Learning a Hierarchical Monitoring System for Detecting and Diagnosing Service Issues

Vinod Nair, Ameya Raul
Microsoft Research India
{vnair, t-amraul}@microsoft.com

Shwetabh Khanduja
Microsoft Research India
t-shwetk@microsoft.com

Vikas Bahirwani, Qihong Shao
Microsoft, Redmond, WA
{vikasba, qishao}@microsoft.com

S. Sundararajan
Microsoft Research India
ssrajan@microsoft.com

Sathiya Keerthi
Microsoft, Mountain View, CA
keerthi@microsoft.com

Steve Herbert, Sudheer Dhulipalla
Microsoft, Redmond, WA
{steve.herbert, sudheerd}@microsoft.com

ABSTRACT

We propose a machine learning based framework for building a hierarchical monitoring system to detect and diagnose service issues. We demonstrate its use for building a monitoring system for a distributed data storage and computing service consisting of tens of thousands of machines. Our solution has been deployed in production as an end-to-end system, starting from telemetry data collection from individual machines, to a visualization tool for service operators to examine the detection outputs. Evaluation results are presented on detecting 19 customer impacting issues in the past three months.

1. INTRODUCTION

Consumer and enterprise services increasingly run on large scale distributed storage and computing platforms built on hundreds of thousands of commodity machines. Service quality can be affected by hardware and software failures, unexpected user load changes, and so on, resulting in slow response, unavailability, violation of service level agreements, and ultimately customer dissatisfaction and negative reputation. Therefore monitoring the service to detect and diagnose issues quickly is an important problem. Recently there has been a steady increase in the literature in this area, from designing incident ticket management systems [13, 15], to building large scale textual log and time series analyzers [8, 20], developing specialized machine learning and data mining algorithms for anomaly detection [4, 9, 19], and many others [1, 2, 6, 18, 22].

There are several challenges in building a service monitoring system. *Telemetry data manageability:* A typical service is heavily instrumented by sensors of various kinds (e.g., performance counters, textual logs, event streams), reporting values of thousands of variables in real time. The monitoring system needs to use this data in a scalable manner to accurately detect and diagnose issues, and present to the service operators only the small subset of data that is actually relevant for an issue. *Complexity of service architecture:* Most services consist of sub-systems, components, sub-components and so on. The architecture induces relationships, often unknown, among the variables which need to be taken into account for accurate detection. Furthermore, the complexity can grow over time as more machines and functions get added, giving rise to new issues.

Currently rule-based detection is commonly used in service monitoring. Manually written rules often involve only one or at most a few variables, so the volume of detections becomes large as the number of variables runs into several thousands. They do not take into account the unknown relationships among the large set of variables, and as a result, they tend to produce a large number of false positives. Also, they do not handle changes in the service behavior over time. We consider a machine learning-based approach to building a detection system. Such an approach should process the large volume of telemetry in a scalable manner to detect issues, and present the user only the relevant subset of data to look at in an easily interpretable form. It should automatically discover the unknown relationships among the variables to reduce false positives. It should make use of any domain knowledge available, such as the architecture of the service, and any knowledge about patterns in individual variables that may be partial indicators of issues (e.g., spikes, step changes, trends, etc. in individual variables). It should use minimal supervision for learning because in practice there may not be a large number of past service issues available as labeled examples, and getting labels is expensive. Furthermore, many issues that should have been detected can go undetected, so even apparently "issue-free" data may not be so. Finally, although a fully automated approach would be ideal, here we do not insist on it, as a semi-automated one with semi-supervised learning is more feasible and still highly valuable in practice.

We propose an approach that addresses the above requirements. The main idea is to build a *hierarchical monitoring*

KDD'15, August 11-14, 2015, Sydney, NSW, Australia.
© 2015 ACM. ISBN 978-1-4503-3664-2/15/08 ...$15.00.
DOI: http://dx.doi.org/10.1145/2783258.2788624.

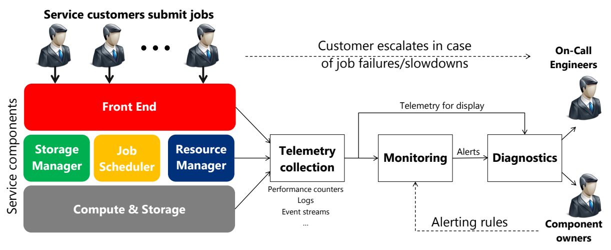


Figure 1: Summary of the service architecture, monitoring infrastructure, and the various types of users involved.

system with several levels of detectors, with detectors in each level taking the detector outputs of the previous level as input. The lowest level (i.e., leaf nodes) detectors take raw telemetry data as inputs. The system has the capability to throw alerts at various levels. The hierarchical structure has several advantages. (1) It helps in reducing the data dimensionality progressively in the hierarchy so that at the higher levels the data volume is low. At the lowest level, where the data volume is highest, the computation is embarrassingly parallel over the leaf nodes, which allows the approach to scale to large data volume. (2) A detector's output at a given level can be interpreted by navigating the paths below it to identify the small relevant subset of telemetry data at the leaf nodes. (3) Domain knowledge about service architecture can be used to shape the hierarchy, while knowledge about unusual patterns in individual variables can be used to design the lowest level detectors. (4) It offers a mechanism to reduce false positives as evidence for an issue can be gathered by identifying relationships among variables at each level. Low level detectors can be inaccurate as they are built using individual variables and may only give a partial indication of an issue. By combining these detectors via their relationships, it becomes possible to build a more accurate detector. (5) It requires minimal supervision. Note that although our approach is general and can be applied to different types of telemetry data, in this paper we focus exclusively on time series data from performance counters.

1.1 Contributions

The main contributions of the paper are:

1. We propose a general framework for building a hierarchical monitoring system that addresses the key requirements of a service monitoring application, making it broadly useful. It can be used with different types of telemetry data sources to detect and diagnose service issues.

2. We demonstrate a principled machine learning approach to building a monitoring system using our framework. The system is used to monitor Microsoft's internal distributed data storage and batch computing service that consists of tens of thousands of machines.

3. We have built an end-to-end pipeline for running our monitoring system, starting from telemetry data collection from individual machines, all the way to a visualization tool which service operators use to examine the detection outputs of our system to quickly focus on where the issues are.

4. We have deployed our system in production and present results on detecting 19 customer impacting issues in the past three months.

Outline: The rest of the paper is organized as follows: we begin with a detailed description of our service monitoring application in section 2. Section 3 gives an overview of our solution approach and its deployment in production. Section 4 describes the details of the machine learning techniques we use. Section 5 presents the evaluation of our approach. We then discuss related work (section 7), and conclude with a discussion of possible extensions of our work (section 8).

