Failure Diagnosis Using Decision Trees

Mike Chen, Alice X. Zheng, Jim Lloyd, Michael I. Jordan, Eric Brewer
University of California at Berkeley and eBay Inc.
{mikechen, alicez, jordan, brewer}@cs.berkeley.edu, jlloyd@ebay.com

Abstract

We present a decision tree learning approach to diagnosing failures in large Internet sites. We record runtime properties of each request and apply automated machine learning and data mining techniques to identify the causes of failures. We train decision trees on the request traces from time periods in which user-visible failures are present. Paths through the tree are ranked according to their degree of correlation with failure, and nodes are merged according to the observed partial order of system components. We evaluate this approach using actual failures from eBay, and find that, among hundreds of potential causes, the algorithm successfully identifies 13 out of 14 true causes of failure, along with 2 false positives. We discuss some results in applying simplified decision trees on eBay's production site for several months. In addition, we give a cost-benefit analysis of manual vs. automated diagnosis systems. Our contributions include the statistical learning approach, the adaptation of decision trees to the context of failure diagnosis, and the deployment and evaluation of our tools on a high-volume production service.

1. Introduction

Fast recovery remains one of the key challenges to designers and operators of large networked systems. Before recovery can take place, however, one must first detect and diagnose the failure. We define failure *detection* to be the task of determining when a system is experiencing problems. Failure *diagnosis*, then, is the task of locating the source of a system fault once it is detected. In this paper, we assume that failures have been detected within the system, and concentrate on the subsequent problem of diagnosis.

In a large scale Internet system, there are many components at work during the lifetime of a request. For example, a failed database query could be caused by a misconfigured application server, a bug in a new version of the application, a network problem, a bad disk on the database server, or a combination of these errors and more. As networked systems grow in size and complexity, it becomes increasingly impractical to examine each component manually for sources of error. Manual diagnosis is time consuming, error-prone, requires much expertise, and does not scale. An automated approach is essential if we want to continue at the current growth rate and improve system availability.

Many systems today record requests with simple runtime properties for monitoring and billing purposes. They typically record properties such as the timestamps, request types, and the frontend machines servicing the requests. Our approach uses aggressive logging to trace request paths through tiered systems, recording the system components and databases used by each request [5]. Using machine learning algorithms, we leverage the recorded runtime properties from a large number of independent requests to simultaneously examine many potential causes.

The following observation from systems operation is crucial in designing our system: the exact root cause of error, such as the line of code containing a bug, is often not required for recovery to take place. The operators only need enough confidence in the diagnosis to apply well-known recovery techniques like reboot, failover, and rollback. Hence these are the goals of our approach:

- **High diagnosis rate.** We need to be able to localize a large percentage of the errors in order to garner confidence from the system recovery operator.
- **Few false positives.** False positives, which are correct behavior of the system that are mistaken for an error, are costly in terms of time wasted on recovery and rediagnosis.
- **Robust to noise.** Few systems are free from failures. It is common to find a small number of failures at any moment on a large system. An ideal algorithm should be able to filter out noise and prioritize faults that are impacting the availability of a system.
- **Near real-time turn-around.** The algorithm needs to be fast enough in order to derive benefits over manual approaches.

In this paper, we present a system that analyzes readily available request traces in order to automatically locate sources of error. We demonstrate our methods at work on eBay's web request log system, and analyze how well it achieves the four goals stated above. Being one of the larger Internet service sites in existence today, eBay makes for an ideal case study; successful deployment on the eBay system may ultimately assure success on smaller sites as well.

We start off by describing the eBay logging framework in section 2, followed by a detailed explanation of our decision tree approach in section 3. Section 4 describes the experimental setup, and section 5 presents some diagnosis results. We conclude with a discussion of our approach and possible improvements in section 6, plus a brief survey of related work in section 7.

2. The eBay Internet Service System

The Centralized Application Logging (CAL) framework at eBay is a central repository of application-level logs. CAL exports an API that enables platform and application developers to record information associated with each request. CAL is the foundation for several of the tools used by the operations team to detect and diagnose failures. CAL is also used by developers for debugging and performance tuning, and for business-related purposes such as fraud detection and business intelligence.

