Experience Report: Log Mining using Natural Language Processing and Application to Anomaly Detection

Christophe Bertero, Matthieu Roy, Carla Sauvanaud and Gilles Tredan
LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
Email: firstname.name@laas.fr

***Abstract*—Event logging is a key source of information on a system state. Reading logs provides insights on its activity, assess its correct state and allows to diagnose problems. However, reading does not scale: with the number of machines increasingly rising, and the complexification of systems, the task of auditing systems' health based on logfiles is becoming overwhelming for system administrators. This observation led to many proposals automating the processing of logs. However, most of these proposal still require some human intervention, for instance by tagging logs, parsing the source files generating the logs, etc.**

In this work, we target minimal human intervention for logfile processing and propose a new approach that considers logs as regular text (as opposed to related works that seek to exploit at best the little structure imposed by log formatting). This approach allows to leverage modern techniques from natural language processing. More specifically, we first apply a word embedding technique based on Google's `word2vec` algorithm: logfiles' words are mapped to a high dimensional metric space, that we then exploit as a feature space using standard classifiers. The resulting pipeline is very generic, computationally efficient, and requires very little intervention.

We validate our approach by seeking stress patterns on an experimental platform. Results show a strong predictive performance ($\approx 90\%$ accuracy) using three out-of-the-box classifiers.

***Keywords*—Anomaly detection, logfile, NLP, word2vec, machine learning, VNF**

I. Introduction

Gathering feedback about computer systems states is a daunting task. To this aim, it is a common practice to have programs report on their internal state, for instance through journals and logfiles, that can be analyzed by system administrators.

However, as systems tend to grow in size, this traditional logging method does not scale well. Indeed, scattered software components and applications produce heterogeneous logfiles. For instance, logging methods such as the common *syslog*, are extremelly flexible in their syntax (see the RFC [7]). Also, different logfiles may gather information with distinct types of information. For instance rule-based logging [4] traces the start and the termination of applications functions, while *syslog* event logging collects system activity. Each of them tends to describe a partial view of the whole system. In particular, [3] shows that event logging, assertion checking, and rule-based logging are orthogonal sources for system monitoring. Moreover, each partial view of the system, even when using the same logging method (or protocol), may not use the same keywords to express normal or erroneous behaviors. This plethora of available logfiles burdens log summarization.

As a result, source code analyzes and communications with application developpers are necessary for troubleshooting or auditing systems [17]. Notwithstanding, such non automatic processes are not acceptable in large computing system because troubleshooting for reconfiguration must be handled online. To address these challenges, a large number of studies proposed approaches to automate and scale up log analysis ([5], [8], [17], [23], [24]). Most approaches require however cumbersome log processing, for instance by manually tagging important events, or by parsing the source code functions to assess the fixed and variable parts of log events.

The contribution of this paper is to propose a new approach departing from this research line and considering log mining as a natural language processing task.

This approach has two main consequences, $i)$ we lose a part of the context by under-exploiting the specificities of each structured sentence according to a predefined pattern and, most importantly, $ii)$ our approach is agnostic to the format of the logfiles. Thus, while considering sets of logfiles as languages, we gain the ability to use modern Natural Language Processing (NLP) methods. In other words, we trade accuracy for volume, preferring the ability to inaccurately process large volumes of logfiles instead of accurately processing some tediously preprocessed logs.

As such, the question we explore in this work is: "What can off-the-shelf Natural Language Processing algorithms bring to log mining? ". We more particularly focus on such questions as "is my system in state A or state B? ". The proposed approach is rather simple and brutal. Instead of precisely tracking the events related to a transition from A to B, we collect large amounts of log events related to systems in states A and B. We then transform the logs into multidimensional vectors of features (using NLP algorithms) and train a classifier on the resulting data. The resulting pipeline is a relatively standard big data application, where we target the realization of classifiers providing accurate information about the target system state. We believe this approach is specifically interesting due to the expensive expertise usually required to preprocess the logs.

We show in this paper, through a series of experiments, that with minimum setup effort and standard tools, it is possible to automatically extract relevant information about a system state. We more particularly use the `word2vec` algorithm of Google [16] for log mining, which is an algorithm for learning high-quality vector representations of words. It notably has been used for NLP in some previous works but not for the analysis of logfiles.

Through experiments, we illustrate the potential benefits of our approach, by providing answers to system administrators' questions when data is massively available. As an illustrative example, we focus on the detection of stress related anomalies over a broad range of configurations. More specifically, we deployed on a virtual cloud environment a virtual network function running a panel of three applications, namely a proxy, a router, and a database, to which we applied a large variety of stress patterns by means of fault injection (high CPU and memory consumption, high number of disk accesses, increase of network latency and network packet losses). We show that by simply analyzing the results of NLP processed logfiles, it is possible to detect stressed behaviors with $\approx 90\%$ accuracy.

In the following, we first present in Section II the rationale of our log mining approach, and describe our use of fault injection for validation purposes in Section II. Then, in Section III we define our case study, the experimental platform on which we deployed it, and the implementation of our approcah on this platform. Section IV presents some promising experimental results. In Section V we discuss our results, and analyze their threats to validity. Section VI describes related works regarding NLP and log mining for detection purposes. Finally, we conclude this paper in section VII.

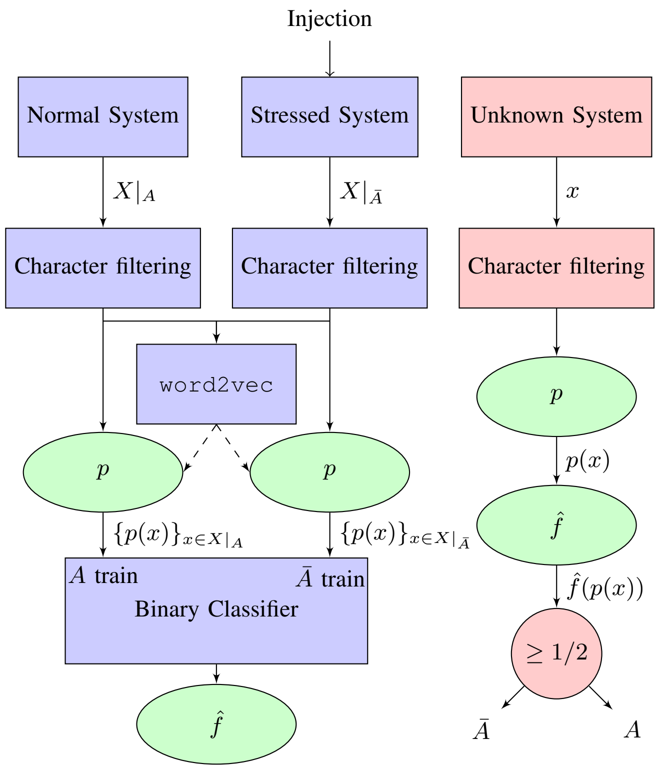

Fig. 1: General approach overview. *Left:* Training. *Right:* Inference.