2. SERVICE MONITORING APPLICATION

The application we consider here is that of monitoring Microsoft's internal distributed data storage and batch computing service. The service is built on several tens of thousands of commodity machines providing fault-tolerant storage and compute. It is used heavily by Microsoft's product groups for running distributed batch jobs on large scale data.

Figure 1 provides a high-level summary of the service and its monitoring infrastructure. The service has a complex architecture with many micro-services, components, subcomponents, and so on. Each of these report their own telemetry. The complexity of the service architecture and the volume of the telemetry necessitates automatic analysis of the data.

Users of monitoring: On-Call Engineers (OCEs) monitor for any issues with the service that need to be immediately addressed to prevent customer impact. They also respond to customer escalations about their jobs failing or executing too slowly (possibly due to service issues), and suggesting work-arounds to customers. Engineering teams who own various components monitor their health and the effect of any updates made to them. Both types of users need to be alerted on any service issues in near-real time so that corrective action can be taken quickly.

Telemetry: *Performance counters* measure quantities such as CPU/memory/disk/network usage, latencies of service operations, number of requests received, etc. at fixed intervals (e.g., once every minute). *Logs* contain various text messages that are generated as the service software executes (e.g., status messages, error messages). *Event streams* are symbolic sequences reporting discrete occurrences in the service, e.g., "Job started". The amount of telemetry is large, e.g., the Storage Manager component by itself reports nearly 26000 counters. (Most of these are accounted by a few hundred unique counters reported by multiple instances of the component.)

Monitoring: As mentioned earlier, a common approach is to use rules, written usually by component owners, to detect issues. Rules are almost always defined on single performance counters, e.g., *If (latency $>$ threshold), then throw alert.* Since one counter provides a limited view of the service behavior, there may not be *any* threshold at which the counter value separates genuine issues from false positives. One is forced to pick a threshold that will detect the issues, but will also result in a large number of false positives. Service operators do not have the time to examine a large number of alerts to pick out the few genuine ones, so the issues are missed anyway. For example, in our application a component team determined that the threshold value for a single-counter rule that would be required to detect issues of interest to them would generate more than 12000 alerts over a four month period. This is simply too high a volume relative to the manpower available.

Diagnostics: Once an issue is caught, an OCE or a component owner needs to understand it to identify a mitigation for the affected customer or apply a hotfix to the affected component. The subset of performance counters relevant to an issue is typically a small fraction of the full set. Currently OCEs and component owners use domain knowledge and significant manual effort to find the right set of performance counters to visualize. As a result, diagnostics is slow and tedious, and prone to error without domain knowledge. One powerful form of domain knowledge is to know how the various performance counters can be grouped together, both by the service component they belong to, and also by the functionality of that component that they are most closely associated with. Such knowledge helps the user identify the relevant group of performance counters to look at and reduces the manual effort.

3. HIERARCHICAL MONITORING SYSTEM

We give an overview of our solution and then describe its deployment in production.

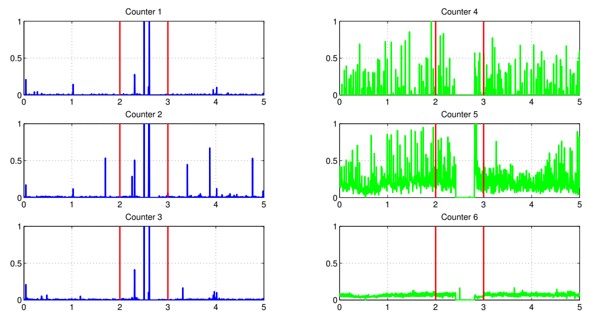
3.1 Solution overview

We introduce the main ideas behind our learning algorithm. For the purposes of this section, we will define a "detector" abstractly as a function that outputs a scalar score measuring how unusual its inputs are (higher score implies more unusual). Applying the detector across time (e.g., in a sliding window manner) results in a time series of scores.

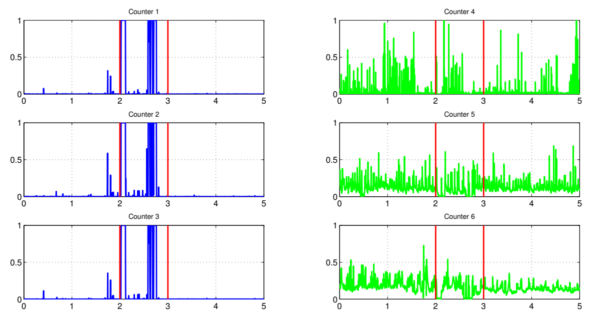
Issue example: We begin with examples of service issues we are interested in learning detectors for. Figure 2 shows two issue examples ((a) and (b)) which manifest in performance counters. In issue (a), two groups of performance counters shown in blue (counters 1-3) and green (counters 4-6) for the same time interval behave unusually around x-axis value 2.5. (Here the groups were identified using domain knowledge.) For issue (a), the blue group shows a spike *and* the green group simultaneously shows a drop. For issue (b), only the blue group shows a spike. So an issue appears as unusual patterns in individual counter time series, but the unusual pattern in any one counter by itself is not sufficient to detect the issue (the same pattern may occur at non-issue times also), resulting in high false positive rates for single-counter detectors. Therefore it is necessary to discover *groups of unusual patterns* that characterize the issue accurately.

Our algorithm (1) detects unusual patterns in counters, (2) automatically discovers groups of related detectors from a large set, and (3) combines the detectors within each group to detect issues characterized by multiple simultaneous unusual patterns in that group. Detectors with similar score time series are likely detecting the same underlying issues, so grouping them together provides more evidence for detecting them. For example, by grouping together detectors for the blue counters in figure 2 and combining them, a detector for issue (b) can be constructed.

To detect *higher-order* issues, such as issue (a) in figure 2 which involves two groups, one can iteratively apply the same grouping and combining steps to build further levels in the hierarchy:



(a) Issue where blue and green counter groups are unusual



(b) Issue where only blue counter group is unusual

Figure 2: Examples of two different service issues involving (a) two counter groups, denoted by blue and green, and (b) only the blue counter group. The plots show the raw counter time series data with time as x-axis and counter value as y-axis. The issue interval is indicated by two vertical red lines.

- **Grouping:** At level i, find groups of related detectors by the similarity of their score time series.

- **Combining:** For each group found in the previous step, combine their scores to define a detector for level $i+1$.

So, a detector at the $i+1^{st}$ level is a group of i^{th} level detectors which when simultaneously output high scores indicate a higher level issue. For example, by grouping together the detectors for the blue and green groups and combining them, it becomes possible to build a detector that outputs a high score for issue (a).

Note that our algorithm is unsupervised, i.e., it does not use manually marked issue intervals in the counter time series to learn detectors. As a result, the algorithm may learn detectors also for unusual patterns that do not correspond to actual service issues. Such detectors will need to be removed using manual analysis, but since the number of detectors is small, this is not a difficult task.

