A User-Friendly Log Viewer for Storage Systems

JAYANTA BASAK and P. C. NAGESH, NetApp, Inc.

System log files contains messages emitted from several modules within a system and carries valuable information about the system state such as device status and error conditions and also about the various tasks within the system such as program names, execution path, including function names and parameters, and the task completion status. For customers with remote support, the system collects and transmits these logs to a central enterprise repository, where these are monitored for alerts, problem forecasting, and troubleshooting.

Very large log files limit the interpretability for the support engineers. For an expert, a large volume of log messages may not pose any problem; however, an inexperienced person may get flummoxed due to the presence of a large number of log messages. Often it is desired to present the log messages in a comprehensive manner where a person can view the important messages first and then go into details if required.

In this article, we present a user-friendly log viewer where we first hide the unimportant or inconsequential messages from the log file. A user can then click a particular hidden view and get the details of the hided messages. Messages with low utility are considered inconsequential as their removal does not impact the end user for the aforesaid purpose such as problem forecasting or troubleshooting. We relate the utility of a message to the probability of its appearance in the due context. We present machine-learning-based techniques that computes the usefulness of individual messages in a log file. We demonstrate identification and discarding of inconsequential messages to shrink the log size to acceptable limits. We have tested this over real-world logs and observed that eliminating such low value data can reduce the log files significantly (30% to 55%), with minimal error rates (7% to 20%). When limited user feedback is available, we show modifications to the technique to learn the user intent and accordingly further reduce the error.

CCS Concepts: • **Information systems** → *Hierarchical storage management*; *Information lifecycle management*; • **Software and its engineering** → *Software maintenance tools*; *Software testing and debugging*

Additional Key Words and Phrases: Log reduction, filtering, learning

ACM Reference Format:
Jayanta Basak and P. C. Nagesh. 2016. A user-friendly log viewer for storage systems. ACM Trans. Storage 12, 3, Article 17 (May 2016), 18 pages.
DOI: http://dx.doi.org/10.1145/2846101

1. INTRODUCTION

System logs are unstructured text messages, logged by several modules of a software system. Users of these logs include administrators, customer support engineers, and the developers. For example, "disk failure" or "network interconnect failure" messages call for urgent administrator intervention. Messages such as "Small Computer System Interface (SCSI) adapter encountered an unexpected bus phase" can help developers triage the bug location [Jiang et al. 2009]. Storage enterprises [Appliance 2007] offer live and remote customer support, by periodically collecting system logs from live

Authors' addresses: J. Basak, NetApp, Cinnabar Hills, EGL, Off Intermediate Ring Road, Bangalore - 560071, India; email: basak@netapp.com; P. C. Nagesh (Current address), The University of Melbourne, Dept. Computing and Information Systems, Australia; email: npanyam@student.unimelb.edu.au.

DOI: http://dx.doi.org/10.1145/2846101

deployments at customer locations. They play a crucial role in proactive repair, alerting, problem forecasting, and diagnosis.

Log files are constantly growing in size as developers introduce more messages as part of software development and maintenance. A study at a leading enterprise [Xu et al. 2010] showed that the number of log printing statements in a product, increased by hundreds per month, at all stages of product development. In our work, we focus on processing weekly logs from an enterprise storage system that typically contains thousands of messages. Large log files impact several resources as detailed below:

—First is the information load on the enterprise engineers. Note that these logs are meant for regular manual inspection by customer support personnel. For an enterprise with thousands of deployments, this cost can be huge.
—Second, finite buffers in both memory and disk are set apart in the storage operating system for log collection from different modules. Unrestricted log size leads to buffer overflow and message dropping without an intelligent filtering. Transmission of logs from customer sites to enterprise repositories consumes network bandwidth.
—Finally, large log files directly increase the storage costs for an enterprise. Therefore, there is a need to restrict log sizes to keep these costs in check.

Log size reduction by reducing log print statements is not feasible for legacy systems and costly for large code bases. Log generation rate also varies dynamically, calling for a runtime log-filtering tool. We observed that log messages are often of very fine granularity. A single event produces multiple messages from different parts of code along the execution path. This forms a sequence of messages with diminishing value to an end user, as the latter messages can be easily inferred. Our primary insight is that messages that could be inferred or guessed from context can be discarded without much consequence to the log user. This corresponds to the intuitive behavior of the log users. They attach maximum value to messages that bring new information and treat those as noise that are already known or expected by them. Based on this rationale, we have built a filter that hides inconsequential messages to shrink the log file in the viewer.

We first relate the utility of a message to the probability of its appearance in due context and detect the inconsequential messages as that with high contextual probability. We use machine-learning-based techniques for computing these probabilities. We developed unsupervised learning techniques that do not need user-labeled data and also mechanisms to leverage any limited forms of user feedback to improve the filtering accuracy in the supervised mode. Finally, we evaluate our mechanism against real-world customer log data.

2. RELATED WORK

The importance of system logs and its benefits have been well studied in the literature [Xu et al. 2010]. In root-cause analysis, prior research [Jiang et al. 2009] has focused on mechanisms to pin point a set of messages that most accurately describes the cause of a given problem. The growing size of logs and the subsequent burden on administrators has been illustrated in the context of Cisco router logs [Kramer 2003], where several tools and utilities are provided for fast parsing and searching through log files. Another approach to dealing with log sizes with the aid of visualization [Shahrestani et al. 2010] is to help reduce the recognition load and to pin point unusual log messages in an interactive and visual log browser.

A related work in log filtering is temporal and spatial filtering of logs [Liang et al. 2007] in supercomputing environments. Their focus is on eliminating redundant log records that refer to the same unit of information (event) but differ with reference to location (spatial redundancy) or time (temporal redundancy). In contrast, our work is about finding obvious or low informative messages that may potentially be a different

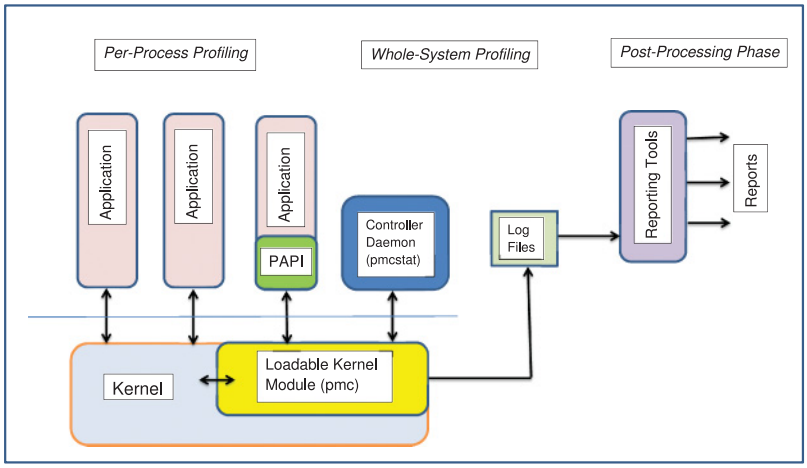

Fig. 1. OS architecture (FreeBSD PMC) highlighting log processing and shipping.

event. As these can be easily guessed from the context, discarding them does not impact the log user.

