

Automated Diagnosis of System Failures with Fa

Songyun Duan
Duke University
syduan@cs.duke.edu

Shivnath Babu
Duke University
shivnath@cs.duke.edu

ABSTRACT

Failures of Internet services and enterprise systems lead to user dissatisfaction and considerable loss of revenue. Since manual diagnosis is often laborious and slow, there is considerable interest in tools that can diagnose the cause of system failures quickly and automatically from system-monitoring data. In this paper, we identify key data-processing challenges in a platform, called *Fa*, that we have developed for automated diagnosis. *Fa* uses monitoring data to construct a database of *failure signatures* against which data from undiagnosed failures can be matched. Two novel challenges we address are to make signatures robust to the noisy monitoring data in production systems, and to output trustworthy confidence estimates for matches. *Fa* uses a new technique called *query-aware clustering* when the signature database has no high-confidence match for an undiagnosed failure. This technique clusters monitoring data based on how it differs from the failure data, and pinpoints the attributes linked to the failure. We demonstrate the effectiveness of *Fa* through a comprehensive experimental evaluation based on failures from a production setting, a variety of failures injected in a testbed, and synthetic data.

1. INTRODUCTION

A recent study [20] found that 72% of the top-40 Web sites suffer user-visible problems, such as slow responses, blank pages or error messages being displayed, items not being added to shopping carts, unexpected database slow-downs, and others. Walmart.com was unavailable for almost 10 hours during the peak U.S. 2006 holiday season. Such deviation of systems from desired behavior may violate *service-level objectives (SLOs)* that specify what an acceptable level of service is. For example, an SLO for an online brokerage may stipulate that all transactions complete within 1 second, regardless of how much middleware, databases, or networks are involved.

SLO violations in a system indicate *failures*. When a system meets all specified SLOs, it is in a healthy state; other-

wise, it is in a failure state. Failures may be caused by a variety of factors including performance problems like resource contention, crashes due to hardware or software faults, and misconfiguration by system administrators. The increasing scale, complexity, and dynamics of modern systems make it laborious and time-consuming to track down the cause of failures manually [8, 14].

At the same time, it is important to diagnose failures and recover systems quickly. Brokerages and banking firms can lose up to \$75,000 per minute of downtime [14]. A 22-hour outage at eBay cost the company more than \$3 Million in customer credits and \$4 Billion in market capitalization. These factors motivate an automated, efficient, and reasonably-accurate tool for diagnosing failures using system monitoring data; a tool that is also easy and intuitive for system administrators to use. However, an automated diagnosis tool faces nontrivial challenges:

- **Noisy data:** Monitoring data collected from production systems contains various types of errors that can mislead diagnosis: (i) natural system variability injects Gaussian noise; (ii) failures may corrupt observations; and (iii) rapid system state transitions cause observations from different states to get mixed up.
- **High dimensionality:** Some of our monitoring datasets have 100-300 attributes per server, posing challenges from an accuracy as well as running-time perspective.
- **Dynamic systems:** Conventional approaches like defining a baseline system behavior, and pinpointing deviations from the baseline do not work when workloads and system configuration change over time.
- **Reuse:** Since failure diagnosis is expensive in large-scale and complex systems, it is valuable to leverage past diagnosis efforts whenever possible; particularly since 50-90% of failures seen are recurrences of previous failures [4].
- **Trust:** Features like reliable confidence estimates and evidence for diagnosis results are important for an automated diagnosis tool to gain administrators' trust; otherwise the tool will not be used in practice.

Our core contribution in this paper is a novel system, called *Fa*, for automated diagnosis of system failures. *Fa* is designed to address the above challenges. We begin with an overview of *Fa* and its new contributions.

	lock_time	num_io	failures	annotation
h1	51.3	76.1	0	
h2	48.5	63.6	0	
l1	95.4	43.5	1	Lock prob
h3	51.9	97.9	0	
h4	49.0	43.6	0	
u1	75.6	83.5	1	?
u2	72.4	83.8	1	?
h5	50.4	13.5	0	
h6	50.8	51.2	0	
l2	89.6	123.2	1	Buffer prob
h7	49.8	119.5	0	
h8	49.3	141.2	0	
h9	72.4	65.4	0	

(a)

	lock_time	num_io	failures	annotation
f1	70.0	80.7	1	?
f2	71.9	85.6	1	?

(b)

Figure 1: Sample data

2. THE FA SYSTEM AND CONTRIBUTIONS

System Monitoring Data: When a system is running, Fa collects monitoring data periodically and stores it in a database. In this paper, we consider monitoring data with a relational schema as shown in Figure 1 (a). For example, Fa uses the *sar* [19] utility to collect more than 100 performance metrics (e.g., average CPU utilization, number of disk I/Os) periodically from Linux servers. Database servers maintain performance counters (e.g., number of index updates, number of full table scans) that Fa reads periodically. Most enterprise monitoring systems like HP OpenView and IBM Tivoli Monitoring collect similar data.

Over a period of time, the monitoring data collected by Fa will contain three types of instances:

- **Healthy data H** , which is monitoring data collected when the system was in a healthy state. Recall that a system is in a healthy state when it experiences no SLO violations; and in a failure state otherwise.
- **Unannotated failure data U** , which is monitoring data collected from failure states of the system where the cause of failure has not been diagnosed so far.
- **Annotated failure data L** , which is monitoring data collected from failure states of the system where the cause of failure has been diagnosed. A successful diagnosis can happen any time after the failure occurs. Upon diagnosis, information about the type and cause of failure is attached as an *annotation* (or metadata) to the corresponding monitoring data. Specifically, the addition of an annotation to an instance t in the unannotated data U , moves t from U to L .

Example 1. Figure 1(a) displays the historic data for a database server collected by monitoring the server at one-minute intervals. In each interval, attribute *lock_time* is the average wait time to acquire locks; *num_io* is the number of disk I/Os; *failures* denotes whether the average response time of database transactions in that interval exceeded a threshold (causing SLO violations) or not; and *annotation* records the cause of each diagnosed failure. In this historic data, healthy data H consists of instances $h1$ – $h9$, annotated data L consists of failure instances $l1$ and $l2$, and unannotated data U consists of failure instances $u1$ and $u2$.

Diagnosis Queries: When the monitored system experiences a failure, an administrator or system-management software can diagnose the cause of the failure by posing a *diagnosis query* to Fa of the form $Q = \text{Diagnose}(F, H \cup U \cup L)$.

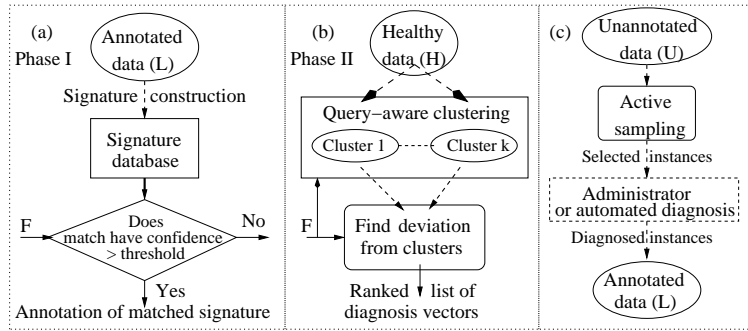


Figure 2: Control and data flow in Fa

- F is monitoring data from the system during the failure (or just before the failure in the case of a system crash).
- $H \cup U \cup L$ is the historic data collected so far.

Example 2. Figure 1(b) shows recent monitoring data F from the same server as in Example 1. The values of the *failures* attribute in F indicate that the server is experiencing some type of failure.

Fa processes a diagnosis query $Q = \text{Diagnose}(F, H \cup U \cup L)$ in two phases, as illustrated in Figure 2.

Phase I (Figure 2(a), Section 3): First, Fa finds whether the failure represented by F is the same as a previously-diagnosed failure in L . That is, Phase I translates Q to $Q_I = \text{Diagnose}(F, L)$. If the diagnosis result produced by this phase has confidence higher than a specified threshold, then Fa returns this result to the issuer of the query; otherwise Fa goes to a more expensive Phase II of diagnosis. Our novel contributions in Phase I include:

- Techniques to construct a database of failure signatures directly from system-monitoring data that includes numeric and categorical attributes.
- Monitoring data from real systems, especially the failure data F , will always contain errors. A signature database is *robust* if it can match F with the right failure signature even when L and F contain some amount of error. Fa generates robust signature databases with two important properties: (i) Slower drop of accuracy compared to competing techniques as error increases; and (ii) *Error-awareness*, which is the ability to do even better when the expected error is known (e.g., based on error models learned from historic data).
- When Fa outputs a matched signature for an undiagnosed failure, it also gives a reliable confidence estimate. That is, the accuracy of Fa’s answer is proportional to the confidence estimate, which is often not true for competing techniques.
- Setting the confidence threshold used to trigger the more expensive Phase II is nontrivial because of a tradeoff: a higher threshold may trigger Phase II more often than needed, while a lower threshold may lower diagnosis accuracy. Fa sets this threshold automatically.

Phase II (Figure 2(b), Section 4): Here, Fa compares F with the healthy data H collected so far, to see whether the cause of the failure can be characterized succinctly as attributes whose values in F deviate from their values in the data representing different healthy states of the system. That is, Phase II translates Q to $Q_{II} = \text{Diagnose}(F, H)$. Our novel contributions in Phase II include:

- Fa uses a new clustering algorithm called *query-aware clustering* that gives special attention to F while clustering H , to identify a minimal set of healthy system states needed for accurate diagnosis of F (i.e., no false positives or negatives). Query-aware clustering outperforms classic (e.g., K-means [23]) and recent clustering algorithms (e.g., *LAC* [11]) on efficiency and diagnosis accuracy.
- In spite of the high-dimensionality of monitoring data, Fa produces concise attribute sets (i.e., almost no false positives) while characterizing the deviation of F from healthy states.

Background Phase (Figure 2(c)): While not our focus in this paper, Fa supports techniques to guide manual diagnosis efforts by actively selecting informative unannotated instances from U for diagnosis. These techniques (described in [13]) run constantly in a background phase, transferring data from U to L to improve the signature database’s accuracy and coverage.

3. PHASE 1: GENERATING AND USING A SIGNATURE DATABASE

We are given L , the historic data with annotations about m distinct failures, denoted A_1, A_2, \dots, A_m . Our goal is to generate a *signature database* from L that contains entries of the form $\langle \text{sig}, A_i \rangle$ where *sig* is a signature for failure A_i . Each failure can be represented by any number of signatures, including zero. As illustrated in Figure 2(a), instances F from an undiagnosed failure can be matched against the database to find the signature nearest to F . The annotation of this signature will be returned along with a confidence estimate in the match if the confidence exceeds a threshold.

3.1 Overview

We will first illustrate our ideas using a series of examples. Suppose L is as shown in Figure 3(a). Each instance has two attributes, x (denoting `lock_time`) and y (denoting `num_io`), plotted along the horizontal and vertical axes respectively; and one of four distinct annotations A_1 (cross), A_2 (plus), A_3 (triangle), or A_4 (circle).

Clustering: One way to generate the signature database is by clustering the data in L using a technique like K-means [23]. Figure 3(a) shows the clusters per failure type. The centroids of these clusters—represented by blue stars (“★”) in Figure 3(a)—become the signatures for the corresponding failure type, giving a signature database SD_1 as shown in Figure 4 (a).

Suppose the query instance $f_1 = \langle 32, 41 \rangle$ in Figure 3(a) is a failure instance that we want to diagnose. f_1 can be matched with SD_1 to find the centroid (signature) nearest to f_1 ; which is a centroid for Failure A_1 (cross) given the data distribution. However, this diagnosis is incorrect since it is obvious from Figure 3(a) that f_1 is an instance of Failure

A_2 (plus). This example illustrates that although clustering-based signatures are conceptually simple, they have drawbacks. (They work poorly on real data in our experiments.)

Separating functions: Instead of clustering, suppose we identify *separating functions* $s_1(x, y)$, $s_2(x, y)$, $s_3(x, y)$, and $s_4(x, y)$ that separate each type of failure instances from the others. These functions can take many different forms.

To convey our ideas while keeping the example simple, we will use a simple form, namely, separating lines in the 2D plane. Figure 3(a) shows s_1 – s_4 as dotted lines. For example, $s_1(x, y)$ separates the instances of Failure A_1 from the others, and has the form: $s_1(x, y) = 1$ if $y > 45$, otherwise 0.

Matrix: Figure 4(b) shows a signature database SD_2 generated using s_1 – s_4 . SD_2 is a matrix with each row representing the signature of some failure. For example, the signature of Failure A_1 is $\langle s_1(x, y) = 1, s_2(x, y) = 0, s_3(x, y) = 0, s_4(x, y) = 0 \rangle$, denoted $\langle 1, 0, 0, 0 \rangle$. Each column represents a separating function. For example, the first column represents $s_1(x, y)$ which maps instances of Failure A_1 to 1, and instances of all other failures to 0.