Figure 3 provides a visual summary of how the algorithm builds the hierarchy. We will refer to the circled numbers in the figure to explain the steps. Consider an example service with two components, A and B, each reporting its own telemetry data consisting of counters, logs, and event streams. (To save space, component B's detectors are shown in less detail than those of A.) *Step 1* is the initialization before the iterations begin. It applies a set of low-level detectors to the raw telemetry data. These detectors will have only a limited view of the service behavior, e.g., an anomaly detector based on a single counter, or a small set of counters within a sub-component. As explained, each detector out-

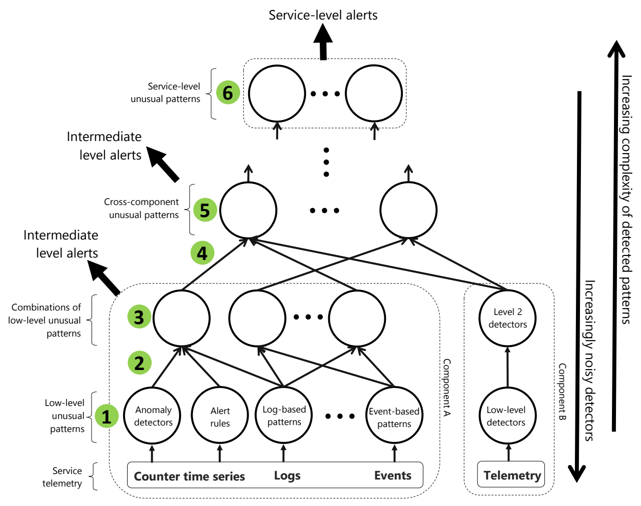

Service-level alerts
Service-level unusual patterns
6
Intermediate level alerts
Cross-component unusual patterns
5
4
Intermediate level alerts
Combinations of low-level unusual patterns
3
2
Level 2 detectors
Low-level unusual patterns
1
Anomaly detectors
Alert rules
Log-based patterns
Event-based patterns
Low-level detectors
Component A
Component B
Service telemetry
Counter time series
Logs
Events
Telemetry
Increasing complexity of detected patterns
Increasingly noisy detectors

Figure 3: Summary of the main steps of the proposed learning algorithm. Circled numbers refer to the different steps which are explained in section 3.

puts a scalar score time series that measures how unusual its inputs are across time.

As shown in the figure, the low-level detectors can take different forms. Various time series anomaly detection algorithms from the literature can be applied to the counter time series. Alert rules written by service experts can also be plugged in, allowing our approach to leverage the domain knowledge encoded in such rules. Services invest heavily in rule-based alerting before machine learning approaches are adopted. We make direct use of that investment by combining rules with ML, rather than treating them as mutually exclusive. Further detectors based on features extracted from logs and event streams, e.g., frequent itemset mining on event sequences, can also be plugged in, again allowing us to build on the rich literature in this area.

The iterative part starts at *Step 2* with the first grouping operation applied, followed by the combining operation in *Step 3*. Section 4 discusses the algorithmic details of both of these operations. Once *Step 3* is complete, the next-level detectors are defined. At the lower levels of the hierarchy the grouping and combining operations may be restricted to be within a component or a sub-component, as shown by the example in the figure where the second-level detectors are within either component A or B. So the resulting second-level detectors still have a narrow view of the service behavior, but broader than the first-level detectors. Note that alerts need not be generated only at the highest level of the hierarchy. If needed, the lower-level detectors can also be used to throw alerts, e.g., to produce alerts at a component level, as shown in the figure.

Steps 4 and *5* show the next iteration of grouping and combining. As the example shows, at this point detectors across components are combined so that the unusual patterns that require combining information from multiple components can be detected. These two steps can be repeated to learn detectors that have visibility across the entire service, as shown in *Step 6*. As more levels are added, the complexity of the detected unusual patterns increases, and we also expect the detectors to become increasingly sparser in terms of how frequently they output high scores.

3.2 Addressing the requirements

Scalability: By restricting the lowest level detectors to individual counters or small groups of counters, the learning at the lower levels can be made embarassingly parallel. For this reason we avoid joint analysis of a large set of counters at the lowest levels. As further levels are added, the number of detector score time series to consider also drops, and therefore information from across a large set of counters can be efficiently combined at the higher levels. At the higher levels, more expensive joint analysis algorithms can be applied without affecting scalability.
Irrelevant variables: By removing detectors that do not fit into any group well in the grouping operation, irrelevant counters and detectors are automatically removed.
Domain knowledge: Restricting the structure of the hierarchy according to the service architecture and the choice of the level at which to combine detectors across component and sub-component boundaries are powerful ways to incorporate domain knowledge about the service into the detector. Use of rules as low-level detectors provides yet another way.
Lack of labeled data: The algorithm does not require labeled examples to learn the detectors. Evaluating the quality of the detectors still requires labeling. Avoiding or reducing that effort is an open research problem. Supervision is also needed in removing any learned detectors that detect unusual patterns which are not of use for monitoring service health.
Interpretability: Once an alert is thrown at the highest level, the user can trace through the hierarchy to find out which branches contributed the most to the alert. This will directly help identify the components and sub-components involved in the issue.

4. MACHINE LEARNING ASPECTS

This section presents the details of our approach. As explained before, there are two main steps that are repeatedly applied to build each level of the hierarchical detector: 1) discover groups of related detectors from the previous level, and 2) for each such group, combine the scores of its members to compute a score for the group. We present specific choices of algorithms for these two steps. Here we use sparse structure estimation for Gaussian Graphical Model [17] with Affinity Propagation [7] to discover groups of related detectors, and use averaging of scores to compute the group score. We also describe one choice for the low-level detector which is designed to detect a change over time in the probability distribution of a counter.

4.1 Sparse Structure Estimation

Consider a p-dimensional Gaussian random vector $X = (X_1, \ldots, X_p) \sim \mathcal{N}(\mu, \Sigma)$ with mean μ, covariance matrix Σ, and precision matrix $K = \Sigma^{-1}$. A *Gaussian Graphical Model* (GGM) is an undirected graph $g = (V, E)$ with a vertex set $V = X_1, ..., X_p$ and an edge set E. The absence of an edge between the vertices X_i and X_j denotes the conditional independence $P(X_i, X_j | X_{-(i,j)}) = P(X_i | X_{-(i,j)}) P(X_j | X_{-(i,j)})$ where $X_{-(i,j)}$ denotes the set of all variables in X except X_i and X_j. It can be shown that an edge $(i, j) \in E$ is absent if and only if $K_{ij} = 0$. This is because the conditional distribution $P(X_i, X_j | X_{-(i,j)})$ is also Gaussian with the covariance matrix

$$\bar{\Sigma} = \frac{1}{K_{ii}K_{jj} - K_{ij}K_{ji}} \begin{bmatrix} K_{jj} & -K_{ij} \\ -K_{ji} & K_{ii} \end{bmatrix}. \qquad (1)$$

Since K is symmetric, $K_{ij} = K_{ji}$. So if $K_{ij} = 0$, $\bar{\Sigma}$ becomes a diagonal matrix and X_i and X_j become conditionally independent. Therefore the problem of estimating the graph structure of the GGM can be turned into the problem of estimating the non-zero entries of K.

