

# Automated Computer System Diagnosis by Machine Learning Approaches \*

Yun Mao

Department of Computer and Information Science  
University of Pennsylvania  
Philadelphia PA 19104  
maoy@cis.upenn.edu

## Abstract

Computer systems are growing more and more complex, towards or even beyond the limit that humans can handle. The consequence of the trend is that systems become more fragile, harder to manage, and therefore have lower availability and usability.

This paper surveys the systems and their algorithms that can automatically learn the behaviors of those complex systems and diagnose the root-causes of failures without the detailed knowledge of the system structures. We propose a unified common framework for automated fault diagnosis. Under the same framework, we present four case studies in different application domains, including program debugging, configuration troubleshooting and Internet service diagnosis, using different machine learning algorithms, such as logistic regression, decision trees, Bayesian estimation, and tree-augmented naïve Bayesian networks. Their advantages and disadvantages are discussed and compared. Finally, we provide some thoughts on the possible future directions.

---

\*Last Update Date: 2005/08/04 03:21:13 CVS Revision: 1.24

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>System Failure Diagnosis</b>	<b>3</b>
2.1	Possible approaches . . . . .	4
2.2	Generic framework for automated diagnosis . . . . .	4
<b>3</b>	<b>Program bug isolation</b>	<b>5</b>
3.1	Overview . . . . .	5
3.2	Algorithm . . . . .	7
3.3	Evaluation . . . . .	9
<b>4</b>	<b>Desktop configuration troubleshooting</b>	<b>10</b>
4.1	Overview . . . . .	10
4.2	Architecture . . . . .	10
4.3	Algorithm . . . . .	11
4.4	Evaluation . . . . .	14
<b>5</b>	<b>Fault localization in Internet services</b>	<b>15</b>
5.1	Overview . . . . .	15
5.2	Algorithm . . . . .	15
5.3	Evaluation . . . . .	17
<b>6</b>	<b>Performance diagnosis by state monitoring</b>	<b>18</b>
6.1	Overview . . . . .	18
6.2	Architecture . . . . .	18
6.3	Algorithm . . . . .	19
6.4	Evaluation . . . . .	20
<b>7</b>	<b>Discussion and future direction</b>	<b>21</b>
7.1	Comparison . . . . .	21
7.2	Lessons and challenges . . . . .	22
7.2.1	Systems and networking . . . . .	23
7.2.2	Machine learning . . . . .	25
7.2.3	Database . . . . .	26
<b>8</b>	<b>Summary</b>	<b>26</b>

## 1 Introduction

Computer systems are growing more and more complicated, towards or even beyond the limit that humans can handle. For example, in Microsoft Windows XP, there are tens of millions of lines of code. More evidences of software complexity growth can also be found in the number of configuration options, programming APIs, software dependencies, etc. Software complexity challenges its availability and usability: complex systems are more prone to crash due to bugs; they are also harder to install, manage and optimize, in part because the configuration space increases exponentially. Studies have shown that technical support contributes 17% of the total cost of ownership of today’s desktop PC [38].

An ideal software system should have an *autonomic* option as mentioned in the autonomic computing grand challenge of IBM research [25] to help users to manage itself: it should detect failures when they occur; it should localize the root causes of the failures after the discovery; it should automatically fix the problems based on the causes; it should tune itself to the best performance according to different user requirements and adapt to dynamic environments.

Autonomic computing is a big, actively evolving topic. In this paper, we focus on the automated fault diagnosis problem: when the system faults are captured, how can the computer automatically locate the root causes? In particular, we investigate the statistical machine learning algorithms in fault localization by studying four cases of fault diagnosis systems in the fields of bug isolation [27], desktop configuration [36], and Internet services [9, 11].

The rest of the paper is organized as follows. In section 2 we define the system failure diagnosis problem, discuss the possible approaches, and propose a generic framework for automated statistical diagnosis. In the next four sections, we present the four case studies, show how they can fit in to the framework, and evaluate their pros and cons individually. Section 7 presents a comparison between the four systems, and shows the lessons and future directions we learn. Finally section 8 summarizes the paper.

## 2 System Failure Diagnosis

Consider a complex system with several layers of architectural abstractions. Each layer may consist of many components that are distributed, and interact with each other in an arbitrary order. Such vertical and horizontal diversities make the application level fault diagnosis very hard. It is not

practical to assume the designers have considered every single possible scenario in the real world, and write bug-free programs. Therefore, it is not hard to imagine that when some fault is detected at the application layer, the user is still unable to figure out what has gone wrong due to insufficient error messages. Or even worse, the computer may not be aware of the fault at all.

## 2.1 Possible approaches

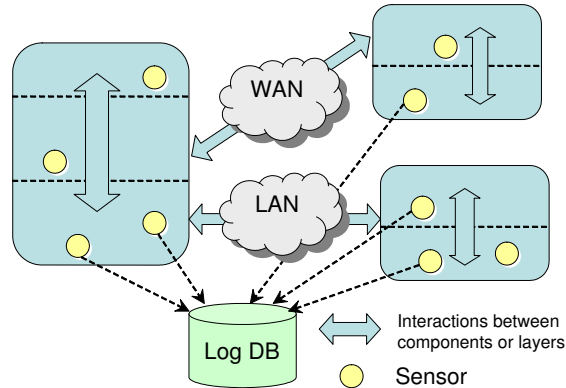
One approach to diagnose computer failures is the white-box approach [4, 6, 12, 31, 24, 3, 1, 14, 23, 34]. The structure of the system or the dependency model is known and is built in an expert system. Rules are specified by human experts to either identify proper system behavior or diagnose known failures. Optionally, when the fault root causes are found, pre-specified rules can also direct computers to take actions to fix the problems. The challenge for this approach is the correctness and the completeness of the rule specification at reasonable cost, in the face of dynamic environment and complex system structures.

Another approach is the black-box (or gray-box to some extent) approach. Faults are detected and causes are identified without detailed knowledge of the system structures and correct behavior. Models are induced from a statistical learning perspective. Such an approach is promising because it requires less human intervention and therefore is more *automatic*. We will focus on proposals based on the black-box approach for the majority of the paper in favor of the automation argument.

## 2.2 Generic framework for automated diagnosis

To diagnose failures, we deploy *sensors* inside the system at lower layers and smaller components. The framework is shown in Figure 1. The sensors monitor the system behavior and runtime properties, which could be arbitrary information that may help diagnose the application faults. For instance, they may include whether a component finishes with success or failure, configuration parameters possibly read from a registry database, or CPU/IO load information. Combining with the information of success/failure at the application layer, the user may be able to induce which sensors produce the most relevant data that may contributed to the failure.