To match a query instance $f = \langle x, y \rangle$ with SD_2 , we compute $\vec{s}(x, y) = \langle s_1(x, y), s_2(x, y), s_3(x, y), s_4(x, y) \rangle$ and find the signature nearest to $\vec{s}(x, y)$ in SD_2 . For example, $\vec{s}(32, 41)$ is $\langle 0, 1, 0, 0 \rangle$ for the query instance $f_1 = \langle 32, 41 \rangle$. (Note that $\langle 0, 1, 0, 0 \rangle$ matches Failure A_2 ’s signature perfectly.) For now, we will measure distances in terms of the *Hamming distance*, namely, the number of bits that are different. Thus, the distances of $\vec{s}(32, 41)$ to the four signatures in Figure 4(b) are respectively 2, 0, 2, 2. Since $\vec{s}(32, 41)$ is nearest to A_2 ’s signature, f_1 is diagnosed correctly.

Handling errors: Now consider the query instance $f_2 = \langle 39, 41 \rangle$ shown in Figure 3(b). f_2 is of failure type A_2 , but has higher error in the x dimension than the instances in L . If we match f_2 against SD_2 , $\vec{s}(39, 41)$ is $\langle 0, 0, 0, 0 \rangle$. Since $\vec{s}(39, 41)$ is equidistant from all the signatures in SD_2 , f_2 will not be diagnosed correctly by SD_2 .

Now suppose we use the signature database SD_3 from Figure 4(c) to diagnose f_2 . SD_3 contains two new separating functions, $s_5(x, y)$ and $s_6(x, y)$. s_5 separates instances of Failures A_1 and A_2 from those of A_3 and A_4 , and s_6 separates instances of A_1 and A_3 from those of A_2 and A_4 ; as represented by the columns for s_5 and s_6 in Figure 4(c). The separating planes for s_5 and s_6 are shown in Figure 3(b). Now, $\vec{s}(x, y) = \langle s_1(x, y), s_2(x, y), s_3(x, y), s_4(x, y), s_5(x, y), s_6(x, y) \rangle$. For f_2 , $\vec{s}(39, 41) = \langle 0, 0, 0, 0, 1, 0 \rangle$. $\vec{s}(39, 41)$ has least Hamming distance to the signature for Failure A_2 in Figure 4(c), so f_2 will now be diagnosed correctly.

Why did SD_3 diagnose f_2 correctly, while SD_2 did not? The reason can be understood from an analogy to *error correction* in telecommunications. Error correction is the ability to reconstruct the original, error-free data at the destination in the presence of errors caused by noise or other impairments during transmission from source to destination. The central idea in error correction is as follows: a bit string b to be transmitted is interleaved with some carefully-chosen extra bits, to transmit a new bit string b' such that a fixed number or less of bit-flip errors during transmission will not convert b' to a new bit string a' that corresponds to the transmitted version of another bit string a , $a \neq b$. (If this case arises, then the destination cannot tell whether a was transmitted or b .)

For f_2 , s_2 predicts 0 instead of the correct 1; causing a 1-bit error. SD_3 is robust to 1-bit errors, but SD_2 is

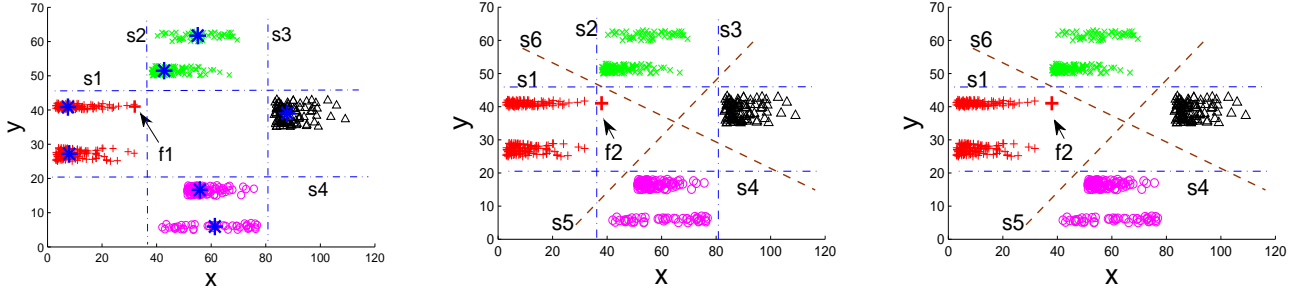


Figure 3: (a) Clustering Vs. separating functions; (b), (c) improving robustness of signature databases

SD ₁			SD ₂					SD ₃						SD ₄							
lock_time	num_io	annotation	s ₁	s ₂	s ₃	s ₄	annotation	s ₁	s ₂	s ₃	s ₄	s ₅	s ₆	annotation	s ₁	s ₂	s ₃	s ₄	s ₅	s ₆	annotation
7.9	27.2	A ₁	1	0	0	0	A ₁	1	0	0	0	1	1	A ₁	$\beta_1=1$	$\beta_2=0$	$\beta_3=0$	$\beta_4=1$	$\beta_5=1$	$\beta_6=1$	A ₁
7.5	40.9	A ₁	0	1	0	0	A ₂	0	1	0	0	1	0	A ₂	1	0	0	0	1	1	A ₁
42.8	51.5	A ₂	0	0	1	0	A ₃	0	0	1	0	0	1	A ₃	0	1	0	0	1	0	A ₂
55.1	61.7	A ₂	0	0	0	1	A ₄	0	0	0	1	0	0	A ₄	0	0	1	0	0	1	A ₃
87.8	39.0	A ₃	s ₁ = 1 if y > 45, otherwise 0					s ₁ = 1 if y > 45, otherwise 0						s ₁ = 1 if y > 45, otherwise 0							
61.3	5.9	A ₄	s ₂ = 1 if x < 38, otherwise 0					s ₂ = 1 if x < 38, otherwise 0						s ₂ = 1 if x < 39, otherwise 0							
65.8	16.7	A ₄	s ₃ = 1 if x > 80, otherwise 0					s ₃ = 1 if x > 80, otherwise 0						s ₃ = 1 if x > 80, otherwise 0							
			s ₄ = 1 if y < 20, otherwise 0					s ₄ = 1 if y < 20, otherwise 0						s ₄ = 1 if y < 20, otherwise 0							
								s ₅ = 1 if 0.81*x - y > -16, otherwise 0						s ₅ = 1 if 0.81*x - y > -16, otherwise 0							
								s ₆ = 1 if 0.36*x + y > 61, otherwise 0						s ₆ = 1 if 0.36*x + y > 61, otherwise 0							

Figure 4: Signature databases: (a) SD₁, (b) SD₂, (c) SD₃, (d) SD₄

not. The Hamming distance between any two signatures in SD_3 is ≥ 3 . Thus, even though $\vec{s}(39, 41)$ was computed as $(0, 0, 0, 0, 1, 0)$ —a 1-bit error from the ideal $(0, 1, 0, 0, 1, 0)$ — $\vec{s}(39, 41)$ still remained nearest to A_2 's signature. However, the Hamming distance between any two signatures in SD_2 is ≥ 2 . Thus, a 1-bit error in $\vec{s}(39, 41)$ leaves it in an ambiguous position for SD_2 ; causing incorrect diagnosis.

The previous example shows that selected redundancy in the set of separating functions can overcome incorrect predictions by some of the functions. Learning more functions increases the cost of generating the signature database. However, that is not a concern since the bulk of this work is done offline, and can be made very efficient with parallel learning of functions. The more pressing issue is that some functions are less reliable than others, and their presence can hurt diagnosis accuracy and confidence significantly.

Functions s_2 and s_3 are two less reliable functions in our example. Note that the data for each type of failure in Figure 3(a) shows larger spread along the x axis than the y axis. Intuitively, there are larger chances of error in the x values. Since s_2 and s_3 separate exclusively along the x axis, they are likely to get their predictions wrong when errors in x arise (like what happened for f_2). We can drop s_2 and s_3 , and generate a new signature database SD_4 that has the four functions s_1 , s_4 , s_5 , and s_6 only (Figure 3(b)). SD_4 is shown in Figure 4 (d). Note that SD_4 will give a perfect match for f_2 .

Takeaway points: Our series of examples show that the following is a powerful representation of the signature database to achieve both good accuracy and robustness to errors:

- *Binary matrix M :* The i th row in M , denoted $M(i, :)$, is the signature of failure A_i , $1 \leq i \leq m$. The j th

column in M , denoted $M(:, j)$, corresponds to the separating function $s_j(\vec{x})$. The number of columns d depends both on m and the built-in error tolerance desired.

- *Separating functions $s_1(\vec{x})$ – $s_d(\vec{x})$:* Each function separates one or more types of failure instances from the others. These functions can take many different forms. Fa uses *Classification and Regression Trees (CART)* [23] that are learned automatically from L .
- *Weights β_1 – β_d for the respective functions:* For robustness to errors, the prediction $s_j(\vec{x})$ from a less reliable separating function s_j is given a smaller weight while computing the distance of $\vec{s}(\vec{x})$ to the signatures. For example, reasonable weights for s_1 – s_6 in SD_4 are $\{1, 0, 0, 1, 1, 1\}$ because s_2 and s_3 are less reliable.

Next, we describe the offline generation of the signature database, its use for diagnosis, and online maintenance.

3.2 Generating the Binary Matrix

There are four rules to generate a valid matrix M :

- (I) Each row should be distinct since no two failures can have the same signature.
- (II) Columns that contain all 0s or 1s should be excluded, since they do provide no differentiation among failures.
- (III) Two columns cannot be the same or complementary since they derive the same separating function. (A 0-1 exchange in a column generates its complementary.)

- (IV) The *radius* r of M , defined as half the minimum Hamming distance over all $\langle M(i, :), M(j, :) \rangle$ pairs, $i \neq j$, should be above a given threshold. Intuitively, the higher the radius, the higher the error-correction ability of M . For a query instance \vec{x} , $\vec{s}(\vec{x})$ can be matched with the correct signature even when up to $r - 1$ separating functions produce wrong predictions for \vec{x} .

We use a random search algorithm to generate M given a threshold R_t on M 's radius. The algorithm is as follows:

1. Generate m random binary vectors of length $d = R_t(2 + \delta)$. δ is a positive integer (we set $\delta = 1$). The expected Hamming distance between each pair of vectors is $d/2$.
2. Remove columns containing all 0s or 1s (Rule II).
3. For any identical or complementary column pair, retain one column only (Rule III).
4. If the matrix generated by Steps 1-3 has a radius smaller than the threshold R_t , then go to Step 1.

This simple algorithm is surprisingly effective. The radius threshold R_t is a design choice best left to the administrator. R_t balances diagnosis accuracy and robustness against the time to generate the signature database—higher R_t means more columns (functions), and hence longer time to generate the database. Based on our empirical observations, $R_t = 5 \log_2(m)$ is a balanced choice, and is Fa's default.

3.3 Generating the Separating Functions

For each column $M(:, j)$ of the matrix, Fa learns the separating function $s_j(\vec{x})$ as a binary classification tree (CART) [23] separating the instances with annotation $M(i, j) = 1$ from the instances with annotation $M(i, j) = 0$. In separate work [12] with many types of functions from statistical machine-learning, we found CARTs to best balance prediction accuracy and learning time.

3.4 Weighting the Separating Functions

Suppose the j th separating function $s_j(\vec{x})$ has weight β_j , and the overall weight vector is $\vec{\beta} = \langle \beta_1, \beta_2, \dots, \beta_d \rangle$. A query instance \vec{x} whose true annotation is A_i will be matched with A_i 's signature $M(i, :)$ if the following condition holds:

$$\min_{k \neq i} \{ \text{Dist}(\vec{s}(\vec{x}), M(k, :); \vec{\beta}) - \text{Dist}(\vec{s}(\vec{x}), M(i, :); \vec{\beta}) \} > 0 \quad (1)$$

Here, Dist is the *weighted* Euclidean distance: $\text{Dist}(\vec{u}, \vec{v}; \vec{\beta}) = \sqrt{\sum_{j=1}^d \beta_j (u_j - v_j)^2}$. For binary vectors \vec{u} and \vec{v} , the Euclidean distance is the square root of the Hamming distance.

The larger the difference in Equation 1, the higher the chances of matching \vec{x} to the correct signature when errors cause variations in $\vec{s}(\vec{x})$. Thus, to make the signature database robust, we want to choose the weight vector $\vec{\beta} = \langle \beta_1, \dots, \beta_d \rangle$ that maximizes this difference over all instances $\langle \vec{x}, A_i \rangle$ in the annotated failure data L . Under the condition that $\|\vec{\beta}\|^2 (= \sum_{j=1}^d \beta_j^2)$ is fixed, this optimization is:

$$\max_{\vec{\beta}} \min_{\langle \vec{x}, A_i \rangle \in L, k \neq i} \{ \text{Dist}(\vec{s}(\vec{x}), M(k, :); \vec{\beta}) - \text{Dist}(\vec{s}(\vec{x}), M(i, :); \vec{\beta}) \}$$

If there exists some weight vector $\vec{\beta}$ that satisfies Equation 1 for all instances $\langle \vec{x}, A_i \rangle$ in L , then the above optimization problem is equivalent to the following one:

$$\min_{\vec{\beta}} \frac{1}{2} \|\vec{\beta}\|^2$$

such that, $\forall \langle \vec{x}, A_i \rangle \in L$ and $\forall k \neq i$:

$$\text{Dist}(\vec{s}(\vec{x}), M(k, :); \vec{\beta}) - \text{Dist}(\vec{s}(\vec{x}), M(i, :); \vec{\beta}) \geq 1 \quad (2)$$

This equivalence is shown in [21] which reports recent results on learning *Support Vector Machines (SVMs)* for complex problems like sequence alignment and grammar learning. ([21] does not consider signatures or failure diagnosis.) The above optimization problem is a general version of the maximum-margin principle used in SVM learning, and can also be solved using SVM learning to generate $\vec{\beta}$ [21].