At eBay, both the application servers and the applications they host are instrumented to log to CAL. They asynchronously write information about each request to the Harvester cluster over persistent TCP connections. The Harvesters are responsible for writing logs to persistent storage. They also publish the logs in real time onto a message bus to make the information available to various analysis and visualization engines. CAL uses load balancing switches to provide high availability and scalability.

The CAL API requires the application developer to mark the start and the end of a request, and also supply the name and the status code of a request (i.e. success or failure). Additional information such as version number, host name, and pool name are automatically recorded by the platform. Because the application servers are threaded, the intermediate logs between the start and the end markers can be associated with the request using the triplet {`thread ID`, `process ID`, `host ID`}. CAL also supports nested requests. For example, an HTTP request may result in multiple database accesses. All of these child database requests are automatically associated with the parent request.

A basic request trace includes the request type, request name, host name, pool name, version, timestamp, and the status of the requests. Most large Internet services record similar information for monitoring and billing purposes. On CAL, this basic trace can be extended to include the databases accessed by each request, providing additional visibility into the runtime behavior and resource dependency of each request.

The CAL system currently services more than two thousand application servers at more than one billion URLs/day and a peak data rate of 200Mbps. It stores 1TB of raw logs per day, or 150GB gzipped.

3. A Decision Tree Learning Approach

There are many possible machine learning approaches toward failure diagnosis. In this paper, we treat the problem as one of finding system components that are correlated with failure. More specifically, we train a decision tree to classify the failed and successful requests that occurred during the faulty period. We then post-process the paths that lead to failure-predicting nodes and extract relevant components. While decision trees [3] are not always the most competitive classifiers in terms of prediction, they enjoy the crucial advantage of yielding human-interpretable results, which is important if the method is to be adopted by real network operators.

3.1. Learning Decision Trees

Here is what a decision tree might look like in our system. Suppose there are two sources of error, one associated with machine `x`, the other associated with request type `y`. Suppose there are 15 failed and 5 successful requests observed on machine `x`, 10 failed and 6 successful requests of type `y`, and 20 other successful requests. Figure 1 shows a possible decision tree learned from this data. Each leaf node is labeled with the majority vote of the data contained at that node. For example, the leftmost path of (`Machine = x`) results in a leaf node predicting failure, with 15 supporting votes (i.e., failures) and 5 dissenting votes from training data. By examining the paths that lead to failure-predicting leaf nodes, one may distinguish the possible sources of error.

Learning a decision tree involves deciding which split to make at each node, and how deep the tree should be. Let X denote our feature vector and y the class label. For binary classification, $y \in \{0, 1\}$, where 1 denotes a failure and 0 success. The vector X may includes features such as the name of the machine, the name of the software build running on that machine, etc. The root node of the decision tree contains all of the data. At each node, the dataset is split according to the values of one particular feature. Splits are picked to maximize the $Gain$ in information. This continues until no further split is possible, or the node contains only one class. After the tree is fully grown, entire sub-

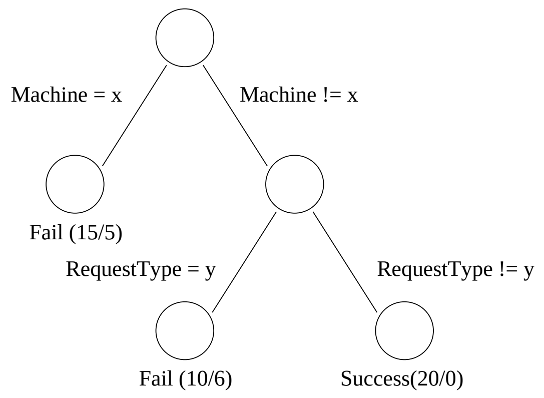

Figure 1. An example decision tree for diagnosis.

branches with low overall $Gain$ value are pruned back in order to avoid overfitting. [1]

There are many possible definitions for the $Gain$ of a split. It is generally observed that the choice of splitting criterion does not affect the ultimate *classification* performance [3]. Our ultimate goal, however, is not classification but path selection. In our experiments, we examine two different splitting criteria, and the selected paths do indeed sometimes differ.

