value distributions are statistically different (Mann-Whitney 95% confidence level) and the median of row i is better than row $i+1$. For recall and precision, higher values are better. For the probability of false alarm, lower values are desired

weakly than its treatment learner counterpart. Both the weaker results and slower runtime can be explained by the very design of Simulated Annealing: because it makes a single initial guess and mutates it, it is unable to try as many combinations as TAR3 or TAR4.1. It must keep trying to make its guess better, and only resets after certain timers expire. It keeps resetting until a certain score threshold is met, which is why it is slower than the other algorithms. If its initial guess is particularly poor, it will never be able to mutate it into something that scores highly. Thus, it will reset until it is able to find a good mutation.

Quasi-Newton performs very well on recall, even tying with TAR4.1 in the rank-sum test. However, it also has the worst probability of selecting false positives. In

Metric	Bicycle			
Runtime (seconds)	Rank	Program	Median	
	1	TAR4.1	0.18	
	2	TAR3	0.23	
	3	QN	8.3	
	4	SA-T4	158.1	
	4	SA-T3	398.5	
Recall	Rank	Program	Median	Quartiles
	1	TAR4.1	40	
	1	QN	30	
	2	SA-T4	24	
	3	TAR3	21	
	4	SA-T3	19	
				0 50 100
P(False Alarm)	Rank	Program	Median	Quartiles
	1	TAR3	6	
	2	SA-T3	11	
	3	SA-T4	22	
	4	TAR4.1	31	
	5	QN	33	
				0 50 100
Precision	Rank	Program	Median	Quartiles
	1	TAR3	54	
	2	SA-T3	32	
	3	TAR4	30	
	4	SA-T4	23	
	4	QN	22	
				0 50 100

Fig. 7 Results on several criterion, sorted by Mann-Whitney rank, for the bicycle dataset. In each quartile chart, the horizontal lines (if any) show the 25 to 75 percentile range, and the black dot represents the median point. Quartiles represent data summarized over ten repeats. Row i is *ranked* higher than row $i-1$ if their value distributions are statistically different (Mann-Whitney 95% confidence level) and the median of row i is better than row $i+1$. For recall and precision, higher values are better. For the probability of false alarm, lower values are desired

fact, its false positive rate exceeds its true positive rate on the bicycle dataset. Quasi-Newton is extremely imprecise, it tries to suggest treatments that contain most of the data rather than making any attempt to fit the treatment to the data. As with Simulated Annealing, Quasi-Newton returns results that are highly dependent on the initial conditions. This is because the data has a tendency to be discrete, while Quasi-Newton assumes continuous conditions. In these situations, the algorithm is likely to become stuck in local minima.

Both Simulated Annealing and Quasi-Newton require favorable initial conditions. This weakness is not shared by the treatment learners because of their highly randomized nature. They do not make any single initial guess, and they do not try to

manipulate their findings. The use of stochastic search algorithms has been criticized because their results may not be optimal; they may miss potentially powerful treatments because they randomly skip around the space of possible solutions. However, the problem that we are trying to solve is inherently not smooth (much less not convex), which means that gradient-based optimization techniques are also likely to miss the optimal solution. This effect is somewhat mitigated by TAR3 and TAR4.1 because they form treatments from a cumulative probability distribution that favors high-scoring ranges.

8 Discussion

While the results show the advantage for using treatment learning algorithms for these kinds of problems, they do not answer *which* one to use. TAR3 and TAR4.1 excel in different areas, and there is a notable tradeoff between the two. This makes it difficult to clearly recommend one over the other.

TAR3 is extremely specific in its recommendations. It tends to produce treatments that maximize the *lift* calculation while just meeting the support requirement. The result of this are treatments with a low recall value and a very high level of precision. While the recall values are weak, reflecting the lower support, the false alarm rate is nonexistent. This alone may be a reason to favor TAR3's treatments. TAR3 will not give you all of the sources of failure, but it will suggest very few false positives.