We formalize the problem as follows. Let  $Y$  denote the random variable of the application level outcome status for a particular sample. Suppose there are  $n$  sensors, and let  $\vec{X} = (X_1, \dots, X_n)$  denote the random variables



**Figure 1.** The generic framework for automated diagnosis

of the states measured by the  $n$  sensors, the values of which are correlated with  $Y$  and may suggest the root cause of the problem. In usual fault diagnosis cases,  $Y$  is binary, which indicate either a successful run (0) or a failed one (1) for the application. In a performance diagnosis problem,  $Y$  might be a continuous variable dictating the performance result, such as throughput or latency, or it can be simplified to a binary proposition: either satisfied or unsatisfied by the user’s definition. We assume  $Y$  is binary for the rest of the paper for simplicity. The automated diagnosis problem is to reason which sensors are the most correlated ones to the application failures.

The answer to the question of how exactly to find the suspicious sensors is different for different statistical models, as we shall see in the following four cases studies later in this paper. However, it is common to fit the problem into machine learning by training a classifier. It is important to note that the goal is not to learn how to classify data points into classes, because it is usually not expensive to directly measure whether the application ends up with success or failure. However, the classifier also comes with a statistical model that describe the relations between the output of the sensors and the application output, which is essential in localizing the suspicious sensors.

### 3 Program bug isolation

#### 3.1 Overview

It is well-known that bugs exist in computer programs. It is not unusual for software vendors to ship final release versions to the end users with known bugs that appeared in their regression tests, not to mention the undiscovered

notation	meaning
$i$	the index variable of sample instances
$j$	the index variable of sensors
$Y$	the random variable indicating the application outcome status of a particular instance
$X_j$	the random variable of the output from the $j$ th sensor
$n$	the total number of sensors (features, attributes)
$\vec{X}$	the random variable vector of all sensors <i>i.e.</i> $\{X_1, \dots, X_n\}$
$y_i$	the application outcome status observed in the $i$ th sample
$x_{ij}$	the output of the $j$ th sensor observed in the $i$ th sample
$\vec{x}_i$	the output vector of all sensors observed in the $i$ th sample
$x_j$	the output of the $j$ th sensor observed in a particular sample
$\lambda$	a tunable parameter, depending on the algorithm details

**Table 1.** The summary of notation usage throughout the paper, unless specified otherwise

ones. Fixing all bugs is very expensive. Many of them only appear in special circumstances and are not easily reproducible. Some of them are non-deterministic, which may disappear or alter their behavior when they are studied (called *Heisenbug* [27]).

Most of existing debugging tools do not deal with those bugs well, and require a lot of human intelligence in setting breakpoints, speculating call stacks, etc. While static analysis [15] can alleviate the problem, many bugs are only observable at runtime.

Liblit *et al.* proposed *statistical bug isolation* [27] to address the problem. In their framework, they monitor and sample code-level behavior, and analyze the correlation with program failures. The code-level behavior is abstracted as *predicates* (*i.e.* assertions), which include signs of function return values, conditional branches, pointer arithmetic, etc. These predicates serve as the “wild guesses” of potential bug sources. Statistical analysis can discover the correlations between bugs and a small number of predicates, therefore it helps the programmers to eventually pinpoint the bugs.

Adding many predicates may slow down the program significantly. A key step forward to limiting the performance impact is to do sampling. Each assertion statement only gets executed in a low probability. The program may be executed by many users, and therefore the cost of evaluating the assertion is amortized. The authors did several pieces of work to optimize the performance of the sampling framework, including global count-down management, control flow based optimization, etc.

## 3.2 Algorithm

The bug isolation problem can be formalized according to the definition in section 2.2.  $y_i$  indicates whether the  $i$ -th run of the program is successful or not; the  $j$ -th sensor corresponds to the counter of the  $j$ -th code-level predicate, which yields  $x_{ij}$  as the number of times when the  $j$ -th predicate is executed *and* observed as `true` in the  $i$ -th run of the program.

The predicates also form into groups due to their mutual exclusiveness. For example, to check the return value of a function `foo()`, three predicates are deployed: `foo()>0`, `foo()<0` and `foo()==0`. Observed in any time, there is one and only one of them as `true`. Such predicates form into a triple group.

Because we knew most of the predicates were irrelevant, we can discard the obvious irrelevant ones by the following rules:

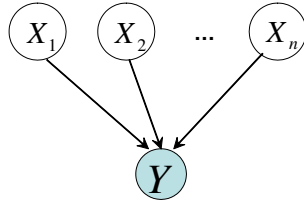
- universal falsehood: discard feature  $j$  if  $\forall i, x_{ij} = 0$ . These sensors probably represent predicates that are always false.
- lack of failing coverage: discard feature  $j_1, j_2$  and  $j_3$  if  $\forall i, x_{i,j_1} + x_{i,j_2} + x_{i,j_3} = 0$  or  $y_i = 0$  when  $x_{i,j_1}, x_{i,j_2}, x_{i,j_3}$  are a triple group. This implies that the assertion tests are not reached in failing runs, because one sensor in each triple must always be true for any sample.
- lack of failing example: discard feature  $j$  if  $\forall i, x_{ij} = 0$  or  $y_i = 0$ . This contains the super set of either the universal falsehood case or the lack of failing coverage case. However, note that it might not be as general as the other two strategies to use in other applications, because “failures always happen when a sensor yields 0” may be a piece of useful information. In bug isolation, discard the information is fine because other sensors in the same triple will still carry the information effectively.

If the bug is *deterministic*, meaning a predicate is always false when the execution is successful, we can further discard the counters that have a non-zero value on any successful run, and view the remaining predicates as the most suspicious ones that related to the failure. In [27], the authors demonstrated the method in debugging `ccrypt`.

When the bug is *non-deterministic*, the above screening method is not effective because with enough number of executions, no predicate will have non-zero value on all successful runs. Instead, we model the problem as a classification problem and a feature selection problem. The output of the

sensors (i.e. the counters of predicates) are viewed as features in machine learning. The goal is to train a statistical binary classifier that takes the input features, uses them as few as possible, and yields the correct program output status (success or failure). Logistic regression [22] is a method of learning such a binary classifier. The graphical model of logistic regression is shown in Figure 2. The output of the logistic function can be viewed as the probability of how likely the data point belongs to class 0 (success) or class 1 (failure). i.e.

$$P(Y = 1|\vec{x}) = \sigma(\beta_0 + \vec{\beta} \cdot \vec{x}) = \frac{1}{1 + e^{-\beta_0 - \vec{\beta} \cdot \vec{x}}} \quad (1)$$



**Figure 2.** The graphical model for logistical regression. The stochastic events are connected by the arrows that link the causes to the effects.

