Guided Problem Diagnosis through Active Learning

Songyun Duan and Shivnath Babu
Department of Computer Science, Duke University, Durham, NC, USA
{syduan,shivnath}@cs.duke.edu

Abstract

There is widespread interest today in developing tools that can diagnose the cause of a system failure accurately and efficiently based on monitoring data collected from the system. Over time, the system monitoring data will contain two types of failure data: (i) annotated failure data L, which is monitoring data collected from failure states of the system, where the cause of failure has been diagnosed and attached as annotations with the data; and (ii) unannotated failure data U. Previous work on wholly- or partially-automated diagnosis focused on L or U in isolation. In this paper, we argue that it is important to consider both L and U together to improve the overall accuracy of diagnosis; and in particular, to proactively move instances from U to L. However, such movement requires manual diagnosis effort from system administrators. Since manual diagnosis is expensive and time-consuming, we propose an algorithm to make the best use of manual effort while maximizing the benefit gained from newly diagnosed instances. We report an experimental evaluation of our algorithm using data from a variety of failures—both single failures and multiple correlated failures—injected in a testbed, as well as with 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 slowdowns, and others. Walmart.com was unavailable for almost 10 hours during the peak U.S. 2006 holiday season. Such deviation of systems or applications 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, otherwise, 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 mistakes by system administrators. The increasing scale, complexity, and dynamics of modern systems is making it harder than ever to track down the cause of failures manually [7, 12, 15].

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 [12]. 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 interest in wholly- or partially-automated tools that can diagnose the cause of a system failure accurately and efficiently. The lack of such tools will continue to raise the total cost of system ownership, which can be as high as 18 times the original purchase price [15].

When a system experiences a failure, a system administrator (*sysadmin*) or system-management software would want to diagnose the cause of the failure quickly. We are building a system called *Fa* to aid this process [11]. Fa's goal is to automate, as much as possible, the process of diagnosing system failures based on monitoring data collected continuously from the system; and if possible, to bring the system back to a working state automatically (*self-healing*).

System Monitoring Data: When a system is running, Fa collects monitoring data periodically and stores it in a database. 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. Most database servers maintain performance counters (e.g., number of index updates, number of full table scans) that Fa reads periodically. Fa collects 100-300 such metrics periodically from the systems that it monitors.

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.

	cpu_util	num_io	failures	Annotation
h1	51.3	76.1	0	
h2	48.5	63.6	0	
h3	51.9	97.9	0	
h4	49.0	43.6	0	
h5	72.4	65.4	0	
h6	50.8	51.2	0	
h7	49.8	119.5	0	
h8	49.3	141.2	0	
h9	50.4	13.5	0	
h10	70.4	43.5	1	Problem 1
h11	82.6	33.5	1	?
h12	20.4	73.8	1	?
h13	73.5	23.2	1	Problem 2

(a)

	cpu_util	num_io	failures	Annotation
f1	70.0	80.7	1	?
f2	71.9	85.6	1	?

(b)

Figure 1. Sample data

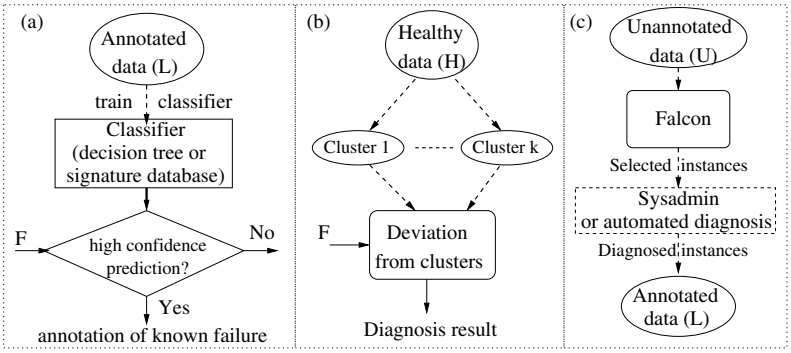

Figure 2. Different phases of diagnosis in Fa

- *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. The annotation corresponding to a failure may also include information on how the failure can be fixed automatically to recover the system back to a healthy state; such annotations enable self-healing [9].

Example 1.1 *Figure 1(a) shows the historic data for a database server collected by monitoring the server at one-minute intervals. In each interval, attribute* `cpu_util` *is the average server CPU utilization;* `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 failure that has been diagnosed. In this historic dataset, healthy data* H *consists of instances* $h_1, \ldots, h_9$*, annotated data* L *consists of failure instances* h_{10} *and* h_{13}*, and unannotated data* U *consists of failure instances* h_{11} *and* h_{12}*.*

Failure Diagnosis using Monitoring Data: When the system experiences a failure, Fa provides support for diagnosing the cause of the failure based on two sets of data:

- Monitoring data F collected from the system during the failure (or just before the failure in the case of a system crash).
- All historic data collected so far, namely, $H \cup U \cup L$. ($\cup$ denotes the union operator.)

Example 1.2 *Figure 1(b) shows recent monitoring data* F *from the same server as in Example 1.1. The values of the* `failures` *attribute in* F *indicate that the server is experiencing some type of failure. To diagnose the cause of this failure, Fa will use* F *as well as the full historic data* $H \cup U \cup L$ *from Figure 1(a).*

To diagnose the cause of the failure represented by a set of instances F, a sysadmin or system-management software poses a *diagnosis query* to Fa of the form Q=*Diagnose*$(F, H \cup U \cup L)$. Fa will process this query in two phases, as illustrated in Figure 2.