TAR4.1, on the other hand, is prone to suggesting treatments with a very high level of support, leading to a higher probability of detection. The problem with TAR4.1's results is that its treatments are not well-fitted to the data, they do a poor job of filtering out noisy factors or unnecessary information. This results in a much higher false alarm rate then that seen in TAR3's predictions.

The results for both treatment learners when asked to solve the problems as designed by the RSE group, regardless of their respective strengths, tend to be low when compared to the results of standard data mining problems in the literature. Note that, in most cases, every single algorithm used in this experiment returned performance values below 50%. This effect is largely due to the type of problem being solved within the RSE group. In the experiments run in this paper, these treatment learners were being asked to look for *any* critical failure (not just *specific* types of critical failures). This has a tendency to blur the results, as the learners must correlate back to a wide range of inputs, perhaps with disjoint ranges, and there is no guarantee that key inputs for an individual type of failure are in the dataset.

To gain a clearer look at their potential performance, we ran one additional experiment, asking the treatment learning and optimization algorithms to look at a specific failure type in isolation for the second RSE project. In this case, we chose to look at the failures in which a critical slideslip angle limit was exceeded. Those results can be seen in Fig. 8. Both TAR3 and TAR4.1 are better able to fit their treatments to the specific problem, resulting in a much lower false alarm rate and almost 100% precision. Interestingly, TAR3 and TAR4.1 performed almost identically, with TAR4.1 maintaining a slightly higher recall and TAR3 a slightly higher precision. TAR3's recall rose significantly, from a 23% median to 36%, while TAR4.1's dropped roughly the same amount, from 48% to 39%.

Metric	Project 2 (error: angle of attack)			
	Rank	Program	Median	
Runtime (seconds)	1	TAR4.1	0.01	
	2	TAR3	0.02	
	3	SA-T3	2.6	
	3	QN	3.1	
	4	SA-T4	3.0	
	Rank	Program	Median	Quartiles (0, 50, 100)
Recall	1	TAR4.1	39	
	2	QN	34	
	3	TAR3	36	
	3	SA-T4	24	
	4	SA-T3	21	
	Rank	Program	Median	Quartiles (0, 50, 100)
P(False Alarm)	1	TAR3	0	
	1	SA-T3	0	
	2	TAR4.1	7	
	3	SA-T4	30	
	4	QN	43	
	Rank	Program	Median	Quartiles (0, 50, 100)
Precision	1	TAR3	100	
	2	TAR4.1	97	
	2	SA-T3	97	
	3	SA-T4	81	
	3	QN	82	

Fig. 8 Results on several criterion, sorted by Mann-Whitney rank, for a specific error type in the second RSE project. In each quartile chart, the horizontal lines (if any) show the 25 to 75 percentile range, and the black dot represents the median point. Quartiles are obtained by summarizing data over ten repeats. Row i is *ranked* higher than row $i-1$ if their value distributions are statistically different (Mann-Whitney 95% confidence level) and the median of row i is better than row $i+1$. For recall and precision, higher values are better. For the probability of false alarm, lower values are desired

This experiment in looking at a specific failure type is an even clearer example of why treatment learning techniques should be used. While there was an increase in precision across the board due to the more precise nature of the problem, the performance of the optimization methods was far below the treatment learners. When the TAR3 objective function was used by the simulated annealer, it performed similarly to the actual TAR3. However, its results were poorer and its runtime was slower. Simulated annealing with the TAR4.1 objective function and the Quasi-Newton method showed particularly poor results. For both of these algorithms, the false alarm rate was higher than the median detection rate.

