Expert Systems With Applications 191 (2022) 116263

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

AutoLog: Anomaly detection by deep autoencoding of system logs

Marta Catillo *, Antonio Pecchia, Umberto Villano

Università degli Studi del Sannio, Benevento, Italy

ARTICLE INFO

Keywords:
System logs
Deep learning
Autoencoder
Anomaly detection
Cybersecurity

ABSTRACT

The use of system logs for detecting and troubleshooting anomalies of production systems has been known since the early days of computers. In spite of the advances in the area, the analysis of log files emitted by real-life systems poses many peculiar challenges. Up-to-date tools, such as log management and Security Information and Event Management (SIEM) products, capitalize on standard data formats, logging protocols and dictionaries of threat signatures, which hardly fit to logs of industrial and proprietary systems.

This paper addresses the analysis of logs emitted by computer systems with a focus on anomaly detection. The proposed approach, named AutoLog, consists in sampling the logs at regular intervals and to compute numeric scores. Scores collected under normative operations are used to train a semi-supervised deep autoencoder, which serves as a baseline to classify future scores. The approach is not constrained by the structure of underlying logs and does not need for anomalies at training time. The results obtained in detecting anomalies of two industrial systems and the public BG/L and Hadoop datasets widely used as benchmarks, indicate that the recall of AutoLog ranges between 0.96 and 0.99, while the precision is within 0.93 and 0.98. A comparative study with isolation forest, one-class SVM, decision tree, vanilla autoencoder and variational autoencoder is conducted to demonstrate the validity of the proposal.

1. Introduction

System logs are sequences of time-stamped text lines that report on informational and abnormal events occurring during the execution of a given system. Almost any computer system appends lines to one or more special files – called *system logs*, *log files* or just *logs*, for short – at run-time. Since the early days of computers, to "grep" log files to search interesting keywords (e.g., *error*, *denied* or *unavailable*) is a well-established mean to monitor and assess dependability of computer systems and to gain direct insight into failures, anomalies and misuse (Cinque, Cotroneo, Della Corte, & Pecchia, 2016; Oliner, Ganapathi, & Xu, 2012). Nowadays, many log-related tasks have converged into up-to-date log management tools and Security Information and Event Management (SIEM) products (Bhatt, Manadhata, & Zomlot, 2014; Miller, Harris, Harper, VanDyke, & Blask, 2010), such as LogRhythm,[1] Splunk[2] and Logstash,[3] which allow to collect and normalize diverse data sources and logs. More importantly, these tools and products implement real-time monitoring and alerting capabilities, which are critical for practitioners and administrators to develop situational awareness from large volumes of run-time logs. In spite of the "technical" advancements in log management, substantial cognitive work by human experts and system administrators is required to traverse the logs in order to pinpoint and to correlate relevant lines for forensics and troubleshooting.

The effectiveness of log management and SIEM installations is intertwined with the completeness of the specifications of *interesting events*, i.e., events that should be detected and followed up for further inspection. Specifications may consist in *hard-coded* catalogs of keywords, regular expressions and rules, which are used to scan system logs and to alert analysts and administrators upon matches. Fig. 1 provides a concrete example of rule from OSSEC,[4] i.e., an open source log monitoring solution. The rule – coded in eXtensible Markup Language (XML) – is intended to raise an alert whenever the phrase "Client sent malformed Host header" (within the `match` tag) is seen in the logs of the Apache web server; the phrase is deemed a potential indicator of a *Code Red attack*. Current products rely on internal representation formats (e.g., MDI Fabric and Common Information Model used by LogRhythm and Splunk, respectively) and a variety of default *log adaptors* along with comprehensive catalogs of rules for many standard protocols and commodity applications that can be encountered in production

* Corresponding author.
E-mail addresses: marta.catillo@unisannio.it (M. Catillo), antonio.pecchia@unisannio.it (A. Pecchia), villano@unisannio.it (U. Villano).

[1] http://logrhythm.com/products/siem
[2] http://www.splunk.com/
[3] https://www.elastic.co/products/logstash
[4] https://ossec.github.io/docs/manual/non-technical-overview.html

https://doi.org/10.1016/j.eswa.2021.116263
Received 13 May 2021; Received in revised form 29 October 2021; Accepted 19 November 2021
Available online 10 December 2021

```
<rule id=''30107'' level=''6''>
  <if_sid>30101</if_sid>
  <match>Client sent malformed Host header</match>
  <description>Code Red attack.</description>
  <info type="link">
    http://www.cert.org/advisories/CA-2001-19.html
  </info>
  <info type=''text''>
    CERT: Advisory CA-2001-19 "Code Red" Worm Exploiting
    Buffer Overflow In IIS Indexing Service DLL
  </info>
  <group>automatic_attack,</group>
</rule>
```

Fig. 1. Example of rule from OSSEC (*apache_rules.xml* file).

environments (e.g., ftp, ssh, telnet, squid, nginx). For other types of proprietary or industrial log files, they require users to perform ad-hoc data preparations. Moreover, default rules must are typically specialized and updated by system administrators and practitioners: setting up well-crafted and effective rules entails substantial **threat knowledge** by domain experts.

This paper addresses the analysis of logs emitted by computer systems, with a focus on the detection of anomalies, such as failures and misuse. We consider the logs of four real-life systems, which range from an **industrial system** in the transportation domain to a **microservices-based installation** implementing a standard multimedia architecture adopted by large *telcos*, up to publicly-available system logs from a Blue Gene/L (BG/L) **supercomputer** and a Hadoop **cluster**. Overall, the systems consist of various nodes and applications interacting through a network, and strongly rely on logs to massively record execution events, traces and dumps of variables. Our work stems from different motivations. There exist many classes of systems where it is hard to fit the concept of pre-established rules, such as the one shown in Fig. 1. Proprietary and vendor-dependent logs, including those generated by the industrial and microservices system addressed by our study, lack standardized practices (Cinque, Cotroneo, & Pecchia, 2018). Moreover, differently from many conventional protocols and standard data sources (e.g., netflows, web servers or intrusion detectors), there is not yet a mature threat model or a default catalog of rules for monitoring the logs of transportation and telco systems. These problems bear the risk of underutilizing logs emitted by real-life systems: in fact, although logs are a goldmine of information, their analysis is still a great challenge. Given the volume of logs, manually inspection is hard and impractical. Anomaly detection techniques can be conveniently used on the top of system logs to overcome the limitations posed by the need for pre-established rules.