The popular decision tree learning algorithm *C4.5* [17] defines the $Gain$ function for feature x_i at node t as:

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

where $H(t)$ denotes the binary entropy at node t, and $H(x_i, t)$ is the sum of entropy of child nodes after making the split based on feature x_i. Entropy is one of many possible measures of "pureness" of the distribution amongst classes. If Z is a discrete random variable and $p(Z)$ its distribution, then the entropy of Z is defined as $H(p(Z)) = -\sum_z p(z) \log p(z)$. The entropy function is maximized when the distribution is uniform, and decreases toward 0 as the distribution skews toward certain feature values. Hence a smaller entropy indicates larger skew in the distribution.

In contrast with *C4.5*, our implementation at eBay currently deploys an algorithm that we call *MinEntropy*. Its $Gain$ function is based on the entropy of a different random variable, and, due to time and resource constraints, it only follows the path that is the most "suspicious." Instead of pruning, it uses an early stopping criterion and stop splitting when the $Gain$ falls below a certain threshold.

Suppose feature x_i has d possible values. In MinEntropy, we look at the probability that a failed request at a particular node t takes on a particular value.

$$P(x_i = j; t) = \frac{\text{\# of failed requests at node } t \text{ with } x_i = j}{\text{\# of failed requests at node } t}$$
$$\text{and} \quad Gain(x_i, t) = -H(P(x_i; t))$$

$P(x_i = j; t)$ represents a multinomial distribution over values of x_i. This method makes the assumption that, for each request contained in node t, determining its value for feature x_i is like tossing a d-sided die with the right empirical distribution. Our goal is to find the particular value of the feature that seems to be correlated with an unusually high number of failures. Thus at each step, we split the tree based on the feature with the lowest entropy, and follow the child node j with the highest failure probability ($P(x_i = j; t)$).

3.2. Failure Diagnosis from Decision Tree Output

Learning the decision tree is only half the battle. We also need to select the important features that correlate with the largest number of failures. We do this using the following four heuristics:

1. We ignore the leaf nodes corresponding to successful requests. Most of them do not contain any failed requests, and are thus useless in diagnosing failures.
2. *Noise Filtering*: We ignore leaves containing less than $c\%$ of the total number of failures. This corresponds to making the reasonable assumptions that there are only a few independent sources of error, and each of them accounts for a large fraction of the total number of failures.
3. *Node Merging*: Before reporting the final diagnosis, we merge nodes on a path by eliminating ancestor nodes that are logically "subsumed" by successor nodes. For example, if machine `x` only runs software version `y`, then the path `(Version=y and Machine=x)` is equivalent to `(Machine=x)`. This kind of "subsumption" between system components defines a partial order on our features, where feature1 $\leq$ feature2 iff all requests containing feature1 also contain feature2.
4. *Ranking*: We sort the predicted causes by failure counts to prioritize their importance.

Applying these heuristics to the example decision tree in Figure 1, the right-most leaf node would be eliminated in step 1 because it does not contain any failures. In step 2, if we have a noise filtering threshold of 10%, then the remaining leaf nodes, with 40% and 60% of the total failures, will both be retained as candidates. In step 3, we produce two predicted sources of error: (`Machine=x`) and (`Machine!=x and RequestType=y`). If none of the

1 When a model trades off performance on test data for better fit of training data, it is said to *overfit*.

Host	DB	Host, Host	Host, DB	Host, SW	DB, SW
2	4	1	1	1	1

Table 1. Summary of combinations of types of faults occurring in our snapshots of the request logs.

Type	Name	Pool	Machine	Version	Database	Status
10	300	15	260	7	40	8

Table 2. Names of features and the number of unique values of each feature in the request logs.

failed requests with with request type `y` is executed on machine `x`, then the machine feature is subsumed by the request type and the two nodes merge into one. The diagnosis is now (`Machine=x`) and (`RequestType=y`). Finally, in Step 4, the two predicted causes are ranked by failure count; (`Machine=x`) causes 15 failures, which places it before (`RequestType=y`) with 10 failures.

4. Experimental Setup

4.1. Data Collection

We have collected 10 most recent one-hour snapshots of logs that are known to contain system faults. Four of these snapshots have two independent faults each, so the total number of faults is 14. Of these 14 faults, 6 are single-host faults, 6 are database faults, and 2 are software bugs. The true causes of the problems are identified by examining post-mortems, operations chat logs, and application logs. Table 1 contains a summary of different types of faults.