- In Phase I (Figure 2(a)), Fa determines 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=*Diagnose*(F, L). If the diagnosis result produced by this phase has high confidence, then Fa returns this result to the issuer of the query; otherwise Fa goes to Phase II of diagnosis.

- In Phase II (Figure 2(b)), Fa compares F with the healthy data H collected so far, to see whether the cause of the failure can be characterized succinctly as deviation of F from the data representing the healthy states of the system. That is, Phase II translates Q to Q_{II}=*Diagnose*(F, H).

Phase I of Diagnosis: Many techniques have been proposed in the literature to implement Phase I which essentially is a *multi-class classification* task [22]. (A survey is given in [9].) Each instance t in L can be viewed as a record that belongs to a unique *class*. In our setting, t's class is its annotation which is determined by the cause of the failure that t comes from. Given the instances in L and their respective annotations, Phase I's task is to predict the correct annotation for F.

The general approach to perform Phase I is to train a *classifier* C [22] from the current set of annotated instances L. C can then be used to predict F's class. A popular classifier used in the literature (e.g., [6]) is a decision tree. The nonleaf nodes in the tree define partitioning conditions on selected attributes in the data (e.g., `cpu_util` > 60). Each

leaf node N is associated with an annotation that is predicted for all instances that satisfy the conditions along the path from the root node to N.

Another popular classifier is a *signature database* (e.g., [4, 9]) that contains a unique signature for each distinct type of failure in L. Intuitively, the signature for a failure type captures the unique set of symptoms for that type. F is compared with each signature in the database. If there is a strong match between F and a particular signature s, then F represents the same failure as s with high confidence.

Phase I leverages previous diagnosis efforts, which is valuable because diagnosis of failures is laborious and expensive in complex, large-scale systems. Furthermore, previous studies indicate that most failures in practice tend to have occurred before. Reference [4] reports that typically half, and sometimes as much as 90%, of all software problems reported by users today are recurrences of known problems, i.e., those whose cause has already been ascertained or is under investigation. At the same time, Phase I will be effective only if failure instances appearing in U are annotated and moved to L in a timely fashion; thereby making L sufficiently comprehensive.

Phase II of Diagnosis: If F corresponds to a previously-unseen or previously-undiagnosed failure, then Phase I will not be able to do a diagnosis with high confidence. In this situation, Phase II of Fa attempts to diagnose F based on the healthy data H. Figure 2(b) illustrates the basic idea, and the details are in [2]. The instances in H are first grouped into clusters such that each cluster represents a distinct healthy state of the system. Fa then characterizes the deviation of F from these healthy states to pinpoint the probable cause of the failure. Similar approaches have been proposed by other researchers, e.g., [3, 8].

While both Phase I and Phase II have their pros and cons, Phase II has a harder task than Phase I. Consequently, diagnosis results from Phase II tend to be less accurate than those from Phase I. In machine-learning terminology, Phase I is a *supervised learning* task and Phase II is an *unsupervised learning* task [22]. In supervised learning, the training data consists of pairs of input objects (L in Fa) and the corresponding outputs (annotations in Fa). The goal is to create a function (e.g., a classifier like decision tree) that can predict the output for any legal input. However, in unsupervised learning, the training data contains input objects only—the outputs are unknown—and the goal is to learn a model that fits the input (e.g., a clustering of the input).

Recall that our monitoring datasets contain 100-300 attributes. Supervised learning often has an accuracy advantage over unsupervised learning on such high-dimensional datasets. Reference [9] empirically evaluates techniques for Phase I and Phase II and comes up with similar observations. For example, Phase II is shown to be prone to misdiagnosis under multiple correlated failures.

A New Background Phase III: The above observations about Phases I and II motivated us to consider a new phase in Fa where we move data actively from U to L, with the goal of increasing the accuracy and coverage of Phase I with the least manual effort. Figure 2(c) illustrates this step. We can move a failure instance t from U to L after annotating t with the cause of the failure that it represents. To get the correct annotation for t, we can leverage the manual diagnosis efforts of sysadmins.

Phase III is implemented using a new algorithm, called *Falcon*, that can select some instances u from U, and pose an *annotation query* to the sysadmin of the form: What are the annotations for the instances in u? To answer this annotation query, the sysadmin will have to actually diagnose the cause of the failure represented by u. She can take the help of Fa's Phase II for this purpose. If the sysadmin is able to respond back with the annotations for u, then Falcon will remove u from U, and add u and its annotations to L. Otherwise, these instances are left in U. Falcon then iterates by selecting a new set of instances from U, and posing a new annotation query to the sysadmin.

Phase III can be run continuously in the background as the system executes, or it can be invoked on demand by the sysadmin—e.g., when she has the time to do more diagnosis, or when she feels that the classifier trained from the current L needs to be improved. As new annotated instances appear in L, the classifier used by Phase I can be retrained on the new L to potentially improve its accuracy and coverage. The main challenge we need to address in Phase III is to design the sequence of annotation queries posed to the sysadmin. Since diagnosis is expensive and time-consuming, our goal is to make the best use of manual diagnosis efforts while maximizing the information gained from the newly diagnosed instances. Addressing this challenge is the focus of this paper.

This rest of this paper is organized as follows:

- Section 2 discusses guidelines that algorithms for Phase III should meet. We then give an overview of how Falcon adheres to these guidelines.
- Sections 3 and 4 describe the components of Falcon.
- Section 5 reports an experimental evaluation based on failures injected in a testbed, as well as synthetic data.
- Section 6 reports related work, and Section 7 concludes.