We propose **deep autoencoding of system logs** (AutoLog) to mitigate the issues above. AutoLog consists in sampling the logs of the system under assessment at regular intervals, and to compute numeric scores by means of a **term-weighting** technique without making any specific assumption on the format of underlying logs. Scores collected solely under *normative operations* are used to train a **semi-supervised deep autoencoder**, which serves as a *baseline model* to classify future scores into normative or anomaly classes. It should be noted that semi-supervised learning does not need to know anomalies at training time: in this respect, detection is pursued by pinpointing the scores that deviate – through the notion of **reconstruction error** of the autoencoder – from the baseline model. This leads to a more general solution to log-based anomaly detection. AutoLog is suited to distributed systems because it is inherently conceived to aggregate scores computed with the log lines gathered from different sources (e.g., log files) at a given time. It is worth noting that current distributed systems may create distinct log files per application or node. In this respect, clock synchronization across the nodes of the system is a requisite to time-stamp and to process the logs correctly. It is well-accepted that a synchronization protocol, such as the Network Time Protocol (NTP), provides accuracies generally in the range of 0.1 ms with fast local area networks (LANs) and computers and up to a few tens of milliseconds in the intercontinental Internet[5]: this delay is negligible when compared to the sampling period of AutoLog, which is in the range ten to hundreds seconds. AutoLog requires no a-priori catalogs or rules of interesting symptoms or patterns. Overall, the approach makes it possible to cope with the lack of threat models for handling proprietary logs and to complement insights from hard-coded detection rules.

AutoLog is evaluated by means of direct experiments with the logs of above-mentioned systems, where we compute and analyze numeric scores reflecting normative operations and anomalies. Experiments are based on a mixture of simulated and spontaneous anomalies. In the industrial and microservices systems we reproduce bruteforce authentication attempts, tampering with data structures and system misuse – inspired by accepted taxonomies in the area – and collect the logs; on the other hand, the BG/L log accounts for hardware and software errors observed over 215 days of operations that were neither induced nor simulated. The Hadoop dataset accounts for different types of service failures in the production environment. Results indicate that the recall of AutoLog at detecting anomalies ranges between 0.96 (industrial system) and 0.99 (microservices system); precision ranges between 0.93 (BG/L) and 0.98 (microservices system). As for BG/L, whose reference dataset is a widely-used benchmark in log analysis and anomaly detection, AutoLog achieves recall, precision and F1 score of 0.98, 0.93 and 0.95: these figures are strongly competitive with other existing methods assessed on the same BG/L dataset. The assessment of AutoLog is complemented by extensive discussion on the use of Principal Component Analysis (PCA) and clustering in our domain; moreover, we compare AutoLog with a wide set of techniques including isolation forest, one-class Support Vector Machine (SVM), decision trees, vanilla autoencoder and variational autoencoder.

The paper is organized as follows. Section 2 presents related work in the area. Section 3 describes the reference systems, our approach to compute scores from logs and available datasets. Section 4 addresses deep autoencoding for anomaly detection, design and training aspects. Section 5 reports the assessment of AutoLog. Section 6 proposes a comparative study of AutoLog with respect to other techniques. Section 7 discusses limitations and threats to validity of our study, and how they have been mitigated, while Section 8 concludes the work and provides future research perspectives.

2. Related work

2.1. Mining quantitative metrics from logs

Mining and analyzing quantitative **metrics** and **scores** inferred from system logs and other monitoring tools has been proven to be successful

[5] https://www.eecis.udel.edu/~mills/ntp.html

for addressing dependability and security problems in a variety of domains. A literature review on the analysis of logs for vulnerability and security is presented in Svacina et al. (2020).

The paper Farshchi, Schneider, Weber, and Grundy (2018) address *sporadic* Cloud operations. The technique correlates logs and Cloud metrics to detect anomalies during operations. Lines in the logs are clustered into higher-level activities through the Pearson product-moment correlation coefficient method. A regression-based technique is used to infer a correlation between lines in the log and changes in Cloud metrics. Correlations are used to formulate assertions, which aim to detect deviations of the system behavior.

Natural language processing (NLP) techniques have been used to address system logs. An anomaly detection technique leveraging NLP is proposed in Bertero, Roy, Sauvanaud, and Tredan (2017); the approach is intended for analyzing logs from a software system in face of different operating conditions. The analysis is performed through the Google *word2vec* algorithm, which allows mapping words into a high dimensional space. Based on the mapping, vectors of features are created to train a classifier and to provide information on the target operating condition of the system.

More recent contributions in this field attempt to obtain better representations than *word2vec* by capturing semantic information hidden in log templates. The LogAnomaly framework (Meng et al., 2019) leverages a novel word representation method based on synonyms and antonyms, *template2Vec*, to effectively represent the words in templates. The LogBERT framework for log anomaly detection (Guo, Yuan, & Wu, 2021) is instead based on Bidirectional Encoder Representations from Transformers (BERT). BERT is a Google-developed Transformer-based machine learning technique for NLP pre-training. By using the structure of BERT, LogBERT expects that the contextual embedding of each log entry can capture the information of whole log sequences.

An approach for mining console logs to detect run-time problems in large-scale systems is presented in Xu, Huang, Fox, Patterson, and Jordan (2008). The approach extracts structured information from console logs and constructs vectors of features.