To learn the classifier, we can maximize the log likelihood of the training data set.

$$LL = \sum_{i=1}^n [y_i \log P(Y_i = 1|\vec{x}_i) + (1 - y_i) \log P(Y_i = 0|\vec{x}_i)] \quad (2)$$

Because most of the predicates are irrelevant and should have no impact on  $Y$ , we would like an additional constraint that forces most of the elements in  $\vec{\beta}$  to zero. This is accomplished by subtracting a penalty term  $\lambda \sum_{j=0}^n |\beta_j|$ , where  $\lambda$  is a tunable *regularization parameter* by the user. Finally the penalized log likelihood function we want to maximize is:

$$LL = \sum_{i=1}^n [y_i \log \frac{1}{1 + e^{-\beta_0 - \vec{\beta} \cdot \vec{x}_i}} + (1 - y_i) \log \frac{1}{1 + e^{\beta_0 + \vec{\beta} \cdot \vec{x}_i}}] + \lambda \sum_{j=0}^n |\beta_j| \quad (3)$$

There is no closed form solution for the optimization problem, but iterative algorithms such as gradient ascent or Newton’s method can be used to estimate  $\beta$ .

Once the model has been trained, the predicates with the largest  $\beta$  coefficients indicate the most suspicious lines of code to look for bugs. Zheng

*et al.* unified the deterministic and non-deterministic versions of the algorithm in [41].

### 3.3 Evaluation

The statistical debugging framework has two main contributions. First, the source-to-source compiler was built for automatically inserting sampled instrumentation to isolate bugs. The well designed compiler framework is easy to use and advantageous over ad-hoc, manual instrumentation methods. Because the predicates are only evaluated at a low probability, the performance impact is small in that it is practical to deploy in the production environment. The experiments showed that the performance overhead is less than 4% for 1/1000 sampling rate in `ccrypt`. Sampled assertions can also alleviate the privacy leakage problem in the instrumentation results collecting process, although it does not offer a complete privacy guarantee. Because the statistics are collected from the real user executions, the bugs that bother the users most may have higher chances to be isolated and fixed. Second, a statistical machine learning algorithm based on logistic regression is proposed to isolate non-deterministic bugs, the kind that are known to be very hard to debug. In the experiment, logistic regression was able to identify the closest line of code to the bug source from 30,150 “wild-guessed” assertions.

The bug isolation framework also has a couple of limitations. First, it does not work very well when there are multiple different bugs in the program. This is partially because different bugs may appear at different frequency and the less common bugs have less impact in the optimization process than the common ones. Second, the framework cannot collect the statistics about the sequential behavior of the program. For example, suppose a program usually calls function `foo()` after `bar()`, but when a bug occurs, `foo()` is never called. The statistical debugging framework is not yet able to capture such anomaly behavior. Third, in a large program, the redundancy between the predicates may also be high. This is known to be problematic to the logistic regression algorithm. Liblit *et al.* have been working on debugging large programs with redundant predicates and multiple bugs [28].

## 4 Desktop configuration troubleshooting

### 4.1 Overview

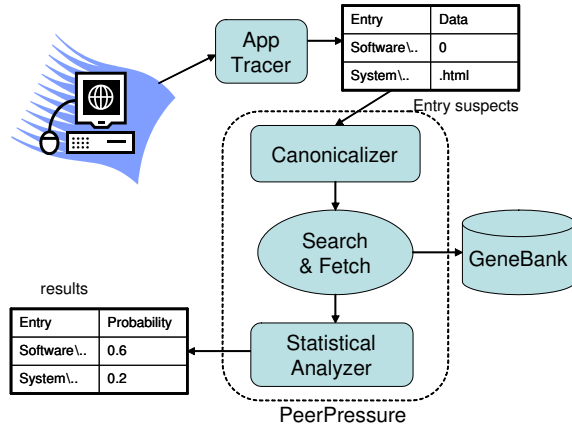
Reasonably well tested software may work just fine in normal cases. However, unexpected environment or non-standard configuration may let the program crash or stop functioning correctly. Many troubleshooting cases in technical support logs are due to misconfiguration. If the mis-configured parameters are persistent states, the errors are not transient in that rebooting the system would not solve the problem. Unfortunately, persistent configurations are quite common, such as the Windows registry and UNIX `.conf` files in `/etc` directory.

There are many ways to have misconfigurations besides a mistaken user’s manual change. For example, software bugs may corrupt an entry with invalid data; applying security patches may introduce incompatible configurations; system tuning software may radically change the configuration as well. As an application may use many configuration parameters, it is hard to manually check every one of them.

Wang *et al.* proposed the PeerPressure system [36] to do automatic troubleshooting on misconfiguration. They focused on the Windows platform and a particular type of configuration data, the Windows registry, although the methodology and lessons learned are applicable to other configuration problems. The idea is to compare the configuration entries of the sick machine to those of many other machines. With the belief that the majority of the machines are healthy, it is possible to automatically identify the suspicious entries on the sick machine.

### 4.2 Architecture

Figure 3 shows the architecture of PeerPressure. The first step is to identify registry entry suspects by using AppTracer on the sick machine to record all the entries that are used as input to the failed application. Note that if the application attempts to access an entry that do not exist yet, it is also recorded because a missing entry may very likely be the root failure cause. These entries are called suspects. Then the related entries are preprocessed by the canonicalizer, which replaces the user names and machine names to constant strings “USERNAME” and “MACHINENAME”, in order to reduce the difference between the suspects and the entries from the majority. Suppose there are  $n$  suspects in total and each entry suspect corresponds to a sensor. The output of the sensor is the value of the canonicalized entry. For example, we know  $y_0 = 1$  because we know the application has failed on the



**Figure 3.** The PeerPressure Architecture

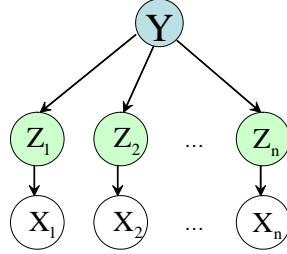
sick machine. We may also have  $x_{01} = \text{image/jpeg}$ ,  $x_{02} = \text{not exist}$ , where sensor 1 corresponds to registry key `.jpg/contentType`, and sensor 2 corresponds to registry key `.htc/contentType`. Now, instead of re-executing the application on many other machines, PeerPressure optimistically assumes that they will access the exact same set of the registry entries. Therefore, the troubleshooting process does not have to have access to the *live execution* environment of other machines. Only the registry database of other machines is necessary. The registry can be fetched from a centralized database, called “GeneBank”, or from some peer-to-peer troubleshooting community [35], depending on its availability. We assume GeneBank is used for simplicity reason. For each different machine  $i$  in the GeneBank, we can obtain  $\vec{x}_i = (x_{i1}, \dots, x_{in})$  as the corresponding values from the registry database. Note that different from the bug isolation example,  $y_i$ , whether the program output is success or failure is unknown. However, it is assumed that most of the machines in the GeneBank do not suffer the same configuration problems, *i.e.*  $y_i = 0, \forall i > 0$ . Finally, PeerPressure uses Bayesian estimation, as described in the next subsection, to estimate which entry suspect is the most suspicious to the application fault.