One way to estimate the graph structure from data for the special case of sparsely connected, high-dimensional GGMs is presented by Meinshausen and Buhlmann [17]. This approach uses the fact that the distribution $P(X_i|X_{-(i)})$ is Gaussian with mean $E[X_i|X_{-(i)}] = -\sum_{j \neq i} \frac{K_{ij}}{K_{ii}} X_j$. This is a linear regression model with X_i as the target variable, all remaining variables $X_{-(i)}$ as inputs, and $\beta_{ij} = \frac{K_{ij}}{K_{ii}}$ as the regression coefficient for the input variable X_j. To identify the non-zero entries of K, for each X_i, learn the regression coefficients β_{ij} from data, and $K_{ij} \neq 0$ whenever $\beta_{ij} \neq 0$.

In particular, Meinshausen and Buhlmann propose using squared loss with L_1 regularization to estimate the coefficients (popularly called *Lasso* [23]). Let's denote the training data matrix as $\mathbf{X}$, containing n independent samples of X as the rows and the p variables as the columns. $\mathbf{X}_i$ is the i^{th} column of this matrix, and $\mathbf{X}_{-i}$ is the matrix containing the remaining columns. Let β_i be the vector of regression coefficients when the i^{th} variable is the target. The optimal regression coefficient vector β_i^* is estimated as

$$\beta_i^* = \arg\min_{\beta_i} \frac{1}{n} \|\mathbf{X}_i - \mathbf{X}_{-i}\beta_i\|_2^2 + \lambda \|\beta_i\|_1, \qquad (2)$$

where λ is a parameter to control the relative contribution of the L_1 regularizer to the objective function. Meinshausen and Buhlmann show that the resulting algorithm is a consistent estimator of the graph structure for the case of *high dimensional data* (including $p \gg n$) and *sparse graph* (the maximum number of edges for any vertex is $O(n^{\kappa})$ with $0 \leq \kappa < 1$). Both of these conditions are relevant for our application – the dimensionality is high due to the large number of performance counters, and the graph is sparse as we expect only a small number of them to be conditionally dependent on each other. So we expect the algorithm to be well-suited for our application. Detailed theoretical results for the algorithm can be found in [17].

In our implementation, for each detector i, we chose λ to be the largest value such that the squared error falls below a fraction above the asymptotic squared error value.

The final output of the algorithm is a $p \times p$ graph adjacency matrix A such that $A_{ij} = 1$ if the algorithm has determined that there is an edge between X_i and X_j, and 0 otherwise. A_{ii} is fixed to be 0 for all i.

We apply the algorithm to the output scores of detectors. In practice a detector score may follow a heavier-tailed distribution than a Gaussian, with the score at zero or near zero most of the time and taking high values only during an unusual event. In such a case, the average squared error term in the Lasso objective function (equation 2) is dominated by the prediction error on the high scores. As a result, the algorithm tends to put edges between pairs of detectors that give high scores simultaneously because they can significantly reduce the average squared error.

4.2 Clustering with Affinity Propagation

Once the graph structure is estimated using the Lasso-based algorithm, we want to find groups of detectors that can be combined to form the next-level detectors in the hierarchy. Our approach is to cluster the detectors based on the graph structure such that densely connected subgraphs form the detector groups. One way to achieve this is to use the graph adjacency matrix as a (binary) similarity score matrix between detector pairs in a clustering algorithm. The output is a clustering of the detectors into different groups.

Here we use Affinity Propagation (AP) [7] for clustering. Given a dataset of points and the pairwise similarities between points, AP selects a subset of them as "exemplars", one per cluster, to act as the representative point of each cluster. The remaining points are assigned to one exemplar each, resulting in a partitioning of the dataset. The exemplars and the assignments are selected to maximize the sum of similarities between points and their exemplars. The solution is found using a message passing algorithm.

Unlike k-means clustering, AP does not require pre-specifying the number of clusters. Instead, it uses a "preference" parameter which controls the cost for a datapoint to be selected as an exemplar. The more negative it is, the larger clusters one gets. This parameter can be used to bias the solution towards small/medium/large number of clusters. It also provides a way to incorporate into the clustering domain knowledge about particular data points that should be used as exemplars.

4.3 Combining scores within a detector group

After detector groups are formed using affinity propagation, the scores of the detectors within a group need to be combined to form a single score, which is the output of that group. Here we use averaging, which keeps the group score interpretable. One consequence of this choice is that the detector group score at any level in the detector hierarchy is a linear function of a subset of low-level detector scores.

Other functions can also be used to combine the scores, including nonlinear and/or learned ones. For example, one can use unsupervised learning algorithms such as One Class Support Vector Machine [21] or Local Outlier Factor [3] to learn a nonlinear combiner function. More generally, it can be useful to jointly learn both the detector groups and the combiner function applied to each group by optimizing a single objective function, instead of treating them as two disjoint steps.

4.4 Distribution change detection as a low-level detector

Many issues affecting the service manifest as abrupt changes in the probability distribution of counters (e.g., a jump in the mean value of a counter). One choice for a low-level detector is to find such changes in a single performance counter.

Let $X(t)$ be a univariate counter time series which is generated by independently sampling at each discrete time step from a distribution $P(X; \theta)$ parametrized by θ. Consider a time interval $t \in [t_0, t_1]$ in which a distributional change occurs at time t_c such that $X \sim P(X; \theta = \theta_0)$ for $t \in [t_0, t_c)$ and $X \sim P(X; \theta = \theta_1)$ for $t \in [t_c, t_1]$. t_c can be estimated as the candidate changepoint $t_a \in (t_0, t_1)$ at which the distribution estimated from samples in $[t_0, t_a)$ is maximally different (according to a chosen dissimilarity function) from the distribution estimated from samples in $[t_a, t_1]$. If the maximum dissimilarity is above a threshold, then the detector reports a change in distribution.