However, there are two reasons why we do not want to use the weights learned from Equation 2:

- There may not be a feasible $\vec{\beta}$ that satisfies Equation 1 for all instances $\langle \vec{x}, A_i \rangle$ in L .
- We want to avoid the classic problem of *overfitting* [23], where $\vec{\beta}$ is too well tuned for L that it performs poorly for undiagnosed failure instances not in L .

Both these issues can be addressed by introducing a *slack variable* $\varepsilon_i \geq 0$ per $\langle \vec{x}, A_i \rangle \in L$ to relax the corresponding constraints in Equation 2; for a new optimization problem:

$$\min_{\vec{\beta}, \varepsilon} \frac{1}{2} \|\vec{\beta}\|^2 + c \sum_{i=1}^{|L|} \frac{\varepsilon_i}{|L|}$$

such that, $\forall \langle \vec{x}, A_i \rangle \in L$ and $\forall k \neq i$:

$$\text{Dist}(\vec{s}(\vec{x}), M(k, :); \vec{\beta}) - \text{Dist}(\vec{s}(\vec{x}), M(i, :); \vec{\beta}) \geq 1 - \varepsilon_i \quad (3)$$

Here, $|L|$ is the number of instances in L . The constant $c > 0$ controls the tradeoff between matching training instances in L correctly and the robustness to errors. Fa uses SVM learning algorithms from [21] to learn the weights $\vec{\beta} = \langle \beta_1, \dots, \beta_d \rangle$ by solving this optimization problem. A good value of c is determined via 5-fold cross validation over L [23].

3.5 Online Use and Maintenance

So far we discussed how the signature database is generated offline from L . We now discuss the online use of the database for diagnosis, and its maintenance as new instances and annotations are added to L . When the database is queried with an undiagnosed failure instance \vec{x} , Fa first computes $\vec{s}(\vec{x}) = \langle s_1(\vec{x}), \dots, s_d(\vec{x}) \rangle$, and then finds the signature nearest to $\vec{s}(\vec{x})$, namely, the signature that minimizes $\text{Dist}(\vec{s}(\vec{x}), M(i, :); \vec{\beta})$, $1 \leq i \leq m$. As shown in Figure 2(a), a confidence estimate *conf* is generated for this match and compared with the confidence threshold C_t . If $\text{conf} \geq C_t$, then the annotation of the matched signature is returned as the diagnosis result; otherwise Phase II is invoked.

Confidence estimate: When \vec{x} is matched with the signature $M(i, \cdot)$, the confidence in this match is defined as:

$$conf = \min_{k \neq i} \{ \text{Dist}(\vec{s}(\vec{x}), M(k, \cdot); \vec{\beta}) - \text{Dist}(\vec{s}(\vec{x}), M(i, \cdot); \vec{\beta}) \} \quad (4)$$

Intuitively, $conf$ is high when the second-nearest neighbor N_2 of $\vec{s}(\vec{x})$ is far from the first-nearest neighbor N_1 , indicating an unambiguous match to N_1 . As the gap between N_1 and N_2 shrinks, the ambiguity in the match to N_1 increases, and the confidence decreases.

To make the confidence estimate easier for administrators to understand, Fa converts it into a value in $[0, 100]$. This conversion is done using an equi-depth histogram (quantiles) with 100 buckets generated from the distribution of confidence estimates over all instances of L . Let p_k , $0 \leq k < 100$, denote the quantiles of this distribution. For a confidence estimate $conf$ from Equation 4, Fa finds i such that $p_i \leq conf < p_{i+1}$; and reports i as the confidence estimate.

Setting the confidence threshold C_t : The value of C_t is critical because a low C_t can lead to incorrect diagnosis, while a high C_t can invoke the more expensive Phase II more often than needed. (Compared to Phase I, Phase II involves higher run-time overhead and more efforts from administrators to interpret diagnosis results.) Fa’s approach is to let the administrator specify the minimum diagnosis accuracy she wants from the signature database. Then, Fa automatically derives the appropriate C_t that gives this diagnosis accuracy while minimizing the chances of invoking Phase II. The algorithm works as follows:

1. Divide L into a *training set* and a *test set*. Generate a signature database SD from the training set (Sections 3.2-3.4). For each instance in the test set, use SD to find the matched signature and confidence estimate.
2. Pick an integer value $x \in [0, 100]$. For test instances whose confidence estimate is $\geq x$, compute the percentage of instances matched to the correct signature in SD . This percentage is the expected diagnosis accuracy when the confidence threshold is x , denoted $acc(x)$. Vary x in $[0, 100]$ to get enough $\langle x, acc(x) \rangle$ points to plot the *accuracy-confidence curve (AC-Curve)* for SD as shown in Figure 5. Note that as x increases, the accuracy is being computed on higher-confidence answers from SD ; so we expect the AC-Curve to be nondecreasing.
3. Pick the minimum x_t in the AC-Curve such that $\forall x, x \geq x_t, acc(x)$ is above the diagnosis accuracy desired by the administrator. x_t is a sample of the desired value of C_t .
4. Repeat Steps 1, 2, and 3 with different training and test sets from L to get multiple independent samples of C_t ; and set C_t to their mean.

Figure 5 is a sample graph from our experiments which plots the AC-Curves for both Fa’s signature database (FA) and the clustering-based signature database (CLUS). If the administrator desires a diagnosis accuracy of 95%, the Fa’s $C_t = 20$ while CLUS’s $C_t = 80$. That is, Fa is four times less likely to trigger Phase II than CLUS.

Incremental Maintenance: When new instances are added to L , we need to update two components of the signature database: its separating functions and its weight vector. Recall that CARTs are used as the separating functions and the weight vector is determined via an SVM learning algorithm. Both the separating functions and the weight vector can be updated efficiently with new instances using an incremental CART learning algorithm [23] and an incremental SVM learning algorithm [6] respectively.

When new annotations are added to L , we also need to update the Matrix. One row (signature) is generated for each added annotation. The existing separating functions can be updated incrementally. If the radius of the new Matrix goes below the threshold R_t specified in Section 3.2, then more columns will need to be added to the Matrix to bring its radius above the threshold. New separating functions will have to be learned from scratch for the newly added columns. As we do not expect frequent additions of new annotations, the time amortized for learning new CARTs over a long interval should be small. Again the amortized time for computing the weight vector from scratch via SVM learning is also small. Note that the weight vector can be updated incrementally if the Matrix does not grow in columns.

3.6 Error-Aware Signature Databases

This section considers how Fa can make the signature database more accurate and robust if it has models that represent errors expected in the monitoring data. These models could be derived from historic data.

Types of Error: We have seen two types of errors in our monitoring data: *Gaussian* and *non-Gaussian*.

- Gaussian error is caused by natural variability in real systems. If we take multiple observations of an attribute from a particular system state, a goodness-of-fit test to a normal distribution will often be positive.
- Two significant causes of error in our monitoring data cannot be modeled by Gaussian distributions: (i) the onset or presence of failure corrupts readings of some attributes (seen with JBoss application server and MySQL); (ii) observations from different states get mixed up in the same instance due to rapid system state transitions, or due to delays in measuring different attributes under system overload. (A third probable cause is incorrect filling of missing values by monitoring tools.)

Error Models: Both the above types of error can be captured using error models. Plenty of literature exists on error models that vary from simple to complex (e.g., [26]). For example, an attribute x_i with Gaussian error can be represented by a new attribute x'_i of the form $x'_i = x_i + \text{Gaussian}(0, \delta_i)$, where δ_i controls the scale of error. The value of δ_i can be learned from historic data. Error modeling is complementary to our robust signature construction techniques.

Error-Aware Matrix: The Matrix’s role is to provide redundancy at the level of separating functions. Intuitively, if more error is expected, then we should have a Matrix with a larger radius (recall Section 3.2). The radius of a given Matrix can be increased by adding columns.

Error-Aware Separating Functions: Algorithms for learning separating functions can be modified to utilize error information. Fa’s CARTs pick attributes to use in decision

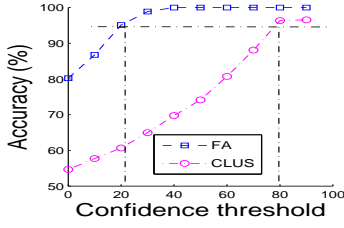


Figure 5: Sample AC-Curve

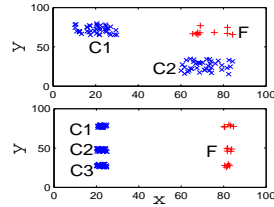


Figure 6: Sample data

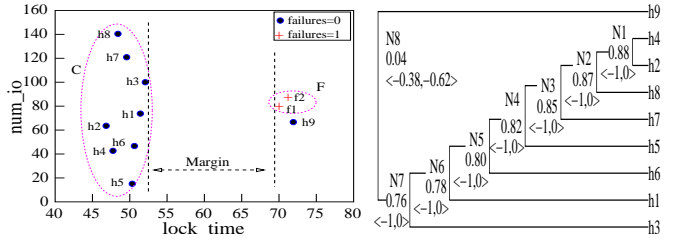


Figure 7: Plot of data in Fig.1 Figure 8: Dendrogram

nodes in the tree based on a metric called *information gain* (*InfoGain*) [23]:

$$\text{InfoGain}(A, x_i) = \text{Entropy}(A) - \text{Entropy}(A|x_i)$$

This classic formula represents the extra information we gain about the annotation A of an instance given the value of attribute x_i in that instance. As expected, attributes with larger *InfoGain* are preferred while picking attributes to use in decision nodes. However, an attribute x_i with higher chances of error—like attribute x in our running example in Section 3.1—should be preferred less because decisions made based on x_i 's value are less reliable. We can achieve this property by rewriting *InfoGain*(A, x_i) as:

$$\text{InfoGain}(A, x_i) = \text{Entropy}(A) - \text{Entropy}(A|x'_i)$$

Here, x'_i is the distribution of x_i once the expected error is added based on the known error model.

Error-Aware Weights: Finally, learning weights for separating functions can be made error-aware by appropriately choosing the set of $\langle \vec{x}, A \rangle$ instances used to learn the weights in Section 3.4. Along with the original instances in L , we can use new instances generated by injecting expected error (based on the error models) into the original instances in L .

4. PHASE II: QUERY-AWARE CLUSTERING

If the instances F to diagnose in a *Diagnose*(F) query correspond to a failure type that was not seen previously, then the match from the signature database will have low confidence. Phase II of diagnosis runs in this setting. The basic approach in Phase II is to determine how F differs from the data H representing the system in healthy states. We will first illustrate the main ideas using a series of examples.

Suppose H consists of the instances in Figure 6(a) shown using the “x” symbol. Each instance has two attributes, x and y , plotted along the horizontal and vertical axes respectively. The figure also shows the failure instances F , indicated using the “+” symbol, in a *Diagnose*(F) query. It is clear from the figure that there are two distinct healthy states of the system: (i) C_1 with $x \in [10, 30]$ and $y \in [65, 80]$, differing from F primarily along the x attribute; and (ii) C_2 with $x \in [60, 80]$ and $y \in [15, 30]$, differing from F primarily along the y attribute. A conventional clustering algorithm like K-means or LAC [11] can identify these clusters in H , and link both attributes x and y to the failure.

Next, suppose H (“x”) and F (“+”) are as shown in Figure 6(b). A conventional clustering algorithm will now group the instances in H into three distinct clusters (C_1 , C_2 , and C_3 in Figure 6(b)). Since each of these clusters differs from F along both the x and y attributes, both attributes will be linked to this failure as well. However, a closer look at

Figure 6(b) indicates that this answer is incorrect. Both the failure data and the healthy data have similar distribution along the y axis, and differ along the x axis only. So, the correct answer should link the failure to x only.

What went wrong in the second example? Conventional clustering algorithms ignore the failure (query) instances F while deciding how to group the instances in H into clusters. Thus, the clusters generated by these algorithms are independent of the failure instances to be diagnosed, causing two major weaknesses: (i) generating clusters that do not give the correct diagnosis, and (ii) generating many more clusters than needed, which can mislead the system administrator. Section 5.3 validates both observations empirically.