II. Approach

A. General approach overview

The approach proposed as the contribution of this paper is presented in Figure 1.

Consider a set of logfiles related to a given system. Each of these logfiles contains a varying amount of lines, each line consisting of one application of the system reporting an event. Each log event (line) is a list of words.

As we consider logfiles as a natural language, we analyze these logfiles using Natural Language Processing tools. As such, we first remove all non alphanumeric characters (as required by `word2vec`) and replace them by spaces, namely `sed 's/[^a-zA-Z0-9]/ /g'`.

Secondly, we use `word2vec` from [16], a popular embedding tool employed by Google to process natural language. In a nutshell, `word2vec` produces a mapping from the set of words of a text corpus (a set of logfiles in our case) to an euclidean space say T. In the case of a 20-dimensions space $T \subset \mathbb{R}^{20}$. Thus, each word of an event gets assigned coordinates in a vector space. The enjoyable property of `word2vec` is its ability to produce *meaningful* embeddings, where similar words end up close, whereas words that are not related to each other end up far away in the embedding space.

Once each word has been mapped to the embedding space T, we define the position of a log event as the barycenter of its words. Following a similar scheme, once all log events from a given logfile have been mapped to points, we define the position of this logfile as the barycenter of the position of its log events. Hence, at the end of the process, each logfile is mapped to a single point in T. This drastic compression has one major interest: it produces a compact and useful input to traditional classifiers. Assuming $\mathcal{X}$ represents the set of all possible logfiles, such mapping can be represented as a function:

$$\begin{aligned} p : \mathcal{X} &\to T \\ x &\mapsto p(x). \end{aligned}$$

Now, assume that one has access to a large set X of observations (logfiles) on the system, corresponding to two states that we would like to characterize, say A and $\bar{A}$. Let $X|_A$ and $X|_{\bar{A}}$ be the corresponding logfiles sets. By the above described process, every observation $x \in X = X|_A \cup X|_{\bar{A}}$ can be assigned to a coordinate $p(x) \in T$.

In a third step, we train a classifier, named $\hat{f}$ hereafter, on $p(x|x \in X|_A)$. A typical such classifier $\hat{f}$ is an approximation of the ideal separation function:

$$\begin{aligned} f : T &\to [0, 1] \\ p(y) &\mapsto \mathbb{P}(A|y). \end{aligned}$$

The training of a classifier requires an available set of labeled data. These labels may be for instance: normal and anomalous. In cases that labeled data is not available, one can generate them by monitoring a system while experiencing normal and anomalous behaviors. Since anomalous behaviors are undesired events and, as such, usually not frequent in recent systems, they need to be synthesized using techniques such as fault injection. In this paper, we generate sets of normal and anomalous behaviors in a controlled manner using fault injection techniques for all anomalous behaviors, as represented in Figure 1.

Once the training is finished, the resulting classifier is used to provide, given any new production logfile x, an inferred state (anomalous or not) $\hat{f}(p(x))$ that we claim is a good approximation of the actual stress status of the system, i.e., $\mathbb{P}(A|x) \simeq \hat{f}(p(x))$. It is actually expressed as a probability and we need to set a limit over which a system is categorized as stressed, say $1/2$ as in Figure 1. In the case x contains unencountered words, those are simply ignored.

III. Case study and experimental platform

A. Case study

We hereby present our case study on virtual network function (VNF) called Clearwater[1] as well as the workload generator used during our experiments to simulate actual users of this target system. This case study was used in our preivous work [19] for anomaly detection based on monitoring data.

It constitutes a meaningful case study in that it deploys several components of different roles (e.g., router, proxy and database). While we apply our approach with no specific configuration nor a priori knowledge of the implementations for each component, we consider that our approach has good chances to generalize to various case studies.

1) Description: The service is an open source VNF named Clearwater. It provides voice and video calls based on the Session Initiation Protocol (SIP), and messaging applications. Clearwater encompasses several software components and we particularly focus our work on Bono, Sprout, Homestead shown in Figure 2.

Bono is the SIP proxy implementing the Proxy-Call/Session Control Functions. It handles users' requests and routes them to Sprout. It also performs Network Address Translation traversal mechanisms.

Sprout is the IMS SIP router, receiving requests from Bono and routing them to the adequate endpoints. It implements some Serving-CSCF and Interrogating-CSCF functions and gets the required users profiles and authentication data from Homestead. Sprout can also call application servers and actually contains itself a multimedia telephony (MMTel) application server, whose data is stored in another Clearwater component not presented in this work (when calls are configured to use its services).

Homestead is a HTTP RESTful server. It either stores Home Subscriber Server (HSS) data in a Cassandra database and masters data (i.e., information about subscribed services and locations), or pulls data from another IMS compliant HSS.

Bono, Sprout, and Homestead work together to control the sessions initiated by users and handle the entire CSCF. Our case study encompasses these three components, each one being deployed on a dedicated virtual machine (VM) of our virtualized experimental platform (see Section III-B).

[1]http://www.projectclearwater.org/about-clearwater/

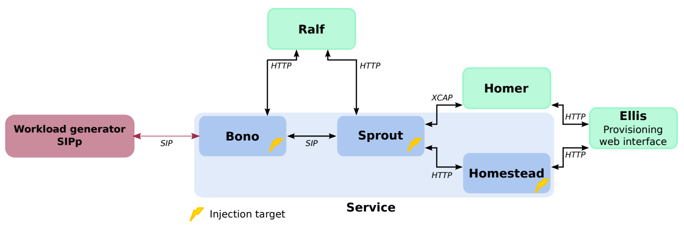

Fig. 2: Clearwater deployment.