A complete request trace contains a basic trace and a database path trace. The basic trace contains 6 features: request type, request name, pool, host, version, and the status of each request. The extended database trace contains the list of databases accessed by each request. (See Table 2 for a summary of the features.) Each hour-long snapshot contains complete request traces from a fixed set of 15 pools, constrained to the minutes during which the faults are known to be present. Each of these one-minute slices contains about 200,000 requests, 0.001%-2% of which has an outcome status of failure.

4.2. Implementation

We have implemented the MinEntropy algorithm in both C++ and Java. The C++ version currently runs on the eBay site and performs automated diagnosis on eBay's production site in less than 10 seconds.

We use implementations of C4.5 and association rules from the machine learning tools package Weka [2] [13], an open-source machine learning package. The experiments are run using JDK 1.4.2 on quad-PIII 2GHz Linux 2.4.18 machines with 4GB of RAM. The algorithms first load the traces into memory, then compute the results.

5. Results

We compare the decision tree approach against a data-mining technique known as "association rules" [2] that simply ranks all possible combinations of features according to their observed probability of request failure.

Each algorithm returns a set of candidates (i.e., combinations of system components) that are deemed to be correlated with user-visible failures. To evaluate their performance, we make use of the *recall* and *precision* metrics. Recall measures the percentage of failure causes that are correctly diagnosed by the algorithm. Precision measures how concise the candidate set is.

During system recovery, each extra component in the candidate may incur added cost to the recovery effort. Hence the definition of precision needs to take into account not only the number of candidates retained, but also whether the list of components in each candidate is as succinct as can be. The number of *false positives* contained in a candidate is taken to be the number of extra components included which are not actual sources of error. Suppose the candidate is (`Machine=x and Version=n and RequestType=z`). If only machine `x` is at fault, then the candidate contains two false positives, one for each extra component. If the fault is in the combination of machine `x` and software version `n`, then the only false positive is `RequestType=z`.

Let C denote the number of effective components in the candidate set. Let N be number of distinct causes of failures, and n the number of correctly identified failure causes. We define:

$$\begin{aligned} Recall &= \frac{n}{N} \\ fpr &= \frac{C-n}{C} \\ Precision &= \frac{n}{C} = 1 - fpr \end{aligned}$$

Perfect diagnosis would have both recall and precision equal to 1.

2 Weka implements a variant of C4.5 called J4.8.

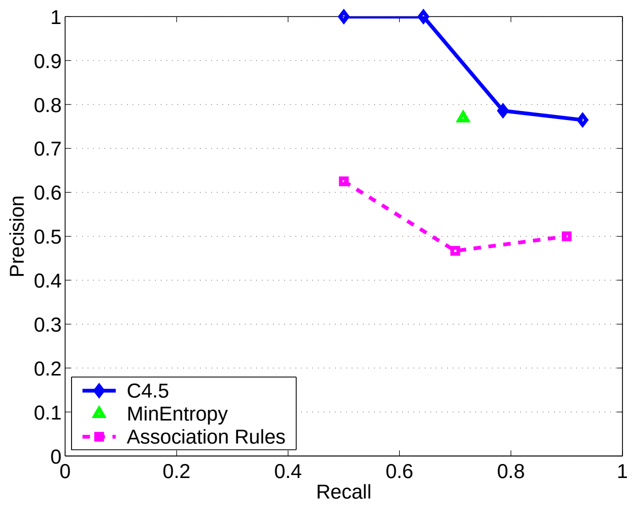

Figure 2. Precision-recall curves for C4.5, MinEntropy, and association rules.

5.1. Results on Basic Request Traces

The basic type of request trace is common to many Internet services for performance monitoring, failure monitoring, and billing. It does not contain any database access information. In a basic trace, a database fault would manifest itself in what appears to be many minor failures.

In our dataset, 6 out of 14 faults are database-related. (See Table 1 for a summary of the types of faults.) The ground truth cause for each database fault is taken to be the name of the request that accesses the database and has the most number of failures. For example, a fault in the feedback database is translated into a software fault of the request name ViewFeedback. Prediction of other related symptoms is counted neither as a correct diagnosis nor as a false positive, and therefore has no effect on precision and recall.