Preserving the relevant log messages and discarding or suppressing the irrelevant messages is also related to the semantic interpretation of messages in general. In literature, semantic meaning has been associated with the word vectors in different contexts with various probabilistic models [Maas and Ng 2010; Sarukkai 2000]. The measure of probability or likeliness of appearance of log messages in the due context is also related to the association rule mining in large databases [Agrawal et al. 1993]. Commercial tools like "Splunk" are available for various types of log mining and association [Alspaugh et al. 2014]. Log analysis has also been successfully used in failure prediction and data-driven system management [Peng et al. 2005; Ziming et al. 2009]. In our context, we use probabilistic models to find out the importance of log messages in the due context. Unlike the existing literature, we also modify the probabilistic models using user input. For example, a certain set of system log messages may be relevant to a particular system engineer and may not be so relevant to some other person. Therefore, the relevance of log messages depends not only on the context but also on the specific user interest.

3. SOLUTION OVERVIEW AND ALGORITHMS

3.1. Log Collection

In this section, we illustrate the log collection and processing activity that enables remote customer support. We describe a typical operating system architecture that incorporates collection, preprocessing, and shipping of log files. Then, we illustrate the presence of obvious or inferable messages in a real-world log. These form the inconsequential messages that can be discarded. Finally, we discuss algorithms for identification of inconsequential messages.

The AutoSupport. AutoSupport [Jiang et al. 2009] refers to other infrastructure that supports archival and monitoring of logs by customer support engineers. Figure 1 depicts the architectural setup of FreeBSD PMC (Performance Monitoring Counters) [Koshy 2007] depicting the log collection and shipping infrastructure. Our semantic filter is housed in a similar operating system architecture and is part of the log processing and reporting tools.

Fig. 2. An extract from a real-world log file showing inconsequential messages.

In this architecture, there is a loadable kernel module that manages the on-CPU performance counters and collects performance data. This module also interfaces with other parts of the kernel to collect the other book-keeping data. A controller daemon manages the entire data collection process. Simple analyses may be run using pmcstat. There are performance APIs (PAPI) that measure the per-process performance. All counters are then processed as a log-file (periodically, e.g., in the interval of 1 hour) and transferred to the reporting tool. The reporting tool collects the log files of periodic intervals and collates those logs over an extended period (e.g., 1 day) and ship over the network.

Example. Consider the example scenario where a disk drive has failed and the system is in RAID rebuild mode. The log file contains error notifications of disk failure first and subsequently more information around RAID rebuild. Often, this condition is accompanied by increased CPU utilization or low spare space in aggregates. For the log user, given the information of disk failure and ongoing RAID rebuild, alerts corresponding to increased CPU utilization or low spare space in aggregates are expected and of least value. Conversely, abnormal increase in CPU utilization during normal operations of the system are least expected and carry maximum value to the end user. In this scenario, the log user finds this information most valuable, as this is an indicator of some anomaly or an impending breakdown of the system. Notice that a message such as "increased CPU utilization" is not noise at all times and is most useful in some contexts and inconsequential in other contexts.

We show in Figure 2 an extract from a real-world log file with inconsequential messages highlighted in gray. A summary of successful completion of a RAID

Table I. Message Structure

Date	Message type	Object identifier
Sun Aug 5 01:08:23 CDT	[diskLabelBroken]	device 2 has a broken label

scrub almost always follows RAID start messages. Similarly, failure of the mailer (*asup.smtp.unreach:error*) leads to an informational message that the destination is not reachable (*asup.smtp.host:info*). Note that, in both cases, the latter events can be inferred from the former, as detected by our algorithm. Some error messages are also marked as gray (inconsequential), because the algorithm is able to predict the error message (*remoteVolume.unreachable:error*) from the preceding messages (*rpc.client.error*). However, equating an error message to an inconsequential message may not be acceptable to the end user. In this work, we regard that as an error or incorrect classification by our algorithm. In future, we plan to address this by incorporating the message severity and exempting the error category messages from filtering.

To summarize, the utility of a message is a function of how predictable it is in the context of the current state of the system. This notion is employed by our filtering algorithm that determines the most inconsequential messages in a log file that can then be discarded. This ensures that the log file size is restricted to the budgeted size without losing its overall effectiveness. In the next section, we describe algorithms for identification of inconsequential messages that are based on unsupervised learning and supervised learning techniques. The former algorithm requires no user-labeled data and learns the parameters from the input data directly. In the latter, we incorporate newer parameters that are learned from user-labeled data and can further improve the classification efficiency.

3.2. Methodology

The log file is a sequence of messages with each message having a structure as described in Table I.

In the log file, each message is tagged with a message type. Essentially these types are generated by the storage operating system to indicate the specific module and error type that is included in the log message. Certain variables like address space and error details are included in the object identifier of the log message. Whenever an engineer looks into the log messages, first she looks into the message type to understand if that message is important or not in the due context. The message type is usually structured text and the message body consists of unstructured text. For root-cause analysis of faults that occurred in the system, it is essential to look into the message body; however, to eliminate the not-so-useful messages, only the message type is sufficient for an engineer.

We consider only the type of the message (i.e., every message in the log file is reduced to a single message type indicator such as disk failure or disk scrub started). We ignore the other fields in the message, such as the object identifier. The full log file is therefore modeled as a sequence of message types $M_0, M_1, \ldots, M_N$ ordered by their timestamps. We define the context of a message (W) as its recent history or the set of w messages preceding it in the log file. We process the log file by a sliding window of length w (5 in our experiment). Formally, we define the notion of utility score of a message M_i as $1 - P(M_i|W)$ where $P(M_i|W)$ is its *prediction probability* conditioned on its context, represented by W. In other words the utility of a message is the inverse of its predictability. Note that when a message can be completely guessed from its context it brings no value (i.e., its prediction probability is 1 and its utility is 0). Conversely, a message that is highly surprising is most useful (i.e., its prediction probability is 0

(1) N : The number of messages in the log file.
(2) L: The sequence of messages ($M_0, M_1, \ldots M_N$).
(3) T: The set of message types ($e, s, \ldots$). Every message belongs to a unique message type.
(4) w: The length of the sliding window.
(5) C_s : The number of windows in the log file that starts with the message "s".
(6) $P(s) = \frac{C_s}{N}$ The absolute probability of seeing the message "s" in the log
(7) C_s^e: The number of windows starting with "s" and containing "e".
(8) $P(e, s) = \frac{C_s^e}{(w-1)*(N-w+1)}$ The joint probability of seeing the message "e" in a sliding window starting with the message "s"
(9) $P(e|s) = \frac{P(e,s)}{P(s)}$ The conditional probability of seeing the message "e", in a given window starting with the message "s"
(10) $\Phi = \frac{\sum_{\forall e \in T} P(s)}{|T|}$.This is the average absolute probability.
(11) $\mu = \frac{\sum_{\forall e, sinT} P(e|s)}{|T|^2}$. This is the average conditional probability

Fig. 3. Probability formulations.

and its utility is 1). Using these sliding windows and message types, we compute the prediction probabilities as detailed below.