2) Workload: IMS workloads can be emulated by means of the SIPp benchmark[2]. The benchmark contains a workload that can be configured with a number of calls per second to be sent to the IMS, and a scenario. The execution of a *scenario* corresponds to a call. A scenario is described in terms of SIP transactions in XML. A SIP transaction corresponds to a SIP message to be sent and an expected SIP response message. A call fails when a transaction fails. A transaction may fail for two reasons: either a message is not received within a fixed time window (i.e., the timeout), or an unexpected message is received. Unexpected messages are identified by the HTTP error codes 500 (Internal Server Error), 503 (Service Unavailable) and 403 (Forbidden).

The scenario run for our experimentations simulates a standard call between two users and encompasses the standard SIP REGISTER, INVITE, UPDATE, and BYE messages. The scenario is available online[3]. Timeouts are set to 10 sec as in similar experimental campaigns [2].

3) Fault injection for training and validation: Fault injection is used in our study for collecting logfiles representing both normal behaviors and stressed behaviors of a target system, in order to provide them as inputs for the training and validation of the classifiers. We emulate errors by means of injection tools that implement systems stressing. These tools were used in our previous work [19].

We call the orchestration of several executions of the target system in presence or not of error emulations an *experimental campaign*. In the following we present the errors that our injection tools emulate and describe the execution of an experimental campaign.

Error emulation. We emulate the following five types of errors, which we will be referring to as *CPU, memory, disk, network packet loss, and network latency* errors respectively:

(1) high CPU consumption,
(2) misuse of memory, i.e., increase of memory consumption,
(3) abnormal number of disk accesses, i.e., large increase of disk I/O accesses and synchronizations,
(4) network packet loss,
(5) network latency increase.

CPU errors. Abnormal CPU consumptions may arise from programs encountering impossible termination conditions leading to infinite loops, busy waits or deadlocks of competing actions, which are common issues in multiprocessing and distributed systems.

[2]http://sipp.sourceforge.net/index.html

[3]https://homepages.laas.fr/csauvana/sipp_scenario/issre2016_sipp_scenario.xml

Memory errors. Abnormal memory usages are common and happen when allocated chunks of memory are not freed after their use. Accumulations of unfreed memory may lead to memory shortage and system failures.

Disk errors. A high number of disk accesses, or an increase of disk accesses over a short period of time, emulate disks whose accesses often fail and lead to an increase in disk access retries. It may also result from a program stuck in an infinite loop of data writing.

Network packet loss and latency errors. Such errors may arise from network interfaces of the target system or from the network interconnection of the virtualized infrastructure hosting the system. We emulate packet losses and latency increases. Packet losses may arise from undersized buffers, wrong routing policies or even firewall misconfigurations. Latency errors may originate from queuing or processing delays of packets on gateways or at the target system level.

From the definition of these error types, an important experimental parameter is the injection intensity, i.e., the expected impact magnitude of the different injections from users points of view. In our study, we present results for the detection of errors with high intensities. In other terms, experimental campaigns perform injections that strongly affect the target system capability to answer users requests.

Table I presents the intensity levels that we calibrated for our Clearwater case study.

Error type	Unit	Intensity level
CPU	%	90
Memory	%	97
Disk	#process	50
Network packet loss	%	8.0
Network latency	ms.	80

TABLE I: Injection intensity levels.

Regarding the memory, disk and CPU injections, the intensity values of errors are constrained by the capacity of the operating systems (OSs) on which are deployed the applications of our case study. In other words, the intensity levels correspond to the maximum resource consumption allowed by the OS before killing the execution of the injection agent.

Considering the remaining types of injections, the corresponding intensity levels is set so as to lead to around 99% of unsuccessfully answered requests when applied in at least one VM. The unsuccessfully answered requests rate can be known from the workload logfiles.

Experimental campaigns. The experimental campaign is conducted using a customizable main script that either launches normal or anomalous executions of the target system. The experimental campaign either launches normal or stressed executions of the target system. An execution, be it normal or anomalous, produces one logfile for each VM of our target system.

We define a campaign to run as many normal executions as the number of stressed executions. The selected number of stressed executions is configured to represent all combinations of different injections (i.e., the injection of each error type, in each VM).

When running an anomalous execution, the configured injection starts after t seconds from the target system boot time, where t is randomly selected in a preconfigured interval. This process adds randomization to the set of collected logfiles, a prerequisite for the generalization of our results.

Additionally, consecutive executions of a campaign are separated by the reboot of all VMs of the target system and the workload in order to be sure to restart from a clean and unpolluted state.

As a result, the parameters of an experimental campaign are as follows: $i)$ target VMs listed in *l_vm*, $ii)$ error types listed in *l_type*, $iii)$ an injection duration set in *inject_duration*, $iv)$ a clean run duration set in *clean_run_duration*, $v)$ an interval of values defining after which time an injection can start after a reboot set in *interval*.

Moreover, a campaign is executed as follows. Each error type is injected in a first VM, then in a second VM, etc. with reboots of the target system and the workload before each new execution. The stressed executions are orchestrated as explained in algorithm 1. Then the same number of normal executions are performed.

Algorithm 1 Orchestration of stressed executions of the target system in an experimental campaign

```
Input: l_vm, l_type, inject_duration, interval,
  clean_run_duration

  start_workload()                          ▷ Clean run
  for vm in l_vm do                         ▷ Runs with injections
      for err in l_type do
          start_workload()
          rand_time = random_int(interval)
          sleep(rand_time)
          inject = Injection(err, inject_duration)
          inject_in_vm(vm, inject)
          stop_workload()
          reboot_vms()
      end for
  end for
```

B. Experimental platform

In the following, we first present the platform on which we run experiments. Then we describe the implementation required to carry out our experiments namely the injection agents, experimental campaign parameters, and the collection of logfiles.

1) Platform: We deployed our target system on a virtualized platform. The platform is composed of a cluster including two hypervisors and several VMs. Four VMs are deployed for our target system: one VM runs the workload and the other three respectively host the components Bono, Sprout and Homestead of Clearwater. The workload VM also has the means to control the experimental campaign launch. Two other VMs are respectively used to store logfiles collected from the target system and to analyze the stored logfiles. The deployment of the VMs is illustrated in Figure 3.