Figure 2 shows the precision-recall curves of C4.5, MinEntropy, and association rules on the basic request traces. The curves are obtained by varying the cutoff threshold for the number of retained candidates. At a failure rate cutoff of 50%, C4.5 returns seven candidate components over ten snapshots.[3] This gives us a precision of 100% at a recall of 50%. As we lower the cutoff threshold, the candidate set grows: 17 candidate components are retained at a cutoff of 5%, bringing us to 93% recall and 76% precision.

Since MinEntropy always produces one and only one prediction, it is a fixed point in this graph. Out of the 13 components returned by MinEntropy, it correctly identify 10 actual faults of the system (one for each snapshot). Hence its precision score is 76.92% at 71.43% recall. This precision score is comparable to that of C4.5 at the 5% cutoff threshold, though its recall score suffers from the single-candidate limitation and is much lower as a result. Thus MinEntropy proves to be a good alternative to C4.5 when resources are scarce.

Both C4.5 and the simplified MinEntropy approach performs much better than association rules at all levels of recall.

3 As described in Section 3.2, a $k\%$ cutoff threshold for C4.5 means that that only paths accounting for $\geq k\%$ of all observed request failures are retained for further processing.

5.2. Experiments on Complete Traces

Each request in the eBay system may access one or more of the 40 databases multiple times. For this experiment, we include database access information in the request traces. Because the causes of database-related failures are now in the trace, we expect the algorithms to correctly diagnose snapshots involving database faults.

Since each request may access a variable number of databases, we take each database to be a binary feature with the values True or False.

To underscore the importance of noise filtering and node merging, we compare the results obtained using un-processed C4.5 paths, C4.5 with noise filtering, and C4.5 with noise filtering and node merging. The cutoff threshold is set to 10% of all observed failures and the results are presented in Figure 3. All three variations of the algorithm correctly identify 13 out of 14 true failures, making the recall rate 93%. In particular, all three variations correctly identify all the true causes in the four cases that have two independent failures.

The importance of post-processing is apparent in the precision scores. Our result-aggregation heuristic increases precision from 18% to 76%. This means that the false positive rate drops from 82% to 24%, which is a dramatic improvement. These statistics suggest that more than 58% of components in the raw decision tree paths are extraneous. These are paths that contain the correct sources of error, but also contain other features that are not related to any faults (cf. the sample decision tree and discussion of node merging in Section 3). The extraneous features were necessary in the construction of the decision tree, but are useless as an indicator of error. Hence post-processing of the decision tree paths is a crucial step.

5.3. How Many Candidates to Keep?

Our post-processing procedure relies on a cutoff threshold in the noise filtering step (c.f. Section 3.2). The cutoff

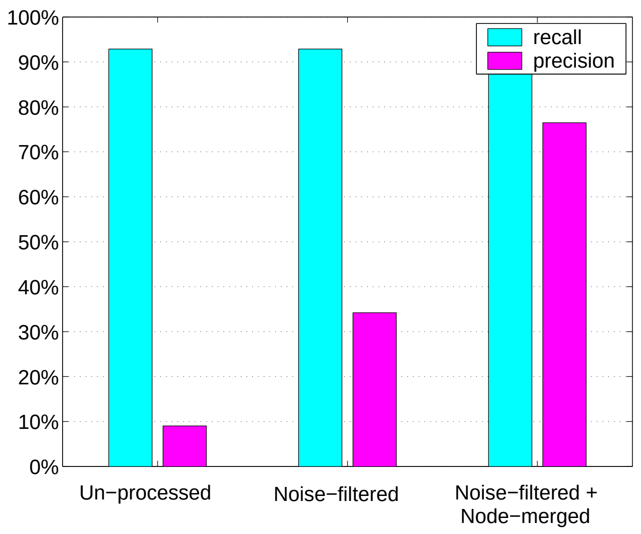

Figure 3. Recall and precision rates of C4.5 with different heuristics on traces containing database accesses.

threshold c is based on the percentage of request failures accounted for by a particular path in the decision tree. Let F be the total number of failed requests observed during one snapshot, and f_t the number of failed requests grouped under leaf node t. If $\frac{f_t}{F} > c$, then the path leading to node t is retained as a candidate. Raising (lowering) c would decrease (increase) the number of retained paths.