Probability Computation. In Figure 3 we describe the different notations used.

The prediction probability of a message is conditioned on its context. Formally, this is denoted as $P(M_i|W) = P(M_i|M_{i-w} \ldots M_{i-1})$ where M_i denotes the message at the ith position in the list L. Note that the context W is a composed of w individual messages preceding the message M_i in the log file. The conditional probability $P(M_i|W)$ is accordingly a function of the conditional probabilities over each message in the context, that is, $P(M_i|W)$ is a function of $P(M_i|M_{i-w})$, $P(M_i|M_{i-w+1}, \ldots, P(M_i|M_{i-1})$. Each of these w messages has a combined influence on what message (M_i) follows them. To accurately combine these individual influences to get the final prediction probability ($P(M_i|W)$), we need to construct a Bayesian network over these w messages such that each message is represented by a vertex, and a directed edge with weight $P(a|b)$ is introduced between two vertices with message types a and b. A static analysis of the code gives the call graph with which we can construct a message flow graph by mapping messages to their originating functions. However, this approach is not feasible in an event-driven programming model where the control transfers across functions are hard to capture. Also, messages are usually emitted through a common message logging framework that further complicates static code analysis to derive an accurate message flow graph. Therefore, we do not attempt to construct a Bayesian network and instead try with three intuitive approaches of combining the individual conditional probabilities that are detailed below:

(1) Disjunctive: In this approach, we assume that the message M_i can be inferred by any one of the individual messages in the context W. The message M_i appears if either M_{i-w} appears OR M_{i-w+1} appears OR $\ldots M_{i-1}$ appears. Therefore, the probability of $P(M_i|W)$ is computed by OR-ing the individual events.
(2) Conjunctive: In this approach, we assume that the message M_i can be inferred only with the full set of messages in the context W. The message M_i appears if M_{i-w} appears AND M_{i-w+1} appears AND $\ldots M_{i-1}$ appears. Therefore, the probability of $P(M_i|W)$ is computed by AND-ing the individual events.

Table II. Three Approaches for Computing Prediction Probability and Threshold Values

Approach	Prediction probability ($P(M_i|W)$)	Threshold (θ)
Disjunctive	$1 - \prod_{j=i-w}^{i-1}(1 - P(M_i|M_j))$	μ
Conjunctive	$\prod_{j=i-w}^{i-1} P(M_i|M_j)$	μ^w
Markovian	$P(M_i - w) * \prod_{j=i-w}^{i-1} P(M_{j+1}|M_j)$	$\Phi * \mu^w$

(3) Markovian: In this approach the sequence of messages $M_{i-w} \ldots M_i$ is modeled as a Markov chain. The message M_i appears if and only if the exact sequence $M_{i-w}, M_{i-w+1}, \ldots, M_{i-1}$ appears. Therefore $P(M_i|W)$ is given by the sequence probability with the Markov assumption that each message only depends on the previous message.

The formulations for the prediction probability with the above three approaches are summarized in Table II. The threshold prediction probabilities in these three approaches are accordingly derived by assuming the average case, that is, when each of the conditional and absolute probabilities encountered take the average values of μ and ϕ, respectively, the prediction probabilities computed are the expected or average case probabilities. These are regarded as threshold values for classifying messages as those with "high" or "low" prediction probabilities.

3.3. Algorithms

3.3.1. Simple Log Filtering. In Algorithm 1, we describe a simple technique to identify inconsequential messages. It computes prediction probabilities of each message and compares it against a threshold (λ) value to decide whether the message is informative or inconsequential. Note that this process is sensitive to window length w. Large window lengths imply increased dependence of the prediction probabilities on many messages or events in the past. Conversely, short windows limit the dependence of probability computations to just the most recent messages. We found that setting of window length to 5 worked best in our experiments.

ALGORITHM 1: Simple Filtering: Identification and Filtering of Inconsequential Messages in the Log File.

Input: Log file L as a list of messages
Input: w : window length
Input: λ: Threshold
Output: Log messages marked as inconsequential or informative
Initialize **foreach** *message* M_i *at position* i *in the list* L **do**
    compute its prediction probability $P(M_i|W)$;
    `//Using Markovian, Disjunctive or Conjunctive approach, whose formula in given in Table II`
end
Mark all messages whose prediction probability $> \lambda$ as inconsequential;

3.3.2. Repetitive Filtering. In this section, we describe a modified algorithm (Algorithm 2) that is less sensitive to window length and the presence of duplicates in the input list of messages. Intuitively, prediction probability is a measure of the dependency of the current message with reference to other message types (example $P(raid\ failure|bad\ disk\ blocks)$. The presence of duplicate messages introduces self-loops across identical message types (for example, $P(raid\ failure|raid\ failure)$). These message types could be originating from different devices or objects that need special

handling. In this work, we only focus on message types. To handle duplicate messages, we identify long subsequences of inconsequential messages and remove duplicates within them. We repeat the whole process of relearning probabilities on this reduced log, until no further duplicate filtering can happen. On the resulting stable log, messages are finally marked as inconsequential or otherwise, based on the final prediction probabilities. Note that messages that are farther apart (i.e., $> W$) did not influence the prediction probabilities previously (Algorithm 1). In the current algorithm (Algorithm 2), multiple repetitions can eliminate duplicates or inconsequential messages between them and bring them closer (i.e., within W distance). This is equivalent to increasing the window length in regions of the input data with duplicates or inconsequential messages.

We use the input parameter λ to control the sensitivity of our algorithm. The threshold $\theta * \lambda$ is a function of the average conditional probabilities (see the formula in Table II) and the input scaling factor λ. More messages are marked as inconsequential when λ is lowered. Conversely, we can increase λ to reduce the number of inconsequential messages. Optimal values of λ can be chosen based on budgeted log size or the desired number of messages to be discarded.