We use Poisson distribution for $P(X;\theta)$ for count variables and Squared Hellinger distance to compute the distance between two distributions P and Q. For Poisson-distributed P and Q with parameters λ_P and λ_Q, respectively, Squared Hellinger distance is

$$D^2(P,Q) = 1 - \exp(-\frac{\rho}{2}(\sqrt{\lambda_P} - \sqrt{\lambda_Q})^2), \tag{3}$$

where ρ is a scaling parameter. The advantages of this distance are: 1) it is a function of the change in mean between P and Q, which makes its detection results more visually interpretable, and 2) it produces a score normalized to the interval $[0,1]$, which is needed when combining detectors to form groups. We use an interval size of 12 hours.

To estimate detection quality, we manually evaluated a set of detections. We selected ten counters belonging to five different types of counters (two of each type) and evaluated the ten time intervals over a period of six months which gives the highest change detection score. We found that the precision was 95%.

Setting ρ: The ρ parameter in the Squared Hellinger distance expression above is set such that the detector score saturates at 1 for only a small percentage of time during a six month period, ensuring that the detector stays within its dynamic range most of the time. This percentage is set to 0.1% for all counters. As a result, the appropriate value of ρ for each counter can be estimated automatically from the data.

Alternative approaches: Other choices for the parametric form of the distribution, such as Pareto, Exponential, Log-Normal, etc., can be also be tried. Counters with heavy-tailed distributions are more common in our application, so we expect distributions such as Pareto to be more useful. One can also attempt to automatically identify the best-fitting distribution type for a counter; we leave this as future work. Instead of assuming a parametric form for the counter distribution, one can also use non-parametric two-sample tests such as Kolmogorov-Smirnov test and Maximum Mean Discrepancy [11], or non-parametric change detection using exchangeability Martingales [12].

5. RESULTS

To illustrate the effective working of the various principles underlying our hierarchical monitoring system, we use the data collected from 11 *volumes* (instances of the Storage Manager component of the service), each of which serves the same functionality. Each volume has 75 counter time series. The change detection algorithm described in section 4.4 is applied on each of the counter time series to form 75 low level detector time series. The hierarchical approach was applied on these to form several detectors at the mid level; a second iteration yields one high level detector. The grouping of the low level detectors to form the mid level detectors is done in one of two ways: (a) *Manual* - where an expert identified the grouping at a broad level; and (b) *Auto* - where the Lasso algorithm combined with AP clustering was applied to get the grouping. The manual approach was mainly included to provide a strong baseline for evaluating Auto.

The results are grouped in three parts. (1) First we illustrate the working of the structure discovery process and its effectiveness in bringing about sparsity and removal of irrelevant variables, thus leading to manageability. (2) Then we show how the hierarchical monitoring process leads to a reduction of manual effort via reduction of false positives while retaining alert quality. (3) We quantify this via suitably chosen plots that relate alert quality and manual effort and evaluate the detectors at various levels.

Effectiveness of structure discovery: Figure 4 shows results associated with the formation of the mid level detectors for one volume. The top left gray scale image describes the weights obtained by the Lasso algorithm. Each row (i-th) corresponds to the Lasso weights associated with the prediction model for the i-th variable. Pure black pixels indicate zero weights while gray pixels indicate non-zero weights; as the weight magnitude becomes larger, the pixel becomes more white. It is clear that the solution is very sparse - only a minimal set of important variables are picked up for predicting each variable. The top right plot is the same gray scale image with the variables re-ordered to show the groupings formed by the AP clustering algorithm. The diagonal blocks shown in red or blue describe the groupings.

Note that the AP algorithm is not perfect (it leaves out several good white pixels from the diagonal blocks), which is expected of any soft clustering algorithm. Since the total number of white pixels in the image is small, an alternative to the AP algorithm is to take each white pixel and define a grouping with just the two variables defining the pixel. This means that the groupings can have an intersection, which is quite fine. In general, finding overlapping clusterings may

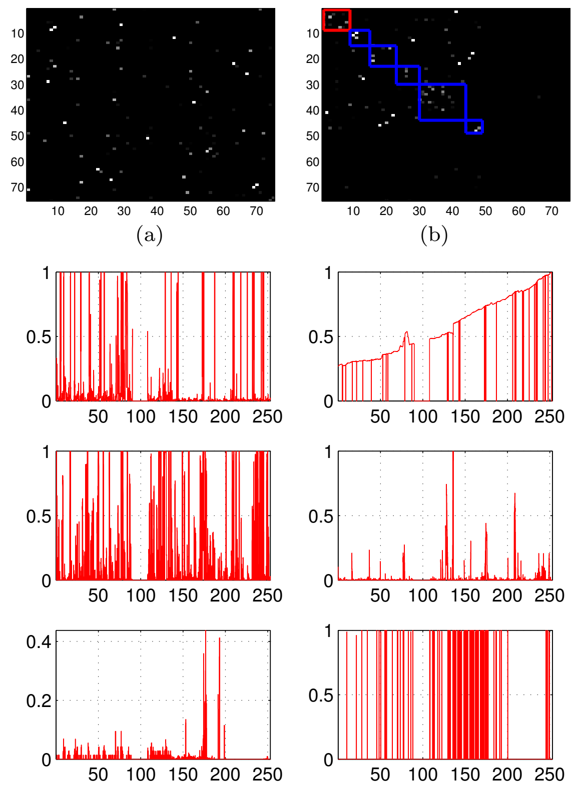


Figure 4: Illustration of the effectiveness of structure discovery (Lasso and AP clustering). See text for details.

be a good idea for avoiding information loss. We will explore these ideas in future work.

It is useful to note from the top right image of Figure 4 that the last 20 variables do not involve any white pixel. This confirms the ability of our method to separate out variables that do not form any grouping with other variables. While it is usually the case that most of these variables are actually irrelevant, some of them could still be useful; such variables could be identified via supervision - either direct manual specification, or, via supervised learning using data that contains time windows containing issues in the system.

Figure 4(c) shows the low level detector time series (left side plots) and their corresponding raw time series (right side plots) corresponding to a selected subset of three of the eight variables that define the top, red diagonal block of the image in Figure 4(b). In the top two sets of plots in Figure 4(c) one can see that the raw time series of those two variables are quite different, but their detector time series are very close. Thus, as recommended by our method, it is always better to use the detector time series and avoid using the raw time series, at the low level.

The bottom variable in Figure 4(c) has an interesting raw time series with great jumpiness. The distribution detection algorithm is quite effective and leads to a detector with only a small number of high peaks. Thus, this is a case in which the low level detector itself is useful. The other variables of the red block of Figure 4(b) have high peaks that are decently matched with those of the bottom variable - this is the good effect caused by Lasso. It turns out that the mid level detector formed by the red block is even better than the bottom variable in terms of restricting the firings to an even smaller set.