Both the data mining and information retrieval fields have weighed in on the tradeoff between precision and recall on numerous occasions (Menzies et al. 2007; Cleland-Huang et al. 2006; Antoniol et al. 2002; Antoniol and Gueheneuc 2005; Marcus and Maletic 2003), never definitively preferring one over the other. For NASA use, both values are highly important. The RSE simulators allow for stochastic parameters, with the wind values being a classic example. Even on a day in which there is no wind, there is a chance for a gust. The model tries to mimic measured parameters for the time of year and day in the launch location. The learners used in these experiments only suggest treatments for parameters we can control or measure, but these uncontrollable stochastic parameters still exist (note, however, that these parameters are likely to be at least loosely correlated with parameters we can control and measure). As a result, the failure boundaries are not well-defined.

In the particular cases that the RSE group is trying to solve, recall equates to the percentage of failures contained within the predicted rule. Obviously, a user would like recall to be high—you want the produced rules to actually predict the failure. For example, if 95% of all parachute failures happen when the easterly winds exceed some parameter in combination with a center of gravity (cg) within some given range, then you would want to know that restricting the allowable wind velocities and cg on launch day will greatly decrease the odds of that kind of parachute failure (the next goal, at this point, would be to find a rule that eliminates the odds of the other 5% of the failures). However, most treatment-finding algorithms can trivially prevent 100% of launch failures simply by specifying that the launch should never happen at all. If the learner decides that the failures occur when the wind velocities are greater than zero, the produced treatment is essentially stating that no launch is safe.

This is why precision is important in addition to recall—high precision values imply that the treatment doesn't trivially satisfy the constraints. Furthermore, precision does more than just prevent trivial solutions; it gives the engineers definitive trade spaces in which to work. In our previous example, we prevented 95% of parachute failures simply by restricting wind velocities in combination with the cg. The rule could have prevented the same 95% of failures by just restricting the cg placement without considering the wind velocities, but would have done so with worse precision. If restricting the cg of the vehicle becomes too expensive, it may be easier to move the launch date and time to make sure that the wind velocities are particularly low on the day of launch.

Given the high importance of both precision and recall on NASA simulations, our recommendation would be to favor neither TAR3 *or* TAR4.1, but to run both and compare the treatments delivered. The runtime advantage that both algorithms have over standard optimization techniques allows for the use of both to quickly explore the treatment space.

9 External validity

These experiments were conducted at NASA Ames Research Center with assistance from NASA contractors and civil servants. Additionally, two of the primary sources of data were from large-scale NASA project simulations. Therefore, a possible threat

to validity exists from the data and environment used for this experiment. The external validity of NASA-based research has been debated by Menzies et al. (2007), Turhan et al. (2008). Basili et al. (2002) have argued that conclusions derived from NASA data are relevant to the overall software industry because NASA contractors are obliged to demonstrate an understanding and adherence to modern industrial best practices. These same contractors service numerous industries. For example, Rockwell-Collins builds systems for both defense contractors and civilian aerospace corporations.

However, the work of other authors is not enough to completely dispel the issue of external validity. This is one of the reasons that the bicycle dataset was included in this experiment. The data, recorded during the operation of a bicycle, was not collected using NASA hardware or at a NASA facility. Despite this separation, the same trends occurred and each of the algorithms performed at a similar efficiency. This replication of trends shows that treatment learning is not a task that has been tuned to NASA data; it, in fact, has applications for both large-scale and small-scale industrial testing.

10 Conclusions and future work

Building a large-scale industrial system is a difficult task. It can cost millions of dollars and require months to years of testing. Early simulation makes a large difference, cutting both the cost and time to market (Sendall and Kozacaynski 2003). However, this early simulation is of limited value if there is not a way to tell which specific factors led to system failures. Experts are expensive and their time is limited, they cannot waste hours sorting through gigabytes of simulation logs. They need a way to limit the number of possible combinations that need to be examined.

Treatment learning, formally a subset of minimal contrast-set learning, is one method of accomplishing this task. A treatment learner gathers evidence from labeled simulation instances and determines the smallest rule that, when imposed, makes it most likely that a specific outcome will occur.