ALGORITHM 2: Repetitive Filtering: Log of the Log File for Removing Inconsequential Messages

Input: Log file L as a list of messages
Input: w : window length
Input: λ: A constant parameter for threshold scaling
Output: Log messages marked as inconsequential or informative
Initialize **foreach** *message M_i at position i in the list L* **do**
    compute its prediction probability $P(M_i|W)$;
    `//As given by the formula in Table II`
end
Find subsequences within L, where each prediction probability exceeds the scaled threshold ($P(M_i|W) \geq \lambda * \theta$) and the subsequence length exceeds N;
`//This is a thick zone of inconsequential messages`
Within this subsequence look for repeating messages and remove all repetitions except the first occurrence of that message;
if *No removal occurred in the above step* **then**
    `//List L has stabilized w.r.t probability computations. Finalize and finish.`;
    Mark all messages with high prediction probability ($P(M_i|W) \geq \lambda * \theta$) as inconsequential.;
    Restore all the messages that were removed in earlier iterations to their original position in the list L, and mark them inconsequential.;
else
    `//Repeat until prediction probability computation over List L stabilizes.`;
    go back to the start of the algorithm.;
end

3.4. Incorporating User Feedback

The previously described probability computations and algorithms do not require any user feedback or labeled data and are therefore referred to as unsupervised learning techniques. These unsupervised techniques can be readily deployed and do not require any explicit training phase. In contrast, supervised learning techniques such as Support Vector Machines (SVMs) [Burges 1998] and Artificial Neural Networks (ANNs) [Jain et al. 1996] are harder to adapt for the purpose of log filtering for the following reasons:

—Feature engineering: A suitable data transformation is required to represent data in the form of input and output pairs, where input is a set of feature or attributes of the log data and output is a label denoting the message as inconsequential or informative. The data transformation in the log files to generate a time invariant <input, output> pair is very difficult.
—Feasibility: The techniques mentioned above such as SVM and ANN require large amounts of training pairs. It is difficult and impractical for an engineer to label large portions of the log files manually.

In our context, we only have access to limited quantities of labeled data corresponding to different regions of a log file. Therefore, we propose a novel mechanism to leverage labels over relatively small portions of the log file and, correspondingly, modify our filter to further reduce error rates. We do it by adding bias to the conditional probabilities of the messages. This method lets us combine the learnings from the log file obtained using unsupervised techniques with the user intent derived from the labeled data.

3.4.1. Collecting User Feedback. In order to collect user feedback, we modify the log viewer to incorporate minor visual elements to enable the user to provide feedback. Note that this log viewer is an internal tool within the enterprise and is utilized by the customer support engineers and software developers. Selective domain experts who have opted to provide feedback see radio buttons next to the individual messages which can be clicked to indicate that the message is inconsequential or informative. A background process collects this feedback to be relayed onto a central repository. Now we describe how this limited user feedback can be utilized to improve the log-filtering accuracy.

As discussed before, messages are marked inconsequential when they are highly predictable, that is, when the given message can be predicted from the context of the preceding messages in the log. These are captured using conditional probabilities across message types, which were computed solely from the log file. We now incorporate an additional "bias" parameter into this conditional probability. This bias provides us the flexibility to alter the conditional probabilities in tune with the user intent. Consider the example of $P(raid\ failure|bad\ diskblocks)$ as computed from the log file. A domain expert may opine that this value or measure of influence of bad disk blocks on an impending raid failure is too high or too low. In order to incorporate such user intent, we modify the probability measure by incorporating a prior bias parameter, which can be fine-tuned to reflect user intent. Below, we present the modified probability formulations:

(1) $P(e|s) = \frac{P(e,s)}{P(s)}$. The conditional probability of seeing the message "e," in a given window starting with the message "e" as defined earlier.
(2) $bias_{[S,E]}$. The bias parameters are defined for every module pairs, shown by the subscript $[S, E]$.
(3) $P(e|s, bias_{[S,E]}) \propto \frac{P(e|s)+bias_{[S,E]}}{1+bias_{[S,E]}}$. The right-hand side of this equation describes how the bias is incorporated into the original probability value. These values are normalized to get the actual probabilities ($P(e|s, bias_{[S,E]})$).

High values of bias push up the probability $P(e|s, bias_{[S,E]})$ closer to the maximum value of 1. Conversely, low values of bias tend to leave the probability score unchanged as defined solely from the log file contents. In our work, we restrict the value of bias to be in the range $[0, 6]$. Note that e and s denote individual messages, and E and S represent the module names to which these messages belong to. For example, *nis.lclGrp.updateSuccess:info* is a message and *nis.lclGrp* is the module name. The bias parameters are defined for every module pairs, shown by the superscript $[S, E]$, to limit

their number. The alternative of defining bias parameters for every pair of message identifiers can result in a very large number of bias parameters. This would lead to too many tunable parameters and render the process of optimization ill defined.

3.4.2. Evolutionary Approach. In this approach, the bias parameters of the model are encoded as a chromosome, and we use a genetic algorithm to search for the best chromosome. The notion of best chromosome is defined by a fitness criterion which corresponds to some desirable property in the problem domain. The genetic algorithm starts off with an initial population of chromosomes and follows an evolutionary approach of combining some members of the population to produce offspring or newer members of the next generation of the population. This is so designed such that good members (chromosomes with higher fitness scores) have a greater chance of participating in this process, whereas bad members (chromosomes with low fitness scores) are likely to be left out. We utilized an elitist model of genetic algorithms and set the initial population size to 80 members. The algorithm stabilized after around 300 generations, when it output the best chromosome. We experimented with different fitness function or optimization criterion and report the findings in the next section. We utilized Pyevolve [Perone 2009], an open-source implementation of genetic algorithms, to search for the best possible values for the bias. More details about genetic algorithms can be found in Whitley [1994].

4. EXPERIMENTS AND EVALUATION

In this section, we discuss our experimental objectives and framework, datasets used, and the metrics for measuring the effectiveness of our semantic filter. We first evaluated the log-filtering approach in a laboratory setup where we implemented it in Python using 2GHz CPU with 4GB RAM. We then shipped our code into a production environment having a Xeon processor with 16GB RAM.

4.1. Evaluation Objectives and Experimental Setup

An ideal log-filtering solution should discard only inconsequential messages. Further, the notion of inconsequential message is subjective and has to be defined by the log user. Note that the converse of an inconsequential message is any informative message that the end user is not sure about discarding. This is the superset of other significant or critical messages. We collected real-world logs obtained from storage systems installed in our customer locations.