2 Preliminaries and Overview

2.1 Guidelines for Phase III

Recall from Section 1 that Fa's Phase III poses a sequence of annotation queries to the sysadmin. For efficiency and ease of use, we require this sequence to adhere to three guidelines G_1, G_2, and G_3 that we discuss next.

Guideline G_1: Each individual annotation query posed to a sysadmin should contain *multiple* instances belonging to a

single type of failure. The intuition behind this guideline is that it will be hard for a sysadmin to diagnose the cause of a failure from a single instance of monitoring data. Multiple distinct instances per failure make it easier to spot patterns both manually as well as when Fa's Phase II is used [2]. It is even more important to ensure (as much as possible) that the instances in an annotation query correspond to the same type of failure. A query that mixes instances from an assorted set of failures will easily confuse sysadmins. The consequences can be higher cost and labor for diagnosis, higher chances of misdiagnosis, and subsequent loss of faith in the usefulness of Phase III.

Guideline G_2: The instances selected in each annotation query should be sufficiently different from the existing annotated instances in L. Adhering to this guideline ensures that manual diagnosis efforts are not duplicated needlessly.

Guideline G_3: The instances selected in each annotation query should be representative of the failures seen in the system that are not covered by the existing annotated instances in L. Adhering to this guideline ensures that manual diagnosis efforts are spent on failures actually seen in the production system.

2.2 Overview of Falcon

Our goal is to design an algorithm that adheres to all three guidelines. Guidelines G_2 and G_3 can be met using techniques for *active learning* from supervised machine learning [14]. In conventional supervised learning, a classifier C is trained on a pool of instances that are all annotated. C can then be used to predict annotations for instances with unknown annotations. Active learning is used when the training pool consists largely of unannotated instances for which getting annotations are costly. This approach has been applied widely, e.g., in text and image classification, speech recognition, and software testing [14].

Starting with a small set of annotated instances, an *active learner* searches the unannotated pool for instances that provide useful information in creating an accurate classifier. Once an unannotated instance t is chosen, a request is made to a human expert (in general, an oracle) to provide the correct annotation for t. The expert annotates t at some cost, and the classifier is retrained on the new set of annotated instances. Based on the newly gained information, the active learner searches the unannotated pool again; and the process repeats.

Notice that an active learner is exactly what we need to implement Phase III. However, conventional active learners (described in Section 3) adhere to guidelines G_2 and G_3, but not to G_1. The complication posed by G_1 is that the cost incurred by the sysadmin to answer an annotation query is very high if the instances belong to more than one type of failure. Falcon[1], illustrated in Figure 3, addresses this issue.

[1]Fa's Active Learning with Clustering Online

Algorithm *Falcon* /* Fa's implementation of diagnosis Phase III */
1. Let L be the current set of annotated failure instances, and U be the current set of unannotated failure instances;
   /* Clustering step to adhere to guideline G_1 */
2. Group instances in U into a minimal set of clusters where each cluster has instances of same failure type with high probability;
   /* Use of active learning to adhere to guidelines G_2 and G_3 */
3. Use an active learner to pick one cluster from the set of clusters generated in the previous step;
4. Pick some $k \geq 1$ instances from the chosen cluster in U to pose an annotation query to the sysadmin;
5. Move the annotated instances returned by the sysadmin, if any, from U to L. Update the classifier trained from L;
6. Go to Step 1;

Figure 3. Outline of our Falcon algorithm

Falcon proceeds in iterations where each iteration picks an annotation query—i.e., a set of unannotated instances from U—that is posed to the sysadmin for annotation. Falcon adheres to guideline G_1 by ensuring that each query contains multiple instances that all correspond to the same type of failure with high probability. To meet this requirement, Falcon first groups instances in U into clusters such that instances from the same failure type go into the same cluster. Section 4 describes Falcon's clustering techniques.

Once the clusters are generated, Falcon uses an active learner to pick one cluster from which all instances in the current annotation query will be chosen. This step requires some modifications to conventional active learners which are designed to pick one instance from a pool of unannotated instances, rather than one cluster from a pool of clusters. In Section 3, we describe some conventional active learners and our extensions that enable Falcon to adhere to guidelines G_2 and G_3.

After picking a cluster, Falcon has to decide which subset of instances from this cluster to include in the annotation query posed to the sysadmin. This decision is discussed in Section 4. The response given by the sysadmin will be annotations for some subset of the queried instances. This subset could range from all queried instances to an empty set. The cost incurred by the sysadmin for finding the annotations adds to the overall cost of Falcon so far. The newly annotated instances will be added to L, and the classifier that Phase I trains from L will be updated. Falcon then proceeds to design the next annotation query.

The next two sections present the details of each step of Falcon in Figure 3.

3 Active Learners

In this section, we first describe three popular active learners from the machine-learning literature. We will then describe how Falcon adapts these learners to also adhere to guideline G_1 from Section 2.1. Conventional active learners

are best described by where they are positioned in the classical "exploration" Vs. "exploitation" spectrum in machine learning [14].

A popular active learner that we consider in Falcon is called the *least-confidence learner (LC)* (or uncertainty sampling) [14]. The unannotated instance $t \in U$ that LC will pick for manual annotation next is the one on which the classifier C trained from the current L is least confident about the true annotation. A generic way to quantify the confidence in C's prediction of t's annotation is to measure how close t is to a *decision boundary* in C. (Intuitively, each side of a decision boundary in a classifier will give a different prediction of t's annotation.) The closer t is to a decision boundary in C, the less C is confident about its prediction of t's true annotation; hence, the larger the uncertainty in t.