We have developed a new algorithmic framework, called *query-aware clustering*, that clusters H with consideration of the instances F to be diagnosed. (That is, the same H may be clustered differently for a different F .) Intuitively, query-aware clustering will place two instances $h_1, h_2 \in H$ into the same cluster iff they have similar deviations from F . This strategy gives the right answer for the example in Figure 6(b), generating a single cluster for H , and linking the failure to attribute x only. The rest of this section describes query-aware clustering.

4.1 Diagnosis Vectors and Margin Classifiers

Fa processes a *Diagnose*(F, H) query by first clustering the healthy data H into a set of clusters C_1, C_2, \dots, C_l , and then outputting the deviation of F from these clusters in the form $\{\langle \vec{w}_1, C_1 \rangle, \langle \vec{w}_2, C_2 \rangle, \dots, \langle \vec{w}_l, C_l \rangle\}$ as the diagnosis result. l depends on the query, and is not a predetermined constant. $C_1 \cup C_2 \cup \dots \cup C_l$ need not include all the instances in H . Thus, outlier instances in H will be ignored.

Each $\vec{w} \in \{\vec{w}_1, \dots, \vec{w}_l\}$ is called a *diagnosis vector*. \vec{w} has the form: $\vec{w} = \langle w_1, w_2, \dots, w_n \rangle$, where each attribute $x_j \in \langle x_1, x_2, \dots, x_n \rangle$ is given a weight w_j such that $-1 \leq w_j \leq 1$ and $\sum_{j=1}^n |w_j| = 1$. Intuitively, $\vec{w}_i \in \{\vec{w}_1, \dots, \vec{w}_l\}$ specifies the weighted list of attributes to which the failure can be localized by comparing the instances in C_i to the failure instances F . C_i serves as the evidence why \vec{w}_i is reported in the diagnosis result.

Computing the Diagnosis Vector: Since we are dealing with very high-dimensional data, a most desirable property of each $\langle \vec{w}, C \rangle$ is to make \vec{w} as concise as possible. That is, the weights of all attributes that do not help differentiate between C and F should be zero. This property enables the system administrator to zoom in quickly on likely causes of the failure without being misled by false positives. Fa uses *margin classifiers* (*MC*) to achieve this property. A margin classifier $MC(F, C)$, $C \subseteq H$, finds the linear combination $\sum_{j=1}^n w_j x_j$ of attributes $\langle x_1, \dots, x_n \rangle$ that produces the maximum separation between C and F . This maximum

Procedure *Margin Classifier (MC)***Input:** Healthy instances C , Failure instances F **Output:** Margin m between C and F , and the diagnosis vector $\vec{w} = \langle w_1, w_2, \dots, w_n \rangle$ that produces the margin

MC solves the following linear program:

1. Variables in the linear program:
 - (i) $X_i, Y_i, 1 \leq i \leq n$, such that output $w_i = X_i - Y_i$
 - (ii) $High, Low$ such that output $m = High - Low$
2. Constraints in the linear program:
 - (i) $\sum_{i=1}^n (X_i - Y_i)t.\mathbf{x}_i \geq High, \quad \forall t \in C$
 - (ii) $\sum_{i=1}^n (X_i - Y_i)t.\mathbf{x}_i \leq Low, \quad \forall t \in F$
 - (iii) $\sum_{i=1}^n (X_i + Y_i) = 1$
 - (iv) $X_i \geq 0, Y_i \geq 0, \quad 1 \leq i \leq n$
3. Optimization objective: Maximize $High - Low$
4. Solve the linear program; Return m and \vec{w}

Figure 9: Margin Classifier ($MC(F, C)$)

separation is called the *margin* between C and F .

Example 3. Consider query $Q = Diagnose(F, H)$ from Example 1 and Figure 7. Let $C = H - \{h9\}$. (h9 is an erroneous observation generated while the system transitioned from a healthy state to a failure state.) The margin between C and F is produced between the two dotted lines in Figure 7: the line `lock_time = 51.9` and the line `lock_time = 70.0`. Thus, $margin = 18.1$. Since the margin is produced along `lock_time`, the diagnosis vector $\vec{w} = \langle w_1, w_2 \rangle$ (corresponding to `<lock_time, num_io>`) that produces the margin is $w_1 = -1$ and $w_2 = 0$. `<1, 0>` also produces the same margin.

Figure 9 shows how a margin classifier $MC(F, C)$, $C \subseteq H$, works by solving a linear program to compute the margin between C and F . $MC(F, C)$ also finds the diagnosis vector that produces the margin. Section 5.3.2 gives an empirical validation of how margin classifiers produce concise and correct diagnosis vectors.

4.2 Strawman: Margin-based Agglomerative Clustering (MAC)

We begin with a strawman algorithm, called *margin-based agglomerative clustering (MAC)*, for query-aware clustering of H . MAC was proposed originally in [16] for analyzing cancer-related microarray data, and we have extended it to process diagnosis queries. MAC starts with an agglomerative hierarchical clustering [23] of the instances in H . The margin from the failure instances F is used as the metric for clustering. (Conventional clustering schemes use distance-based metrics like Euclidean distance.) Each instance in H is first placed in its own active cluster. In each iteration, MAC computes $MC(C_i \cup C_j, F)$ for each pair $\langle C_i, C_j \rangle$ of clusters among the remaining active clusters. MAC then picks the cluster pair $\langle C_{i'}, C_{j'} \rangle$ that gives the maximum margin with respect to F , and merges (agglomerates) them together to form a single combined cluster. The merged clusters $C_{i'}$ and $C_{j'}$ are no longer considered active. This process is repeated until all instances are merged into a single cluster. The entire process can be represented as a *dendrogram*, which is a tree with the instances in H as leaves, and each new cluster formed by MAC as a nonleaf node.

Example 4. Figure 8 shows the dendrogram generated by MAC for the healthy data in Example 1 and Figure 7. The

margin (computed after normalizing the data) and diagnosis vector for the cluster at each nonleaf node are also shown.

We can generate clusters from the dendrogram by selectively deleting nonleaf nodes which will partition the dendrogram into a forest of trees. The instances comprising the leaves of each tree form a cluster C that will be output as a $\langle \vec{w}, C \rangle$ pair in the query result after computing $MC(F, C)$.

Consider a node P in the dendrogram with child nodes L and R . Let the clusters corresponding to these three nodes be C_p, C_l , and C_r respectively. (Note that C_l and C_r were merged to form C_p in the dendrogram, i.e., $C_p = C_l \cup C_r$.) Let the margin and corresponding diagnosis vector for these three clusters be $\langle m_p, \vec{w}_p \rangle$, $\langle m_l, \vec{w}_l \rangle$, and $\langle m_r, \vec{w}_r \rangle$ respectively. We will delete P if any of the following conditions is satisfied: (i) $Margin(C_l, F, \vec{w}_p) < (1 - \alpha)m_l$, or (ii) $Margin(C_r, F, \vec{w}_p) < (1 - \alpha)m_r$. Here, $Margin(C, F, \vec{w})$ denotes the margin (separation) of a cluster of instances C from the failure instances F along the diagnosis vector \vec{w} . α is a small positive constant, e.g., $\alpha = 0.2$.

Note that \vec{w}_p is the diagnosis vector that gives the margin for the combination of C_l and C_r . Intuitively, if the margin of C_l (C_r) along \vec{w}_p is significantly less than the margin of C_l (C_r), then merging C_l with C_r is diluting the “clusteredness” of C_l (C_r) with respect to the failure instances F . (Note that $MC(C_l, F)$ computes C_l ’s maximum margin across all possible diagnosis vectors.)

Example 5. When the dendrogram in Figure 8 is partitioned, the nonleaf node N_8 will be deleted, to generate two output clusters $\{h1, h2, h3, h4, h5, h6, h7, h8\}$ and $\{h9\}$; precisely what we expect based on Figure 7. Recall that h9 is an erroneous observation generated during system transition.

MAC is inefficient: MAC requires $O(|H|^2)$ invocations of MC since MAC starts by invoking $MC(\{t_1, t_2\}, F)$ for every pair of instances t_1, t_2 in H . Thus, MAC scales poorly with $|H|$, but it gives good diagnosis accuracy; we will validate both observations empirically in Section 5.3.

4.3 Partition-Check-Merge (PCM) Algorithms

We now propose an algorithmic framework that combines the good features of MAC, which is accurate, but inefficient, with those of conventional Distance-based Partitional¹ Clustering (DPC) algorithms that are efficient, but less accurate. We later describe clustering algorithms that instantiate this framework. This new framework is called *Partition-Check-Merge (PCM)* because it has the following structure:

- One or more *partitioning phases* that use an efficient DPC algorithm to partition the data progressively into more and more clusters until the check in Step 2 is satisfied. This progressive cluster refinement is achieved by increasing the input parameter k to the DPC algorithm that specifies the number of clusters to generate.
- One or more *checking phases* that perform checks, namely, evaluating the current partitioning of instances to see whether this partition is good enough to be the set of clusters produced during an intermediate stage of MAC. If a check succeeds, then PCM moves to the merging phase; otherwise, partitioning is continued, possibly with a larger k .

¹Intuitively, partitional algorithms work in a top-down fashion, while agglomerative algorithms work bottom-up.

- A *merging phase* where the current set of clusters are merged progressively, like in MAC, to possibly consolidate several small clusters into a minimal set of clusters (representing diagnosis vectors and evidence) that can be output in the query result.

A degenerate case of PCM is one where the check never succeeds, so the partitioning phase eventually places each instance into a separate cluster. In this case, the merge phase will resemble running MAC from scratch. However, for most monitoring datasets, the check will succeed much earlier—e.g., once k becomes equal to or larger than the best k for the data—avoiding the $O(|H|^2)$ complexity of MAC. If the partitioning phase generates more clusters than optimal, then the merging phase will glue back clusters that should not have been split in the first place; at some loss of efficiency. In effect, PCM can be as accurate as MAC, while leveraging the efficiency of DPC. The challenge in PCM is in the implementation of the check phase. Next, we discuss two concrete instantiations of PCM.

4.4 PCM-Conservative (PCM-C)

PCM-Conservative (PCM-C) (Figure 10) uses a conservative implementation of check to process a $Diagnose(F, H)$ query. For each instance $t \in H$, PCM-C first computes m_t , the individual margin between t and the failure instances F .

For partitioning data instances, PCM-C does DPC using the LAC [11] algorithm (Line 7 in Figure 10). Suppose the clusters $\{C_1, \dots, C_k\}$ are produced by a partitioning step. For each $C \in \{C_1, \dots, C_k\}$, let $\langle m_C, \vec{w}_C \rangle$ be the margin and corresponding diagnosis vector for C and F . PCM-C’s check phase lists an instance $t \in C$ as *covered by C* if $Margin(\{t\}, F, \vec{w}_C) \geq (1 - \alpha)m_t$. (Recall from Section 4.2 that $Margin(\{t\}, F, \vec{w}_C) \leq m_t$, since m_t is t ’s maximum margin across all vectors.) That is, t is covered by the cluster C that t was assigned to by DPC if t ’s margin along C ’s diagnosis vector is close enough to t ’s individual margin.

If t is covered by C , then (i) t will not be considered again during partitioning (Line 21), and (ii) t will be associated with C in the input to the merge phase. PCM-C iterates through the partitioning and merge phases until all instances get covered. If check finds that the current set of clusters $\{C_1, \dots, C_k\}$ do not cover a significant fraction of the remaining instances, then partitioning is redone with a larger k ; currently, we double k when this situation arises (Lines 16-17). Thus, while PCM-C starts with a small default value of k , k will get incremented automatically if required.

Once all instances in H have been covered by a set of clusters generated by DPC, these clusters are input to the merge phase. Merge does MAC-style agglomerative clustering—starting with these clusters as the leaves of the dendrogram, instead of the $|H|$ individual instances—to generate the final query output.

4.5 PCM-Eager (PCM-E)

We found empirically (Section 5.3) that PCM-C tends to generate many clusters as input to the merge phase. While merge can glue back clusters that should not have been split, PCM-C’s merge remains inefficient because of the quadratic dependence on the number of input clusters. PCM-E tackles this problem.