### 4.3 Algorithm

The learning problem is a little different from the bug isolation problem because we only have one negative example, the sick machine, in the training data set. In the sample set, the machines are *unlabeled*, although it is assumed that the majority of them are healthy. Therefore, supervised binary

classification learning algorithms cannot be directly applied.

To make the problem more tractable, in PeerPressure, the following assumptions are made: (1) there is only one sick entry among the suspects. (2) the value of an registry entry is independent of other entries given the machine’s health status.<sup>1</sup>



**Figure 4.** The graphical model of a naïve Bayes network in PeerPressure

Because of the independence assumption, the probability can be modeled by a naïve Bayes network. The graphical model is illustrated in Figure 4. We introduce a set of random variables  $\{Z_j\}_{j=1}^n$ , where  $Z_j$  suggests whether the suspect  $X_j$  is sick ( $Z_j = s$ ) or healthy ( $Z_j = h$ ).  $P(Z_j = s) + P(Z_j = h) = 1$ . Because we know that when the machine is healthy, all the entries are healthy, we have

$$P(Z_j = h|Y = 0) = 1$$

When the machine is sick, according to the assumption, the probability of each entry being sick is

$$P(Z_j = s|Y = 1) = \frac{1}{n},$$

and therefore

$$P(Z_j = h|Y = 1) = \frac{n-1}{n}$$

Now, if we know the probability distribution of  $P(X_j|Z_j)$ , the conditional probability tables (CPTs) of the network are all known, and we can use

---

<sup>1</sup>These two assumptions made in [36] are contradictory to each other, because given one entry known as the sick one, the probability of other entries being the sick ones is zero. Fortunately, we found that (later confirmed with the authors of the paper) the contradiction has little impact to the result. To make the assumptions sound, we can weaken the first one to “the prior probability of each entry being sick is  $\frac{1}{n}$ ” and the rest of the analysis and derivation will still hold without contradiction. Theoretically, the analysis under the new assumptions is even compatible with multiple sick entry cases. However, when multiple sick entries are sick, they usually are dependent on each other, and therefore the second assumption will be less likely to hold.

Bayesian inference to compute the probability of the  $j$ -th entry being sick given the machine is sick and the value of the  $j$ -th entry is  $x_{0j}$ :

$$\begin{aligned} \textit{Guilty}(X_j) &= P(Z_j = s | \vec{X} = \vec{x}_0, Y = 1) \\ &= P(Z_j = s | X_j = x_{0j}, Y = 1) \end{aligned}$$

We can compute the *Guilty* function for each suspect, and pick the one that has the largest guilty value. To simplify the notation, when the context is clear, we focus on one suspect and drop all the sensor indices:  $x_0$  is the registry entry value at the sick machine;  $n$  is the total number of suspicious registry keys;  $k$  is the number of sample machines in the GeneBank;  $c$  is the number of possible sample values for the suspect;  $m$  is the number of samples in  $\{x_i\}_{i=1}^k$  that match the suspect's value  $x_0$ ;  $Z$  is the random variable suggesting whether the suspect is sick ( $Z = s$ ) or healthy ( $Z = h$ ).  $P(X = x_0)$ ,  $P(Z = s)$  and  $P(Z = h)$  can be abbreviated to  $P(x_0)$ ,  $P(s)$  and  $P(h)$  respectively. The *Guilty* function after notation simplification is:

$$\textit{Guilty}(x_0) = P(s|x_0, Y = 1) \quad (4)$$

$$= \frac{P(x_0|s)P(s|Y = 1)}{P(x_0|s)P(s|Y = 1) + P(x_0|h)P(h|Y = 1)} \quad (5)$$

The only items that are unknown in Eq. (5) are  $P(x_0|s)$  and  $P(x_0|h)$ , *i.e.* the distribution of  $P(X|Z)$ . To model  $P(X|h)$ , we know that if a value is healthy, the probability it occurs should conform to the observation of the sample set. We can assume that  $P(X|h)$  is multinomial over all  $c$  possible values,  $v_1, v_2, \dots, v_c$ . A multinomial distribution is a probability distribution that is dictated by a multi-sided die, where side  $j$  has label  $v_j$ . The probability of throwing the die and getting value  $v_j$  is  $p_j$ . Because one side is always up when a die is thrown,  $\sum_{j=1}^c p_j = 1$ . We assume that the prior distribution of  $\{p_j\}$  is a Dirichlet distribution [19] with parameter  $\lambda$ . Then after observing the samples  $\{x_1, \dots, x_k\}$ , we can use Bayesian estimation to calculate the *posterior probability*:

$$P(x_0|h) = \frac{m + \lambda}{k + c\lambda} \quad (6)$$

Because we know little about what the sick entry may look like due to the lack of sick machine training data set, we assume each possible value of the registry entry has an equal prior probability<sup>2</sup>:

$$P(x_0|s) = \frac{1}{c} \quad (7)$$

---

<sup>2</sup>If the sick machine training data set is available, we can also use the Dirichlet-multinomial distribution to model and estimate the posterior probability  $P(X|s)$ .

Now putting them all together in Eq.(5), we have

$$Guilty(x_0) = P(s|x_0, Y = 1) = \frac{k + c\lambda}{k + c\lambda n + cm(n - 1)} \quad (8)$$

#### 4.4 Evaluation

PeerPressure is a black-box approach to diagnosis configuration problem of desktop applications. Comparing to previous work [37], its main contribution is to eliminate the manual steps in the troubleshooting process of both identifying healthy machines and specifying correct registry values. This new feature automates the troubleshooting process and makes the system easy to use for ordinary users. In [36], the authors showed 20 real-world troubleshooting cases that PeerPressure was evaluated against. The result was promising: PeerPressure was exactly accurate in 12 cases and quite effective in 19 out of 20 cases, even when the number of the suspects was as large as 26,308 in some cases.

PeerPressure makes the following assumptions to keep itself effective:

- The sick machine is not highly customized by the user. A highly customized machine has distinct registry values in many entries. The canonicalization procedure cannot cope with those entries yet. Therefore, more false positives will be reported by PeerPressure. Sometimes, the machine itself customizes entries, which makes such case more common. The network configuration is such an example.
- The unsuccessful execution is not caused by another application’s misconfiguration. For example, a Web browser may fail to open an image because a mis-configured firewall rule or a VPN setting. AppTracer will not capture such configurations, and therefore cannot diagnose it.
- The user knows how to reproduce the problem. In other words, the trouble is caused by a deterministic bug. A non-deterministic bug imposes challenges to both AppTracer and the statistical estimation algorithm, because the proposed models are not directly applicable. Fortunately, non-deterministic bugs caused by configuration problems are rare.
- The registry entries are conditionally independent. Aside from the contradiction discussed in the previous subsection, the entry independence assumption is unlikely to hold when two or more entries are sick. It is unclear how to change to Bayesian estimation model accordingly to model dependent registry entries.