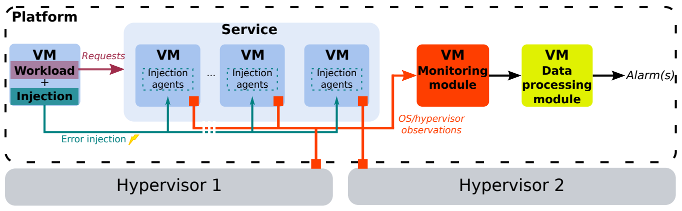

Fig. 3: Virtualized platform.

The platform is a VMware vSphere 5.1 private cloud composed of 2 servers Dell Inc. PowerEdge R620 with Intel Xeon CPU E5-2660 2.20 GHz and 64 GB memory. Each server has a VMFS storage. Each VM deployed for the target system implementation has 2 CPUs, a 10 GB memory, a 10 GB disk and runs the Ubuntu OS. VMs are connected through a 100 Mbps network.

2) Fault injection: Injections in the target system are carried out by injection agents installed in these VMs. There is one injection agent for each error type in each VM of a target system. Agents are run and stopped through an SSH connection orchestrated by the campaign main script. They emulate errors presented in Section III-A3 by means of a software implementation.

CPU and disk errors are emulated using the stress test tool `stress-ng`[4]. CPU injections run 2 processes (there are 2 cores in each VM) running all the stress methods listed in the tool documentation. The percentage of loading is set according to the intensity level of the injection.

Disk injections start several workers writing 50 Mo and 50 workers continuously calling the *sync* command, with an *ionice level* of 0. The number of writing workers is set according to the intensity level of the injection.

Memory injections are run by means of a python script reserving memory space while continuously checking whether the amount of memory space reserved by the script corresponds to the amount set by the intensity level of the injection.

Finally, we use the Linux kernel tools *iptables* and *tc* for the injection of network latencies on the POSROUTING chain, and *iptables* on the INPUT chain for the injection of packet losses. All network protocols are targeted.

3) Experimental campaigns parameters: An experimental campaign corresponds to the execution of a customizable main script that starts the workload of our target system, and either makes clean run of this target system or makes runs while performing injections in the target system VMs.

The parameters of the experimental campaigns we run are as follows. The injection duration is calibrated so as to affect several instances of workload executions (an execution lasts less than 1 sec). We calibrated the injection duration to be 10 min long in order to collect around 5000 lines of logfile for each clean run and injection. Also, we calibrated the clean run duration to be 30 min. Finally, we calibrated the start of injections to be randomly selected in the interval from 1 to 10 min. This interval allows the VMs to stabilize after a reboot.

[4]http://kernel.ubuntu.com/~cking/stress-ng/

```
Apr 18 06:44:37 cw-011 restund[1368]: stun
server ready
Apr 18 06:44:37 cw-011 bono[1284]: 2005 -
Description: Application started. @@Cause:
The application is starting. @@Effect:
Normal. @@Action: None.
Apr 18 06:45:01 cw-011 CRON[1521]:
(root) CMD (/usr/lib/sysstat/sadc 1 1
/var/log/sysstat/clearwater-sa`date +%d` >
/dev/null 2>&1)
```

Fig. 4: Example of *syslog* events.

Our experimental campaign parameters are summarized in Table II.

Campaign parameters
• l_vm $= \{Bono, Sprout, Homestead\}$
• l_type $= \{CPU, memory, disk, latency, packet\_loss\}$
• injection_duration $= 10\,\text{min}$
• clean_run_duration $= 10\,\text{min}$
• interval $= [1:10]\,\text{min}$

TABLE II: Injection campaign parameters of the four experimentations.

4) Logfiles collection: The logfiles that we use in this study are generated by the Linux-based Ubuntu OS using *syslog*, the standard tool for message logging. Events are logged with a predefined pattern containing in that order the date of the event issue, the hostname of the equipment delivering the event, the process delivering the event, a priority level, the id of the process delivering the event and finally the message containing free-formatted information. For instance, no performance metrics of the system are logged. A example of *syslog* events is provided in Figure 4.

Results of previous studies [3] show that *syslog* event logging is the more suitable method to use in this context, although a combination of the several methods increases the failure coverage. The syslog facility has the advantage to gather several applications events.

During experimental campaigns, logfiles are collected by means of agents (they are represented by orange squares in Figure 3) and stored in a database for later analysis.

IV. Results

In this section, we quantitatively study the effectiveness of the presented approach by presenting the analysis results over 660 logfiles. After briefly introducing the considered metrics, we will detail the obtained results.

The main research question we seek to answer is: *Using only syslog files as input, how accurately can our algorithm distinguish Stressed and non Stressed systems?* The secondary questions are i) how sensitive are the results to the parameters used to calibrate the models of our approach? and ii) what is the ability of our approach to issue quick decision on a system state?

A. Materials and Metrics

Using the testbed presented in Section III-B we generate a set of 660 logfiles that will constitute the basis of our models training. Exactly half of these (330) originate from normal unstressed system executions. The other half captures systems with injected faults. More precisely, we ran 22 replications for each of the 5 injection campaigns over each of the 3 target VMs of our case study, for a total of $(22*3*5) = 330$ stressed logfiles.

Word2Vec training: To establish the `word2vec` training set, we use the concatenation of all 660 logfiles from which we removed all non alphanumeric characters.

`word2vec`, originally designed for NLP tasks, can be tuned with a number of different options. The most important parameter is the embedding space dimension $dim(T)$, its impact is detailed in Section IV-B2. The other parameters mostly allow to setup filters in order to optimize the computation. We deactivated all of them to keep the maximum amount of information available to the classifier. Finally, from the two methods proposed in the implementation of `word2vec`, namely `skip-gram` and `cbow` (defining whether the source context words should be predicted from target words or the opposite[5]), we chose `cbow` because of its simplicity, in order to provide an "as-simple-as-possible" solution.

Given the relatively small size of our text corpus (compared to all the English texts available on the web, namely `word2vec`'s original usecase), and the well known efficiency of the `word2vec` implementation, the overall computation is tractable on a standard computer (see Section IV-B3). Therefore, the philosophy behind implementation choices is the following: keep it simple, and keep the maximum amount of information.