The Authors of Campos, Vieira, and Costa (2018) present a study on the use of machine learning for failure prediction. The study uses datasets of failure and non-failure data, which consist of numeric features representing the system behavior.

A log-based abnormal task detection approach for Apache Spark is presented in Lu et al. (2017). The approach leverages a set of features extracted from the logs of Spark, e.g., execution time and data locality of each task, to detect where and when abnormalities occur.

The work Zoppi, Ceccarelli, and Bondavalli (2016) presents an anomaly detection approach for Service-Oriented Architectures (SOAs), which aims to cope with SOAs' dynamics by collecting metrics at different system layers.

The technique proposed in Oprea, Li, Yen, Chin, and Alrwais (2015) aims to detect early-stage infections targeting enterprise networks. The technique leverages network logs, such as Domain Name System (DNS) and web proxy logs, and computes scores for domains contacted by known compromised hosts. A graph-theoretic approach —namely *belief propagation*— is used to identify domains that are indicative of malware infections.

2.2. Deep autoencoding for anomaly detection

The model that we use in AutoLog to capture a normative baseline is based on **deep autoencoding**. This was first used for anomaly detection in Hawkins, He, Williams, and Baxter (2002), becoming progressively more and more popular in recent years. Since then, several studies have applied autoencoders for anomaly detection. For example, in Sakurada and Yairi (2014) autoencoders, denoising autoencoders, PCA, and kernel PCA methods are compared according to their performance. The use of deep learning models for anomaly detection is surveyed in Ruff et al. (2020) and Pang, Shen, Cao, and Hengel (2021).

With respect to security applications, autoencoders typically underlie hybrid designs of anomaly detectors. In Fahimeh and Heikkonen (2018) the Authors leverage a deep autoencoder to create an anomaly-based intrusion detection system (IDS), and evaluate its performance by means of the KDD-CUP'99 dataset. In particular, their model is based on a stacked autoencoder, and uses in the training phase a greedy unsupervised layer-wise training mechanism (Hinton, Osindero, & Teh, 2006).

The work Shone, Ngoc, Phai, and Shi (2018) proposes an intrusion detection model that is a combination of deep and shallow learning. In particular, it uses a non-symmetric deep autoencoder (NDAE) for unsupervised feature learning, and a classification model based on stacked NDAEs and the Random Forest algorithm. The performance of this solution is evaluated on the KDD-CUP'99 and NSL-KDD datasets. In Catillo, Rak, and Villano (2020) the Authors describe a semi-supervised learning technique that provides two different training phases. They test the approach on CICIDS2017 dataset by exploiting the double loop learning concept with the aim of reducing the number of false positives.

Authors in Nguyen, Lim, Divakaran, Low, and Chan (2019) propose a framework for detecting and explaining anomalies in network traffic. They leverage a variational autoencoder in order to detect anomalies. They demonstrate the validity of the proposal on the recent University of Granada (UGR) dataset (Macià-Fernández, Camacho, Magàn-Carriòn, García-Teodoro, & Theròn, 2018). The method is effective for detecting a variety of attacks.

An autoencoder-based approach is proposed by the Authors of Aygun and Yavuz (2017). In particular, they describe two deep learning-based anomaly detection models using an autoencoder and a denoising autoencoder to detect zero-day attacks.

In the context of industrial anomaly detection, it is worth mentioning the Autoencoder-based Payload Anomaly Detection (APAD) method (Kim, Jo, & Shon, 2020). Its Authors propose two payload anomaly detection approaches by leveraging an autoencoder: they are required at operative level and product process management level, respectively.

In Liu et al. (2021), instead, a semi-supervised anomaly detection method is proposed, based on the encoder–decoder–encoder paradigm. The Authors show the effectiveness of their proposal by means of an extensive experimentation conducted on their Aluminum Profile Surface Defect (APSD) dataset. Deep SAD (Ruff et al., 2020) is a deep methodology for semi-supervised anomaly detection. Its Authors propose the use, in addition to the labeled normal samples commonly exploited by semi-supervised approaches, of a small set of labeled anomalies to obtain performance improvements.

The work Su et al. (2019) proposes a stochastic recurrent neural network approach for multivariate time series anomaly detection. It captures temporal dependence between multivariate observations and applies a variational algorithm for representation learning. The approach aims to reconstruct input data by the representations and use the reconstruction probabilities to determine anomalies. Different from our work, experiments are done on numeric datasets and detection achieves an overall F1 score of 0.86.

It is worth noting that autoencoders are also used in domains other than anomaly detection. For example, paper Zhao, Hao, Tang, Chen, and Wei (2021) proposes a conditional variational autoencoder to solve a highly imbalanced classification problem, where the training data points of the minority classes are rare. In Zhang, Lu, and Wang (2021) the Authors attempt to solve the generalized feature selection problem. Also in this case they use an autoencoder to reduce the dimensions of data while maintaining a high-quality representation as well.

2.3. Our contribution

While deep autoencoding methods are increasingly used for anomaly and intrusion detection in well-structured network traffic records – as for many of the papers referenced in Section 2.2 –

we take a different perspective by addressing text lines of system logs. The analysis of system logs to detect anomalies by "traditional" methods (i.e., not based on deep learning) is presented in He, Zhu, He and Lyu (2016); the paper also includes the evaluation of six anomaly detection methods (three supervised and three unsupervised) on publicly-available datasets. Log anomaly detection based on deep learning is instead surveyed in Yadav, Kumar, and Dhavale (2020).