There are many different ways to select the threshold c. Here we explore two approaches: (1) selection based on a metric that combines recall and precision and (2) selection based on a metric that measures the expected recovery time.

The F-score is defined as the harmonic mean of precision and recall:

$$\text{F-score} = \frac{Precision * Recall}{Precision + Recall}$$

Figure 4 plots the F-scores (averaged over the 10 basic trace snapshots) for C4.5 against various values of c. The 5% threshold returns the highest F-score with a value of 0.4194. This puts us at the rightmost end of the recall-precision curve in Figure 2.

Alternatively, we can pick c to optimize a recovery cost function. For each component of the system, we know (1) whether or not it contains a real source of error, and (2) whether or not our algorithm labels the path as a potential source of error. Let Y denote the ground truth label of the component, where $Y = 1$ if and only if it is truly correlated with error. Let $\hat{Y}$ be the label given by our algorithm.

Let a be the amount of time it takes to run the automatic diagnosis algorithm, r the time it takes to perform the re-

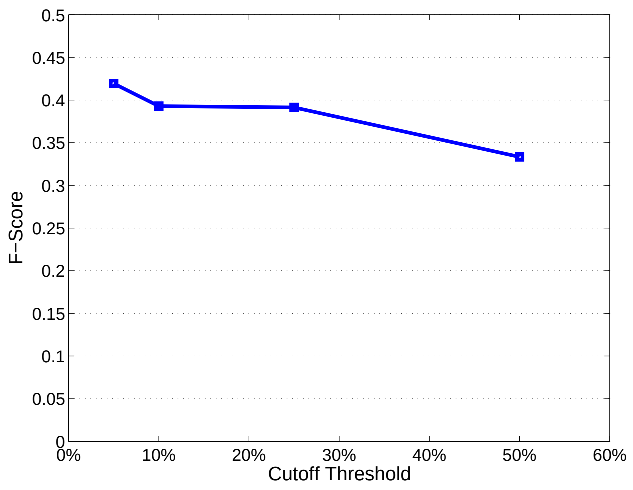

Figure 4. Average F-score vs. cutoff threshold for C4.5 on the basic trace dataset.

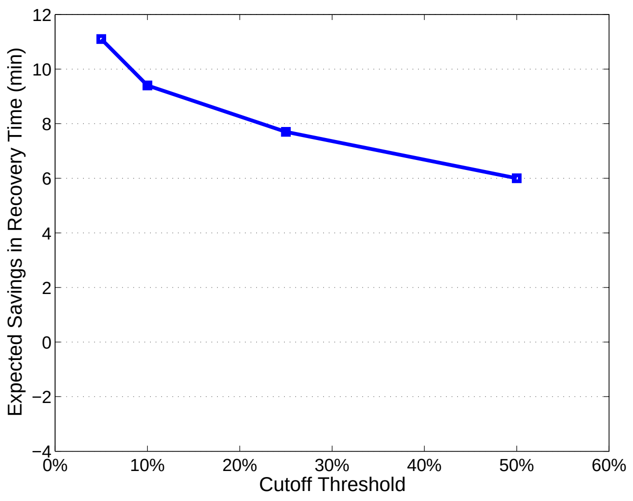

Figure 5. Savings in recovery time for C4.5.

covery, v the time it takes to verify whether or not the recovery action fixed the bug, and m the time for a human operator to manually examine the system and locate the bug. Each of the four combinations of Y and $\hat{Y}$ has a certain cost in terms of recovery time:

- $Y = 1, \hat{Y} = 1$: requires time $(a + r + v)$;
- $Y = 1, \hat{Y} = 0$: requires time $(a + v + m + r)$;
- $Y = 0, \hat{Y} = 1$: requires time $(a + r + v)$;
- $Y = 0, \hat{Y} = 0$: requires time (a).