LC is good at exploitation—namely, acquiring annotations for instances near decision boundaries so that the boundaries can be refined—but, it does not conduct exploration where the goal is to acquire annotations for instances so as to create new decision boundaries if required. Pure exploitation will not find regions of the input space that contain many unannotated instances for which the current classifier learned from L predicts the true annotation incorrectly. Exploration searches for such regions. A popular active learner in the exploration category that we consider in Falcon is called *Kernel Furthest First*; discussed in Section 3.

The third type of active learner that we consider balances exploration and exploitation by defining a probability—varied suitably over time—of choosing whether to explore or exploit whenever an annotation query has to be chosen [14]. The rest of this section gives the details of how we implemented these three active learners in Fa.

3.1 Least-Confidence Sampling (LC)

LC first learns a classifier C from the current set of annotated failure instances L. For each instance $t \in U$, LC then uses C for two things: (i) predicting t's (unknown) annotation; and (ii) estimating the confidence in this prediction. The details of confidence estimation are specific to the type of classifier we train from L, and works as follows for the two types of classifiers discussed in Section 1:

- While predicting the annotation of an instance $t \in U$, a decision tree classifier can compute the probability of t having each possible annotation from the space of all annotations (i.e., failure types). The annotation predicted for t will be the one with the highest probability. The confidence in this prediction is the difference in probabilities between the most probable annotation and the second most probable annotation.
- A signature-based classifier will predict t's annotation to be the same as that of the signature whose distance to t is minimum, i.e., t's *nearest neighbor* in the signature database. (The distance metrics we consider will be defined momentarily.) The confidence in this prediction is $d_2 - d_1$, where d_1 is t's distance to its nearest neighbor in the signature database, and d_2 is t's distance to its second nearest neighbor in the database.

Notice that both the above differences estimate the distance to a decision boundary. When LC has to pick k, $k \geq 1$, instances to pose an annotation query to the sysadmin, it will pick the k instances from U whose predictions have the lowest confidence. Ties are broken randomly.

3.2 Kernel-Furthest-First Learner (KFF)

For each instance $t \in U$, KFF computes t's distance to its nearest neighbor in L, i.e., the instance in L that t is closest to among all instances in L. A popular metric for estimating distances is the L_2 norm (or Euclidean distance). The L_2 norm for a pair of instances t and t' is $\sqrt{\sum_{i=1}^{n}(t.A_i - t'.A_i)^2}$, where $A_1, \ldots, A_n$ are the data attributes in each instance. Another distance metric, which we use by default, is the *cosine distance*: $1 - \cos(\theta)$. Here, θ is the angle between instances t and t' treated as vectors. $\cos(\theta) = \frac{\langle t,t' \rangle}{||t||\ ||t'||}$, i.e., the inner product of t and t' normalized by the product of their lengths. When KFF has to pick k, $k \geq 1$, instances to pose an annotation query to the sysadmin, it will pick the k instances from U with the largest distance to their nearest neighbor in L. Ties are broken randomly.

3.3 Hybrid Learner (Hybrid)

Whenever an annotation query has to be chosen, Hybrid decides whether to do an exploration with probability p, or to do an exploitation with probability 1-p [14]. KFF is used if exploration is chosen, and LC is used if exploitation is chosen. A simple option is to use a fixed p. Reference [14] describes a better approach which we implemented as Hybrid in Falcon. Hybrid varies p dynamically such that p is high initially, and p is reduced gradually as the classifier trained from L becomes more accurate. After each exploration step, Hybrid estimates how "successful" this step was. Intuitively, if the exploration was successful, then p should be kept high; otherwise it should be reduced.

Let C_b and C_a be the classifiers trained from the set of annotated instances L before and after an exploration step. Let V_b (V_a) be the vector containing the annotations predicted by C_b (C_a) for the instances in $L \cup U$. If V_a is significantly different from V_b—which can be computed using the distance metrics from Section 3.2—then Hybrid estimates

that the exploration was successful; otherwise it reduces p. Intuitively, the measure of success of an exploration step is the magnitude of change produced in the predictions of the classifier trained from L. The full details of Hybrid are given in [14].

3.4 How Falcon Uses an Active Learner

The conventional active learners discussed so far in this section work at the level of individual instances in U. However, recall from Section 2.1 that guideline G_1 requires Falcon to work at the level of clusters in U, where each cluster contains instances of the same failure type with high probability. When LC is used as the active learner, Falcon will compute the confidence of each cluster as the average confidence across all instances in that cluster. The cluster with the least confidence is chosen for the annotation query in Step 3 of Falcon in Figure 3.

A similar approach is used for KFF. Here, for each cluster, Falcon will compute the average, over all instances in the cluster, of the distance to the nearest neighbor in L. The cluster with the largest average distance will be chosen for the annotation query. Note that Hybrid uses one of LC or KFF in each iteration.

4 Clustering in Falcon

We first considered conventional *distance-based* clustering techniques to group instances in U into clusters that contain instances of the same failure type with high probability. However, these techniques performed poorly, so we developed a new technique that we call *time-based chunking*. Both techniques are described next.

4.1 Distance-based Clustering

We will give a brief description of *K-means* which is one of the most commonly-used distance-based clustering techniques [22]. K-means uses an iterative refinement algorithm that starts by partitioning the input instances in U into k initial sets, either at random or using some heuristic. It then calculates the mean instance, or *centroid*, of each set; and constructs a new partition of the instances in U by associating each instance with its closest centroid. Any of the distance metrics from Section 3.2 can be used for measuring the distance between instances. The centroids are recalculated for the new clusters, and the algorithm is repeated by alternate application of these two steps until the instances no longer switch clusters or the centroids no longer change.