## 5 Fault localization in Internet services

### 5.1 Overview

Failures not only happen in isolated programs or desktop computers, but also occur in distributed, component-based systems, such as Internet services. In such systems, a service request from the user is served by many components, presumably distributed among computer clusters or farms. For example, a transient network failure may make the application server unreachable, an upgrade of the database software may conflict with the configuration of the application server, etc. As the systems grow in terms of complexity, it is inevitable to avoid all kinds of errors or even to manually examine every single component.

To this end, Chen *et al.* proposed an automated approach to diagnose failures in such systems using decision trees [9]. For each request, along with the result (success or failure), more detailed information are recorded as well, such as the software version number, host name, and database name. The logging system is called Centralized Application Logging (CAL) framework, which is a central repository of application-level logs. The log information of each request is asynchronously written to the Harvester cluster over persistent TCP connections. The goal is to induce the root causes of the known failures based on the request log data. Chen *et al.* have implemented such a system in eBay's Internet service.

### 5.2 Algorithm

The Internet service diagnosis problem can be formalized according to the definition in section 2.2.  $y_i$  indicates whether the  $i$ th request is successful (0) or not (1); the  $j$ -th sensor corresponds to the  $j$ -th attribute or feature in the application log. In the implemented system, there are 6 basic sensors: the request type, request name, pool name, hostname, software version, and the status of each request. There are also 40 binary sensors for database access, which correspond to 40 distinct databases in the whole system. The sensor output of a request is true if and only if the request accesses the corresponding database.

A decision tree in this problem is a binary classifier. It takes input from the sensors described above, reaches its decision (a successful request or a failed request) by performing a sequence of tests. Each internal node in the tree corresponds to a test of the features, and branches from the node are labeled with the possible values of the test. Each leaf node dictates a final decision. Such a tree presentation is very natural to humans, and therefore

is easy to interpret *why* the result is as predicted. The interpretability is essential in diagnosis because the ultimate goal is to know which feature corresponds to the failure and not to know whether the request has succeeded.

The decision tree learning algorithms are well studied in the literature. In [9], the C4.5 algorithm [33] is adopted. The key step of learning a decision tree is to decide, at a given node, which feature and testing value to choose to split the data. Splits are chosen to maximize a *Gain* function, until no further split is possible or necessary. In C4.5, the *Gain* function at node  $t$  is:

$$Gain(x_j, t) = H(t) - H(x_j, t),$$

where  $H(t)$  is the binary entropy at node  $t$  and  $H(x_j, t)$  is the sum of entropy of children nodes after splitting based on the  $j$ -th feature  $X_j = x_j$ .  $H(t) = -\sum_{\vec{x}_i \in t} P(\vec{x}_i) \log P(\vec{x}_i)$ . The algorithm does a full search to find the maxima, which means, at each node, the *Gain* function will be evaluated for all left features, in each of which, for all possible values.

Instead of the C4.5 algorithm, in the actually deployed eBay system, a revised algorithm called *MinEntropy* is adopted. Its *Gain* function is based on the entropy of a different random variable, and only follows one of two possible paths, which is considered more “suspicious”. *MinEntropy* also stops splitting when the *Gain* function falls below a certain threshold, instead of pruning as in C4.5. Suppose feature  $X_j$  has  $d$  possible values. We examine the probability that a failed request at a node  $t$  takes on a particular value,

$$P(X_j = z|t) = \frac{|\{\vec{x}_i\}|}{|\{\vec{x}_k\}|}, \forall i, k \text{ such that } x_{ij} = z, y_i = 1, y_k = 1$$

and

$$Gain(X_j) = -H(P(X_j|t))$$

Intuitively, *MinEntropy* splits the tree based on the feature with the lowest entropy, and follows the child node  $j$  with the highest failure probability.

As noted earlier, learning decision trees is not the final objective. The final goal is to select the important features that correlate with the largest number of failures. Several post processing heuristics in [9] are proposed:

1. *noise filtering*: Leaves in the decision tree containing less than  $\lambda\%$  of the total number of failures are ignored. This is based on the assumption that only a few independent sources of error are expected, and each of them accounts for a large fraction of the total number of failures.

2. *node merging*: Before reporting the final diagnosis, nodes on a path are merged by eliminating ancestor nodes that are subsumed by successor nodes. i.e. if  $\forall \text{request } i, P(x_{ij}|x_{ik}) = 1$  and node  $X_j = x_j$  is the ancestor of node  $X_k = x_k$ , then  $X_j$  is merged into  $X_k$ . One example is that machine A only runs software version B. If `version=B` is the ancestor of `machine=A`, then it can be removed without losing useful diagnosis information.

Finally, the predicted causes of a request failure are the nodes along with the decision path in the decision tree, ranked by the failure counts.

### 5.3 Evaluation

Among four cases studies in this paper, this diagnosis system by decision trees developed was the only one that was running on a production platform, the eBay Web site. The advantage of the decision tree algorithms is that their results are straightforward for human to understand: the decision trees are just like a bunch of if-then-else statements. The proposed post-processing rules are also keys in addition to the learning algorithms. Applying node merging and noise filtering made false positive rate dropped from 82% to 24% in the test.

However, the decision tree learning approach also has some drawbacks. First, although the algorithms themselves are simple, they are not efficient in terms of running time. The deployed system adopted the MinEntropy algorithm instead of C4.5, to trade accuracy for efficiency. In most of the accuracy experiments conducted in [9], only C4.5 was evaluated and studied. Second, although each data set studied contained about 200,000 requests, only 0.001%-0.002% of them had an outcome status of failure. The number of sensors captured was also small: 6 in the basic data sets and 46 in the complete data sets, comparing to thousands or tens of thousands features in the PeerPressure and bug isolation project. It is unclear whether the decision tree algorithms can scale up to support more sensors, which may or may not be redundant, and more failure instances, where data could be more noisy and might add more uncertainty in both the decision tree learning and post-processing procedures. The time complexity is also a concern for scalability.

## 6 Performance diagnosis by state monitoring

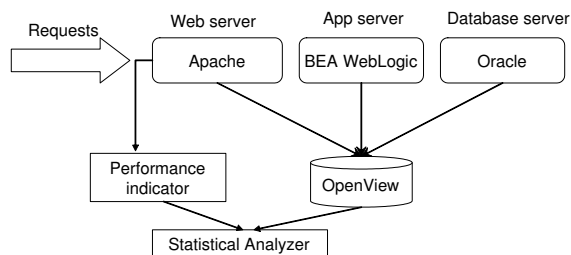
### 6.1 Overview

A complex system may not only fail mysteriously at a given task, but also encounter performance problems. For example, a typical case is the violation of Service Level Objectives (SLO) in Internet services. The SLO is a performance agreement between the service provider and its customers. Therefore, instead of observing failed requests, observing average-case response time above a certain threshold is also considered as problematic. Diagnosing performance problems is hard in that there are so many factors to influence the performance: workload, CPU, disk, software configurations, network cross traffic, etc.