Although the TAR3 learner has been used in prior publications, it has never been benchmarked against optimization techniques on real-world applications. The goal of this research is to comparatively assess two different treatment learning techniques (TAR3 and TAR4.1) against two standard optimization algorithms (a Quasi-Newton method and Simulated Annealing) on real-world industrial projects. Three sets of data were used, two from large NASA projects and one from the operation of a bicycle. Each algorithm was executed multiple times over each dataset and performance statistics were collected. The results show that treatment learning shows better performance when compared to standard optimization algorithms for these sorts of problems. Both TAR3 and TAR4.1 are orders of magnitude faster than standard techniques. TAR3 demonstrates the lowest false positive rate and highest precision, while TAR4.1 produces the highest recall. Thus demonstrating the superiority of treatment learning over standard optimization algorithms for such design improvement. As both precision and recall are important for such NASA simulations, we favor neither treatment learner; rather, we advocate the use of both TAR3 *and* TAR4.1 to provide a pool of design suggestions.

The immediate research direction for the treatment learners will center around improvements to the internal heuristics. One idea proposed has been to change the discretization technique. Currently, both TAR3 and TAR4.1 use a simple equal-bin scheme. This naive approach is likely to miss important curves in the data space. Experiments are being conducted with various alternative schemes, including recursive cliff-based methods (Fayyad and Irani 1993). Other planned improvements center around optimization of the source code. There are still numerous memory issues that should be addressed and the code should be re-engineered to follow the highest industrial programming standards.

Both of our treatment learners ignore the time-dependency of the recorded data. This is a potential weakness when looking at the failure of a complex system, where the exact cause of a failure may not always be present at the moment where the effect of that failure causes the system to cease functioning. A potential avenue for future research could include incorporating a Markov Model or a Linear Dynamical System into the data processing steps (Bishop 2007) and modifying TAR3 and TAR4.1 with the ability to use these models in their analysis. The treatment learners should consider extreme or mean values over some period of time, whether a particular system mode was ever entered into, and other key events. A goal for this analysis would be to find a way to use sequential data within the machine learning techniques in order to automatically identify interesting time-dependent factors.

Acknowledgements This research was conducted at West Virginia University and the Ames Research Center under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

References

Acevedo, A., Arnold, J., Othon, W., Berndt, J.: ANTARES: Spacecraft simulation for multiple user communities and facilities. In: AIAA Modeling and Simulation Technologies Conference and Exhibit, pp. 2007–6888 (2007)

Agrawal, R., Imeilinski, T., Swami, A.: Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIGMOD Conference, Washington, DC, USA (1993). Available from http://citeseer.nj.nec.com/agrawal93mining.html

Antoniol, G., Gueheneuc, Y.: Feature identification: a novel approach and a case study. In: ICSM 2005, pp. 357–366 (2005)

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng. **28**(10), 970–983 (2002)

Austin, P., Grootendorst, P., Anderson, G.: A comparison of the ability of different propensity score models to balance measured variables between treated and untreated subjects: a Monte Carlo study. Stat. Med. **26**, 734–753 (2007)

Basili, V., McGarry, F., Pajerski, R., Zelkowitz, M.: Lessons learned from 25 years of process improvement: the rise and fall of the NASA software engineering laboratory. In: Proceedings of the 24th International Conference on Software Engineering (ICSE) 2002, Orlando, Florida (2002). Available from http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/83.88.pdf

Bay, S.B., Pazzani, M.J.: Detecting change in categorical data: mining contrast sets. In: Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining (1999). Available from http://www.ics.uci.edu/pazzani/Publications/stucco.pdf

Bishop, C.: Pattern Recognition and Machine Learning. Springer, New York (2007)

Boehm, B., Papaccio, P.: Understanding and controlling software costs. IEEE Trans. Softw. Eng. **14**(10), 1462–1477 (1988)