Falcon's active learner will pick one of the clusters generated by K-means, as discussed in Section 3.4. The annotation query posed to the sysadmin will consist of one *representative instance* R, which is the centroid of this cluster, and $k-1$ *supporting instances*, which form a random sample of $k-1$ instances from this cluster.

As we report in Section 5, K-means performed poorly when used in Falcon. Recall that the monitoring datasets collected by Fa contain 100-300 attributes, i.e., these datasets are high dimensional. Distance-based clustering suffers from the *curse of dimensionality* in high dimensional spaces [10]. For any pair of instances in such spaces, it is highly likely that there are some attributes on which the instances are highly distant from each other. Thus, the clusters generated by K-means from U tend to be impure in that they contain instances from different failure types.

4.2 Time-based Chunking

We developed a different technique to address the problems with distance-based clustering. In the system management domain, it is reasonable to expect a strong time-based correlation among annotation values. That is, it is more likely that two failure instances that are close together in time belong to the same failure type, compared to two failure instances that are distant in time. This property can be leveraged while clustering instances in U. However, the challenge that we need to solve is how to identify *change-points* in U—where one type of failure finishes, and another type of failure starts.

Often, there are external indicators that make it easy to detect change-points. For example, instances from two different types of failure may be separated in time by a long intermediate phase where the system was in a healthy state. While such external indicators are useful, we cannot rely solely on such indicators to identify all change-points. For example, a type of failure may be workload dependent, causing failure instances to be interleaved naturally with healthy instances. We have developed a technique to identify change-points that can leverage external indicators where available, but does not depend on them.

The basic idea behind our technique is to use patterns in the confidence estimates of instances in U that are close together in time. As discussed in Section 3.1, these confidence estimates are generated by first training a classifier C on the current set of annotated instances L, and then using C to predict the annotation and associated confidence for each instance $t \in U$.

Figure 4 illustrates the type of patterns we hope to see, namely, the values of confidence estimates for instances of the same failure that are contiguous in time are relatively close to one another (compared to the confidence estimates for instances of other failure types). The x axis in Figure 4 corresponds to the instances in U laid out in increasing order of timestamp. These instances are from a real experiment, and were generated in our testbed by injecting different failure types at different points of time. The top graph in the figure shows the actual change-points in the data that we generated by changing the type of failure injected.

A small random sample of the failure data was used as

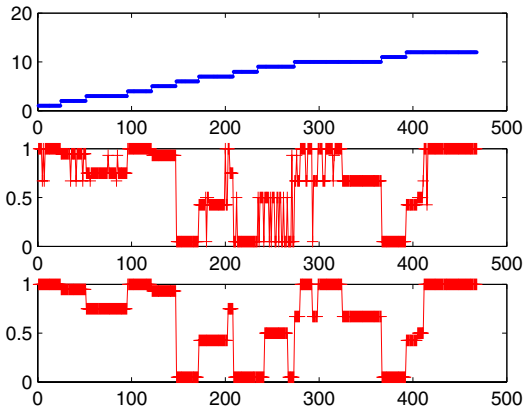

Figure 4. The topmost figure shows the actual change-points. The middle and bottom figures show estimated confidence values before and after time-based chunking.

the current set of annotated instances L, and a decision tree classifier C was trained. The middle graph in Figure 4 shows the confidence estimate from C for each respective instance in the top graph. Note that for most failure types, the majority of confidence estimates for instances of that type tend to fall within a small range of one another. Thus, most of the actual change-points in the data can be captured by change-points in the confidence-estimate plot. This idea forms the crux of time-based chunking.

Time-based chunking can be implemented in many ways, e.g., using recent change-point detection techniques like [1, 13]. We have implemented a relatively straightforward technique that has worked satisfactorily so far. We scan the instances in U in increasing order of timestamp, grouping the instances into chunks as follows. Let N be the current chunk, and let e be the confidence estimate of the earliest instance in this chunk in our scan. The scan will close chunk N, and start a new chunk, when it finds an interval that contains many instances whose confidence estimates fall outside $[e-\delta, e+\delta]$. Here, δ is a user-specified constant. Thus, the confidence estimates of all instances in chunk N will lie in $[e-\delta, e+\delta]$, and will be approximated as e. The bottom graph in Figure 4 shows the chunks generated by our technique, which are reasonably accurate.

Once Falcon's active learner picks one of the generated chunks (as discussed in Section 3.4), the annotation query posed to the sysadmin will consist of one representative instance R, which is the instance at the center (in time) of this chunk, and $k-1$ supporting instances around R that belong to the same chunk.

5 Experiments

Our experiments are run with monitoring data from a controlled testbed developed using software from the Berkeley/Stanford Recovery-Oriented Computing (ROC) project [16]. The testbed runs a multitier Web service named *Rubis* [18]—an auction service modeled on eBay—running on a JBoss application server (with an embedded Web server) and a MySQL database server. The monitoring data we collect primarily includes the number of procedure invocations per minute of various Java modules (*Java beans*) in the application server while the Web service is in operation. We employ a failure-injection tool [5] to systematically inject failures into Rubis and JBoss. The types of failures injected in our experiments include Java exceptions, message drops, deadlocks, JNDI corruptions, data corruptions, memory leaks, and infinite loops. In addition to single independent failures, we also injected multiple correlated failures because such failures are common in large-scale production systems [5]. The testbed is run on a machine with 1 GHz CPU and 1 GB memory.