We implemented an interface with the AutoSupport logs where we suppress the inconsequential messages. A snippet of the interface is shown in Figure 4. All inconsequential messages are flagged as “Spam” and suppressed in the view. There is a “+/−” toggle button with every batch of suppressed messages. A user can still view the suppressed messages if she presses the toggle button where the suppressed messages are displayed in the gray background. If she presses the toggle button again, then the inconsequential messages are removed from the view. With every message there is an option to provide feedback if the user considers that message as “Spam” or “Not Spam.” The feedback is collected (provided the user selects one) for every message and used for further computation in the supervised mode in the background. If the user does not provide any feedback for a message, then the label generated by the unsupervised filtering is considered to be the right label.

With the aid of domain experts drawn from our developers, quality assurance engineers, and customer support professionals, we manually classified a random sample of 3,000 log messages as inconsequential or otherwise. This labeled dataset was then divided into two equal parts referred to as the training dataset and the testing dataset. The training dataset is used for learning bias parameters for incorporating user

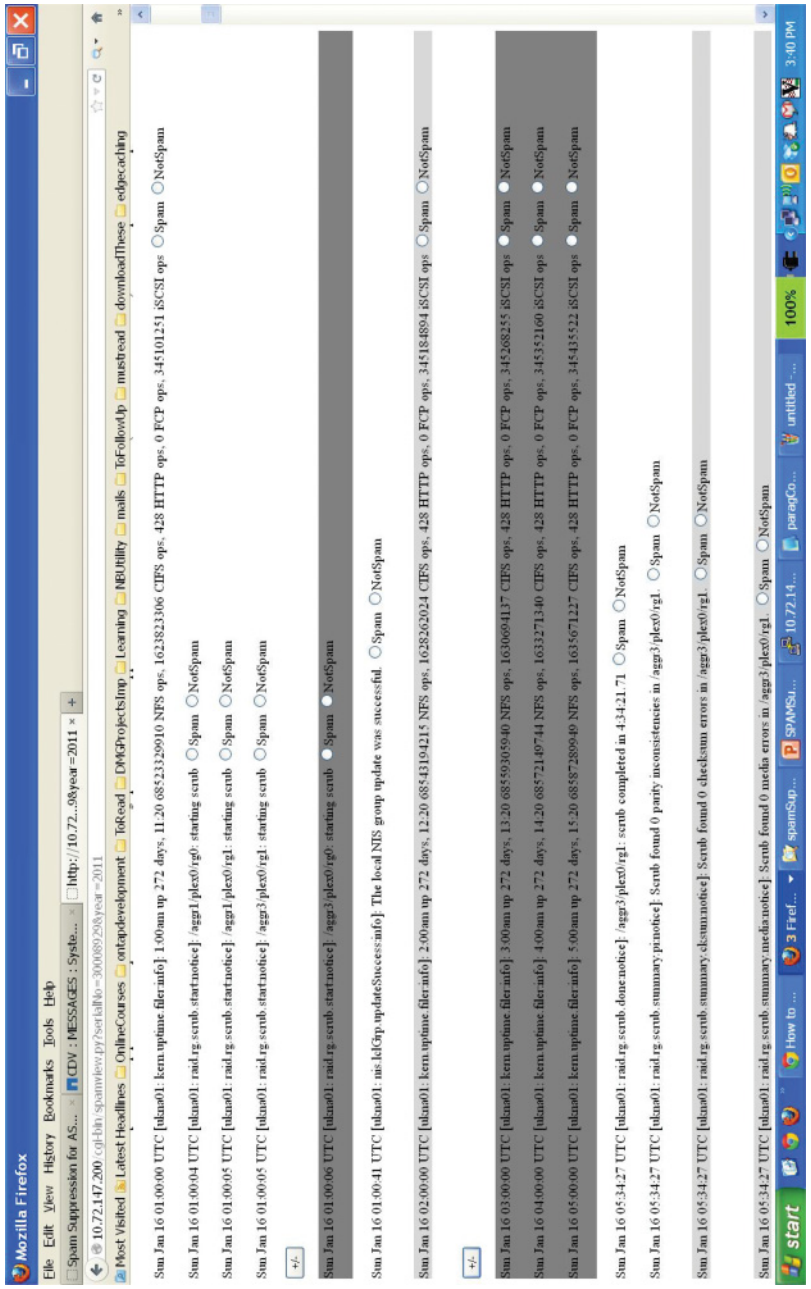

Fig. 4. A snippet of the log viewer user interface.

feedback. All metrics reported for our filtering algorithms are over the testing dataset only. This practice also illustrates that the learning from the training dataset is general and is applicable to other datasets as well, as can be seen from results over the testing dataset.

ROC Curve. Our main objective is to measure the performance of our filtering algorithms against this labeled data. The performance of a threshold based binary classifier is measured by Receiver Operating Characteristics (ROC) [Fawcett 2006]. Primarily, our filter identifies if a message is inconsequential (True) or informative (False). When such decisions are correct, that is, they agree with the manually obtained labels, they are considered true positives (*TP*) and true negatives (*TN*), respectively. If not, they are considered as false positives (*FP*) and false negatives (*FN*), respectively. The true positive rate ($TPRate = \frac{TP}{TP+FN}$) denotes the ratio of the number of correctly retrieved inconsequential messages to the total number of inconsequential messages. The false-positive rate ($FPRate = \frac{FP}{FP+TN}$) denotes the ratio of the number of informative messages that are incorrectly identified as inconsequential to the total number of informative messages.

The ROC curve is the curve obtained by plotting the FPRate versus the TPRate. For every classifier, there are chances that negative samples are classified as positive and that results in false positive samples. If a classifier randomly assigns labels to the samples, then it is expected that FPrate = TPRate. For an ideal classifier, TPRate = 1 even with FPRate = 0. The area under the ROC curve (AUC) is a measure of how accurate a classifier is. For a random decision maker, the expected AUC = 0.5, whereas for an ideal classifier AUC = 1.

In the next section, we report ROC curves for the different algorithms. When $\lambda = 0$ every message is classified as inconsequential. This corresponds to a TPRate and FPRate 1 and 1, respectively. Similarly, when λ is very high, it scales up the threshold ($\lambda * \theta = 1$) and no message is classified as inconsequential. This corresponds to a TPRate and FPRate 0 and 0, respectively. The ROC curve contains all the intermediate points between these two extremes, which are obtained by varying λ. This corresponds to increasing or decreasing the number of messages labeled inconsequential and the resulting reduction in the log file size. Within a storage system, too, λ can be tuned as per the desired log reduction rate. The area under the ROC curve is a measure of overall filter performance. A higher area under curve denotes higher TPRate with lower FPRate. For an ideal filter, area under curve is 1.0.