Recall that PCM-C’s check phase will list an instance t as covered only if t ’s margin along the diagnosis vector of the cluster C that t was assigned to by DPC is close enough

Algorithm Partition-Check-Merge-Conservative (PCM-C)

Input: $Diagnose(F, H)$ query; Value of α (default is 0.2)

Output: Result in the $\{\langle \vec{w}_1, C_1 \rangle, \langle \vec{w}_2, C_2 \rangle, \dots, \langle \vec{w}_l, C_l \rangle\}$ format

```

/* First compute the individual margin of each instance  $t \in H$  */
1. Compute  $\langle m_t, \vec{w}_t \rangle = MC(\{t\}, F)$  for each instance  $t \in H$ ;
2.  $k = \text{default value}$ ; /* LAC’s number of clusters parameter */
3.  $Rem\_pts = H$ ; /* instances not assigned to clusters yet */
4.  $Coverings = \phi$ ; /* assignment of instances to clusters */
5. While ( $Rem\_pts \neq \phi$ ) {
6.   /* Partitioning phase */
7.   Partition  $Rem\_pts$  with LAC into clusters  $\{C_1, \dots, C_k\}$ ;
8.    $Outliers = \phi$ ;
9.   /* Check phase: Lines 10-22 below */
10.  For (each instance  $t \in Rem\_pts$ ) {
11.    Let  $C_i$  be the cluster that  $t$  was assigned to by LAC;
12.    If ( $Margin(\{t\}, F, \vec{w}_{C_i}) \geq (1 - \alpha)m_t$ )
13.      Mark  $t$  as covered by  $C_i$ ;
14.    Else Add  $t$  to  $Outliers$ ;
15.  } /* end for */
16.  If ( $\frac{|Outliers|}{|Rem\_pts|} > 0.9$ )
17.     $k = k \times 2$ ; /* increase  $k$ ,  $Rem\_pts$  is unchanged */
18.  Else {
19.    For (each cluster  $C_i \in \{C_1, \dots, C_k\}$  that covers instances)
20.      Add  $C_i$  and the instances  $C_i$  covers to  $Coverings$ ;
21.     $Rem\_pts = Outliers$ ; /* remove covered instances */
22.  } /* end else */
23. } /* end while */
24. /* Merge phase */
25. Initialize a partially-built dendrogram with the clusters in
26.    $Coverings$  as the leaves of the dendrogram;
27. Proceed with MAC using this partially-built dendrogram;

```

Figure 10: PCM-Conservative (PCM-C)

to t ’s individual margin. *PCM-Eager (PCM-E)* relaxes this condition as follows: PCM-E’s check lists t as covered if t ’s margin along the diagnosis vector of *any* of the clusters generated by DPC so far is close enough to t ’s individual margin. As before, if t is covered by C , then t will be assigned to C in the input to the merge phase. (Note that DPC may not have assigned t to C .) Intuitively, PCM-E reduces DPC’s role to identifying significant diagnosis vectors \vec{w} from the data. The $Margin(\{t\}, F, \vec{w}) \geq (1 - \alpha)m_t$ condition is used to associate instances with each \vec{w} , creating clusters that are input to the merge phase. The rest of PCM-E is similar to PCM-C.

4.6 Filtering and Ranking Diagnosis Results

Fa takes the set of $\langle \vec{w}_i, C_i \rangle$ pairs generated by query-aware clustering, and outputs the final diagnosis result as a filtered and ranked list.

- **Filtering:** Fa removes $\langle \vec{w}_i, C_i \rangle$ pairs where the cluster size $|C_i|$ does not satisfy a minimum support threshold; similar to support thresholds in frequent-itemset mining. Clusters composed of outliers get eliminated here.
- **Ranking:** The remaining $\langle \vec{w}_i, C_i \rangle$ pairs are ranked in decreasing order of cluster size $|C_i|$.

5. EXPERIMENTAL EVALUATION

5.1 Experimental setting

All the diagnosis techniques described in this paper were implemented in the Fa system. We evaluate these techniques in the context of a three-tier Web service composed of a Web server, an application server, and a database server. The evaluation is based on common types of failures in each tier. Table 1 summarizes our datasets and failure scenarios.

Failures injected in a testbed: We have implemented a testbed that runs *Rubis* [18]—a multitier auction service modeled after eBay—on a JBoss application server (with an embedded Web server) and a MySQL DBMS. It has been reported that software problems and operator errors are the common causes of failure in Web services [17]. We inject such failures into a running Rubis instance using a comprehensive failure-injection tool [5]. This setting makes it easy to study the accuracy of Fa’s diagnosis algorithms because we always know the true cause of each failure.

Specifically, we can inject 3 independent causes of failure—software bugs, data corruption, and uncaught Java exceptions—into any of the 25 Java modules (*enterprise Java beans (EJBs)*) that comprise the component of Rubis running in the application server. Using this mechanism, we can inject 75 distinct single-EJB failures and any number of independent multiple-EJB failures (concurrent single-EJB failures). Intuitively, multiple-EJB failures are harder to diagnose. The *Rubis-60*, *Rubis-complex*, *Rubis-bug* and *Rubis-jndi* monitoring datasets in Table 1 are from this setting. These datasets contain the number of times each distinct EJB procedure call is invoked per minute.

We can also inject failures caused by contention for CPU, memory, and disk resources. The *OLTP-single* and *OLTP-multi* monitoring datasets in Table 1 are collected from a MySQL DBMS running an OLTP workload, where we injected resource contention to cause failures. The datasets record OS metrics (e.g., CPU utilization, paging), DBMS performance counters (e.g., number of index accesses and table scans), and transaction-level performance metrics (e.g., average transaction response time) per minute.

Real failures in a production system: *Software aging*—progressive degradation in performance caused by, e.g., memory leaks, unreleased file locks, and fragmented storage space—is a common cause of system failure, especially in Web servers [22]. The *Dolphin* and *ECE* datasets were collected from two production servers at Duke over the course of two months. This data records OS metrics at 10-minute intervals. Both servers crashed once or more during this period due to aging of different resources, as found by a previous study [22]. We validate Fa’s automated diagnosis results with the results from this human-intensive study.

Synthetic data: *Synthetic* is a complex dataset (*PENDIG-ITS*) from the UCI machine-learning repository. Rather than generating our own synthetic data, we decided to use this dataset for the purpose of experimental repeatability.

5.2 Evaluation of Phase I

Queries: We consider $Diagnose(F, L)$ queries over the Rubis-60, Rubis-complex, and Synthetic datasets. By default, L contains 60% of the failure instances in each dataset, and is used to generate the signature database. The remaining 40% of the failure instances are used to query the signature database to compute its diagnosis accuracy (% of times the correct annotation is returned).

Techniques: We compare four techniques: (i) CLUS, sig-

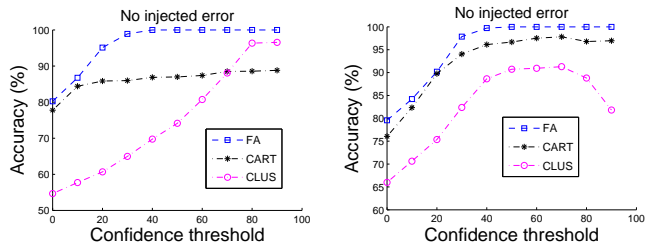


Figure 11: AC-Curves for Rubis-60, Rubis-complex

nature database implemented using K-means clustering with 10 clusters per annotation²; (ii) FA, Fa’s signature database; (iii) FA-EA, Fa’s error-aware signature database; and (iv) CART, a multi-class classifier implemented using classification and regression trees. By treating each annotation in L as a distinct class label, a multi-class classifier learned from L can predict the annotation of a new failure instance. We chose CART over other multi-class classifiers for three reasons: (i) CARTs are being used for diagnosis in production settings like eBay.com [7]; (ii) CARTs provide a principled way to compute confidence estimates; and finally, (iii) Fa uses CARTs as separating functions.

5.2.1 Comparing Accuracy-Confidence Curves

The goal of Phase I is to provide high diagnosis accuracy while invoking Phase II only when required. The Accuracy-Confidence Curves (AC-Curves) in Figure 11 show how well each technique meets this goal. No error is injected into the query instances (unlike the experiments in Section 5.2.3). Recall the definition of confidence estimates, confidence thresholds, and AC-Curves from Section 3. To diagnose a query instance, CARTs compute a probability distribution over annotations, and output the most-likely annotation. The confidence estimate is the difference in probabilities between the most-likely annotation and the second most-likely annotation, mapped to $[0, 100]$ as discussed in Section 3.5.

Suppose the administrator wants a diagnosis accuracy of 90%. Then, Figure 11(a) for Rubis-60 shows that the confidence thresholds (C_t) for FA and CLUS can be set to 20 and 80 respectively. Since FA’s C_t is 4 times less than that of CLUS, FA is four times less likely to invoke Phase II at the same accuracy level. More interestingly, CART is unusable when required accuracy is 90%. FA maintains its superior performance for Rubis-complex, while CLUS now becomes unusable when the required accuracy is over 90%.

Figure 11 used our default setting where for each query instance (\vec{x}, A) , the signature database contains at least one signature for annotation A . That is, we evaluated how good the signature database is in diagnosing previously-seen failures (which does not mean previously-seen instances). This setting is practical because as much as 90% of all software failures reported by users today are previously-seen failures [4]. We will now consider query instances whose correct annotations are not in the signature database. An accurate response from Phase I in this case is an answer with confidence below the threshold C_t ; thereby invoking Phase II.

To create this setting, we divide the instances in Rubis-60 into two groups with nonoverlapping annotations: Group G_1 with 40 annotations and Group G_2 with the remaining 20

²We also tried the state-of-the-art LAC clustering [11], and got similar results.

Name	a	i	Description of data and failures
1. Rubis-60	110	8184	Contains annotated data about 60 distinct single-EJB failures injected in our testbed
2. Rubis-complex	110	1797	Contains annotated data about 14 distinct multiple-EJB failures injected in our testbed
3. Synthetic	16	10992	Synthetic annotated data about 10 distinct failures; patterns in the data are complex
4. Dolphin, 5. ECE	43	4881	OS-level data collected for 55 days from two heavily-used departmental servers at Duke
6. Rubis-bug	110	900	Data access by the <i>BuyNow</i> EJB gets null result occasionally (bug in application logic)
7. Rubis-jndi	110	1500	JNDI naming-directory entry of the <i>SB_SearchItemsByRegion</i> EJB gets corrupted
8. OLTP-single	42	3660	Occasional CPU contention caused by an application on OLTP server (no disk contention)
9. OLTP-multi	28	696	Both CPU and disk contention caused separately by an application on the OLTP server

Table 1: Monitoring datasets used in the evaluation. Columns a and i are the number of attributes and instances respectively. Datasets 1-3 are used to evaluate Phase I, and datasets 4-9 to evaluate Phase II

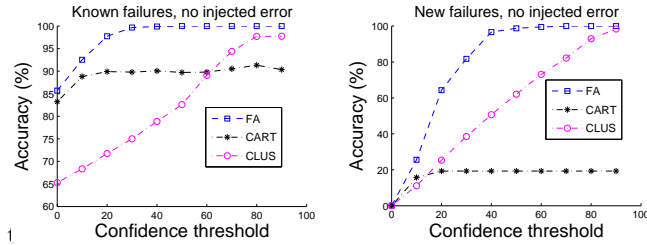


Figure 12: AC-Curves for Rubis-60 with two groups: (a) existing annotations, (b) new annotations

annotations. A subset of the instances from G_1 are used to construct the signature database. The remaining instances in G_1 (previously-seen failures) and the instances in G_2 (new failures) form the set of query instances. Figures 12(a) and (b) show the diagnosis accuracy of different techniques in these two cases. The behavior of signature databases (FA and CLUS) for both types of failures is as we saw in Figure 11(a). However, CART performs poorly on the new failures, which shows a key advantage of using signature databases for Phase I rather than multi-class classifiers. Intuitively, signature databases have a better chance of detecting when a failure does not have the symptoms of any previously-seen failure. Our algorithm for setting C_t (described in Section 3.5) considers both types of plots in Figure 12.

5.2.2 Verification of Error Models

Section 3.6 has discussed the Gaussian error model. To verify the presence of Gaussian error in our monitoring data collected for an attribute (e.g., CPU utilization), we took multiple measurements of this attribute from a particular system state. The normal probability plot of these values in Figure 16 and a hypothesis test for goodness of fit to a normal distribution (Matlab’s Lillietest) prove that Gaussian error exists in the monitoring data.