Reduction of manual effort while retaining alert quality: Figure 5 plots the detector time series at the three levels of the hierarchy. The time interval between the two vertical red lines denotes an interval where an issue occurred. For illustration we use a fixed threshold of 0.5 on each detector to define its firing. (Even if the thresholds are changed, the conclusions given below remain nearly the same.) Each firing leads to an alert that adds to the manual effort to investigate and resolve it. Clearly, the number of firings reduces significantly as we move up the hierarchy, and becomes manageably very small - just two - at the high level. These firings also lie in the red interval, thus neatly detecting the issue. It is important to note that this does not mean that firings at the lower levels are unimportant. For example, if we look at the third variable in the mid level, it fires several times around time=150. Other mid level detectors do not fire around the same time, and so the high level detector does not fire here. But it is quite possible that the firings of this variable is indicative of an issue with a mid level component. As pointed out earlier, such firings at lower levels can be determined as important using appropriate supervised learning.

Evaluation of alert quality and manual effort at various levels: Figure 6 plots various quantities accumulated over all 11 volumes of the platform. Here, *#Firings* is the number of times a specified threshold is crossed. Human experts labeled 19 issues in the 11 volumes by locating time intervals where issues occurred in the system. A detector is said to successfully identify an issue if it fires within that issue's time interval. We define *Recall* as the fraction of the set of 19 issues identified by a detector. The figure compares

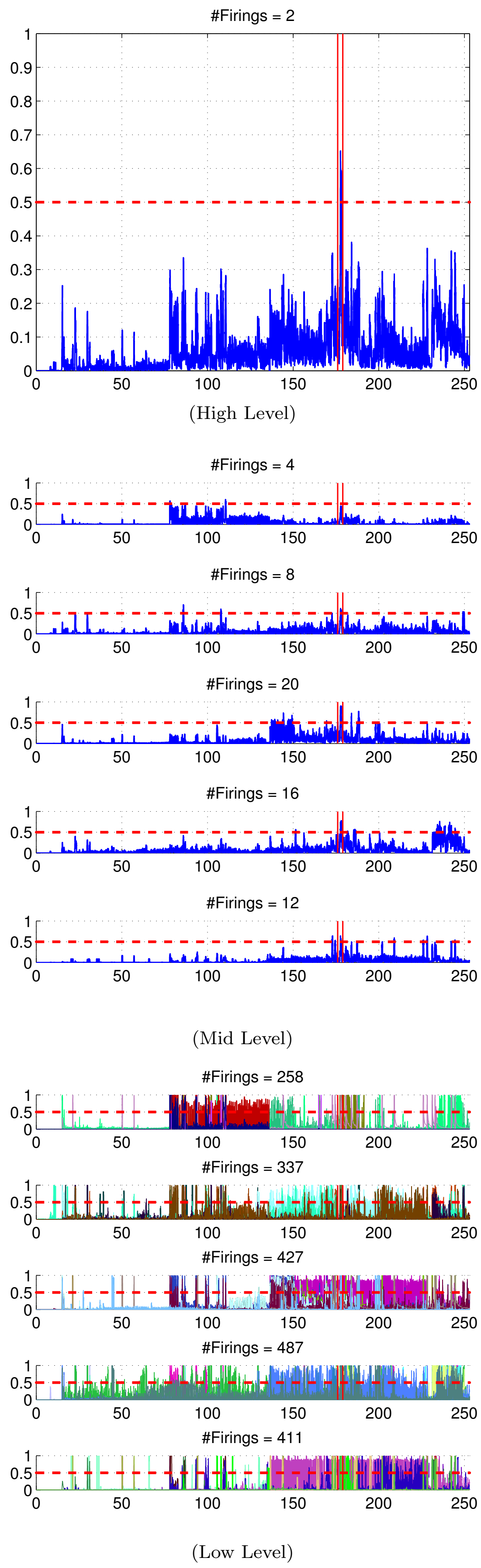


Figure 5: Reduction of the number of firings along levels of the hierarchy. See text for details.

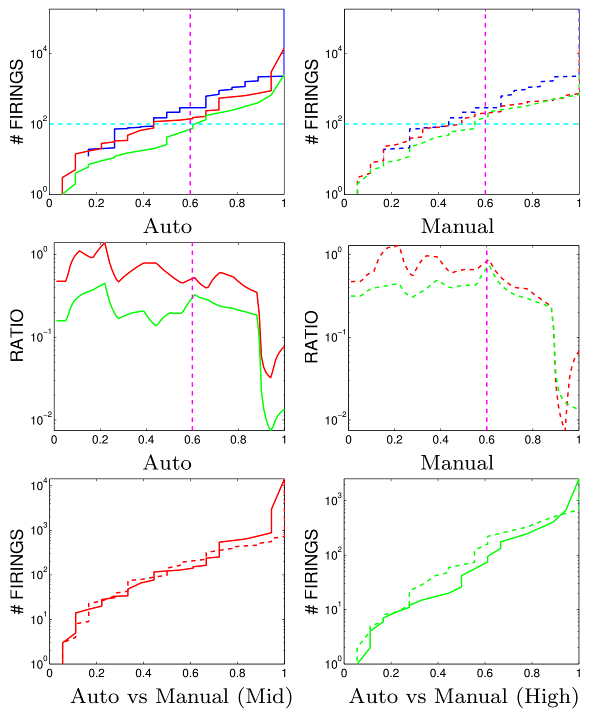


Figure 6: Performance of the hierarchical approach. See text for details.

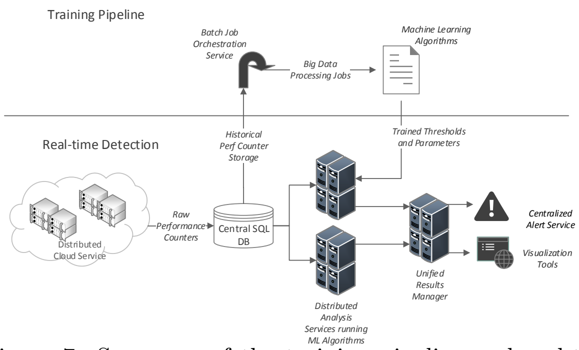
the performance of detectors at the low (blue), mid (red) and high (green) levels. *Ratio* refers to the ratio of the number of firings of mid or high level to that of the low level; thus, numbers much smaller than 1 indicate much better manageability compared to the low level. The top plots of Figure 6 show the behavior of #Firings against Recall - each curve is drawn by decreasing the threshold of each detector from a large value to a small value. The curves for the Auto and Manual methods are shown respectively as continuous and dotted lines. The plots in the middle row show the variation of Ratio as a function of Recall. Note the use of log scale in the vertical axes of these plots. In all these plots, lower a curve is, the better it is in terms of manageability: a specified recall can be achieved with lesser number of firings.