4.2. Results and Inferences

4.2.1. Filtering Algorithms. We experiment with the three approaches of filtering algorithm, namely the Conjunctive, Disjunctive, and Markovian approaches, to determine the best. Also, we experimented with the two flavors of the algorithm with simple filtering (Algorithm 1) and repetitive filtering (Algorithm 2) techniques and plot their ROC curves in Figures 5 and 6, respectively. The area under curve for each of these curves is presented in Table III. We note that the Markovian approach is the best of the three approaches for computing prediction probabilities.

4.2.2. Incorporating User Feedback. In this section, we show experimentally that the filtering accuracy can be improved by incorporating user feedback. We modify the probability formulations to incorporate prior bias indicative of user intent as discussed in Section 3.4.2. We learned the optimal values of the bias parameters over the training data using an evolutionary approach with an open-source implementation of genetic algorithms [Perone 2009]. As the Markovian approach was shown to be superior to other approaches, we discard the other alternatives, namely the Conjunctive and

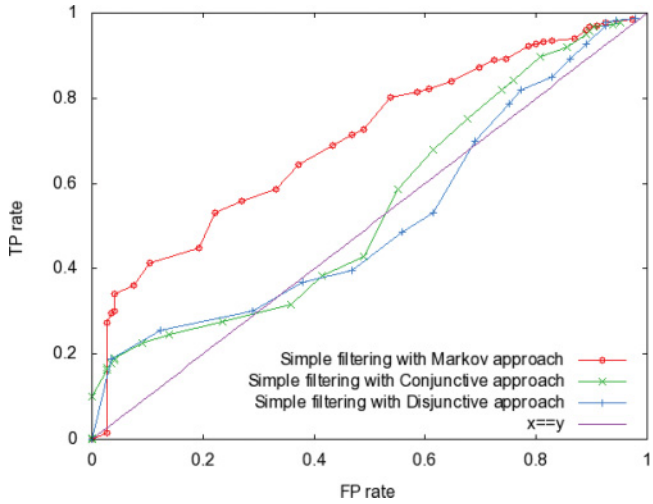

Fig. 5. ROC curve for simple log filtering with different approaches.

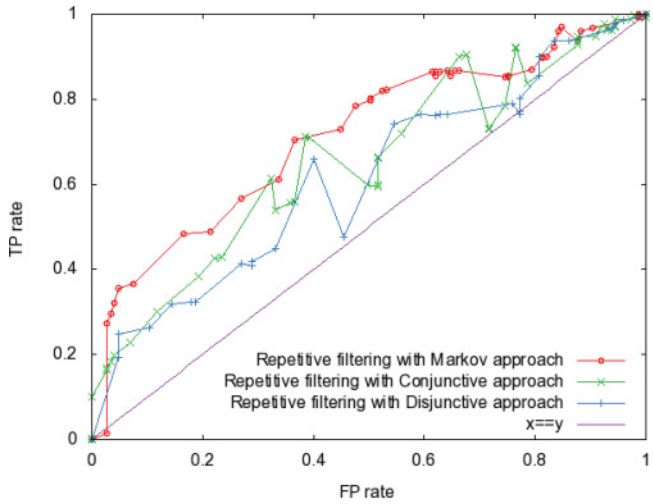

Fig. 6. ROC curve for repetitive log filtering with different approaches.

Table III. Area Under Curve for the Two Filtering Algorithms' Different Approaches

Algorithm	Markov	Conjunctive	Disjunctive
Simple Filtering	0.673	0.499	0.498
Repetitive Filtering	0.712	0.650	0.613

Disjunctive approaches. We show the ROC curves of the simple and repetitive filtering algorithms after incorporating user feedback and contrast it with the ROC curve of the unsupervised learning mode.

The evolutionary approach of optimization seeks to find the best possible parameters (bias values) for a given fitness criterion. Essentially, the genetic algorithm evaluates the given fitness function for different values of chromosomes (bias parameters) generated through an evolutionary approach. The total number of chromosomes evaluated

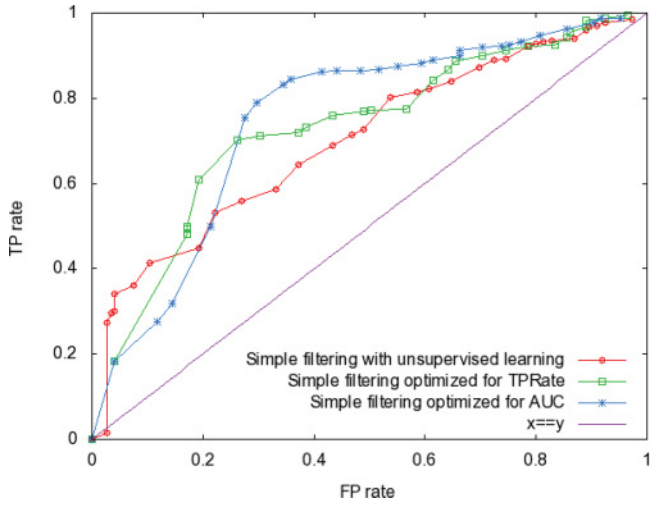

Fig. 7. ROC curves for simple filtering algorithm incorporating user feedback.

is the product of the number of generations and the size of the population in each generation, which can be large. To ensure timely completion of the optimization process, we need to define the fitness function such that it can be quick and lean. We evaluate the following different fitness functions for optimizing the filter performance.

(1) Optimizing for AUC: In this optimization criterion, we are seeking to maximize the area under the ROC curve for the filtering algorithm. This criterion translates to optimizing the overall performance of the filter. In this approach, the fitness function returns the area under the ROC curve by first finding a large set of tuples <TPRate,FPRate> to obtain a smooth ROC curve and then estimating the area under it. The multiple points of the ROC curve are obtained by varying λ in the filtering algorithm. This process has to be repeated for every chromosome value supplied by the genetic algorithm. As repetitive filtering (Algorithm 2) is computationally more expensive we used only simple filtering (Algorithm 1) in the fitness function. The resulting best chromosome is used to report the improvements with user feedback for both the algorithms.
(2) Optimizing for TPRate: In this mode, we are seeking to optimize the TPRate in a small region where FPRates is approximately around 20%. This criterion is with the assumption that up to 20% of FPRate can be tolerated and therefore we are interested in maximizing the TPRate in this region. We utilize the repetitive filtering algorithm and fix the scaling factor (λ) to a fixed value that corresponds to FPRate being 20% in unsupervised mode. The fitness function is set to return the TPRate corresponding to this fixed scaling factor and the given chromosome (bias parameters) using repetitive filtering (Algorithm 2).