A common approach is to build an expert system that incorporates a priori model of structures and behaviors, and use rules to guide performance diagnosis [3, 1, 14]. However, it is very expensive to build the models and the rules, and more expensive to maintain them manually when the system is evolving.

Cohen *et al.* propose another approach to address the diagnosis problem. Instead of manually specified and managed, the models are learned by statistical machine learning algorithms in an automated fashion [11]. The models are trained by monitoring low level system states of the entire system and the corresponding performance, *i.e.* the SLOs. If the SLO violations are observed, the learned models can be used to infer which system metrics are most correlate to the violations.

### 6.2 Architecture



**Figure 5.** The architecture of automated performance model induction based on system states monitoring in a 3-tier Internet service system

The learning and monitoring architecture for the 3-tier Internet service system is illustrated in Figure 5. The 3-tier system is distributed into three

components: Web server, application server and database server. They are running Apache, BEA WebLogic and Oracle respectively on the OS of Windows 2000 Server. At each tier, sensors are deployed to measure performance related system states, such as CPU time spent in user node, variance of user CPU time, physical disk reads, the number of packets that the network interface has sent/received, etc. These states are measured, and sent to a centralized management software HP OpenView [23]. On the other hand, the logs from the Web server are retrieved by a performance indicator, which analyzes the log and estimate average request response time, and decides whether there are violations based on the given SLO definition. The statistical analyzer will combine the information from both the performance indicator and the metric database (OpenView), to learn a model that describes the relationship between various metrics and the SLO. After the model is learned, it can be used to do diagnosis inference for a particular instance of SLO violation.

### 6.3 Algorithm

The performance diagnosis problem can be formalized according to the definition in section 2.2.  $y_i$  indicates whether the average requests in the  $i$ -th time period are SLO compliant (0) or not (1);  $x_{ij}$  is the output of  $j$ -th sensor measured in the  $i$ th period. In the test, there are 124 sensors in total and the sampling interval is 15 seconds.

With  $k$  training samples  $\{\vec{x}_i, y_i\}_{i=1}^k$ , we can learn the model by training a classifier that estimate  $P(Y|\vec{X} = \vec{x})$ . If  $P(Y = 1|\vec{x}) > P(Y = 0|\vec{x})$  then it is likely that a SLO violation has happened.

In [11], a Tree-Augmented naïve Bayesian Network (TAN) model [17] is used. TAN is a Bayesian network model with special network structures. Similar to the naïve Bayesian network, the root node  $Y$  is the parent of all other nodes. Different from it, a node  $X_j$  can have another parent node  $X_{p_j}$  in the model. Comparing to the naïve Bayes networks, TANs do not assume conditional independence between the input features given  $Y$ , and therefore are more suitable to represent complex relationships. Many system metrics measured in the system are correlated, and thus the TAN model is more appropriate. It also inherits the benefit of interpretability of Bayesian networks. Once the TAN model is learned, meaning both the Bayesian network structure and its conditional probability distributions (CPD) are known, we can use Bayesian inference to estimate arbitrary probability density. Interpretability is required because the ultimate goal is *not* to answer whether requests violate the SLO because it is already cheap to know the exact an-

swer by examining the logs, but to answer what has caused the performance degradation.

The detailed algorithms of training the TAN model can be found in [17] and [11]. The probability at each node is modeled as conditional Gaussian distribution. Only a small subset of the features are selected in the model. Feature selection is done by a greedy strategy: at each step, select the metric that is not yet selected and yields maximal improvement in the classification accuracy of the resulting model over the sample data.

After training the model, we can classify a new point  $\vec{x}$  by evaluating:

$$\log \frac{P(Y = 1|\vec{x})}{P(Y = 0|\vec{x})} = \sum_j \log \left[ \frac{P(x_j|x_{p_j}, Y = 1)}{P(x_j|x_{p_j}, Y = 0)} \right] + \log \frac{P(Y = 1)}{P(Y = 0)} > 0 \quad (9)$$

If Eq. 9 holds, then it indicates a SLO violation, or a compliance otherwise. Note that the “contribution” of a particular metric  $x_j$  is

$$Guilty(x_j) = \log \frac{P(x_j|x_{p_j}, Y = 1)}{P(x_j|x_{p_j}, Y = 0)}.$$

A positive value suggests that  $x_j$  is “guilty” for the SLO violation. For a given violation instance, more than one metric could be guilty, the larger  $Guilty(x_j)$  is, the more likely that metric  $x_j$  reveals a true cause of the performance problem.

## 6.4 Evaluation

The paper by Cohen *et al.* was the first paper that trying to *automatically* learn statistical models that correlate system metrics to application layer performance. A key step forward is to remove the need of knowledge from human experts to construct the models. The proposed TAN model is more sophisticated than the naïve Bayesian network model, but is still relatively cheap to do learning and inference. More importantly, TAN models inherit the properties of Bayesian network models: interpretability and modifiability. Although the modifiability of TAN models is not yet explored in the paper, the interpretability is essential to infer the correlation between system metrics and SLO violations.

However, the evaluation methodology in [11] were mostly focused on the accuracy of the classifier. The metric used was *balanced accuracy* (BA), which averages the probability of correctly identifying compliance with the probability of detecting a violation. While it is good to know whether the classifier is accurate, such a evaluation metric does not directly answer the

ultimate question of whether the model could diagnose the true performance bottlenecks in a 3-tier Internet service system.

Because the SLO violations are defined by the averaged response time over requests in an interval, the model is not able to reason against individual requests. For example, if certain type of requests are rare, but have very bad performance, their cost reflecting on the system states may be amortized by other requests and therefore they are hard to detect.

## 7 Discussion and future direction

### 7.1 Comparison

Despite the high level similarity of the four case studies, they indeed target at different practical problems. Hence, it is impossible to give a quantitative comparison of their experimental results, although such comparisons against the same problem can be found elsewhere [32]. Here we present a qualitative comparison of the four machine learning algorithms, which may also serve as applicability guidelines in other application domains.

- *failed samples in the training data?* Logistic regression, decision trees and TAN require a reasonable amount of failed samples in the training data set to learn the classifier. On the other hand, the Bayesian estimation technique used in PeerPressure only needs successful examples to learn the stochastic model.
- *Gaussian distribution or not?* If the data is continuous, we may need to check if the distribution conforms to Gaussian distribution. The TAN models work with the Gaussian assumption well. If it is not Gaussian, logistic regression usually has better accuracy.
- *human interpretability?* If the models learned from statistical data often need to be examined by human operators who has little background knowledge in statistical analysis, decision tree algorithms produce understandable models that can be easily visualized. The trees are a series of if-then-else statements which is straightforward to understand for humans. TAN and logistic regression are interpretable, but to a lesser degree.
- *time complexity?* Learning the TAN models is probably the most efficient algorithm if the data obeys Gaussian distribution. Features are chosen based on a greedy algorithm which finishes quite fast. Logistic

	logistic regression	Bayesian estimation	decision trees	TAN
time complexity	B	A	C	A-
human interpretability	B	B	A	B
lack of failed samples?	N/A	B	N/A	N/A
dependent features?	A	N/A	B	A
handle non-Gaussian?	B	C	C	C

**Table 2.** A summary of the qualitative comparison among the four machine learning algorithms in section 7.1.

regression requires non-linear optimization and has no closed form solution. The iterative methods such as stochastic gradient ascent and Newton’s method usually take long time to converge to global minima. Finally, decision tree algorithms are also not efficient if the number of the features and the number of training examples are large. The worst case time complexity is exponential in tree height [30].