Boetticher, G.: An assessment of metric contribution in the construction of a neural network-based effort estimator. In: Second International Workshop on Soft Computing Applied to Software Engineering, Enschade, NL (2001). Available from: http://nas.cl.uh.edu/boetticher/publications.html

Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and regression trees. Technical report, Wadsworth International, Monterey, CA (1984)

Cai, C.H., Fu, A.W.C., Cheng, C.H., Kwong, W.W.: Mining association rules with weighted items. In: Proceedings of International Database Engineering and Applications Symposium (IDEAS 98) (August 1998). Available from http://www.cse.cuhk.edu.hk/kdd/assoc_rule/paper.pdf

Cleland-Huang, J., Settimi, R., Zou, X., Solc, P.: The detection and classification of non-functional requirements with application to early aspects. In: RE 2006, pp. 36–45 (2006)

Cornford, S.L., Feather, M.S., Hicks, K.A.: DDP a tool for life-cycle risk management. In: IEEE Aerospace Conference, Big Sky, Montana, pp. 441–451 (March 2001)

Dechter, R.: Constraint Processing. Morgan Kaufmann, San Mateo (2003)

Eruhimov, V., Martyanov, V., Tuv, E.: Knowledge discovery in databases: PKDD 2007. In: Constructing High Dimensional Feature Space for Time Series Classification, pp. 414–421. Springer, Berlin (2007)

Fayyad, U., Irani, I.: Multi-interval discretization of continuous-valued attributes for classification learning. In: Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, pp. 1022–1027 (1993)

Feather, M., Cornford, S., Hicks, K., Kiper, J., Menzies, T.: Application of a broad-spectrum quantitative requirements model to early-lifecycle decision making. In: IEEE Software (2008). Available from http://menzies.us/pdf/08ddp.pdf

Fischer, B., Schumann, J.: Autobayes: a system for generating data analysis programs from statistical models. J. Funct. Program. **13**, 483–508 (2003)

Gay, G., Menzies, T., Jalali, O., Mundy, G., Gilkerson, B., Feather, M., Kiper, J.: Finding robust solutions in requirements models. Autom. Softw. Eng. **17**(1), 87–116 (2010)

Gigerenzer, G., Goldstein, D.G.: Reasoning the fast and frugal way: models of bounded rationality. Psychol. Rev. 650–669 (1996)

Gill, P.E., Murray, W., Wright, M.H.: Practical Optimization. Academic Press, San Diego (1981)

Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison–Wesley, Reading (1989)

Gu, J., Purdom, P., Franco, J., Wah, B.: Algorithms for the satisfiability (sat) problem: a survey. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 19–152. American Mathematical Society, Providence (1997)

Gundy-Burlet, K., Schumann, J., Barrett, T., Menzies, T.: Parametric analysis of ANTARES re-entry guidance algorithms using advanced test generation and data analysis. In: 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space (2007)

Gundy-Burlet, K., Schumann, J., Barrett, T., Menzies, T.: Parametric analysis of a hover test vehicle using advanced test generation and data analysis. In: AIAA Aerospace (2009)

Holte, R.C.: Very simple classification rules perform well on most commonly used datasets. Mach. Learn. **11**, 63 (1993)

Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)

Hu, Y.: Treatment learning: implementation and application. Master's thesis, Department of Electrical Engineering, University of British Columbia (2003)

Jing, H., George, R., Tuv, E.: Contributors to a signal from an artificial contrast. In: Informatics in Control, Automation and Robotics II, pp. 71–78. Springer, Berlin (2007)

Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **4598**, 671–680 (1983)

Kohavi, R., John, G.: Wrappers for feature subset selection. Artif. Intell. **97**(1–2), 273–324 (1997)

Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. **18**(1), 50–60 (1947). Available on-line at http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.aoms/1177730491

Marcus, A., Maletic, J.: Recovering documentation-to-source code traceability links using latent semantic indexing. In: Proceedings of the Twenty-Fifth International Conference on Software Engineering (2003)