In Figure 7 we plot the ROCs obtained by evaluating the simple filtering algorithm (Algorithm 1) over the testing data and with the bias values optimized for AUC and TPRate. The three curves correspond to unsupervised learning and the prior bias set to optimized AUC and optimized TP rate, as discussed above.

Similarly, in Figure 8, we plot the ROCs obtained by evaluating the repetitive filtering algorithm (Algorithm 2) over the testing data.

Note the improved ROCs after incorporating user feedback as compared to unsupervised learning for both the filtering algorithms. We summarize the area under curves

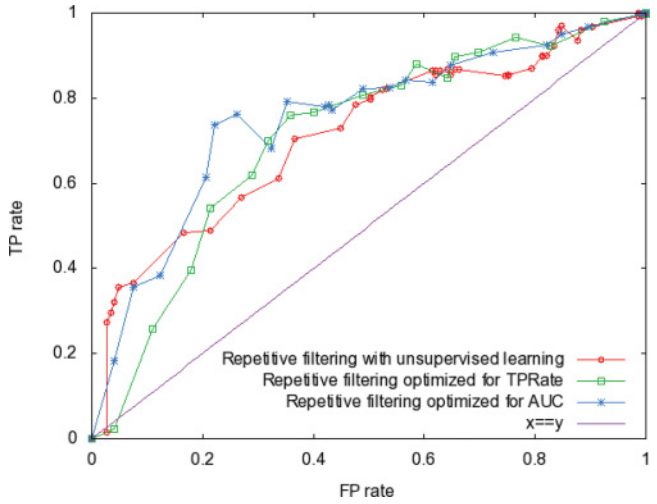

Fig. 8. ROC curves for repetitive filtering algorithm incorporating user feedback.

Table IV. Area Under the Curve for the Different Algorithms after Incorporating User Feedback

Algorithm	Unsupervised learning	Supervised learning optimized for TPRate	Supervised learning optimized for AUC
Simple filtering	0.673	0.694	0.701
Repetitive filtering	0.712	0.708	0.749

for the different approaches in the Table IV. We infer that repetitive filtering algorithm is superior to simple filtering algorithm both in the unsupervised mode and after incorporating user feedback. We also note the effect of incorporating user feedback using two different optimization criterion, namely optimizing for TPRate two gains in area under the curve for the two algorithms are similar when they are optimized for using the optimization criterion Further,

4.2.3. Systems Implication. In the previous sections, we viewed the log filter as a binary classifier and accordingly used ROC curves to measure its performance from a machine-learning perspective. The primary purpose of our filter is to identify and discard inconsequential messages in the log to shrink it to the budgeted log size. Therefore, we define the log reduction rate as a relevant metric for our filter and discuss its implications from a systems perspective in this section.

TPRate is indicative of correctness, as it is a measure of the fraction of inconsequential messages correctly retrieved ($\frac{TP}{TP+FN}$). FPRate is indicative of error as it a measure of the fraction of informative messages incorrectly labeled as inconsequential ($\frac{FP}{FP+TN}$). The reduction in log size can be computed as the proportion of inconsequential messages to the full size of the log file ($\frac{TP+FP}{TP+TN+FP+FN}$). In this section, we plot the log reduction rates against the error incurred (FPRate) and discuss the systems aspects of the usage of this filter. We only discuss the repetitive filtering approach as it is shown to be superior to be simple filtering approach as given by the area under the curves (Table IV).

In Figure 9, we show the log reduction rates obtained with the repetitive filtering algorithm in the unsupervised learning mode and then after the incorporation of user feedback with different modes of optimization. Note that the log reduction rate is closely related to TPRate and varies only slightly. This curve illustrates the different

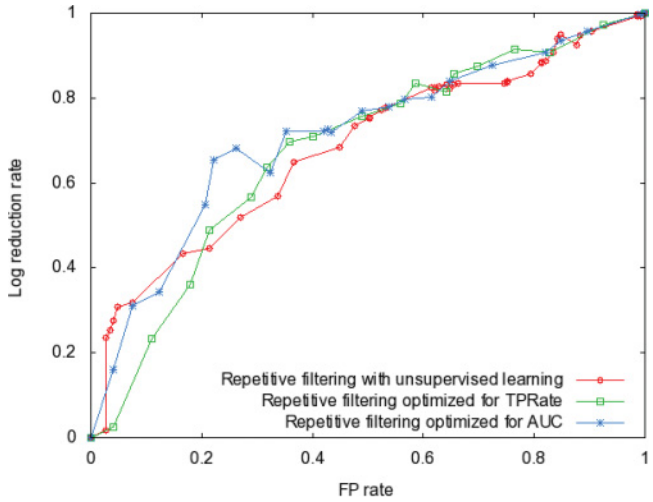

Fig. 9. Variation of log size reduction with error rates.

log reduction rates obtained by tuning the filter and and the resulting error or the amount of informative messages that are incorrectly discarded. For relaxed budgets, the classifier is set to high values of λ. This offers low FPRates, that is, nearly 0% of informative messages are incorrectly deemed to be inconsequential and discarded, but there is still a significant amount of log reduction rates. When the budgets are tighter, aggressive filtering is required to reduce log size. In this scenario, the classifier is set to low values of λ. This offers high TPRates, that is, nearly 100% of inconsequential messages are identified and discarded, but with an increased error, that is, more loss of informative messages. For example, we can achieve a log reduction of up to 35% with an error less than 12%.

Different installations of the system undergo different forms of usage and can vary widely in the budget constraints for the log size and in the log generation rate they experience. Therefore, log filtering needs to be tunable to produce different levels of log reduction rate. We show in Figure 9 that we can tune our log filter by varying λ to achieve the full range of the log reduction rate $[0 - 100\%]$. Accordingly, in an actual system, λ can be tuned to a suitable value. Also, this value is specific to the given log file, as the nature of log messages or the statistical distribution of log messages varies across different log files. This can be observed by fixing λ and experimenting with different log files and recording the resultant log reduction rates. To illustrate this, we recorded the log reduction rates obtained with a fixed λ over 35 log files obtained from different systems and from the same time frame. We fixed $\lambda = 200$, as this value offered low error rates and about 40% log reduction in the previous experiment (Figure 6). We plot the distribution of the resultant log reduction rates in Figure 10.

Resource Requirements. The algorithm requires small amounts of memory to hold the conditional probabilities across different message types. In our system, the number of different message types in a log typically is in a few hundred and the total memory requirement therefore is less than 1MB. We observed that the repetitive filtering step in our algorithm typically repeats for a maximum of 50 iterations and the complete processing takes less than a minute on a normal server class machine. Therefore, it is practical to include this filter within log pre-processing, prior to its transmission. The learning phase for supervised algorithms required about 6 hours.