Name	#Attributes	#Instances	#Distinct failures
1. Rubis-1	105	1334	9
2. Rubis-2	105	1796	14
3. Synthetic	10	528	11

Table 1. Monitoring datasets

Monitoring Datasets: Table 1 summarizes the monitoring datasets generated using failure injection. Rubis-1 contains independent failures and two-way correlated failures. Rubis-2 contains independent failures as well as two-way, three-way, and four-way correlated failures. As we have knowledge of the failure type injected, each failure instance in Rubis-1 and Rubis-2 is annotated with its actual cause; providing ground truth for evaluating the diagnosis accuracy of our techniques. To test our approach on datasets with complex patterns, we also consider a synthetic multiclass dataset (which is actually the VOWEL dataset from the UCI machine-learning repository [21]).

In each monitoring dataset, 30% of the instances are used as the independent test data to compute the accuracy of the classifier trained from the current set of annotated instances in L. The rest of the instances appear in the pool of unannotated instances U. We randomly pick 20% of the instances from U and add their annotations to generate the initial annotated dataset L.

We consider two experimental settings: (I) all the failure types that appear in U also appear in the initial annotated data L, and (II) some types of failure that appear in U are missing from the initial annotated data L. For setting II, the fraction of known failure types in the initial L is 75% of all the failure types in $L \cup U$.

Annotation Query: Recall from Section 4 that each annotation query poses one representative instance R and $k-1$ supporting instances for the sysadmin to diagnose; $k = 10$ in our experiments. In response, we assume that the sysadmin annotates the representative instance R, and also all

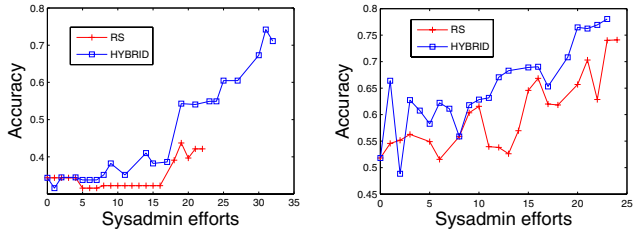

Figure 5. Setting I: (a) Rubis-1, (b) Rubis-2

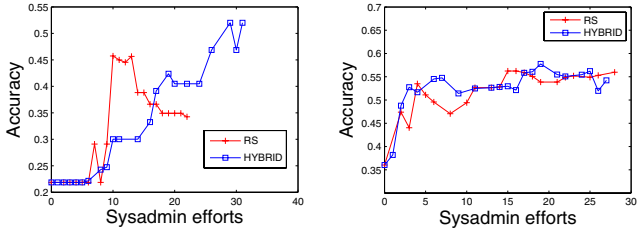

Figure 6. Setting II: (a) Rubis-1, (b) Rubis-2

supporting instances that have the same annotation as R.

Sysadmin Effort (Diagnosis Cost): We approximate the manual diagnosis effort required to answer an annotation query as the number of distinct failure types among the k instances in the query. The intuition here is that sysadmins may be misled while diagnosing the failure type represented by instance R if the supporting instances belong to one or more different failure types. Thus, it is highly desirable to submit a query with k instances from the same failure type.

Algorithms and Defaults: The Falcon algorithm can be implemented with various combinations of clustering algorithms (K-means or time-based chunking) and active learners (LC, KFF, or Hybrid). Decision trees are our default classifier since the decision paths from root to leaf nodes help understand and verify the diagnosis results [6]. We set 10 as the default for the number of clusters in K-means clustering. The combination of time-based chunking and hybrid learner is our default strategy for the Falcon algorithm.

Evaluation Metrics: The accuracy metric preferred for classifiers in the system management domain is called *balanced accuracy* [17]. Suppose there are N types of annotations, and for each annotation A_i, the classifier makes M_i' correct predictions for the M_i instances in the test data that have annotation A_i. Then, the balanced accuracy is $\frac{1}{N} * \sum_{i=1}^{N} \frac{M_i'}{M_i}$. In all the plots in this section, points on the x-axis record the cumulative diagnosis effort of sysadmins for the annotation queries submitted so far. The corresponding value on the y-axis records the balanced accuracy of the current classifier (i.e., the classifier trained on the current L) on the test data. The maximum number of annotation queries in our experiments is set to 30. A good algorithm will enable the classifier to achieve high balanced accuracy at minimal diagnosis effort from sysadmins.

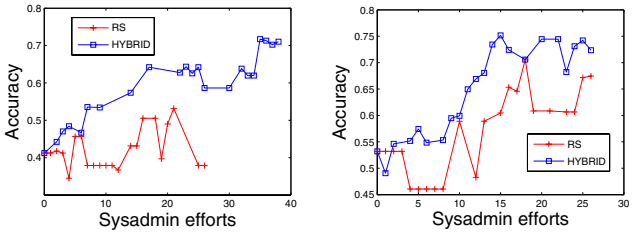

Figure 7. Setting I: (a) Unbalanced Rubis-1, (b) Unbalanced Rubis-2

5.1 End-to-End Validation: Falcon Vs. Random Sampling

We first compare our Falcon algorithm using the default active learner (Hybrid) with random sampling (RS). RS generates annotation queries as follows: a representative instance is picked randomly from U, and its $k-1$ neighbors in time are used as supporting instances.

Figures 5 and 6 plot the performance of Falcon and random sampling in experimental settings I and II respectively. In setting I, it is clear that (i) Falcon requires less diagnosis effort from sysadmins than random sampling, and (ii) Falcon achieves much better balanced accuracy for the same number of annotation queries. In setting II, we do not have information about the failure types not seen in L. Thus, it requires some exploration effort to get annotated instances for failures not seen in L, in order to improve L's coverage. Random sampling is good at exploration, hence it performs comparable to Falcon for Rubis-2 in setting II.