From word coordinates to logfile coordinates: The output of `word2vec` is a file containing the coordinates of the $233k$ distinct words of our training corpus in T. To transform logfiles into coordinates in T, we explored two standard strategies:

bary In the barycenter approach, we first compute the position of each line of a logfile, defined as the average position of all the words it contains. Then, the position of the file is defined as the average of all its line:

$$p(f) =_{\text{def}} 1/|f| \sum_{l \in f} 1/|l| \sum_{w \in l} p(w).$$

tfidf Term frequency - inverse document frequency is a standard metric of information retrieval. Compared to the barycenter approach, words are weighted by their frequency in the document. That is, a frequent (common) word will proportionally have less weight than a rare word when computing the average position of a logfile. We relied on the `scikit-learn`[6] standard implementation of the function.

The output of this step is a matrix of $660 \times dim(T)$ entries decorated with their corresponding target labels (stressed, unstressed system).

[5]See one implementation explaination https://www.tensorflow.org/tutorials/word2vec. Last read on 13/08/2017.

[6]http://scikit-learn.org/

Classifiers: Binary classifiers are amongst the most common and understood classifiers in machine learning. We restricted our study to three simple and state of the art approaches: Naive Bayes, Random Forests and Neural Networks. We relied on the following `scikit-learn` library implementations: Random Forest Classifier, MLP Classifier, and Gaussian NB. All these algorithms belong to the class of supervised algorithms. In other words, they require labeled training data, although we could have used unsupervised approaches such as the ones tested in [8], i.e., Principal Components Analysis and Invariant mining.

Again, the philosophy of our approach is to refrain from fine tuning those implementations and to assess the global strategy as a hole. We therefore used the default parameters on all these algorithms.

Classifier Assessment: To assess the classification accuracy, we used the standard 10-fold validation approach. We first randomly divided the training set in 10 equal sized chunks. Each possible group of 9 chunks was used to train our classifier while the remaining chunk was used as a test.

Let $\{X_i\}_{1 \le i \le 10}$ be a partitioning of X into 10 chunks. Let X_j be the tested chunk, and let T_j (resp. F_j) be the subset of stressed (resp. unstressed) logs of X_j. The set of true positives TP_j for X_j is defined as:

$$TP_j = \{x \in X_j \text{ s.t. } \hat{f}_j(x) \ge 1/2 \wedge x \in T_j\}.$$

Logs that belong to stressed machines and to which the classifier $\hat{f}_j$ (trained using $\cup_{i \neq j} X_i$) assigned a probability greater than 1/2 of being stressed are true positives for X_j. Similarly, the set of false positives FP_j for X_j (logs belonging to unstressed machines but detected as more likely stressed) is defined as:

$$FP_j = \{x \in X_j \text{ s.t. } \hat{f}_j(x) \ge 1/2 \wedge x \in F_j\}.$$

Notice that the true negative and false negative sets are symmetrically defined.

To get a closer look at $\hat{f}_j$, one can use Receiver Operating Characteristics (ROC). That is, let $s \in [0, 1]$ be a "safety level" one wants to apply to $\hat{f}$-based decisions. Let $X_j^s = \{x \in X_j, \hat{f}_j(x) \ge s\}$ be the subset of X_j containing only the logs detected as stressed with probability at least s. For each value of s, it is thus possible to define a true positive rate $TPR_s = |X_j^s \cap T_j|/|T_j|$ and a false positive rate $FPR_s = |X_j^s \cap F_j|/|F_j|$. The graphical representation of the obtained $\{FPR_s, TPR_s\}$ couples provides a precise visual description of $\hat{f}$'s performance, as in Figure 5 that will be presented shortly hereafter.

B. Results analysis

In the following, after exploring the detailed results obtained using a typical trained classifier, we study the impact of the embedding host space dimension. We then study the runtime overhead of our approach.

1) Accuracy: Figure 5 presents the ROCs obtained on a typical configuration. More precisely, in this setup, we used $dim(T) = 20$ and explored various aggregation/classifier configurations. The results are very good, with Neural Network

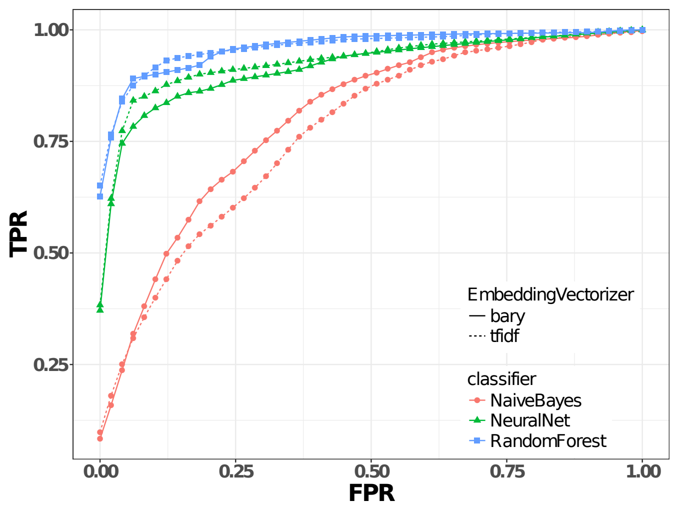

Fig. 5: Receiver Operating Characteristic of 3 classifiers, for $dim(T) = 20$. This plot shows the True Positive Rate of every classifier as a function of the False Positive Rate of the same classifier.

and Random Forest exhibiting a strong classification accuracy ($> 95\%$ AUC). The aggregation technique (i.e., based on tf-idf or barycenter) has little impact. Naive Bayes performs considerably better than random (77% and 81% AUC for tf-idf and barycenter resp.), but is visibly less precise than the other two classifiers. These very good results confirm the soundness of the approach.