Recent trends in the intersection of deep learning and log analysis put forth the use of Long Short-Term Memory (LSTM) approaches. The work Yang, Qu, Gao, Qian, and Tang (2019) proposes *nLSALog*, an anomaly detection framework that leverages log files as data source. The framework models the log as a natural language sequence and uses LSTM – built on the top of nominal training data – to detect security anomalies. The Authors of Yuan et al. (2020) propose Adaptive Deep Log Anomaly Detection (ADA), which aims to detect security-related anomalies in system logs, leveraging deep neural networks with LSTM and dynamic adaptive thresholds. The approach in Du, Li, Zheng and Srikumar (2017), named *DeepLog*, uses a deep neural network to model a system log as a natural language sequence. DeepLog learns patterns from normative executions in order to detect anomalies. Similarly, the Authors of Zhang et al. (2016) parse streamed console logs to detect early warning signals for IT system failure prediction by means of log pattern extraction.

The studies mentioned above capitalize on LSTM, which *does* allow to capture dependencies over sequences of lines in a given log file, but at the cost of *complex* and *error-prone* data preparation and clustering of similar lines of the logs into common templates (or patterns). VeLog – close to our work – is an anomaly detection method based on variational autoencoders (Qian, Ying, & Wang, 2020). VeLog needs to generate the order and number of log execution matrices: a new log sequence is labeled as an anomaly if the order of log execution and number of log execution are predicted to be abnormal. As LSTM-based approaches, VeLog strongly depends on sequences of lines.

Two autoencoders along with Isolation Forest are used for unsupervised anomaly detection in logs in Farzad and Gulliver (2020). The same Authors propose in Farzad and Gulliver (2021) the extraction of features from log messages using an autoencoder, successively exploiting an LSTM, Bidirectional Long–Short Term Memory (BLSTM) or a Gated Recurrent Unit (GRU) to classify the extracted features. The paper Wadekar, Gupta, Vijan, and Kazi (2019) presents a solution utilizing a hybrid Convolutional Autoencoder–Variational Autoencoder (CAE–VAE) architecture. Keys derived from individual entries of log files are grouped in discrete event sequences, and a likelihood metric is used as an anomaly score.

Differently from the majority of papers cited above, AutoLog does not depend on the notion of sequence of lines in the logs. We address heterogeneous – and potentially distributed – logs by extracting vectors of numeric scores, which make it possible to apply a wide category of classifiers and deep learning models. While doing so, AutoLog embeds no application knowledge and makes no assumptions on the format and sequences of underlying lines in the logs. As for the computation of the metrics, differently from Farshchi et al. (2018) and Xu et al. (2008), our proposal does not require supplementary data sources or the availability of the source code of the applications in hand; with respect to Bertero et al. (2017) and Oprea et al. (2015), we do not target a specific type of operation. Accordingly, AutoLog can be applied and ported across different systems, as we did in our study.

3. Systems and log analysis approach

In the following, we describe the reference systems, our log analysis approach to compute quantitative scores from logs and available datasets of scores – from normative operations and anomalous conditions – used to conduct the experiments.

3.1. Reference systems

Our study leverages four systems, which range from a **proprietary information system** in the transportation domain by a top industry vendor,[6] to a **microservices-based installation**, up to publicly-available system logs – common benchmarks in the literature – from a Blue Gene/L (BG/L) **supercomputer** and a Hadoop **cluster**. A brief description of the systems is provided below:

- **Industrial system.** It consists of seven nodes within a local area network (LAN). The nodes host a variety of applications that handle *transportation-related* data, such as current vehicle positions and expected routes, and implement several critical functions, which include *vehicle monitoring*, *route computation*, *trajectory tracing*, *alerting* and *human–machine interfaces*; each node hosts one or more applications. Noteworthy, the system is operated with *vendor-provided* client applications, which serve as workload generator. The workload generated by the clients consists of a mixture of service requests that mimic representative usage profiles of the system in production.
- **Microservices system.** The system consists of a Clearwater[7] microservices installation, which implements the standard IP Multimedia Subsystem (IMS) architecture being adopted by large *telcos* for IP-based voice, video and messaging services. Microservices are connected through a LAN environment and each characterized by a unique IP address. An "anchor" microservice – named *bono* in Clearwater – serves as interface to external clients; other key functions include a Session Initiation Protocol (SIP) router, a database of profile data and a RESTful server that allows authentication credentials and users profiles to be retrieved. As the previous system, Clearwater is exercised with representative usage scenarios and workloads,[8] such as registering and deleting accounts, sending SIP messages and creating endpoints.
- **Supercomputing system**. We use publicly-available system log data[9] of a BG/L supercomputer from the Lawrence Livermore National Labs (LLNL) consisting of 131072 processors, whose reference dataset (Oliner & Stearley, 2007) has become a consolidated benchmark in the area of log analysis and anomaly detection. Interested readers are referred to Adiga et al. (2002) for an overview of the BG/L supercomputer.
- **Hadoop system**. We use publicly-available logs[10] generated by a Hadoop cluster with 46 cores running distributed applications backed by the Hadoop Distributed File System (HDFS). Hadoop is a Big Data framework that allows for the distributed processing of large data sets; it is widely used and studied in the literature given its relevant use by industry. Interested readers are referred to the reference website[11] for any additional information.

System logs collected in distributed deployments – such as those considered in our study – typically result from the "intertwinement" of heterogeneous lines recorded by a variety of daemons, applications and nodes over the time. Log lines reporting the events may be either centralized at a unique location, as for BG/L, or sparse across different files. For example, our study leverages 16 log files from the industrial system and 13 log files from the microservices system; similarly, the Hadoop dataset consists of several log files populated by the system during a given timeframe. Overall, the logs addressed by our study encompass a mixture of formats that consist of a small number of standard fields (e.g., *time-stamp*, *hostname*, *severity*) and —mostly— unstructured text messages.

[6] The name of the vendor is not disclosed due to confidentiality reasons.
[7] http://www.projectclearwater.org/
[8] https://github.com/metaswitch/clearwater-live-test/
[9] https://www.usenix.org/cfdr-data
[10] https://github.com/logpai/loghub/tree/master/Hadoop
[11] http://hadoop.apache.org/

Table 1
Examples of log lines from the reference systems.