- *structure-learning or parameter-learning?* In logistic regression and Bayesian estimation, the structures of the graphical models are determined. Only the parameters of the models need to be learned. In contrast, in TAN and decision trees, both the structures of the networks and their parameters need to be learned. Structure-learning is considered to be a harder problem than parameter-learning in general because of the learning complexity. TAN models and decision trees deal with the complexity by using some sort of greedy strategies.

	number of sensors (features)	number of samples
logistic regression	30150	4390
Bayesian estimation	8-26308	5-87
decision trees	6-40	200000
TAN	124	8640

**Table 3.** A comparison among the four machine learning algorithms in which different number of features (i.e. dimensionality) and learning samples are used.

## 7.2 Lessons and challenges

The diagnosable systems are not limited to the four cases we studied. Other examples range from BGP fault diagnosis in Internet routing [26], content

distribution networks, to sensor networks and beyond. However, we do learn several interesting lessons from the representatives and see some future challenges that are still not well addressed.

### 7.2.1 Systems and networking

In terms of systems and networking research field, when we are aware that the complexity is inevitable in large-scale computer systems and faults and bugs will happen no matter how careful we are to design and implement the systems, a valid question to ask is: what are the design principles to follow, with runtime fault diagnosis support in mind?

- *conceptual simplicity as the rule of thumb*: Automated failure diagnosis is not a panacea for software complexity. No failure diagnosis algorithm is able to identify and localize every fault. As the system scales up, the degree of complexity inevitably increases. Therefore, we cannot afford more complexity by implementing unnecessary functionality or maintaining hardly used backward compatibility. The complexity factor must be considered at the design stage. For example, if the requirement of a distributed system is only to scale up to a limited number of nodes, the exotic peer-to-peer routing algorithms might not be chosen over the centralized algorithms, even though the P2P algorithms are known to scale to a larger scale. In fact, simple system design often leads to better effectiveness of fault diagnosis algorithms.
- *path visibility*: An assumption in the our automated diagnosis framework is that we know how to match low level sensor outputs to high level application results. This is not trivial in concurrent, layered, or networked systems. For example, in a 3-tier Web service system, it is easy to deploy a sensor monitoring database queries. However, knowing the outcome of a database query (low level information) does not necessarily imply to know which particular Web request (high level goal) has generated the query, because the causality path is not explicitly maintained. Several diagnosis systems try to indirectly infer such paths by either implementing tracing mechanisms based on application specific knowledge [20, 5], or analyzing message-level network traces between distributed black-boxes [2]. Unfortunately, these indirect methods come with cost and accuracy limitations. If the system designers have the path visibility in mind [10], the systems would be more diagnosis-friendly. The cost of adding such support might be small. The system only needs to maintain a unique identifier at the

highest level, and make sure that each layer always passes the id to the lower layer or other components, and make it available to the sensors.

- *cross layer modeling*: Large-scale networked systems have multiple layers, each of which represents a different level of abstraction. To make the diagnosis process effective, sensors should be deployed at multiple layers and should be modeled and analyzed together. Such benefit may be amplified the most in a networked system. For example, to debug a file transfer program in wireless networks, we might find that the SLO performance violation has strong correlation to both retransmission timeout events at the transport layer and the packet corruption event at the physical layer, which suggests that the cause is likely to be a high burst error rate (BER) instead of congested network links. Modeling cross layer information may impose a challenge for the machine learning algorithms because presumably multi-layered outputs are more redundant, noisy, and complicated to model.
- *OS support for standardized error/event report*: The current sensor selection and deployment are still rudimentary. In bug isolation [27], even though the assertion sensors are widely deployed, their types are restricted to signs of function return values and pointer arithmetic, therefore important conditions may still be left out. Other diagnosis systems only monitor limited amount of application information, simply based on the designers' instincts. We argue that the people who know the best about where and what to monitor are the ones that design and implement the system themselves. However, due to the lack of OS support for mechanisms to collect, store and manage such useful information, it is either discarded, or saved in some application logs by some ad-hoc method in unknown or unstructured format, which is hard for the diagnosis software to automatically examine. OS support for typed, structured, efficient error and event report would greatly alleviate the problem.
- *configuration management*: As the misconfiguration becomes one of the main reasons for application faults [36], the system designers should pay more attention to it. The plain text `.conf` style configuration files are still prevalent in \*nix systems, but we really need to move on to a more disciplined approach, which provides typed persistent storage, keep track of recent critical changes, and validates the configuration based on known models [16]. An embarrassing example is that in PeerPressure, the authors had to manually develop heuristics to fig-

ure out the same value with different representations. For instance, 1, “#1”, “1” all represent the same value, but with different types! A well managed configuration system would not have such silly problems. Windows registry and GConf [18] are moving towards the goal on desktop computers, but there is still a long way to go.

- *privacy and security*: Privacy is an important issue when the system statistics are collected across different administrative domains, e.g. the router configuration of different ISPs, desktop application configuration of different users [36], or the users’ program statistics [27]. Revealing the sensor outputs from a user might leak sensitive, confidential information. The sampling technique can alleviate the problem by adding uncertainty in the data, but it does not completely solve it. The language-based information flow approach [40] may be helpful, although it also faces hard challenges like timing channels. On the other hand, motivating incentives for users to be willing to share the data should also be investigated to facilitate the data collection [29].

### 7.2.2 Machine learning

Although the case studies have demonstrated the effectiveness of several learning algorithms, there are still many interesting problems remained unsolved, some of which raise new challenges that are not important concerns in traditional machine learning fields, such as pattern recognition, natural language processing, robotics, etc.

- *evaluating and updating models*: A complex system is evolving during its lifetime: hardware upgrades, software upgrades and request pattern changes are all common examples in real world. To learn a model that is capable of expressing all relevant behaviors is quite unrealistic. Therefore, we must be able to evaluate the current model, and adapt the model to the changing environment as well. The transition from offline to online learning also imposes an efficiency requirement to the learning algorithms.
- *learning with missing data*: All four case studies presented in this paper assume data collected are complete. In the real world, there might be missing values in the sensor outputs  $\vec{x}$ , e.g. due to the nature of asynchronized systems. A practical learning algorithm should be able to cope with missing values. The EM algorithm [13] might be promising. On the other hand, the outcome of applications  $y$  may be missing

too, *i.e.*  $\vec{x}$  may not be labeled. It is possible because sometimes it may be costly to label the data, e.g. when human labor has to be involved. There is a whole class of unsupervised learning algorithms that may be suitable for the task. Candidates include clustering, dimensionality reduction, factor analysis, etc. For example, the Pinpoint system [8] has demonstrated the effectiveness of a clustering algorithm in fault diagnosis.