One can have a more detailed look at the origin of misclassifications. Table III exhibits the confusion matrix of Neural Network (using barycenter and $dim(T) = 20$). Although around 90% of the targets get correctly categorized, one can see that the errors are slightly leaning towards false positives (that is, an unstressed system is wrongfully categorized as stressed). Although this is not the purpose of this study, it is possible to exploit this imbalance for an overall better classification accuracy (for instance by raising a $1/2$ limit over which a system is categorized as stressed). The stress patterns are not very homogeneously detected, with Latency stress being 7 times more efficiently detected than CPU stress. However, because of the accuracy of the considered classifier, these results only concern a small number of events, and therefore have a low statistical power. Table IV presents the misclassified entries by application: all three applications (namely Bono, Sprout and Homestead) yield to similar classification accuracy.

TABLE III: Confusion matrix for the Neural Network classifier, using $dim(T) = 20$, and barycenter: detailed by stress type

Stress Type	Detected As Stressed (True)	Detected as Unstressed (False)
No Stress	0.115	0.885
Packet loss	0.939	0.061
Latency	0.985	0.015
Memory	0.939	0.061
Disk	0.970	0.030
CPU	0.893	0.106

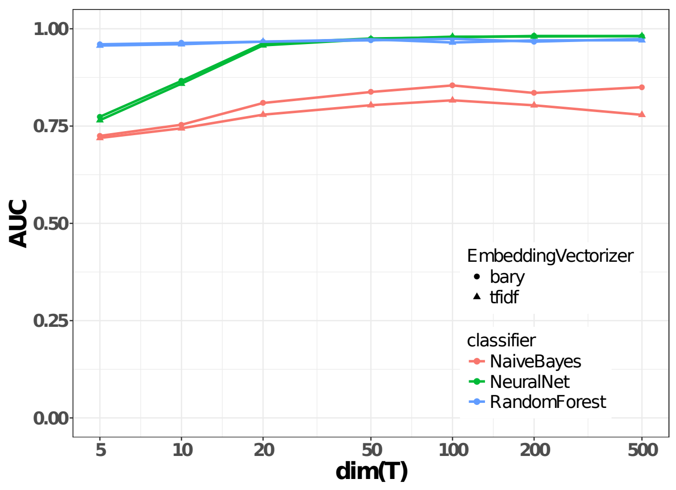

Fig. 6: Area Under the ROC Curves (AUC) capturing the performance of our classifiers, as a function of the number of dimensions of the embedding space

TABLE IV: Confusion matrix for the Neural Network classifier, using $dim(T) = 20$, and barycenter: detailed by application

Target Machine	Requests	Number of misclassifications	Success Rate (%)
Bono	220	19	91.4
Sprout	220	17	92.3
Homestead	220	20	90.9

2) Parameters sensitivity: We here focus on two choices of importance: the dimension of the embedding space $\dim(T)$, and the classifier algorithm. To compare our classifiers, we use the Area Under Curve (AUC) measure. In a nutshell, it measures the area under the ROC of a classifier. That is, an AUC of 1 denotes a perfect classification, while an AUC of 0 denotes a worse than random prediction. It is also commonly presented, given a random positive (stressed) and random negative (unstressed) example, as the probability for the classifier to rank the negative example below (that is, less stressed) the positive example. The ROC AUC is know to well summarizes ROC curves [1].

Figure 6 provides the AUC measures for our 3 considered classifiers for various embedding space dimensions. As expected, increasing the number of dimensions increases the classification accuracy: more information helps. This increase is however very limited: apart from Neural Network, where increasing dimensions from 5 to 20 has a visible impact, classifier accuracies all stay stable for $dim(T) > 20$. This is good news, as such parameter can be hard to tune a priori.

More generally, this figure confirms the previous observations: classification is very accurate, especially using Neural Network and Random Forest, with AUCs consistently scoring above 0.95.

3) Timing performance: When selecting a classifier, the expected classification accuracy is the most important criteria. However, in operational contexts, another crucial criteria is the computational complexity of both training and prediction.

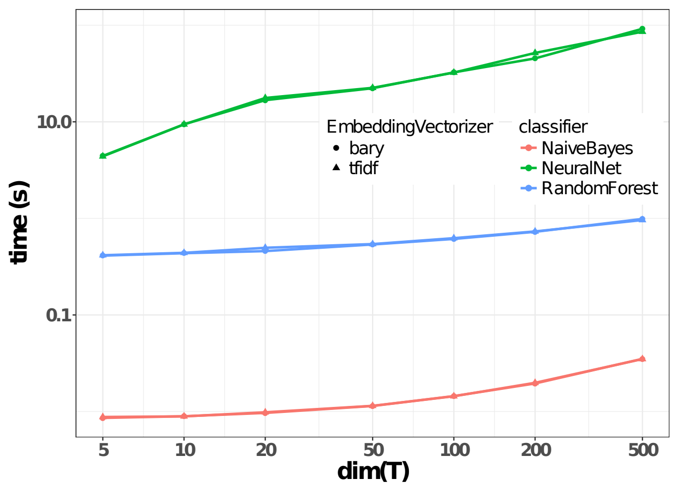

Fig. 7: Training wall time of the classifiers on 660 instances, for varying embedding space dimensions. Notice the log-log scale.

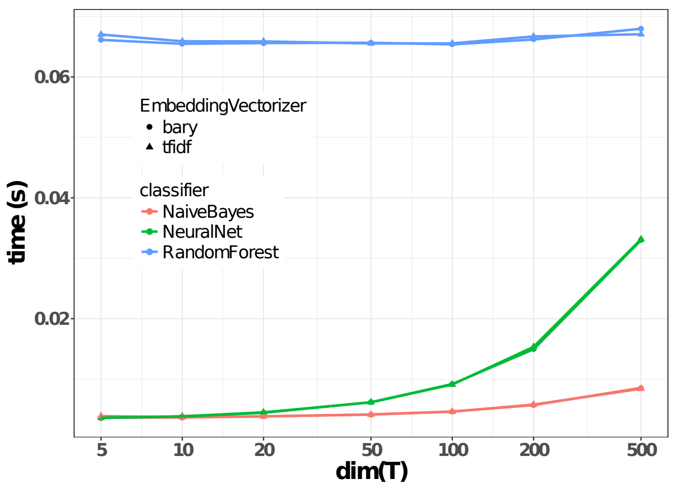

Fig. 8: Time taken for a trained model to issue one prediction. Notice the log-lin scale.

To provide some insights, we recorded wall clock times of the training of machine learning models (Figure 7) and of individual prediction of these models (Figure 8) operations. Those were performed on classical Macbook Pro with 16 GB of RAM and a quad-core Intel i7.