Log line	System
`21 00:08:36.558 [Thread-024] Time: FunctionName: Warning: Nodatabase connection found for key '01234'`	Industrial
`[05/21/16 00:07:11.902] Message: received=0 × 20 datasize=256 dataid=4`	Industrial
`May 21 00:09:39 NODE1 ntpd [11080]: synchronized to 192.168.56.101 stratum 2`	Industrial (`syslog`)
`6-11-2018 14:18:32.644 UTC Error bono.cpp:1337: Route header flow identifier failed to correlate`	Microservices
`2005-06-03-15.42.50.363779 R02-M1-N0-C:J12-U11 RAS KERNEL INFO instruction cache parity error corrected`	BG/L
`2015-10-17 15:37:57,902 INFO [main] org.apache.hadoop.mapreduce.v2.job-history.JobHistoryUtils: Default file system [hdfs://msra-sa-41:9000]`	Hadoop

Table 1 provides examples of real log lines from the reference systems. In the industrial system we leverage both logs in vendor-dependent formats and operating system logs – typically available at `/var/log/syslog` in Linux-based systems – according to the widely-used `syslog` protocol.[12] It is worth noting that the first two lines in Table 1 – although from the same vendor – reveal different formats, such as the time-stamp, lack of *thread id* in the second line and a different syntax for representing key–value pairs. The microservices system and BG/L rely on their own log formats as well. For example, BG/L is based on an effective reliability, availability and serviceability (RAS) logging framework via a centralized DB2 database (Oliner & Stearley, 2007); lines in the log are accompanied by various flags, such as originating location (e.g., rack and node) and severity (e.g., `INFO`, `ERROR` and `FATAL`). On the other hand, Hadoop uses `log4j`[13] to handle the logs, which introduces a further log format in the context of our analysis. The inherent heterogeneity of purposes and formats of logs that can be encountered in real-life systems is a challenge to practitioners. In this study we propose a *uniform* analysis and detection approach than can be ported across different types of logs.

3.2. Overview and log analysis approach

3.2.1. Overview of AutoLog

AutoLog consists in sampling system logs with period P and then following up with two *pipelined* steps: (i) computation of numeric scores from the log lines, which can be handled more conveniently for machine learning than the original raw text shown in Table 1, and (ii) anomaly detection based on the scores. Fig. 2 shows the overall architecture of AutoLog, where the steps mentioned above are named *scoring* and *anomaly detection*, respectively. The approach relies on a database – shown in Fig. 2 – of normative *chunks*, i.e., log lines windows of duration P, obtained during normative system operations. The database is meant to be populated before using AutoLog; moreover, numeric scores computed from the normative chunks in the database are used to train the anomaly detector, i.e., *training phase* in Fig. 2. It is worth noting that log analysis is a "moving" target: software upgrades or changes to the configurations may alter meaning and character of the logs during the lifetime of a given system (Oliner & Stearley, 2007). While the implementation of a *re-training* component is not within the perspectives of the paper at this time, the architecture presented in Fig. 2 is suited to keep up with changes in the logs. When needed, the support database of AutoLog can be augmented or updated by operations engineers with new batches of normative chunks; in turn, the anomaly detector is re-trained with the scores computed from the newly-added chunks. Updating the database has no specific impact on the overall functioning of our method if not the time taken to insert the new chunks and training the detector. The description of the *scoring* component is addressed in Section 3.2.2, while the anomaly detection approach is detailed in Section 4.

3.2.2. Scoring component

Let LE_i (with $1 \leq i \leq \mathrm{N}$) denote a Logging Entity (LE), i.e., a component of the target system (such as a daemon, an application, a node or a set of nodes) that emits lines in the logs. A system may "natively" store the lines emitted by the entities into distinct log files, e.g., one file per application, microservice or container; alternatively, lines from all the entities are centralized at a unique location, and can be dissected later on by source, such as in BG/L. Regardless how files are physically organized and stored, each LE produces its own timeline of log lines, which are denoted by ↑ in Fig. 2. The **scoring component** acquires the log lines emitted by the LEs in the form of batches of fixed-length windows – named **chunks** – of period P. It is worth noting that the approach we use for handling logs is in-line with the well-consolidated *micro-batch* stream processing pattern adopted by up-to-date massive data processing frameworks, such as Spark streaming.[14] At a given time, the scoring component processes the chunks belonging to the same sampling period, i.e., *current chunks* in Fig. 2; once processed, current chunks are queued to the *past chunks* and scoring moves on to the next period.

Current chunks are fed to the scoring component represented in Fig. 2, which applies *parsing* and *term weighting* to the chunks in order to compute a vector of numeric scores s_i ($1 \leq i \leq N$), i.e., one per LE_i; it is worth noting that a new vector of scores is generated for every period P. **Parsing** is a common data preparation step in log analysis (He, Zhu, He, Li & Lyu, 2016) and allows removing the variable tokens of the log lines while preserving the constant parts. At this stage, we also remove special characters and punctuation, such as #, ?, and %. On the other hand, **term weighting** has been successfully used in the past to operate on text logs (Stearley & Oliner, 2008). Given a chunk after parsing, term weighting is done by (i) tokening the log lines of the chunk into terms,[15] (ii) counting the occurrences of the terms within the chunk, (iii) computing a numeric score for the chunk based on the occurrences of the terms. Tokenization is applied to all the lines, including those pertaining to negations in the log files.