As mentioned in Section 3.6, some errors cannot be captured by the Gaussian distribution, e.g., when observations from different system states get mixed into an instance due to rapid system state transition. We verify such a situation with Figure 17 and 18. Figure 17 is a *cluster timeline* generated from a subset of the OLTP-single dataset when clustered using LAC with $k = 2$. A cluster timeline shows the progress of time on the x -axis and the current cluster identifier on the y -axis. In the OLTP-single dataset, CPU contention happens on the OLTP server in a periodic fashion with a period of 4 minutes; this pattern is clear from the cluster timeline. Figure 18 is a cluster transition dia-

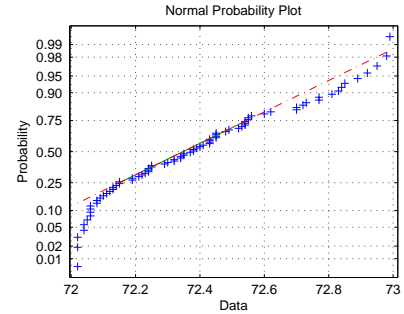


Figure 16: Normal probability plot for attribute $CPU_utilization \in (72, 73)$ in OLTP-single

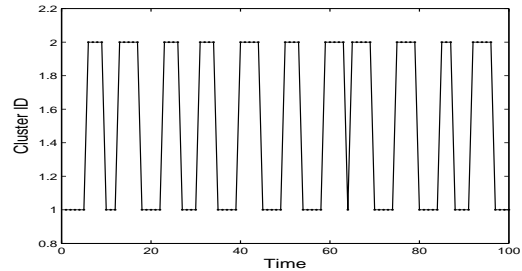


Figure 17: Cluster timeline for a subset of OLTP-single

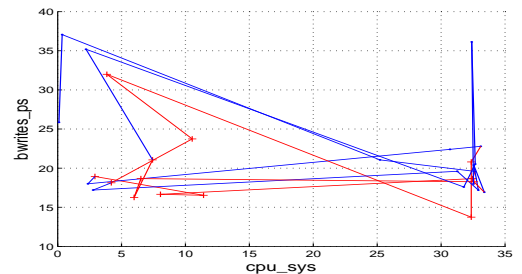


Figure 18: Cluster transition diagram for a subset of OLTP-single

gram that shows how the diagnosis vectors change over time. (This figure is best viewed in color.) Like Figure 17, Figure 18 was also generated from a subset of the OLTP-single dataset when clustered using LAC with $k = 2$. The x -axis and y -axis are two relevant attributes in the data, namely, cpu_sys , the CPU utilization (in OS space) on the OLTP server, and $bwrites_ps$, the number of disk blocks written

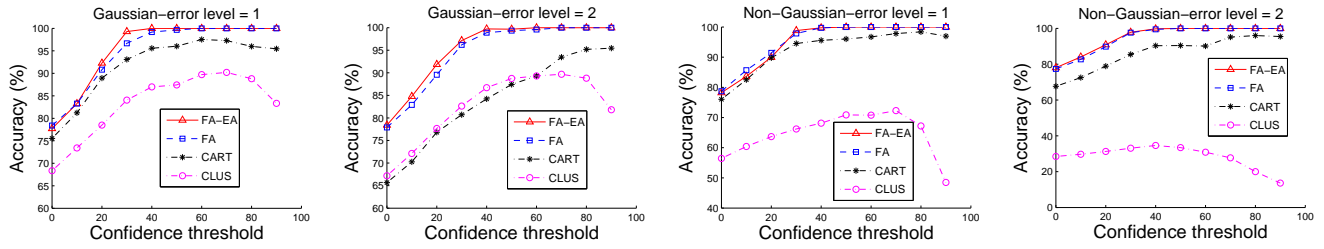


Figure 13: AC-Curves for Rubis-complex: (a), (b) Gaussian error; (c), (d) Non-Gaussian error

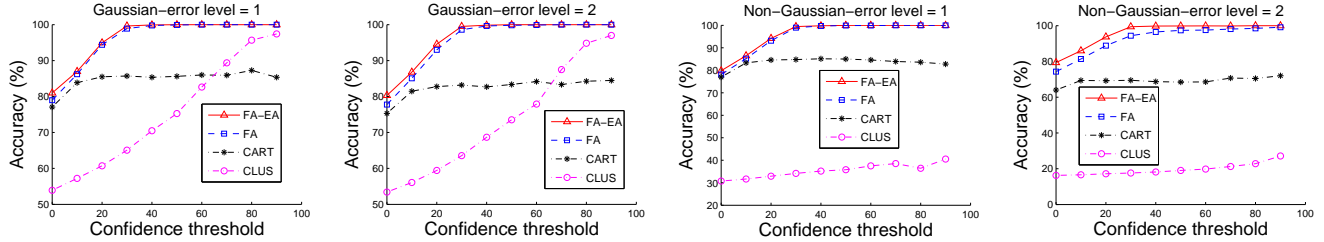


Figure 14: AC-Curves for Rubis-60: (a), (b) Gaussian error; (c), (d) Non-Gaussian error

per second on the OLTP server. Points belonging to the two clusters in the data are indicated respectively using a red “+” symbol and a blue “.” symbol. A line L_{p_1, p_2} from point p_1 to p_2 in Figure 18 indicates that the system was in state p_1 , and then transitioned to state p_2 in the next measurement interval; the color of L_{p_1, p_2} is the same as the color of point p_2 in Figure 18. Figure 18 illustrates some interesting aspects of the OLTP-single dataset:

- The red “+” points predominantly have smaller `cpu_sys` than the blue “.” points.
- The system tends to stay in the red (blue) state for four measurement intervals, and then transitions to the blue (red) state.
- There are *transitional points* in the data. These are points that are collected when the system is transitioning from one state to another, so these points may belong to different (similar) clusters, but have similar (different) attribute values.

Transitional points contain non-Gaussian errors that are affected by which states are involved in the transition. We adopt a probabilistic model to describe such non-Gaussian errors. This model assigns a probability p_i to attribute x_i that specifies how probable the reading of x_i is an incorrect value, a value in its observation range that is not the true value. p_i represents the scale of non-Gaussian error for x_i .

5.2.3 Comparing Robustness to Error

The query instances used so far to compute the accuracy of our diagnosis techniques were taken directly from the monitoring data. We now inject errors into these query instances based on error models to study how accuracy degrades as error increases.

Note that the model we use for Gaussian error is described in Section 3.6 and that for non-Gaussian error is described in Section 5.2.2. In our experiments with Gaussian-error model, the parameter δ_i for x_i is set to $0.2 * rand * error_level$

multiplied by x_i 's true value, where *rand* is a random value within $[0, 1]$ and $error_level \in \{1, 2, 3, 4\}$ controls the scale of errors. For non-Gaussian error model, the parameter p_i for x_i is set to $0.2 * rand * error_level$. *rand* is to make different attributes have different scales of errors. Roughly speaking, for an attribute x : (i) Gaussian error of level e means that observations of x have a variance of $10e\%$ from their true value; and (ii) non-Gaussian error of level e means that each observation of x has a $10e\%$ chance of being an arbitrary value from the range of values of x .

Figure 13 shows the AC-Curves for error levels 1 and 2 for Gaussian and non-Gaussian error in Rubis-complex. (Note that Figure 11(b) is the AC-Curve at error level 0.) It is clear from comparing these graphs that the gap between FA and CLUS/CART increases as the error increases. CLUS is highly sensitive to non-Gaussian errors. Further evidence is provided by Figure 19 where the confidence threshold C_t is set at 40. Figure 19 shows the accuracy of different techniques as the error level, both for Gaussian and non-Gaussian, increases from 0 to 4 for Rubis-complex. Also note that FA's performance is very close to that of the FA-EA algorithm which has knowledge of the expected error. The observations validate FA's robustness to error.

Figure 14 shows the AC-Curves for error levels 1 and 2 for Gaussian and non-Gaussian error in Rubis-60. Note that Figure 11(a) is the AC-Curve at error level 0. Figure 20 shows the accuracy of different techniques as the error level, both for Gaussian and non-Gaussian, increases from 0 to 4 for Rubis-60.

Figure 15 shows the AC-Curves for error levels 1 and 2 for Gaussian and non-Gaussian error in Synthetic. Figure 21 shows the accuracy of different techniques as the error level, both for Gaussian and non-Gaussian, increases from 0 to 4 for Synthetic.

5.2.4 Scalability with Number of Annotations

Figure 22 shows the trend as the number of annotations—distinct single-EJB failures—is increased from 20 to 80. Since we can generate at most 75 distinct single-EJB failures (re-

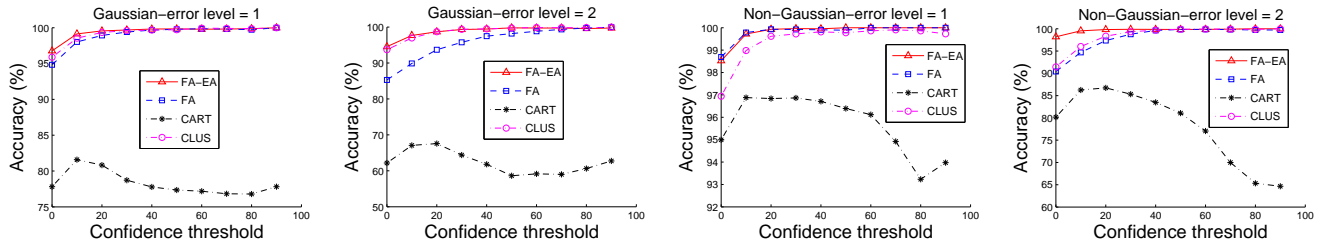


Figure 15: AC-Curves for Synthetic: (a), (b) Gaussian error; (c), (d) Non-Gaussian error

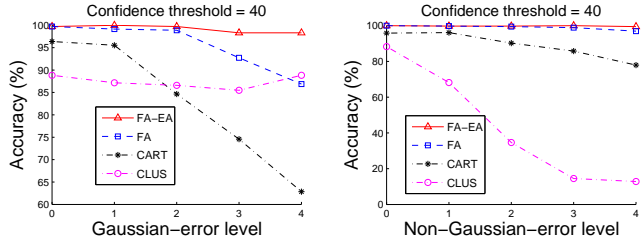


Figure 19: Robustness curves for Rubis-complex: (a) Gaussian error, (b) non-Gaussian error

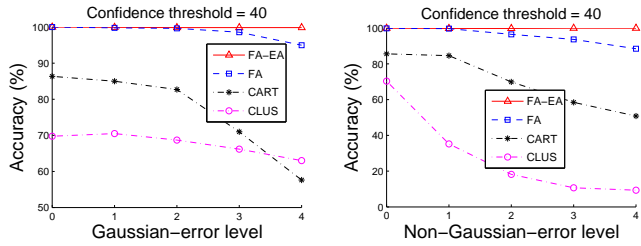


Figure 20: Robustness curves for Rubis-60: (a) Gaussian error, (b) non-Gaussian error

call Section 5.1), the 80-failure dataset contains 5 multiple-EJB failures as well. The gap between FA and CLUS/CART increases as the number of failures increases.

In the offline phase of signature database generation, FA is less efficient than CART or CLUS. If the separating functions are not learned in parallel, FA can take an order of magnitude more time to generate the database than CART or CLUS. However, these offline efforts make FA much better in the online phase because: (i) FA is comparable to CLUS and CART in the time for Phase I; and (ii) FA invokes Phase II much less often.

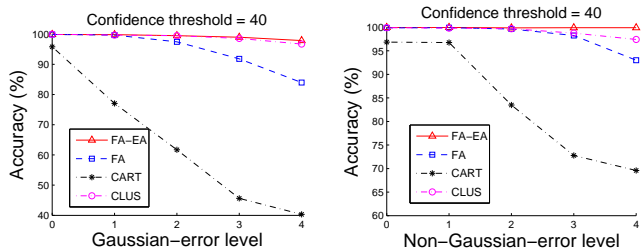


Figure 21: Robustness curves for Synthetic: (a) Gaussian error, (b) non-Gaussian error

5.3 Evaluation of Phase II

Queries: We now evaluate the processing of $Diagnose(F, H)$ queries in Phase II. For datasets 4-9 listed in Table 1, H contains the historic healthy monitoring data and F contains 5-10 instances from the listed failures. For Dolphin and ECE, F contains 5 instances collected just before each server’s first crash. We consider two cases for OLTP-multi, one where F contains failure instances from CPU contention, and the other where F contains failure instances from disk contention. We can evaluate the accuracy of diagnosis results since the cause of failure in each case is known.

Algorithms and Defaults: We evaluate four algorithms for Phase II: (i) MAC (Section 4.2), (ii) PCM-C (Section 4.4), (iii) PCM-E (Section 4.5), and (iv) *LAC-Silhouette* (*LAC-S*). *LAC-S* applies LAC on H after computing the number-of-clusters parameter k that maximizes a validity index called *Silhouette* [3]. *Silhouette* aims to maximize the inter-cluster distances (the average distance of pairs of points from different clusters) and minimize the intra-cluster distances (the average distance of pairs of points from the same cluster). For each cluster $C \subseteq H$ generated by LAC, *LAC-S* outputs $\langle \vec{w}, C \rangle$ computed using $MC(F, C)$.

5.3.1 Comparing Running Times

Table 2 shows the running time of our algorithms on the different datasets. Each reported time was averaged over 10 runs. For *LAC-S*, the time shown is the time to compute silhouette indices for 10 different values of k . This time is an optimistic estimate of the running time of *LAC-S* because we expect that more than 10 choices of k will have to be explored before finding the k that maximizes the silhouette index. We currently try all values of $k \in [2, 30]$. The best k is reported in *LAC-S*’s column in Table 2.

Because of its poor scalability, we had trouble running MAC on the full version of all but the smallest dataset (OLTP-multi) in Table 1. Therefore, the times for MAC are for scaled-down versions of the datasets. The scaled-down size is shown in MAC’s column in Table 2. The times for PCM-C and PCM-E are split into the time for the partitioning and checking phases, denoted T_p in Table 2, and the time for the merge phase, denoted T_m . The following trends are clear in Table 2.