Interestingly, these figures provide a new perspective on our classifiers. Results confirm the reputation of each of those models: Naive Bayes is very simple, it is quickly trained and provides fast answers. Neural Network is a considerably more complex model whose training requires significantly more time. However, once trained it is able to answer reasonably fast. Contrariwise, Random Forest is quickly trained but requires considerably more time to issue predictions. Issuing a prediction requires on average $66ms$ (resp. $5ms$ and $11ms$) for Random Forest (resp. Naive Bayes and Neural Network).

Not surprisingly, increasing $dim(T)$ comes with a computational cost (as it increases the number of features on which each model is trained), but since Section IV-B1 shows that $dim(T) = 20$ is already sufficient to obtain accurate results, we conclude that this approach is computationally tractable. The most prominent decision is the choice of the classifier: although the simplest possible classifier (Naive Bayes) provides cheap and reasonable answers, more efficient classifiers like Random Forest or Neural Network will cost a bit more, either at training time, or at prediction time.

To conclude, this results section explored the performance of three state of the art classifiers exploiting the log positions. These classifiers exhibit a strong performance for a reasonable cost. The most important parameter, the dimension of the host space $dim(T)$, is not very sensitive: values ranging from 20 to 200 will roughly deliver the same performance. Although many parameters could be precisely tuned to optimize the classifiers, we believe these good results obtained using mostly default values of COTS tools already validate the soundness of our approach. More precisely, these show the extremely powerful effect of the `word2vec` embedding applied to logs: it allows to summarize each logfile to a single point in T while keeping enough information to allow an efficient classification.

V. Discussion

Our approach leaves one common question of all machine learning approaches intact: how general are the learned models? In other words, are the classifiers built in this context able to provide accurate answers in different contexts, application environments, under different injection campaigns? Although this question is definitely of interest, we argue its scope goes well beyond this paper. Philosophically, this study shows that it is easy to train efficient classifiers. But informally, a classifier is only as good as its training data. The availability of labelled training data can clearly limit the applicability of our approach. The advantage of fault injection if to gather relevant labeled datasets in a short time period. Although it enables to evaluate our approach in a straghforward manner this implemention can be cumbersome. However, while we rely on fault injection to gather datasets, other sources exist : user-based feedback, crowed sourced datasets, and crash reports of large scale deployments.

In our previous work [19] we analyzed monitoring counters such as CPU consumption or number of disk accesses for anomaly detection. Results from counter-based detection showed a good predictive performance that is yet not fully aligned with the results of this study. For instance, latency errors were significantly harder to detect. In this study, we show that by solely mining *syslog* files we could detect anomalies with high accuracy for all types of anomalies. Consequenlty, we believe our approach is largely promising. As for future work, we plan to study an hybrid approach leveraging both logging and counter-based data in order to further evaluate their potential complementarity. what type of logs enhance or weaken the efficiency of our approach.

Finally, results presented in this paper show that our approach detects with the same accuracy the stresses injected in either type of application of our case study (i.e., proxy, router and database). In other words, the analysis of system related logs such as syslog is an efficient way to summarize application behaviors for stress detection with no regard to the type of application. We believe however that syslog events are not enough to derive application dataflows that may allow to detect other types of anomalies or more importantly for administrators, to diagnose the origin of an anomaly. Consequently, we need to explore in future work other types of logs, notably the ones generated by our case study application.

VI. Related work

In this study, we use a `word2vec`-based method for log mining with a validation-purposed application of detecting stressed behaviors in computing systems. `word2vec` is a method for learning high-quality vector representations of words. It has been used for NLP in some previous works but not for the analysis of logfiles. In comparison, our previous work [19] focuses on anomaly detection based on monitoring data collected by means of a specific software agent, deployed beforehand on target machines, and providing numerical metrics on the system behavior. Here we exploit the default system-produced textual logs to predict stress. Beside the deep technical differences, our approach allows different use-cases, like post-mortem analysis of the behavior of the several processes being executed in the targeted systems.

Consequently, in the following we present separately several works related to NLP and other works related to logfiles analysis for detection purposes.

NLP applications. In the literature, most of the NLP algorithms are used for document processing [26] to isolate references of a given subject in a document and detect the sentiments of the writer, or to exploit tweets [11] to detect cyber-attacks such as distributed denial of service.

To the best of our knowledge, relatively few works exploit NLP for a different purpose than document analysis. We provide here a quick summary of these non-traditional uses of NLP. In [15], the authors use a NLP technique called Latent Semantic Indexing to identify source code documents that match a user query expressed in natural language. They use the same technique in [14] to detect similar piece of code (i.e., duplicated functions) in software systems code. In addition, Latent Dirichlet Allocations are used for a similar purpose in [20]. NLP is also applied on network packet payloads for network intrusion detection in [18]. In [10], customers accesses to businesses URLs are analyzed using a `word2vec`-based method to propose better services to customers. Finally, NLP is also used to detect design and requirement debts [13] from comments of ten open source projects.

Log mining for detection purposes. Although some works propose new methods to generate relevant log events as in [4], logfiles still gather a wide range of events and evaluating their information in the execution context or weighting their gravity is still intricate. For instance, the authors of [17] analyze a wide range of logs with engineers and compare events signaling failures to the engineers feedback on actual failures. It turns out that the number of actual failures is lower than the failures reported by logs. Also they point out that syslog message severity level is of "dubious value", and that it is essential to take into account the operational context during which log events are collected. Nevertheless, logfiles analysis for anomaly (e.g., crash, fault, OS stressing...) detection in computing systems has been widely studied and it is still an active research field, in particular when considering the ever more complex recent computing systems.

Execution traces of streaming applications are analyzed in [9] in order to detect anomalies. The authors analyze traces by means of the merging pattern mining method applied on patterns of events (i.e., lines of traces). Then they build a graph representing the dataflow between the different computing units of the application. Likewise, in [21] the authors analyze the temporality of execution traces in order to derive system states from their estimated control flows. The authors of [25] also work on the ordered nature of logfiles. They exploit time series potentially hidden behind logs events for failure symptoms detection. They use a probabilistic modeling using a mixture of Hidden Markov Models (HMM) to represent different time windows (i.e., sessions) of logs event. They propose a new method for the learning of the HMM mixture working online.