Let x_t denote the number of occurrences of the **term** t in the chunk of a given LE_i (with $1 \leq t \leq T$, where T is the total number of terms). The score of the chunk is given by:

$$s = \sqrt{\sum_{t=1}^{T} (e_t \cdot \log_2(1 + x_t))^2} \qquad (0 \leq e_t \leq 1) \tag{1}$$

where e_t is the *entropy* of the term t. The **entropy** is computed from both x_t and by counting the occurrences of t in the set of $(M - 1)$ normative chunks of LE_i stored in the support database presented in Section 3.2.1. Fig. 3 exemplifies the role of the database at supporting the computation of the entropy. For each LE_i, the database contains (*term*, *count*) pairs computed from $(M - 1)$ chunks collected under normative operations; noteworthy, (*term*, *count*) pairs are arranged by originating chunk, each represented by a dotted box and labeled with

[12] https://tools.ietf.org/html/rfc5424
[13] https://logging.apache.org/log4j/
[14] https://spark.apache.org/streaming/
[15] A term is a sequence of characters separated by one (more) whitespace(s).

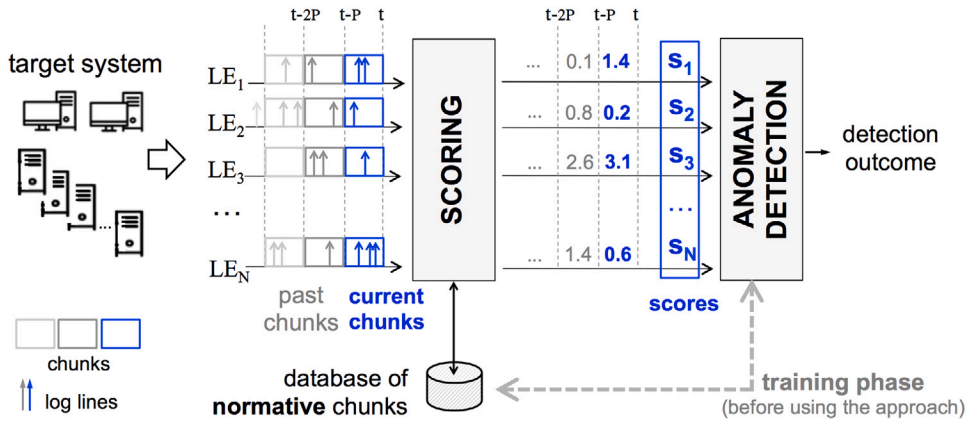


Fig. 2. Overview of AutoLog.

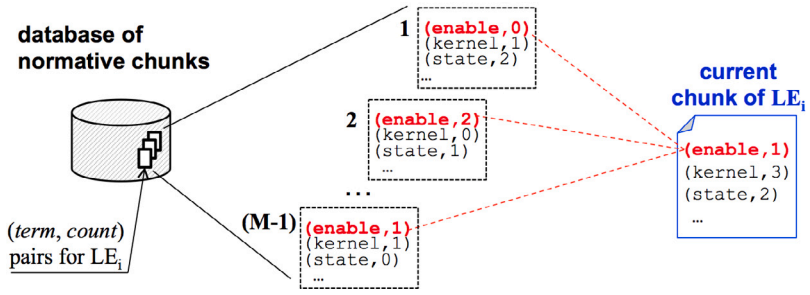


Fig. 3. Role of the support database and term-count representation of the chunks used for computing the entropy.

1, 2, …, $(M-1)$ in Fig. 3. As previously mentioned, the database is populated before the use of the approach. Once the scoring component is fed with a new chunk —"current" chunk of LE_i in Fig. 3— it generates the (*term*, *count*) pairs and retrieves the counts of each term from the database, such as for `enable` in Fig. 3. That said, the value of the **entropy** e_t is given by:

$$e_t = 1 + \frac{1}{\log_2(M)} \sum_{j=1}^{M} p_{t,j} \log_2(p_{t,j}) \qquad p_{t,j} = \frac{x_{t,j}}{\sum_{j=1}^{M} x_{t,j}} \tag{2}$$

where M is the total number of chunks, i.e., $(M-1)$ chunks from the database plus the chunk targeted by the scoring component (represented in the rightmost part of Fig. 3), $x_{t,j}$ is the number of occurrences of the term t in the chunk j, and $p_{t,j}$ is the fraction of term t's total occurrences that are in the chunk j (with $1 \leq j \leq M$). We assume that the chunk targeted by the scoring is the Mth chunk: accordingly, $x_{t,M} = x_t$ in Eq. (2). Beside the arrangement into vectors for anomaly detection, each score produced by the scoring component is tagged with (i) a unique numeric identifier, which tracks the vector the score belongs, (ii) the originating logging entity, and (iii) time-stamps of first and last log lines in the chunk used to compute the score. Although not intended for anomaly detection, administrators can leverage this accessory data to traverse system logs upon the occurrence of an anomaly.

The database of normative chunks serves as a baseline of the system behavior. In this respect, the score obtained by applying Eqs. (1) and (2) quantifies the extent an arbitrary chunk resembles (or not) the baseline of terms that are emitted by the system under regular operations. The *higher* the score, the *larger* the difference between the terms of the chunk and "typical" lines emitted by LE_i, which may be seen as an indication of an anomaly. For example, Table 2 shows some real chunks from BG/L and the corresponding scores computed by our technique after parsing and term weighting. The first chunk is assigned 0.23, i.e., a relatively low value, because it reports on detected and corrected errors, which are purely informative and can be noted in the logs of BG/L almost at any time. On the other hand, the latest two chunks point to real anomalies: they achieve a high score because the log lines consist of terms that are infrequent across the logs. It is worth noting that the chunk shown by the bottom row of Table 2 results from the interleaving of different anomalous lines, whose "aggregated" effect is a score higher than the individual lines themselves.

The weighting approach used here is also known as **logarithmic entropy** because – according to Eq. (1) – the dominance of the terms is mitigated by $\log_2$ and then scaled by the their entropy. The application of information entropy as a measure for the uncertainty in a data set and detection purposes is well consolidated (Holzinger et al., 2014). Moreover, there is an extensive body of literature on term weighting and related applications. Interested readers are referred to introductory references, such as Quan, Wenyin, and Qiu (2011) and Salton and Buckley (1988), for additional information on the topic.