- MAC is very inefficient because of $O(|H|^2)$ MC calls.
- PCM-E is by far the most efficient algorithm. Note that PCM-E’s T_m is usually significantly better than that of PCM-C. This trend is because PCM-E’s aggressive strategy to map points to clusters leads to a much lower number of clusters being input to the (quadratic) merge phase. Furthermore, PCM-E’s T_p

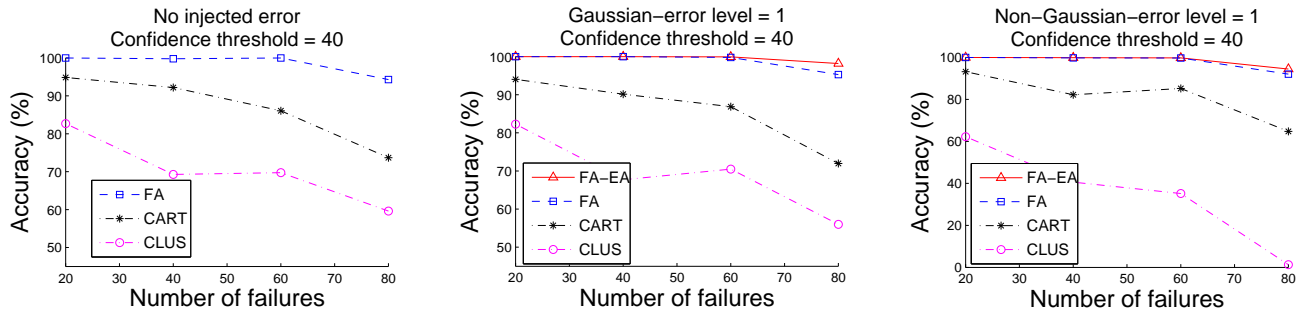


Figure 22: Trend as the number of failures increases

Dataset	LAC-S	MAC	PCM-C	PCM-E
Dolphin	260.3 $k=19$	5137.0 ($ H =1000$)	87.4 $T_p=23.5$ $T_m=63.9$	32.7 $T_p=18.8$ $T_m=13.9$
ECE	229.6 $k=2$	5187.9 ($ H =1000$)	89.1 $T_p=18.9$ $T_m=70.2$	26.1 $T_p=17.2$ $T_m=8.9$
Rubis-bug	28.3 $k=3$	1318.5 ($ H =600$)	45.8 $T_p=40.1$ $T_m=5.7$	34.7 $T_p=28.8$ $T_m=5.9$
Rubis-jndi	75.1 $k=2$	1399.6 ($ H =600$)	108.2 $T_p=92.6$ $T_m=15.6$	95.8 $T_p=87.0$ $T_m=8.8$
OLTP-single	214.5 $k=2$	5116.0 ($ H =1000$)	173.3 $T_p=137.4$ $T_m=35.9$	88.1 $T_p=70.6$ $T_m=17.5$
OLTP-multi, $F=$ CPU contention	7.2 $k=15$	1526.1	7.1 $T_p=4.3$ $T_m=2.8$	3.8 $T_p=3.2$ $T_m=0.6$
OLTP-multi, $F=$ Disk contention	6.2 $k=14$	1516.1	7.4 $T_p=3.3$ $T_m=4.1$	4.1 $T_p=3.6$ $T_m=0.5$

Table 2: Comparing running times (seconds)

is usually better than that of PCM-C because PCM-E’s aggressive strategy gets all points covered in fewer iterations of the partitioning and checking phases.

- PCM-E typically matches or outperforms LAC-S, which is because the silhouette computations in LAC-S perform $O(|H|^2)$ distance computations.

5.3.2 Comparing Diagnosis Accuracy of Phase II

Table 3 summarizes the diagnosis accuracy of our algorithms on the datasets. Full details of each query result, e.g., diagnosis vectors, margins, and rankings are in Appendix A. Numbers like 1st and 2nd in Table 3 indicate the smallest rank of a cluster C whose diagnosis vector gives non-zero weights to attributes relevant to the failure (smaller rank is better). The non-zero weights in this diagnosis vector are shown for PCM-E, with the weights for attributes relevant to the failure shown in bold font. Each cell also shows the % size of C with respect to the number of historic points $|H|$, and the number of $\langle \vec{w}, C \rangle$ pairs in the result after filtering with a support threshold of 2%. The following trends are clear in Table 3.

- PCM-C and PCM-E give the best accuracy in all cases.
- MAC gives good accuracy in most cases. Recall from Section 5.3.1 that MAC uses only a subset of the full

historic data since we had to scale down the datasets to get MAC to run in reasonable time.

- LAC-S gives poor accuracy in many cases.

As shown in Table 3, the diagnosis accuracy of LAC-S is poor for the Dolphin dataset when we use $k=19$ for which the silhouette cluster validity index is maximized. We tried LAC on this dataset with different values of k ranging from 2 to 30. In most cases, the relevant attribute—the attribute measuring available swap space, since the failure in Dolphin is because of swap space exhaustion—was not part of the diagnosis result. In the few cases where the relevant attribute was part of the diagnosis result, it appears in some lowly-ranked cluster (as for $k=19$ in Table 3) and/or as one among many attributes with nonzero weight in a diagnosis vector. Furthermore, as we increase k , LAC reports more and more clusters in the result, each cluster with its own diagnosis vector; so we can’t be confident about any of the output clusters or diagnosis vectors.

On the other hand, note from Table 3 that both PCM-C and PCM-E report the relevant attribute (as one of two attributes) in the diagnosis vector of the top-ranked cluster, which contains close to 40% of the total historic data. This result illustrates the power of PCM’s query-aware clustering. The Dolphin data contains many patterns because of the general effects of software aging, causing LAC’s query-unaware clustering to generate clusters that are not relevant to the query-specified failure points.

We have also compared PCM-E with two correlation-based techniques [7, 8] and two baselining-based techniques [2] proposed in previous work. The results of PCM-E are superior to all these techniques. Because of space constraints, these results, with descriptions of the previous techniques, are given in Appendix B.

5.3.3 Conciseness of PCM-E’s Diagnosis Vectors

Diagnosis vectors from PCM-E usually have very few attributes of nonzero weight, even for high-dimensional datasets. This property makes it easy to interpret results. The last column of Table 3 gives the diagnosis vector of the relevant cluster produced by PCM-E. For example, PCM-E’s diagnosis vector for Rubis-bug contains only 2 attributes (out of 110) with nonzero weights: one with weight 0.39, and the other with weight 0.61. (Sum of absolute weights is 1.) The latter attribute pinpoints the buggy Java bean. PCM-E’s diagnosis vector for Dolphin contains only 2 attributes (out of 41) with nonzero weights: one with weight 0.77 and the other with weight 0.23; both pinpoint the swap space

Dataset/Result	LAC-S	MAC	PCM-C	PCM-E	Diagnosis vector for PCM-E
Dolphin	8th, 5%, 14	Not found	1st, 39%, 8	1st, 37%, 7	2 nonzero weights, (0.77, 0.23)
ECE	Not found	2nd, 9%, 2	1st, 68%, 3	1st, 73%, 4	2 nonzero weights, (0.75, 0.25)
Rubis-bug	2nd, 25%, 3	1st, 14%, 7	1st, 19%, 3	1st, 21%, 6	2 nonzero weights, (0.61, 0.39)
Rubis-jndi	Not found	1st, 43%, 2	1st, 21%, 2	1st, 48%, 5	1 nonzero weight, 1
OLTP-single	Not found	2nd, 10%, 2	1st, 33%, 4	2nd, 26%, 5	4 nonzero weights, (0.3, 0.34, 0.11, 0.25)
OLTP-multi,CPU	3rd, 10%, 15	2nd, 25%, 4	2nd, 19%, 4	2nd, 26%, 5	3 nonzero weights, (0.49, 0.43, 0.08)
OLTP-multi,Disk	1st, 20%, 14	1st, 69%, 4	1st, 70%, 4	2nd, 74%, 4	3 nonzero weights, (0.55, 0.24, 0.19)

Table 3: Comparing diagnosis accuracy (see Section 5.3.2)

exhaustion problem causing the crash.

6. RELATED WORK

Fa differs from all previous work on automated diagnosis in three significant ways: (i) integration of Phase I (diagnosing recurrent failures) and Phase II (diagnosing failures not seen previously); (ii) considering robustness of diagnosis to errors in monitoring data; and (iii) providing reliable confidence estimates.

Diagnosis Phase I: Automated diagnosis of recurrent problems has been considered in previous work, e.g., [25, 4, 24, 9]. [25] builds a multi-class classifier on system event traces to classify previously-solved problems. We have empirically shown the advantages of signature databases over multi-class classifiers, especially in terms of robustness. [24] gives signature-generation techniques which assume that symptoms of each failure have been identified in the monitoring data; which is impractical in the settings we consider where only raw monitoring data is available. [4] considers a very different type of monitoring data—function call stacks from system failures—and gives stack matching and indexing algorithms. [9] extracts indexable signatures from failure data by finding metrics that differentiate a failure state from the healthy states. Instead, Fa captures the difference between one or more failure states from other failure states using annotation information which is ignored in [9].

Fa’s signature database generation resembles solving a multi-class classification problem using an ensemble of binary classifiers (e.g., see [1]). However, our focus on robustness to errors and reliable confidence estimates, and our new weighting algorithm in Section 3.4, differentiate Fa’s Phase I from previous work by the machine-learning community.

Diagnosis Phase II: Unlike query-aware clustering, previous work on Phase II of diagnosis predominantly takes one of the *correlation-based* (e.g., [7, 8]) or *baselining-based* approaches (e.g., [2]). [7] applies decision-tree learning techniques to rank different system components based on their correlation with system failures. [8] applies Bayesian-network learning techniques to correlate performance metrics with high-level system behavior. [2] proposes a heuristic to represent the baseline behavior of a Web service; and applies anomaly detection techniques to categorize deviation from the baseline behavior. We have compared PCM-E empirically on real monitoring datasets with the algorithms in [7, 8, 2]; the experimental results are reported in Appendix A). PCM-E performed the best due to the noisy and dynamic nature of the monitoring data.

Failure diagnosis in database systems: Oracle’s recent ADDM tool [10] shows the growing interest among database vendors on automated diagnosis of database failures. Fa differs from ADDM in two ways: (i) Fa targets a broader class of systems (e.g., multitier services); and (ii) ADDM relies on a static knowledge base gathered by Oracle experts over the years, while Fa automates the process

of generating the signature database from monitoring data. ADDM and Fa can complement each other.

7. CONCLUSION

We showed how Fa’s new contributions address the five challenges from Section 1 for an automated diagnosis tool:

- *Noisy data:* Our empirical evaluation (Sections 5.2.3 and 5.3.2) showed how Fa maintains high diagnosis accuracy in the presence of errors in the monitoring data.
- *High dimensionality:* Fa can pinpoint the 1-2 attributes related to a failure even in the presence of 50-100 attributes (Section 5.3.3).
- *Dynamic systems:* Both Fa’s signature database generation and query-aware clustering can deal with multiple healthy and failure states and rapid state transitions.
- *Reuse:* Fa’s techniques for Phase I increase reuse of previous diagnosis results by enabling high accuracy while minimizing invocations of Phase II (Section 5.2.1).
- *Trust:* Our empirical evaluation showed how Fa’s confidence estimates and diagnosis vectors are very reliable.

8. REFERENCES

- [1] E. L. Allwein, R. E. Schapire, and Y. Singer. Reducing multiclass to binary: A unifying approach for margin classifiers. In *ICML 2000*.
- [2] P. Bodik et al. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *ICAC’05*.
- [3] N. Bolshakova and F. Azuaje. Cluster validation techniques for genome expression data. *Signal Processing*, 83(4), 2003.
- [4] M. Brodie, S. Ma, G. M. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn. Quickly finding known software problems via automated symptom matching. In *Proceedings of ICAC*, pages 101–110, 2005.
- [5] G. Candea et al. Automatic failure-path inference: A generic introspection technique for internet applications. In *Proc. of IEEE Workshop on Internet Applications*, 2003.
- [6] G. Cauwenberghs and T. Poggio. Incremental and decremental support vector machine learning. In *NIPS*, pages 409–415, 2000.
- [7] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *ICAC’04*.
- [8] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI*, Dec. 2004.
- [9] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, Indexing, Clustering, and Retrieving System History. In *SOSP*, Oct. 2005.

- [10] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis and tuning in Oracle. In *CIDR*, 2005.
- [11] C. Domeniconi et al. Locally adaptive metrics for clustering high dimensional data. *Data Mining and Knowledge Discovery*, 14(1), 2007.
- [12] S. Duan and S. Babu. Processing forecasting queries. In *VLDB*, 2007.
- [13] S. Duan and S. Babu. Guided problem diagnosis through active learning. In *ICAC*, 2008. (To appear).
- [14] P. Horn. Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Corp., 2001. <http://www.research.ibm.com/autonomic>.
- [15] Density Estimation. <http://www.stat.sc.edu/rsrch/gasp/density/>.
- [16] K. Munagala et al. Cancer characterization and feature set extraction by discriminative margin clustering. *BMC Bioinformatics*, 2004.
- [17] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it. In *USENIX Symposium on Internet Technologies and Systems*, 2003.
- [18] *Rice University Bidding System*. rubis.objectweb.org.
- [19] *Performance monitoring tools for Linux*. <http://perso.wanadoo.fr/sebastien.godard>.
- [20] *Business Internet Group*. The black Friday report on Web application integrity. San Francisco, CA, 2003.
- [21] I. Tsochantaridis, T. Hofmann, T. Joachims, and Y. Altun. Support vector machine learning for interdependent and structured output spaces. In *ICML 2004*.
- [22] K. Vaidyanathan and K. S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, 2(2), 2005.
- [23] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition, June 2005.
- [24] A. Yemini and S. Kliger. High speed and robust event correlation. *IEEE Communication Magazine*, 1996.
- [25] C. Yuan et al. Automated known problem diagnosis with event traces. In *EuroSys*, 2006.
- [26] X. Zhu and X. Wu. Class noise vs. attribute noise: A quantitative study. *Artif. Intell. Rev.*, 22(3):177–210, 2004.