Automatic techniques based on machine learning or statistics algorithms have been widely used for this matter, as in [6] where the authors propose a new approach for disk failure prediction. More precisely, they analyze by means of a Support Vector Machine (SVM) model, sequences of syslog events based on syslog tag numbers sequences or key strings in events. In [22], the author proposes a new algorithm for the clustering of log events and implements a tool based on it named SLCT. Logfiles parsing is exploited in [24]. The parsing uses log patterns identified from a static analysis of source code. Then, two types of features are computed from the entire available logfiles, and they are fed to the PCA-based anomaly detection algorithm for an offline detection. A log extractor for anomaly detection is studied in [12]. The extractor uses log clustering based on the Levenshtein editing distance to evaluate the similarities amongst log events strings (i.e., two strings are close together if there is a minimal number of actions to change the first string into the other). Templates are then extracted from log clusters. Finally, a sequence of log events matching patterns is created and feed to a machine learning algorithm. The Naive Bayes, and Recurrent Neural Networks are evaluated.

VII. Conclusions and Future work

In this paper, we tackled the problem of anomaly detection by mining logs produced by running systems. Differently to previous studies, we develop a *linguistic* approach by considering logs as regular plain text documents. This enables to exploit recent NLP techniques to extract information from the grammatical structure and context of log events. Logfiles are represented as a set of features that can be processed by standard machine learning algorithms. As such this approach shifts the burden of log preprocessing toward the collection of representative datasets. It is a good trade when data is massively available like in recent distributed systems.

Our experimental campaigns on different components of a VNF rely on fault injection to synthetize anomalous behaviors and collect relevant datasets on demand. We more particularly focus on the case of stress detection and show that strong predictors ($\approx 90\%$ accuracy) are easily trained with no human intervention in the loop. Even though we focus on stress detection in this work, our approach is fitted for computing systems administrators for the online detection of any type of anomaly.

As for future work, we plan to explore unsupervised classifiers that would not restrain our approach scope to labelled training data and mostly known anomalies. *Syslog* files are used in this study, however we plan to inquire about what type of logfiles (e.g., `dmesg`, application logs...) enhance or weaken the efficiency of our approach. Also, we plan to extend our study to more precise online event troubleshooting while combining this detection approach with our previous work on counter-based detection [19].

REFERENCES

[1] A. P. Bradley, "The use of the area under the roc curve in the evaluation of machine learning algorithms," *Pattern recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[2] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "Nfv-vital: A framework for characterizing the performance of virtual network functions," in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, Nov 2015, pp. 93–99.

[3] M. Cinque, D. Cotroneo, R. D. Corte, and A. Pecchia, "Characterizing direct monitoring techniques in software systems," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1665–1681, Dec 2016.

[4] M. Cinque, D. Cotroneo, and A. Pecchia, "Event logs for the analysis of software failures: A rule-based approach," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 806–821, June 2013.

[5] M. Farshchi, J. G. Schneider, I. Weber, and J. Grundy, "Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, Nov 2015, pp. 24–34.

[6] R. W. Featherstun and E. W. Fulp, "Using syslog message sequences for predicting disk failures," in *Proceedings of the 24th International Conference on Large Installation System Administration*, ser. LISA'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–10.

[7] R. Gerhards, "The Syslog Protocol," RFC Editor, RFC 5424, March 2009.

[8] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, Oct 2016, pp. 207–218.

[9] O. Iegorov, V. Leroy, A. Termier, J. F. Mehaut, and M. Santana, "Data mining approach to temporal debugging of embedded streaming applications," in *2015 International Conference on Embedded Software (EMSOFT)*, Oct 2015, pp. 167–176.

[10] R. Kanagasabai, A. Veeramani, H. Shangfeng, K. Sangaralingam, and G. Manai, "Classification of massive mobile web log urls for customer profiling analytics," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 1609–1614.

[11] R. P. Khandpur, T. Ji, S. Jan, G. Wang, C.-T. Lu, and N. Ramakrishnan, "Crowdsourcing cybersecurity: Cyber attack detection using social media," *arXiv preprint arXiv:1702.07745*, 2017.

[12] C. Liu, "Data analysis of minimally-structured heterogeneous logs : An experimental study of log template extraction and anomaly detection based on recurrent neural network and naive bayes." Master's thesis, KTH, School of Computer Science and Communication (CSC), 2016.

[13] E. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[14] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, Nov 2001, pp. 107–114.

[15] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *11th Working Conference on Reverse Engineering*, Nov 2004, pp. 214–223.

[16] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[17] A. Oliner, "What supercomputers say: A study of five system logs," in *Proceedings of DSN 2007*, 2007.

[18] K. Rieck and P. Laskov, "Detecting unknown network attacks using language models," in *Proceedings of the Third International Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, ser. DIMVA'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 74–90.

[19] C. Sauvanaud, K. Lazri, M. Kaâniche, and K. Kanoun, "Anomaly detection and root cause localization in virtual network functions," in *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*, 2016, pp. 196–206.

[20] T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk, "Topicxp: Exploring topics in source code using latent dirichlet allocation," in *2010 IEEE International Conference on Software Maintenance*, Sept 2010, pp. 1–6.

[21] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Salsa: Analyzing logs as state machines," in *Proceedings of the First USENIX Conference on Analysis of System Logs*, ser. WASL'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 6–6.

[22] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proceedings of the 3rd IEEE Workshop on IP Operations Management (IPOM 2003) (IEEE Cat. No.03EX764)*, Oct 2003, pp. 119–126.

[23] Y. Watanabe, H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto, "Online failure prediction in cloud datacenters by real-time message pattern learning," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, Dec 2012, pp. 504–511.

[24] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 117–132.

[25] K. Yamanishi and Y. Maruyama, "Dynamic syslog mining for network failure monitoring," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD '05. New York, NY, USA: ACM, 2005, pp. 499–508.

[26] J. Yi, T. Nasukawa, R. Bunescu, and W. Niblack, "Sentiment analyzer: extracting sentiments about a given topic using natural language processing techniques," in *Third IEEE International Conference on Data Mining*, Nov 2003, pp. 427–434.