We can draw the following conclusions. (a) The mid level detectors are overall better than the low level detectors in reducing #Firings, equal in some places and much better in other places. (b) The high level detector is clearly superior to the low and mid level detectors. To get a quantification, (i) horizontal lines are drawn at #Firings=10^2 to get a feel for improvements in Recall for a specified number of firings (fixed amount of inspection time for the system's operators); and (ii) vertical lines are drawn at Recall=0.6 to get an idea of the reduction in #Firings at the same Recall.

The bottom row of plots compares Auto and Manual separately at the Mid level (left plot) and the High level (right plot). While the two methods are close at the mid level, Auto is overall much better than Manual at the High level. This is a clear indication of the effectiveness of the automatic method defined by Lasso and AP clustering.

6. DEPLOYED SYSTEM

This section explains the training pipeline and the real-time detection service we have implemented to deploy our solution in production (fig. 7). Performance counters are constantly updated on every machine; however, since there are tens of thousands of machines, these raw counters must be aggregated across time and various machine types for any viable real-time health assessment. Our counter aggregation service generates standard aggregate values such as sum, min, max, and average at specified time intervals (e.g., once per minute). These aggregates are stored in a central database for up to 15 days for real-time querying. Additionally, this data is archived in separate historical counter storage for one year for any offline analysis.



Training Pipeline
Batch Job Orchestration Service
Big Data Processing Jobs
Machine Learning Algorithms
Real-time Detection
Historical Perf Counter Storage
Trained Thresholds and Parameters
Distributed Cloud Service
Raw Performance Counters
Central SQL DB
Distributed Analysis Services running ML Algorithms
Unified Results Manager
Centralized Alert Service
Visualization Tools

Figure 7: Summary of the training pipeline and real-time detection service for deploying our solution in production.

Training Pipeline: The counter data in historical counter storage form the input to the offline learning of the hierarchical detector. A list of past service issues are used for evaluation. The output is a hierarchical detector.

Real-time detection: The goal of our detection service is to detect anomalies as soon as they occur. To achieve this, the service executes the following every 15 minutes:

1. It queries the central database for most recent twelve hours of counter data.

2. The Distributed Analysis module then applies the learned detector (output by the training pipeline) on the counters. The results computed by the detector are used to give health assessments for various components of the service being monitored.

3. Results for each component are sent to the Unified Results Manager, which in turn sends alert decisions to the Centralized Alert Service.

4. Centralized Alert Service sends alerts to relevant stakeholders.

Unified Results Manager: Acting as a bridge between the Distributed Analysis module and the Centralized Alert Service as shown in figure 7, this module groups the detector results, suppresses frequent alert fires, and sends summary reports. Additionally, it sends the detector results to visualization tools used by engineers to debug service issues.

Honoring real-time expectations: To honor real-time expectations, our process must complete in fifteen minutes or less. This necessitates the implementation to be scalable for both data access (querying the central DB) and processing (computing detector results). Our service implements a distributed architecture to achieve scale-out with components communicating with each other using industry standard REST Protocol.

Interactive Monitoring Tool: We have built a tool for visualizing the health of the system, the components and

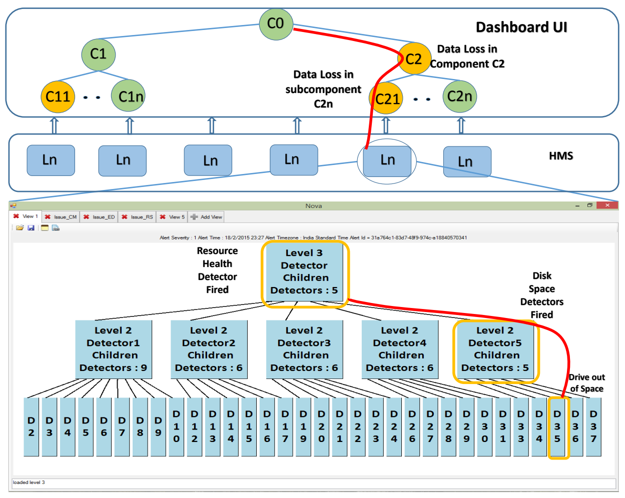

Dashboard UI
C0
C1
C2
Data Loss in Component C2
C11
C1n
Data Loss in subcomponent C2n
C21
C2n
Ln
HMS
Nova
Resource Health Detector Fired
Level 3 Detector Children Detectors : 5
Disk Space Detectors Fired
Level 2 Detector1 Children Detectors : 9
Level 2 Detector2 Children Detectors : 6
Level 2 Detector3 Children Detectors : 6
Level 2 Detector4 Children Detectors : 6
Level 2 Detector5 Children Detectors : 5
Drive out of Space
D2 D3 D4 D5 D6 D7 D8 D9 D10 D12 D13 D14 D15 D16 D17 D19 D20 D21 D22 D23 D24 D26 D27 D28 D29 D30 D31 D33 D34 D35 D36 D37
loaded level 3

Figure 8: Tool interface for visualizing the hierarchical monitoring system. The red line shows a navigation path for a *data loss* issue starting from component C2 that reports *data loss* and narrowing down to a *drive going out of space* in sub-component C21.

the sub-components in a hierarchical fashion. See Figure 8. The health information is colour coded with colours ranging from green (good health) to red (bad health). The health color information of any node in the hierarchy encodes the aggregated health score computed from the detectors feeding to the node. The health of a higher level component is computed as a weighted combination of the preceding level lower components and, weights are assigned according to the severity of issues that arise in different components. For example, *data loss* could be a more serious issue than *delay in job execution*. Users can monitor the health of any component and navigate down (e.g., to counter level details and views) as they see degrading health or unhealthy signs. As users navigate down, lower level health information (e.g., *response health*, *load health*, *disk access health*) get exposed. For example, *response health* can be computed using alerts signalled through unusual delays in responses (as measured through latency counters). Thus, navigating through a path gives a complete of view of the issues and the components involved in producing the unhealthy state of a higher level component. Furthermore, users can mark patterns or events to detect as they find changing behaviors during their investigation process. Detectors for such patterns get added to the system subsequently. Similarly, users can provide feedback when any harmless unusual patterns get detected, as such patterns falsely score a healthy state as unhealthy. Such labels can be used to adapt the machine learning models.