APPENDIX

A. DETAILS OF DIAGNOSIS RESULTS

In this section we give the details of the results summarized in Table 3. The results are given per dataset.

A.1 Dolphin

The most relevant attribute for the Dolphin dataset measures available swap space (`usedSwapSpace`), since the failure in Dolphin is because of swap space exhaustion. Also relevant are attributes that are related to the temporary space available, e.g., `tmpDirUsed` and `tmpDirAvail`.

LAC-S: There are 14 clusters with a support of 2%. The diagnosis vector of the top-ranked cluster has weight 0.1 for `tmpDirUsed`, and three other attributes. This cluster has a margin of 0.38. The number of tuples in this cluster is 667, of which 100% comes from points that are not categorized as failures. The 8th-ranked cluster has weight 0.19 for `userSwapSpace`, and seven other attributes. This cluster has a margin of 0.1. The number of tuples in this cluster is 260, of which 98% comes from points that are not categorized as failures.

MAC: There are 3 clusters with a support of 2%. The relevant attributes are not part of the diagnosis vectors of these three clusters.

PCM-C: The top-ranked cluster has a diagnosis vector with weight -0.74 for `tmpDirUsed` and weight -0.26 for `usedSwapSpace`. This cluster has a margin of 0.39. The number of tuples in this cluster is 1923, of which 96.5% comes from points that are not categorized as failures. There are 8 clusters with 2% support.

PCM-E: The top-ranked cluster has a diagnosis vector with weight -0.77 for `tmpDirAvail` and weight -0.23 for `usedSwapSpace`. This cluster has a margin of 0.39. The number of tuples in this cluster is 1790, of which 96.2% comes from points that are not categorized as failures. There are 7 clusters with 2% support.

A.2 ECE

The failure is because of exhaustion of free memory. The relevant attribute in the data is `realMemFree`.

LAC-S: There are 2 clusters with a support of 2%. The relevant attribute is not part of the diagnosis vectors of these clusters.

MAC: The 2nd-ranked cluster has a diagnosis vector with weight -0.99 for `realMemFree`. (`realMemFree` is also part of the top-ranked cluster, but with a very small weight.) This cluster has a margin of 0.48. The number of tuples in this cluster is 94, of which 89.4% comes from points that are not categorized as failures. There are 2 clusters with 2% support.

PCM-C: The top-ranked cluster has a diagnosis vector with weight -0.98 for `realMemFree`. This cluster has a margin of 0.47. The number of tuples in this cluster is 3200, of which 85.1% comes from points that are not categorized as failures. There are 3 clusters with 2% support.

PCM-E: The top-ranked cluster has a diagnosis vector with weight -0.74 for `realMemFree`. This cluster has a margin of 0.27. The number of tuples in this cluster is 3450, of which 84.7% comes from points that are not categorized as failures. There are 4 clusters with 2% support.

A.3 Rubis-bug

The relevant attributes measure the number of invocations per minute for different procedures of the *BuyNow* Java module in the application server.

LAC-S: The 2nd-ranked cluster has a diagnosis vector with weight 1 for the invocation of the `getItemId` procedure in the *BuyNow* Java module. This cluster has a margin of 0. The number of tuples in this cluster is 363, of which 50.1% comes from points that are not categorized as failures. There are 3 clusters with 2% support.

MAC: The top-ranked cluster has a diagnosis vector with weight 1 for the invocation of the `getItemId` procedure in the *BuyNow* Java module. This cluster has a margin of 0.1. The number of tuples in this cluster is 83, of which 98.8% comes from points that are not categorized as failures. The 2nd-ranked cluster has a diagnosis vector with weight 1 for the invocation of the `getNextBuyNowID` procedure in the *IDManagerHome* Java module which invokes procedures in the *BuyNow* module (and thus, is affected by the failure of the *BuyNow* module.) This cluster has a margin of 0.11. The number of tuples in this cluster is 75, of which 100% comes from points that are not categorized as failures. There are 7 clusters with 2% support.

PCM-C: The top-ranked cluster has a diagnosis vector with weight -0.26 for the invocation of the `getItemId` procedure in the *BuyNow* Java module. This cluster has a margin of 0.07. The number of tuples in this cluster is 173, of which 98% comes from points that are not categorized as failures. The 3rd-ranked cluster has a diagnosis vector with weight 1 for the invocation of the `getNextBuyNowID` procedure in the *IDManagerHome* Java module which invokes procedures in the *BuyNow* module (and thus, is affected by the failure of the *BuyNow* module.) This cluster has a margin of 0.06. The number of tuples in this cluster is 48, of which 98% comes from points that are not categorized as failures. There are 3 clusters with 2% support.

PCM-E: The top-ranked cluster has a diagnosis vector with weight -0.21 for the invocation of the `getItemId` procedure in the *BuyNow* Java module. This cluster has a margin of 0.07. The number of tuples in this cluster is 186, of which 97.8% comes from points that are not categorized as failures. The 2nd-ranked cluster has a diagnosis vector with weight 1 for the invocation of the `getNextBuyNowID` procedure in the *IDManagerHome* Java module which invokes procedures in the *BuyNow* module (and thus, is affected by the failure of the *BuyNow* module.) This cluster has a margin of 0.06. The number of tuples in this cluster is 121, of which 99% comes from points that are not categorized as failures. There are 6 clusters with 2% support.

A.4 Rubis-jndi

The relevant attributes measure the number of invocations per minute for different procedures of the *SearchItemsByRegion* Java module in the application server.

LAC-S: There are 2 clusters with a support of 2%. The relevant attributes are not part of the diagnosis vectors of these clusters.

MAC: The top-ranked cluster has a diagnosis vector with weight 0.73 for the invocation of the `create` procedure in the *SearchItemsByRegion* Java module. This cluster has a margin of 0.11. The number of tuples in this cluster is 257, of which 96.1% comes from points that are not categorized as failures. There are 2 clusters with a support of 2%.

PCM-C: The top-ranked cluster has a diagnosis vector with weight 0.92 for the invocation of the create procedure in the *SearchItemsByRegion* Java module. This cluster has a margin of 0.09. The number of tuples in this cluster is 316, of which 99.4% comes from points that are not categorized as failures. There are 2 clusters with a support of 2%.

PCM-E: The top-ranked cluster has a diagnosis vector with weight 1 for the invocation of the create procedure in the *SearchItemsByRegion* Java module. This cluster has a margin of 0.07. The number of tuples in this cluster is 717, of which 95.7% comes from points that are not categorized as failures. There are 5 clusters with a support of 2%.

A.5 OLTP-single

The relevant attribute measures CPU utilization on the OLTP server.

LAC-S: There are 2 clusters with a support of 2%. The relevant attribute is not part of the diagnosis vectors of these clusters.

MAC: The 2nd-ranked cluster has a diagnosis vector with weight 1 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.76. The number of tuples in this cluster is 103, of which 0% comes from points that are not categorized as failures.

PCM-C: The top-ranked cluster has a diagnosis vector with weight 0.48 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.61. The number of tuples in this cluster is 1222, of which 88.9% comes from points that are not categorized as failures. There are 4 clusters with a support of 2%.

PCM-E: The top-ranked cluster has a diagnosis vector with weight 0.3 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.01. The number of tuples in this cluster is 1162, of which 78.66% comes from points that are not categorized as failures. There are 5 clusters with a support of 2%.

A.6 OLTP-multi (CPU-based Failure)

The relevant attribute measures CPU utilization on the OLTP server.

LAC-S: The 3rd-ranked cluster has a diagnosis vector with weight 0.47 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.34. The number of tuples in this cluster is 71, of which 28.2% comes from points that are not categorized as failures. There are 15 clusters with a support of 2%.

MAC: The 2nd-ranked cluster has a diagnosis vector with weight 0.58 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.33. The number of tuples in this cluster is 176, of which 42.6% comes from points that are not categorized as failures. There are 4 clusters with a support of 2%.

PCM-C: The 2nd-ranked cluster has a diagnosis vector with weight 0.47 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.34. The number of tuples in this cluster is 130, of which 42.3% comes from points that are not categorized as failures. There are 4 clusters with a support of 2%.

PCM-E: The 2nd-ranked cluster has a diagnosis vector with weight 0.57 for the attribute measuring CPU utilization (in OS space) on the OLTP server. This cluster has a margin of 0.33. The number of tuples in this cluster is 179, of which 39.1% comes from points that are not categorized

as failures. There are 5 clusters with a support of 2%.

A.7 OLTP-multi (Disk-based Failure)

The relevant attributes measure disk-usage (e.g., number of bytes or disk blocks read or written) on the OLTP server.

LAC-S: The top-ranked cluster has a diagnosis vector with weight -1 for the attribute measuring disk blocks read per second on the OLTP server. This cluster has a margin of 0.81. The number of tuples in this cluster is 137, of which 64.96% comes from points that are not categorized as failures. There are 14 clusters with a support of 2%.

MAC: The top-ranked cluster has a diagnosis vector with weight -1 for the attribute measuring disk blocks read per second on the OLTP server. This cluster has a margin of 0.8. The number of tuples in this cluster is 482, of which 44.8% comes from points that are not categorized as failures. There are 4 clusters with a support of 2%.

PCM-C: The top-ranked cluster has a diagnosis vector with weight -0.74 for the attribute measuring disk blocks read per second on the OLTP server. This cluster has a margin of 0.29. The number of tuples in this cluster is 485, of which 45.6% comes from points that are not categorized as failures. There are 2 clusters with a support of 2%.

PCM-E: The top-ranked cluster has a diagnosis vector with weights -0.47 , -0.23 , and 0.24 for the attributes measuring disk blocks read per second, disk blocks written per second, and number of OS-level writes per second respectively on the OLTP server. This cluster has a margin of 0.04. The number of tuples in this cluster is 513, of which 41.3% comes from points that are not categorized as failures. There are 4 clusters with a support of 2%.

Diagnosis approach	Causative attributes found
Correct result	Primary attribute: usedSwapSpace attribute (see Section 5.3.2); Secondary attributes: tmpDirAvail, tmpDirSize, or tmpDirUsed
PCM-E	usedSwapSpace ($w_1 = -0.23$), tmpDirAvail ($w_2 = -0.77$) in first cluster with 37% points
Metric attribution	None of the primary or secondary attributes in top-5 attributes in diagnosis vector
CART-based diagnosis	None of the primary or secondary attributes in top-5 attributes in diagnosis vector
KDE-based diagnosis	Attribute usedSwapSpace has weight = 1.0 and is one among five attributes in the diagnosis vector with weight = 1.0, and one among 20 attributes with weight > 0.99
Gaussian-based diagnosis	None of the primary or secondary attributes in top-5 attributes in diagnosis vector

Table 4: Comparison with previous approaches

B. EXPERIMENTAL COMPARISON WITH PREVIOUS WORK

We used the Dolphin dataset in Table 1 to compare our algorithms for diagnosis query processing with the following algorithms from previous work:

- The *metric-attribution approach* from [8] first learns a Tree-augmented Bayesian network classifier [23] using $H \cup F$, and then infers which attributes in A_1, \dots, A_n correlate highly with the attribute that tracks failures in the data (e.g., `failures` in Example 1). The highly-correlated attributes are output as the single diagnosis vector in the query result. Note that this technique does not partition the data into one or more clusters; the entire data is considered to be a single cluster. We did not consider the extended approach in [9]—which first partitions $H \cup F$ into windows of fixed size W , and then applies metric attribution per window—because there is no principled approach for choosing the window size W .
- The CART-based diagnosis approach [7] learns a decision tree T from $H \cup F$, and then outputs the attributes used in the top-level decision nodes in T as part of the single diagnosis vector. Partitioning H into clusters is not considered in [7].
- The baselining-based approach described in [2] first captures the distribution D_{A_i} of each attribute A_i , $1 \leq i \leq n$, using the historic data H . The distribution can be approximated by a Gaussian distribution or using *Kernel Density Estimation (KDE)* [15]. Then, [2] computes the probability of D_{A_i} having produced the measurements of A_i in F . If this probability is low, then A_i is included in the output diagnosis vector with a weight equal to the inverse of this probability.

The experimental results on the Dolphin dataset are summarized in Table B. For the metric attribution, CART-based diagnosis, and baselining-based approaches, we include the five most relevant metrics from the output diagnosis vector. Note that the results show a clear advantage of our PCM-E algorithm over the previous approaches in terms of diagnosis accuracy.