3.3. Available datasets

Vectors of scores are computed from the logs reflecting normative operations and anomalies of the systems presented in Section 3.1. In this respect, experiments are based on both simulated and spontaneous anomalies.

3.3.1. System logs

Industrial and microservices systems. System logs from normative **(NORM)** operations are obtained by exercising the systems with the client applications presented in Section 3.1. In addition, we collect the logs from five independent scenarios, each reproducing a different anomaly. The occurrence of the anomaly is intertwined with the normative operations; after its beginning, each anomaly stays on up to 30 min. We adopt a uniform model of anomalies in the both the industrial and the microservices system; anomalies are spiked in as follows:

Table 2
Examples of chunks and corresponding scores from BG/L .

Log lines in the chunk	Score

`1 ddr errors(s) detected and corrected on rank 0, symbol 18, bit 6`
`CE sym 18, at 0 × 0deb5a60, mask 0 × 02`
`total of 1 ddr error(s) detected and corrected`
	0.23

`ddr: Unable to steer rank=0, symbol=5 - rank is already steering`
`symbol 4. Due to multiple symbols being over the correctable error threshold,`
`consider replacing the card1`
	3.68

`machine check interrupt`
`instruction address: 0 × 001544bc`
`... omitted ...`
`rts panic! - stopping execution`
	6.76

- Authentication **(AUTH)**: bruteforce attempts to gain unauthorized access to a system. For both the systems in hand, authentication-related anomalies are elicited by attempting to guess the credentials of a legitimate user. This is achieved by means of remote logins via the frontend applications exposed by the systems to external clients (e.g., the human–machine interface of the industrial system).
- Log deletion **(DEL)**: deletion of the content of a log. It should be noted that this is a typical action performed by an attacker to cover his/her traces.
- Hang **(HANG)**: the system becomes unresponsive and no service/output is provided within an acceptable timeframe. For example, in the case of the industrial system, the anomaly is injected by stopping the *database service*. As a result, the remaining applications of the system stay up; however, they are not able to successfully fulfill the client requests.
- Modification **(MOD)**: tampering with the data structures handled by the system through abnormal alterations of their fields. For the industrial system, modification is implemented by changing the expected routes of the vehicles; similarly, in the microservices system we alter the telephone account records.
- Denial of Service **(DOS)**: abnormal usage of the system functions over the nominal capacity with the aim of disrupting service. In both systems, this is achieved by increasing the number of client applications, and thus frequency and volume of the service requests.

Although reproduced, anomalies are inspired by the well-consolidated attack phases reported in Ruiu (1999); moreover, they cover a mixture of diverse scenarios that range from bruteforce authentication attempts to tampering with OS resources and misuse of system functions.

Supercomputing system. It is standard practice to log messages in a supercomputing system, such as BG/L. Log lines in the BG/L log account for hardware and software errors at all levels. Different from the systems presented above, the BG/L log contains spontaneous – neither induced or simulated – anomalies observed over 215 days of operations at LLNL (Oliner & Stearley, 2007).

Hadoop system. Hadoop logs used in this study are collected during different executions of data processing applications distributed across a cluster of machines. Logs pertain to both normative executions of the applications and different types of service failures in the production environment, such as *machine down*, *network disconnection* and *disk full*. The dataset is publicly-available as a benchmark for log-based anomaly detection; details can be found in the proposing paper Lin, Zhang, Lou, Zhang, and Chen (2016).

3.3.2. Sampling and labeling

System logs are sampled into chunks in order to compute the scores for anomaly detection. We set a period $P = 10$ s for the industrial, microservices and Hadoop system, which is a balanced trade-off between the latency of the detection and the need for ensuring a suitable number of lines per chunk after each sampling round. As for BG/L, we conduct a sensitivity analysis beforehand. This stems from the long-standing observation that anomalies tend to cause multiple and redundant log lines, which should be grouped together. A common technique to address this issue is the **tuple heuristic** (Hansen & Siewiorek, 1992), which groups the lines whose time distance is lower than a threshold into the same "tuple". The threshold is determined by testing different time values and counting the resulting number of tuples (i.e., the tuple count). Fig. 4 shows how the tuple count of the anomalous log lines of BG/L varies with respect to the threshold. Experimental studies demonstrate that a good choice for the threshold is the value right after the "knee" of the curve – dotted line in Fig. 4 – where the tuple count flattens sharply (Hansen & Siewiorek, 1992). Based on the result, we assume $P = 300$ s for BG/L.

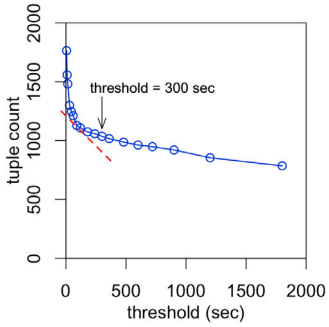


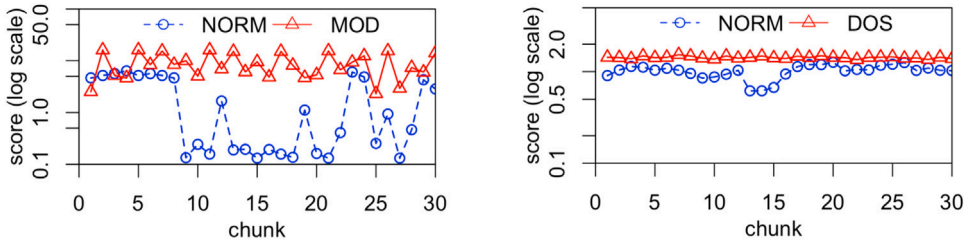
Fig. 4. Tuple count of anomalous log lines in BG/L.