7. RELATED WORK

The literature in the area of service monitoring and diagnosis can be classified into two categories. Due to space limitation, we briefly discuss a few representative works from each of these categories. The first category covers application oriented works that consider one or more application scenarios (e.g., data centers, internet sites, online services) and develop some heuristics and/or data mining based approaches to solve a selected set of problems. Bodik et al. [2] propose the identification of finger prints that characterize different faulty states of data centers and use such finger prints to classify and identify performance crises quickly. Chen et al. [6] suggest a decision tree based approach to detect failures in large Internet sites such as eBay. Gabel et al. [10] present an unsupervised approach to detect faulty machines by comparing machines that perform the same task. Lin et al. [13] propose a hierarchical approach to cluster alerts and incident tickets using text content in large scale enterprise IT infrastructure scenarios. Chen et al. [5] present a subspace modelling approach that discovers a mapping between workloads and system internal measurements and use this mapping to detect system failures even under varying workload conditions. Fu et al. [8] study the problem of identifying performance issue metrics (e.g., access delay in databases) given a large collection of system metrics for an online services system. Roy et al. [20] describe an automated system for mining service logs to identify anomalies. Lou et al. [15] present a software analytics approach to implement an incident management system that investigates faulty incidents and suggest actions to take. To the best of our knowledge, none of the works develop a principled hierarchical monitoring system that can (a) handle high dimensional time series data, and (b) take system architecture and experts domain knowledge into account.

The second category covers methods that solve more primitive problems such as discovering patterns or detecting anomalies in time series data, text logs and event streams. Fu et al. [9] and Lou et al. [14] develop data mining based approaches to detect anomalies in job execution log files. Liang et al. [22] address the problem of generating system events from logs, with the resulting event streams subsequently used to detect anomalies using techniques such as finite state automaton or frequent item-set mining. Luo et al. [16] suggest to find correlation between time series and event data, and show how such discovered correlations help in the incident diagnosis process. The problem of detecting anomalies in multivariate time series has been studied by several researchers [4, 19]. Qiu et al. [19] propose a Lasso based approach to discover dependency structure among variables and detect anomalies in high dimensional time series data. Chandola et al. [4] suggest a sub-space based approach to convert multivariate time series to univariate time series and use that to detect anomalies. Discovering Motifs or shapelets and detecting rare occurrences of them in time series data have been other related areas of research [1, 18]. All these approaches are relevant to our work from the viewpoint of building low level detectors in our hierarchical monitoring system.

8. DISCUSSION & CONCLUSION

In this paper we have proposed a principled hierarchical approach that scales well to service monitoring applications having large amounts of telemetry data. This approach has been successfully deployed for monitoring a large distributed computing platform.

We briefly cover several important aspects of the system that have possible extensions.

(1) Low level detectors: They broadly belong to three categories: (a) general purpose (e.g., change detection methods, nearest neighbor based local outlier detectors), (b) standard patterns (e.g., frequently occurring spikes, bursts, trends) and (c) user-specified patterns or rules that are specified through a tool. While the first two categories of detectors and using expert rules as detectors are reasonably wellunderstood, identification and detection of newer patterns can be semi-automated through a motif discovery process [18] and experts suggesting patterns to detect.

(2) Structure discovery models: While our current implementation uses linear models based on the Gaussian assumption, improvements through non-linear and non-Gaussian models are possible. At each level, our method forms non-overlapping groups of variables. Modifying it to allow overlapping groups is easy and can be useful too.

(3) Learning: The learning process can be made more powerful and sophisticated. Our current system is unsupervised. But supervision data usually comes in as specific issues (e.g., machine overload or data loss) get identified by users. Weighted combinations of detectors at various levels can be set up to detect the issues. The supervision data can be used to adjust these weights as well as guide the structure discovery process. In essence, semi-supervised learning can become an important aspect over time. This set up has the potential to pick up important lower level detectors that are useful for detecting issues; see also the discussions in section 5.

(4) Interaction: Human interaction is useful to improve the quality of machine learning solutions. For example, experts can interact with the system to remove some variables from groups during the structure discovery process.

9. REFERENCES

[1] N. Begum and E. J. Keogh. Rare time series motif discovery from unbounded streams. *PVLDB*, 8(2):149–160, 2014.

[2] P. Bodík, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *EuroSys 2010*, pages 111–124, 2010.

[3] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. LOF: Identifying density-based local outliers. *SIGMOD Rec.*, 29(2):93–104, May 2000.

[4] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection for discrete sequences: A survey. *IEEE Trans. Knowl. Data Eng.*, 24(5):823–839, 2012.

[5] H. Chen, G. Jiang, and K. Yoshihira. Failure detection in large-scale internet services by principal subspace mapping. *IEEE Trans. Knowl. Data Eng.*, 19(10):1308–1320, 2007.

[6] M. Y. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. A. Brewer. Failure diagnosis using decision trees. In *ICAC*, pages 36–43, 2004.

[7] B. J. Frey and D. Dueck. Clustering by passing messages between data points. *Science*, 315:972–976, 2007.

[8] Q. Fu, J. Lou, Q. Lin, R. Ding, D. Zhang, Z. Ye, and T. Xie. Performance issue diagnosis for online service systems. In *SRDS*, pages 273–278, 2012.

[9] Q. Fu, J. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*, pages 149–158, 2009.

[10] M. Gabel, R. Glad-Bachrach, N. Bjorner, and A. Schuster. Latent fault detection in cloud services. Technical Report Technical Report, Microsoft Research, 2011.

[11] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola. A kernel two-sample test. *in JMLR*, 13(1):723–773, Mar. 2012.

[12] S.-S. Ho and H. Wechsler. A martingale framework for detecting changes in data streams by testing exchangeability. *IEEE PAMI*, 32(12):2113–2127, 2010.

[13] D. Lin, R. Raghu, V. Ramamurthy, J. Yu, R. Radhakrishnan, and J. Fernandez. Unveiling clusters of events for alert and incident management in large-scale enterprise it. In *KDD*, pages 1630–1639, 2014.

[14] J. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining invariants from console logs for system problem detection. In *USENIX*, 2010.

[15] J. Lou, Q. Lin, R. Ding, Q. Fu, D. Zhang, and T. Xie. Software analytics for incident management of online services: An experience report. In *IEEE/ACM ASE*, 2013.

[16] C. Luo, J. Lou, Q. Lin, Q. Fu, R. Ding, D. Zhang, and Z. Wang. Correlating events with time series for incident diagnosis. In *KDD*, pages 1583–1592, 2014.

[17] N. Meinshausen and P. Buhlmann. High-dimensional graphs and variable selection with the lasso. *The Annals of Statistics*, 34(3):1436–1462, June 2006.

[18] A. Mueen. Time series motif discovery: dimensions and applications. *Data Mining and Knowledge Discovery*, 4(2):152–159, 2014.

[19] H. Qiu, Y. Liu, N. A. Subrahmanya, and W. Li. Granger causality for time-series anomaly detection. In *ICDM*, pages 1074–1079, 2012.

[20] S. Roy, A. C. Koíg, I. Dvorkin, and M. Kumar. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *ICDE, 2015*, 2015.

[21] B. Scholkopf, J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation*, 2001.

[22] L. Tang and T. Li. Logtree: A framework for generating system events from raw textual logs. In *ICDM*, pages 491–500, 2010.

[23] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society (Series B)*, 58:267–288, 1996.