It is possible that failure instances are *unbalanced* in production environments. That is, some types of failure may occur more commonly or persist for longer periods than other types. To create this situation, we replicate instances from some failure types in our monitoring data, thereby making $L \cup U$ unbalanced. Figures 7 and 8 compare Falcon with random sampling in the two settings. It is clear that Falcon is now consistently better than random sampling: random sampling can keep picking instances with one or more frequent failure types, say A_i, although the extra information contained in the newly annotated instances of failure A_i drops significantly.

These experiments show that Falcon can perform significantly better than simple strategies for guiding the diagnosis efforts of sysadmins.

5.2 Comparing Clustering Methods

Figures 9 and 10 compare time-based chunking (CHUNK) with K-means clustering when they are combined with an active learner (LC) in our two experimental settings. Note that time-based chunking is much better than K-means clustering. Our monitoring datasets contain a large number of attributes (105 attributes in Rubis-1) or have complex patterns (Synthetic). Distance-based cluster-

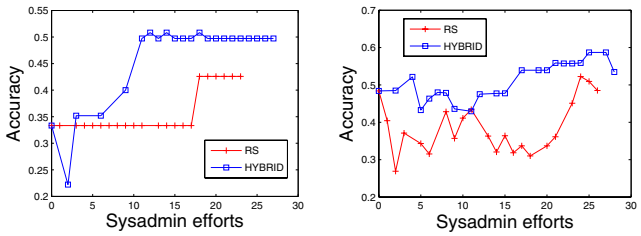

Figure 8. Setting II: (a) Unbalanced Rubis-1, (b) Unbalanced Rubis-2

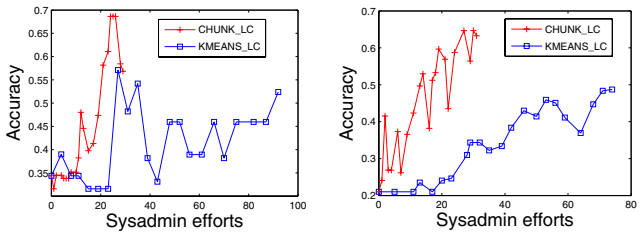

Figure 9. Setting I: (a) Rubis-1, (b) Synthetic

ing like K-means puts instances from different failures into the same cluster. Therefore, the instances from any cluster picked by the active learner tend to contain several distinct failure types, incurring high diagnosis cost per annotation query because of the extra burden placed on the sysadmin.

5.3 Comparing Active Learners

We now compare the three active learners described in Section 2, namely, LC, KFF, and Hybrid. Figures 11 and 12 compare these active learners for datasets Rubis-2 and Synthetic in the two respective settings. Figure 13 considers Setting II for Rubis-1 in the regular and unbalanced cases. The results are along expected lines: Hybrid is able to match up to the best of LC and KFF in each setting. For example, KFF performs very well (and Hybrid matches it) in Figure 13(b) because exploration is important in this setting.

6 Related Work

There has been plenty of previous work on wholly- or partially-automated techniques for diagnosing performance and availability problems in systems. However, previous techniques tend to focus on one of what we identified in Section 1 as Phases I and II of diagnosis. To the best of our knowledge, ours is the first work to focus on Phase III, where the goal is to make the best use of the manual diagnosis efforts of system administrators while maximizing the information gained from the newly diagnosed instances.

References [4] and [24] are recent examples of work on Phase I that build different forms of classifiers to map current failures to previously-diagnosed failures. There has also been work on constructing signatures to characterize different system states [8, 23]. Reference [8] extracts indexable signatures from system states, which are characterized by correlations between low-level system metrics and

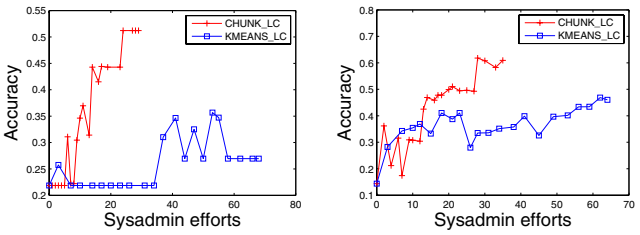

Figure 10. Setting II: (a) Rubis-1, (b) Synthetic

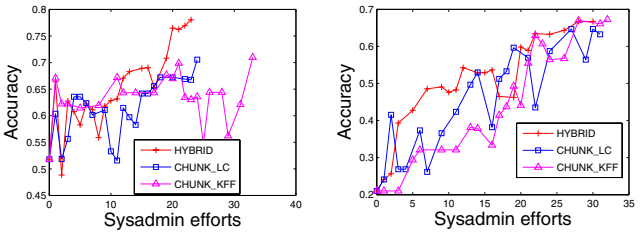

Figure 11. Setting I: (a) Rubis-2 (b) Synthetic

the overall performance metric. From these signatures, a searchable database of historical system states can be created to identify recurrent problems. Reference [23] utilizes signatures to represent correlations between failures and their symptoms in networked systems.