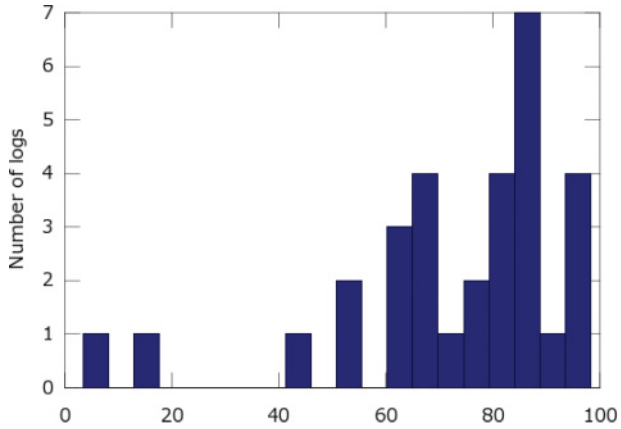

Fig. 10. Distribution of log reduction rates.

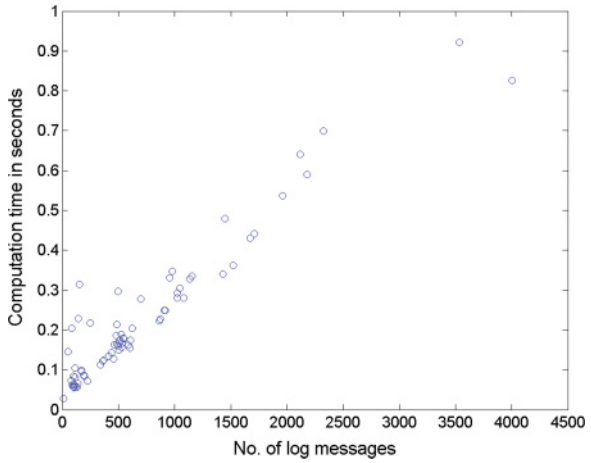

Fig. 11. Distribution of computation overhead with number of log messages.

Real Deployment Statistics. We installed the viewer in real production system within the company. Figure 11 shows the computation overhead for the log reduction with the number of log messages for one typical system. We limited the number of iterations in the repetitive filtering to 5. We obtain similar performance for other systems using unsupervised repetitive log filtering.

5. SUMMARY AND CONCLUSIONS

Presenting system log messages to the users in a user-friendly manner is very important task for diagnostics. In system logs, many messages are not useful in the due context, and it is necessary to hide these messages in the viewer to a user. The redundancy of a message also depends on the intent of the user and she may mark some of the messages as redundant or useful. Considering all these different scenarios, we have designed a user-friendly log viewer. We have demonstrated an unsupervised learning-based classifier that identifies the most inconsequential messages in a log file and filters them to shrink the log file to an acceptable size in order to present to a user. We further enhanced the capability of the classifier using supervised log messages with partial feedback from the user to further improve the experience of the user in mining

the log messages. We have tested it on real-world data sets and demonstrate significant log size reduction (30%) with minimal error rates (7%). Classifier effectiveness can be further improved by broadening the notion of context by including other information, such as the device identifier and timestamps.

ACKNOWLEDGMENTS

The authors thank Siddhartha Nandi for his insightful comments and support throughout this work. The authors also acknowledge Parag Deshmukh and Ajay Bakhshi for their help and support. Special thanks to Madhumita Bharde for taking this work forward towards product delivery.

REFERENCES

R. Agrawal, T. Imieliński, and A. Swami. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*. ACM New York, NY, 207–216.

S. Alspaugh, Beidi Chen, Jessica Lin, Archana Ganapathi, Marti Hearst, and Randy Katz. 2014. Analyzing log analysis: An empirical study of user log mining. In *28th Large Installation System Administration Conference (LISA14)*. USENIX Association, Seattle, WA, 62–77.

Christopher J. C. Burges. 1998. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.* 2, 2 (1998), 121–167.

Tom Fawcett. 2006. An introduction to ROC analysis. *Pattern Recog. Lett.* 27, 8 (2006), 861–874.

Anil K. Jain, Jianchang Mao, and K. Moidin Mohiuddin. 1996. Artificial neural networks: A tutorial. *Computer* 29, 3 (1996), 31–44.

Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. 2009. Understanding customer problem troubleshooting from storage system logs. In *FAST*, Vol. 9. 43–56.

J. Koshy. 2007. PMC based Performance Measurement in FreeBSD. Retrieved from http://people.freebsd.org/~jkoshy/projects/perf-measurement.

Time Kramer. 2003. Effective Log Reduction and Analysis Using Linux and Open Source Tools. Retrieved from http://www.giac.org/paper/gsec/3144/effective-log-reduction-analysis-linux-open-source-tools/105234.

Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. 2007. An adaptive semantic filter for blue gene/L failure log analysis. In *Proceedings of the 3rd International Workshop on System Management Techniques, Processes, and Services (SMTPS)*.

Andrew L. Maas and Andrew Y. Ng. 2010. A probabilistic model for semantic word vectors. In *Proceedings of the Workshop on Deep Learning and Unsupervised Feature Learning, NIPS*, Vol. 10.

Network Appliance. 2007. Proactive health management with autosupport. http://www.netapp.com/us/media/wp-7027.pdf.

W. Peng, T. Li, and S. Ma. 2005. Mining logs files for data-driven system management. *ACM SIGKDD Explor. Newslett.* 7, 1 (2005), 44–51.

Christian S. Perone. 2009. Pyevolve: A Python open-source framework for genetic algorithms. *SIGEVOlution* 4, 1 (Nov. 2009), 12–20. DOI:http://dx.doi.org/10.1145/1656395.1656397

R. R. Sarukkai. 2000. Link prediction and path analysis using Markov chains. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Netowrking*. North-Holland Publishing Co., Amsterdam, The Netherlands, 377–386.

S. A. Shahrestani, M. Feily, R. Ahmad, and S. Ramadass. 2010. Discovery of invariant BOT behaviour through visual network monitoring system. In *Proceedings of the 2010 Fourth International Conference on Emerging Security Information, Systems and Technologies*. 182–188.

Darrell Whitley. 1994. A genetic algorithm tutorial. *Stat. Comput.* 4, 2 (1994), 65–85.

Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2010. Experience mining google.s production console logs. In *Proceedings of the SLAML* (2010).

Z. Ziming, L. Zhiling, B. H. Park, and A. Geist. 2009. System log pre-processing to improve failure prediction. In *Proceedings of the IEEE / IFIP International Conference on Dependable Systems & Networks (DSN'09)*. 572–577.

Received January 2015; revised July 2015; accepted November 2015