- *distributed learning*: In a distributed system, the sensor outputs are naturally produced distributedly in the network nodes. However, most learning algorithms are centralized: they require the data to be gathered together to a centralized node before processing them. To reduce the communication cost and take advantage of the end node processing power, an ideal learning algorithm would run in a distributed manner (Guestrin *et al.* provided such an example in [21]).

### 7.2.3 Database

A complicated system can produce a huge amount of sensing data. The logging system at eBay currently services more than two thousand application servers at more than one billion URLs/day and a peak data rate of 200Mbps. It stores 1TB of raw logs per day [9]. It is inefficient to store and query data at such a large scale in plain text log format and use ad-hoc scripts to process. Relational databases also are inappropriate because the log files do not have a predefined fixed schema, ACID semantics in relational database are expensive and unnecessary, and the query is continuous and time related. Streaming database [7] seems to be a promising direction [39].

## 8 Summary

This paper motivates the need for automated diagnosis in complex computer systems, and defines a common framework that captures the similarity of four seemingly different diagnosis systems. The approach of each work is presented, and its strengths and limitations are discussed and compared. Finally, the lessons learned from the case studies and possible future directions in the field are identified and presented.

## Acknowledgments

I would like to thank Zack Ives for chairing my WPE-II committee, as well as Lawrence Saul and Jonathan M. Smith for sitting on the committee. I would also like to thank Moises Goldszmidt, Ben Liblit and Helen Wang for kindly answering my questions regarding their papers. Finally, John Blitzer's comments improved this paper.

## References

- [1] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.
- [2] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 74–89, New York, NY, USA, 2003.
- [3] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems (TOCS)*, 19(4):483–518, 2001.
- [4] Gaurav Banga. Auto-diagnosis of field problems in an appliance operating system. In *Proceedings of USENIX Annual Technical Conference*, San Diego, CA, USA, June 2000.
- [5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 259–272, San Francisco, CA, USA, December 2004.
- [6] M. Burgess. A site configuration engine. *Computer Systems*, 1995.
- [7] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Vijayshankar Raman, Fred Reiss, and Mehul A.

- Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of CIDR*, 2003.
- [8] Mike Chen, Emre Kiciman, Eugene Fratkin, Eric Brewer, and Armando Fox. Pinpoint: Problem determination in large, dynamic, internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (IPDS Track)*, Washington D.C, 2002.
- [9] Mike Chen, Alice Zheng, Jim Lloyd, Michael Jordan, and Eric Brewer. Failure diagnosis using decision trees. In *Proceedings of International Conference on Autonomic Computing (ICAC)*, New York, NY, USA, May 2004.
- [10] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 309–322, San Francisco, CA, USA, March 2004.
- [11] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeff Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, USA, December 2004.
- [12] A. Couch and M Gilfix. It’s Elementary, Dear Watson: Applying Logic Programming to Convergent System Management Processes. In *Proceedings of 13th USENIX Large Installation System Administration Conference (LISA '99)*, 1999.
- [13] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1):1–38, 1977.
- [14] Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin, and Amin M. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems USITS*, March 2003.
- [15] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of 18th ACM SOSP*, 2001.

- [16] Nick Feamster and Hari Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, May 2005.
- [17] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29:131–163, 1997.
- [18] GConf configuration system. <http://www.gnome.org/projects/gconf/>.
- [19] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. Chapman and Hall/CRC, 1995.
- [20] Thomas Gschwind, Kave Eshghi, Pankaj K. Garg, and Klaus Wurster. Webmon: A performance profiler for web transactions. In *Proceedings of the Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, Newport Beach, CA, USA, June 2002.
- [21] Carlos Guestrin, Peter Bodi, Romain Thibau, Mark Paski, and Samuel Madden. Distributed regression: an efficient framework for modeling sensor network data. In *IPSN'04: Proceedings of the third international symposium on Information processing in sensor networks*, pages 1–10, New York, NY, USA, 2004. ACM Press.
- [22] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer-Verlag, 2001.
- [23] HP OpenView. <http://www.hp.com/openview/>.
- [24] A. Keller and C. Ensel. An Approach for Managing Service Dependencies with XML and the Resource Description Framework. *Journal of Network and Systems Management*, 2002.
- [25] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
- [26] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C. Snoeren. IP fault localization via risk modeling. In *Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, May 2005.
- [27] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM*

*SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

- [28] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 12–15 2005.
- [29] Ratul Mahajan, David Wetherall, and Thomas Anderson. Negotiation-Based Routing Between Neighboring ISPs. In *Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, May 2005.
- [30] J. Kent Martin and D. S. Hirschberg. The time complexity of decision tree induction. Technical Report ICS-TR-95-27, 1995.
- [31] R. Osterlund. PIKT: Problem Informant/Killer Tool. In *Proceedings of 14th USENIX Large Installation System Administration Conference (LISA)*, 2000.
- [32] Claudia Perlich, Foster Provost, and Jeffrey S. Simonoff. Tree induction vs. logistic regression: a learning-curve analysis. *Journal of Machine Learning Research*, 4:211–255, 2003.
- [33] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
- [34] IBM Tivoli Business Systems Manager. <http://www.tivoli.com>.
- [35] Helen J. Wang, Yih-Chun Hu, Chun Yuan, Zheng Zhang, and Yi-Min Wang. Friends Troubleshooting Network: Towards Privacy-Preserving, Automatic Troubleshooting. In *Proceedings of IPTPS'04*, Feb 2004.
- [36] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, USA, December 2004.
- [37] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of 17th USENIX Large Installation System Administration Conference (LISA '03)*, pages 159–172, Oct 2003.

- [38] Web-to-Host: Reducing the Total Owner of Cost. [http://www.wrq.com/assets/products\\_tolly\\_tco.pdf](http://www.wrq.com/assets/products_tolly_tco.pdf). The Tolly Group, WRQ Inc.
- [39] Wei Xu, Peter Bodik, and David Patterson. A Flexible Architecture for Statistical Learning and Data Mining from System Log Streams. In *Workshop on Temporal Data Mining: Algorithms, Theory and Applications at The Fourth IEEE International Conference on Data Mining (ICDM'04)*, Brighton, UK, Nov 2004.
- [40] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 1–14, Banff, Canada, October 2001.
- [41] Alice X. Zheng, Michael I. Jordan, Ben Liblit, and Alex Aiken. Statistical debugging of sampled programs. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.