One may want to pick a cutoff threshold c which would would minimize the expected system recovery time $\mathrm{E}[T]$ under the distribution $P_c(\hat{Y}|Y)$:

$$\begin{aligned}\mathrm{E}[T] \quad = \quad & P_c(\hat{Y}=1|Y=1)\cdot(a+r+v) \\ & +P_c(\hat{Y}=0|Y=1)\cdot(a+v+m+r) \\ & +P_c(\hat{Y}=1|Y=0)\cdot(a+r+v) \\ & +P_c(\hat{Y}=0|Y=0)\cdot(a)\end{aligned}$$

The expected amount of time saved by using automatic instead of manual diagnosis is then $m+r-\mathrm{E}[T]$. The probability distribution $P_c(\hat{Y}|Y)$ at various cutoff thresholds c can be estimated directly from our experiments:

$$P_c(\hat{Y}=1|Y=1) = Recall_c$$
$$P_c(\hat{Y}=0|Y=1) = 1 - Recall_c$$
$$P_c(\hat{Y}=1|Y=0) = \frac{\text{\# of false positives}}{\text{\# of non-faulty components}}$$
$$P_c(\hat{Y}=0|Y=0) = \frac{\text{\# of true negatives}}{\text{\# of non-faulty components}}$$

We plot the expected amount of time saved, $m+r-\mathrm{E}[T]$, using reasonable values for a, r, v, and m. Based on our conversations with operations teams at several large Internet sites, we set $m = 15$ minutes, $r = 3$ minute, $v = 1$ minutes, and $a = 1$ minute. Figure 5 shows the expected recovery time for the faults in our basic trace dataset. On average, the 5% cutoff threshold saves 11.1 minutes over manual diagnosis.

6. Discussion

We have demonstrated the applicability of decision trees to this specific task of failure diagnosis. While there are other classifiers with perhaps better failure prediction performance, decision trees return easily interpretable lists of suspicious system components. There has been much related work in the area of feature selection [1] [12], and specifically in the context of decision trees [10]. In our context, however, it is not sufficient to just include the set of features used in all of the paths leading to failed requests. Rather, we have demonstrated that post-processing of the candidate paths is necessary to eliminate costly false positives.

The domain of failure diagnosis from request logs is characterized by the ready availability of large amounts of unlabeled data. Labeled data, such as the snapshots used in our experiments here, are relatively scarce. The related problem of failure detection concentrates on labeling snapshots as either faulty or normal. One could perhaps make use of unlabeled snapshots within failure diagnosis itself. For instance, one could enrich the noise filtering step with a notion of "normal" amount of noise, learned from statistics derived from unlabeled data.

Another problem is in detecting whether or not any cause is discovered at all. This is necessary when the features contained in the traces do not include the actual sources of error, such as the database faults presented above. In this case, we need to be able to distinguish between a decision tree whose leaf nodes seem to be just noise, versus a decision tree whose leaf nodes contain useful information regarding the cause of failures. A possible approach is to examine the distribution of failed cases among the leaf nodes. If examples of failed requests are concentrated in a small number of leaf nodes, then we would likely get high quality results. Otherwise, if failures are evenly spread among a large number of leaf nodes, then chances are we do not have the right feature that causes the root failure. We can measure the "spread" of the failure distribution by looking at the entropy of the leaf nodes corresponding to failures.

7. Related Work

There has been much work in the field of failure diagnosis, though most previous work explicitly models causal or dependence interactions between the various components of the system. Our approach, in comparison, makes only implicit use of the underlying structure during the node merging phase. It is also necessary to stress that, though we have made references to "cause-finding," we do not attempt to infer any causal relationships between any of the components and the outcome. There has been much work in causal network modeling [18], and also on inferring causal relationships from observational data [8]. However, that is not the approach taken in this paper.

There are many commercial management systems that aid failure diagnosis, such as HP's OpenView [9] and IBM's Tivoli [16]. These systems typically either employ expert systems with human-generated rules or rely on the use of dependency models [7, 11]. However, these systems do not consider how the required dependency models are obtained. More recent research has focused on automatically generating dependency models based on dynamic observations. Brown et al. [4] use active perturbation of the system to identify dependencies and use statistical modeling of the system to compute dependency strengths. The dependency strengths can be used to order the potential root causes. However, the approach is intrusive and less suitable for diagnosing a production site.