Menzies, T., Hu, Y.: Data mining for very busy people. In: IEEE Computer (November 2003). Available from http://menzies.us/pdf/03tar2.pdf

Menzies, T., Sinsel, E.: Practical large scale what-if queries: case studies with software risk assessment. In: Proceedings ASE 2000 (2000). Available from http://menzies.us/pdf/00ase.pdf

Menzies, T., Dekhtyar, A., Distefano, J., Greenwald, J.: Problems with precision. IEEE Trans. Softw. Eng. (September 2007). Available from http://menzies.us/pdf/07precision.pdf
Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Trans. Soft. Eng. (January 2007). Available from http://menzies.us/pdf/06learnPredict.pdf
Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: Equation of state calculations by fast computing machines. J. Chem. Phys **21**, 1087–1092 (1953)
Oakley, J., O'Hagan, A.: Probabilistic sensitivity analysis of complex models: a Bayesian approach. J. R. Stat. Soc. B **66**(3), 751–769 (2004)
Orrego, A.S.: Sawtooth: Learning from huge amounts of data. Master's thesis, Computer Science, West Virginia University (2004)
Quinlan, R.: C4.5: Programs for Machine Learning. Morgan Kaufman, San Mateo (1992). ISBN: 1558602380
Rose, K., Smith, E., Gardner, R., Brenkert, A., Bartell, S.: Parameter sensitivities, Monte Carlo filtering, and model forecasting under uncertainty. J. Forecast. **10**, 117–133 (1991)
Saltelli, A., Chan, K., Scott, E.M.: Sensitivity Analysis. Wiley, New York (2000)
Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M., Tarantola, S.: Global Sensitivity Analysis: The Primer. Wiley, New York (2008)
Schumann, J., Gundy-Burlet, K., Pasareanu, C., Menzies, T., Barrett, T.: Tool support for parametric analysis of large software systems. In: Proc. Automated Software Engineering, 23rd IEEE/ACM International Conference (2008)
Schumann, J., Gundy-Burlet, K., Pasareanu, C., Menzies, T., Barrett, A.: Software V&V support by parametric analysis of large software simulation systems. In: 2009 IEEE Aerospace Conference (2009)
Sendall, S., Kozacaynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003)
Sims, C.: Matlab optimization software. QM&RBC Codes, Quantitative Macroeconomics & Real Business Cycles (March 1999)
Spear, R., Grieb, T., Shang, N.: Parameter uncertainty and interaction in complex environmental models. Water Resour. Res. **30**(11), 3159–3169 (1994)
Taylor, B.J., Darrah, M.A.: Rule extraction as a formal method for the verification and validation of neural networks. In: IJCNN '05: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, vol. 5, pp. 2915–2920 (2005)
Torkkola, K., Tuv, E.: Ensembles of regularized least squares classifiers for high-dimensional problems. In: Feature Extraction, pp. 297–313. Springer, Berlin (2006)
Towell, G., Shavlik, J.: Extracting refined rules from knowledge-based neural networks. Mach. Learn. **13**, 71–101 (1993)
Turhan, B., Menzies, T., Bener, A.B., Di Stefano, J.: On the relative value of cross-company and within-company data for defect prediction. In: Empirical Software Engineering (2009). Available from http://menzies.us/pdf/08ccwc.pdf
Tuv, E., Borisov, A., Torkkola, K.: Best subset feature selection for massive mixed-type problems. In: Intelligent Data Engineering and Automated Learning—IDEAL 2006, pp. 1048–1056. Springer, Berlin (2006)
Uribe, T., Stickel, M.: Ordered binary decision diagrams and the Davis-Putnam procedure. In: Proc. of the 1st International Conference on Constraints in Computational Logics, pp. 34–49. Springer, Berlin (1994)
Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann, San Mateo (1999)
Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques, 2nd edn. Morgan Kaufmann, San Mateo (2005)