Previous work on automated or semi-automated diagnosis based on unannotated monitoring data (i.e., Phase II) predominantly takes one of the *correlation-based* or *baselining-based* approaches. Recent examples of the correlation-based approach include [6, 7]. [6] applies decision-tree learning techniques to rank different system components based on their correlation with system failures. [7] applies Bayesian-network learning techniques to correlate performance metrics with high-level system behavior. Reference [3] is a recent example of the baselining-based approach where a heuristic is proposed to capture and represent the baseline behavior of a Web service; and two techniques—one based on the χ^2 statistical test, and another based on naive Bayesian networks—are proposed to detect and categorize deviation from the baseline behavior. In [2], we describe a new clustering algorithm that pays particular attention to the failure data while clustering the healthy data; see Figure 2(b). Reference [9] empirically evaluates techniques for Phase I and Phase II, and points out many hurdles that Phase II faces (e.g., failure propagation).

Active learning and change-point detection are two techniques that we leverage in our algorithm for Phase III. Reference [14] contains a survey of existing active learners, including detailed descriptions of the active learners that Falcon uses. References [1] and [13] are recent techniques for change-point detection that Falcon could leverage in future.

7 Conclusions

In this paper, we showed that diagnosis of failures based on system monitoring data consists of multiple phases.

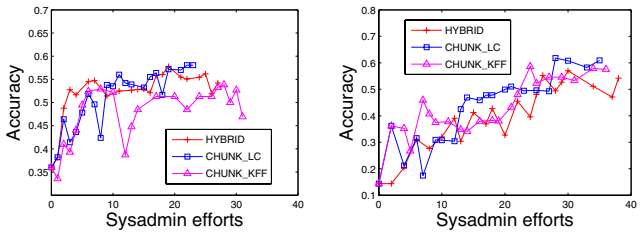

Figure 12. Setting II: (a) Rubis-2, (b) Synthetic

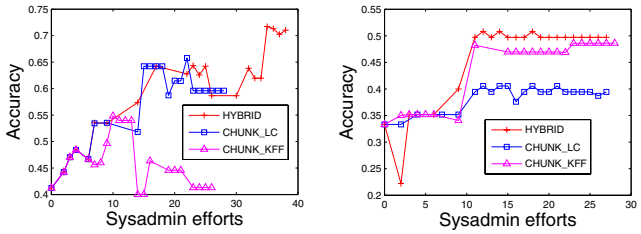

Figure 13. Setting II: (a) Rubis-1, (b) Unbalanced Rubis-1

Phase I of diagnosis uses annotated failure data L, which is monitoring data collected from failure states of the system where the cause of failure has been diagnosed and attached as annotations with the data. Previous work has shown that this phase is extremely valuable—because failures often reoccur—and accurate. However, this phase can be effective only if unannotated failure instances are diagnosed accurately, and moved to L in a timely fashion. Such movement requires manual diagnosis effort from system administrators. Since manual diagnosis is expensive and time-consuming, we proposed an algorithm to make the best use of manual effort while maximizing the benefit gained from newly diagnosed instances. An experimental evaluation using data from a variety of failures—both single failures and multiple correlated failures—injected in a testbed, and with synthetic data, showed the effectiveness of our algorithm.

References

[1] C. C. Aggarwal. A framework for change diagnosis of data streams. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003.

[2] S. Babu, S. Duan, and K. Munagala. Processing diagnosis queries: A principled and scalable approach (poster). In *Proc. of the Intl. Conf. on Data Engineering*, Apr. 2008.

[3] P. Bodik et al. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proc. of IEEE Intl. Conf. on Autonomic Computing*, 2005.

[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 *Proc. of IEEE Intl. Conf. on Autonomic Computing*, June 2005.

[5] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic Failure-Path Inference: A Generic Introspection Technique for Internet Applications. In *Proc. of 3rd IEEE Workshop on Internet Applications*, June 2003.

[6] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Proc. of the First IEEE Intl. Conf. on Autonomic Computing*, June 2004.

[7] I. Cohen et al. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. of Operating Systems Design and Implementation*, Dec. 2004.

[8] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, Indexing, Clustering, and Retrieving System History. In *Proc. of the ACM Symp. on Operating Systems Principles*, Oct. 2005.

[9] B. Cook. Towards self-healing multitier web services. M.S. thesis, Duke University, 2007.

[10] C. Domeniconi, D. Gunopulos, S. Ma, B. Yan, M. Al-Razgan, and D. Papadopoulos. Locally adaptive metrics for clustering high dimensional data. *Data Mining and Knowledge Discovery*, 14(1), 2007.

[11] S. Duan and S. Babu. Proactive identification of performance problems. In *Proc. of the 2006 ACM SIGMOD Intl. Conf. on Management of Data*, June 2006. Demonstration.

[12] P. Horn. Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Corp., 2001. http://www.research.ibm.com/autonomic.

[13] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, Sept. 2004.

[14] T. Osugi. Exploration-based active machine learning. M.S. thesis, University of Nebraska, 2005.

[15] *The RADical Approach to Next-Generation Information Services: Reliable, Adaptive, Distributed*. Invited talk by David Patterson, University of California, Berkeley, at ACM SIGMOD 2006.

[16] D. Patterson et al. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical report, UC Berkeley Computer Science, UCB//CSD-02-1175, 2002.

[17] R. Powers, M. Goldszmidt, and I. Cohen. Short term performance forecasting in enterprise systems. In *Proc. of the 2005 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 2005.

[18] *Rice University Bidding System*. http://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] *UCI Machine Learning Repository*. http://www.ics.uci.edu/~mlearn/MLRepository.html.

[22] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition, June 2005.

[23] A. Yemini and S. Kliger. High speed and robust event correlation. *IEEE Communications Magazine*, 1996.

[24] C. Yuan et al. Automated known problem diagnosis with event traces. In *EuroSys*, Apr. 2006.