The authors of [19] present a fault localization system that models faults in an end-to-end service system. The dependence graph models different layers within each host and linkage pattern between hosts. Each layer is associated with multiple possible failure modes. After observing certain symptoms in the system, belief propagation algorithms are run on the graph, and the posterior beliefs are examined to pick out the most likely causes for the symptoms.

In [20], a network management system is built to monitor exceptional events. A causal model of the system is built by experts with domain knowledge, containing symptom nodes and problem nodes. A codebook is then built that optimally compresses the symptoms given a designated set of problems one wishes to monitor. Fault correlation then becomes a decoding problem given a certain set of observed events.

In the approach taken by [15] and [14], the network is again modeled as a Bayesian network. The authors investigate the problem of designing and sending the optimum number of "active probes" so as to achieve a certain level of accuracy in diagnosis. In addition, efficient local approximation techniques is applied to the inference task on the Bayes net, and accuracy is shown to degrade gracefully under increasing noise in the network.

Our earlier work, Pinpoint [6] and ObsLogs at Tellme Networks [5], also dynamically trace request paths through tiered systems. Pinpoint uses clustering to correlate application components with failures.

8. Conclusion

We have presented a new approach to diagnosing failures in large systems. We record the runtime properties of each request and apply statistical learning techniques to automatically identify the causes of failures. The key to this ability is a large amount of requests and runtime information, which enables meaningful statistical analysis.

We validate our approach using actual failure cases from eBay. The MinEntropy algorithm has been deployed at eBay for several months. For single-fault cases, it correctly identifies 100% of faults with a false positive rate of 25%. The C4.5 decision tree algorithm performs well in both single- and multi-fault cases. When applied to request paths that includes databases accessed, it correctly identifies 93% of faults with a false positive rate of 24%.

We are currently exploring a variety of statistical learning algorithms to improve the diagnosis performance. We are also experimenting with streaming versions of these algorithms to be deployed on production systems. Finally, we plan to extend our approach to diagnose wide-area system failures.

References

[1] A. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, pages 245–271, 1997.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 207–216, Washington, D.C., 1993.

[3] L. Breiman, J. H.Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[4] A. Brown, G. Kar, and A. Keller. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. In *Seventh IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001.

[5] M. Chen, A. Accardi, E. Kıcıman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based Failure and Evolution Management. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, 2004.

[6] M. Chen, E. Kıcıman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *International Computer Performance and Dependability Symposium*, 2002.

[7] J. Choi, M. Choi, and S. Lee. An alarm correlation and fault identification scheme based on OSI managed object classes. In *IEEE International Conference on Communications*, Vancouver, BC, Canada, 1999.

[8] G. F. Cooper. A simple algorithm for efficiently mining observational databases for causal relationships. *Journal of Data Mining and Knowledge Discovery*, 1(1-2):245–271, 1997.

[9] H. P. Corporation. HP Openview. `http://www.hp.com/openview/index.html`.

[10] G. H. John, R. Kohavi and K. Pfleger. Irrelevant features and the subset selection problem. *Machine Learning: Proceedings of the Eleventh International Conference*, pages 121–129, 1994.

[11] B. Gruschke. A new approach for event correlation based on dependency graphs. In *5th Workshop of the OpenView University Association*, 1998.

[12] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *JMLR Special Issue on Variable and Feature Selection*, 3(Mar):1157-1182, 2003.

[13] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools with Java implementations*. Morgan Kaufmann.

[14] I. Rish and M. Brodie and N. Odintsova and S. Ma, G. Grabarnik. Real-time problem determination in distributed systems using active probing. In *Network Operations and Management Systems*, 2004.

[15] I. Rish, M. Brodie, and S. Ma. Accuracy vs. efficiency tradeoffs in probabilistic diagnosis. In *AAAI-2002, Edmonton, Alberta, Canada*, 2002.

[16] IBM. Tivoli Business Systems Manager, 2001. `http://www.tivoli.com`.

[17] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[18] J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2000.

[19] M. Steinder and A. Sethi. End-to-end service failure diagnosis using belief networks. In *Network Operations and Management Symposium*, 2002.

[20] A. Yemini and S. Kliger. High speed and robust event correlation. *IEEE Communication Magazine*, 34(5):82–90, May 1996.