Table 3 shows the number of unique unigrams, bigrams and trigrams (thus providing a glimpse of each log corpus), logging entities, sampling period and chunks by system. For the industrial, microservices and Hadoop system, logging entities relate to distinct log files; as for BG/L, entities consist of the set of nodes allotted to the same rack. As a remark, AutoLog produces *one score per chunk*: in consequence, – given a dataset – the number of scores is equal to the number of chunks shown by the rightmost column of Table 3. The number of chunks is determined by the number of logging entities, sampling period and duration of the logs in hand.

Fig. 5 shows the scores of two logging entities, i.e., LE_3 for the industrial system (Fig. 5(a)) and LE_6 for the microservices system (Fig. 5(b)), under normative operations and the occurrence of an anomaly over 30 subsequent chunks (i.e., five minutes). It can be noted that the anomalies, i.e., *MOD* and *DOS* (▵-*marked* series), make the score to considerably deviate from the normative ones (∘-*marked* series). It is worth noting that normative scores are computed from the logs elicited through representative client applications and benchmarks, which generate mixtures of interleaving requests. In this respect, the normative scores reflect the variability of the workload at the time logs were collected. Most notably, the y-axis of Fig. 5 is in log scale: while it allows appreciating fluctuations around low values of the scores, their variability is over-emphasized.

Table 3
Number of unique *n*-grams, logging entities (LE), sampling period and chunks by system.

System	Corpus statistics			LE	Period	Number of chunks
	unigrams	bigrams	trigrams			
Industrial	898029	2382190	7235966	16	10 s	22640
Microservices	2370889	5129706	8689044	13	10 s	12116
BG/L	5632912	12207650	21771832	67	300 s	3473548
Hadoop	152068	328295	541476	28	10 s	114884



(a) Industrial system (LE_3). (b) Microservices system (LE_6).

Fig. 5. Scores of two logging entities within normative and anomalous conditions.

Scores are arranged into vectors according to the approach in Section 3.2.2. Furthermore, we accompany each vector by a **label**, which denotes whether the vector relates to normative or anomalous conditions. For the industrial and microservices systems the label is either *NORM* (normal) or any of *AUTH*, *DEL*, *HANG*, *MOD*, *DOS*. As for BG/L and Hadoop, it must be noted that the public logs used in our study are labeled. In fact, logs were tagged with a label established by domain experts in consultation with the system administrators: we rely on this information to label the vectors produced by our approach, i.e., *NORM* or *ANOM* (anomaly). Whilst AutoLog *does not* need the labels at training time, labels are intended to be used for evaluating the effectiveness of both AutoLog and other techniques assessed in the comparative study.

4. Anomaly detection method

4.1. Background

AutoLog hinges on the use of a **deep autoencoder**. An autoncoder (AE) is a feedforward neural network where the output layer has the same dimension of the input layer. In fact, the purpose of an AE is to "reconstruct" the input at the output layer. It is possible to design different types of autoencoders (Goodfellow, Bengio, & Courville, 2016). In particular, deep learning can be applied to autoencoders: multiple hidden layers are used to provide depth. The resulting network is known as *deep* or *stacked autoencoder* (Vincent, Larochelle, Lajoie, Bengio, & Manzagol, 2010).

An autoencoder learns the input representation in a different feature space (Goodfellow et al., 2016). The learning task forces the autoencoder to catch the most relevant features of the training data at the **bottleneck** layer, i.e., the middle hidden layer, so that the input can be reconstructed at the output layer. Therefore, the AE consists of two parts: *encoder* and *decoder*. The encoder learns an efficient representation of the given input by using its hidden layer(s); the decoder, instead, produces the reconstruction of the input by using the encoded information in its hidden layer(s). Therefore, the decoder mirrors the encoders in the number of hidden layers and neurons.

An **encoder** is a deterministic map f_θ that transforms an input vector s into its representation y, where $\theta = \{W, b\}$, W is the weight matrix and b is the bias vector. On the other hand, a **decoder** is the g_θ function that maps backwards the representation y to the output z, i.e., the reconstruction of the input vector s. In Eq. (3) the variables W and b represent the weight and bias values for the encoding phase. Similarly, in Eq. (4) the variables W' and b' are the weight and the bias for the decoding phase.

$$f_\theta(Ws + b) = y \tag{3}$$

$$g_\theta(W'y + b') = z \tag{4}$$

The **reconstruction error (RE)** measures the difference between the reconstructed, i.e., z, and the original version of the input, i.e., s. It is important to feed data to an autoencoder and tune it until it is well trained to reconstruct the input with minimum error. Therefore, the measure of faithful reproduction of input data is defined by the RE, which is computed as follows:

$$RE = \frac{1}{N}\sum_{i=1}^{N}(z_i - s_i)^2 \tag{5}$$

where z_i and s_i (with $1 \leq i \leq N$) denote the components of the output and input vector, and N is the number of components. Overall, the key characteristics of autoencoders are:

- *Data dependency*. Autoencoders will only be able to reconstruct data similar to that on which they have been trained. This principle is the basis of our anomaly detection approach (more on this later).
- *Output lossy*. This means that the reconstructed output will be degraded compared to the original input.
- *Self-contained learning*. Autoencoders are trained automatically from data examples. It is not necessary to do the extra work of preparing extra labels or data.

An autoencoder can be used for dimensionality reduction, in a way similar to Principal Component Analysis (PCA) (Almotiri, Elleithy, & Elleithy, 2017). It is worth noting that PCA uses linear algebra to make the transformation of the coordinates; in contrast, an autoencoder performs non-linear transformations by means of non-linear activation functions and multiple layers. Hence the use of an autoencoder is particularly attractive when the data points are complex and non-linear in nature (Song, Liu, Huang, Wang, & Tan, 2013). This is the case of the datasets in hand, as it will be shown in Section 5.1.

4.2. Use of the deep autoencoding in AutoLog

Fig. 6 shows a representation of the AE in the context of AutoLog and it reproduces input and output of the *anomaly detection* block in

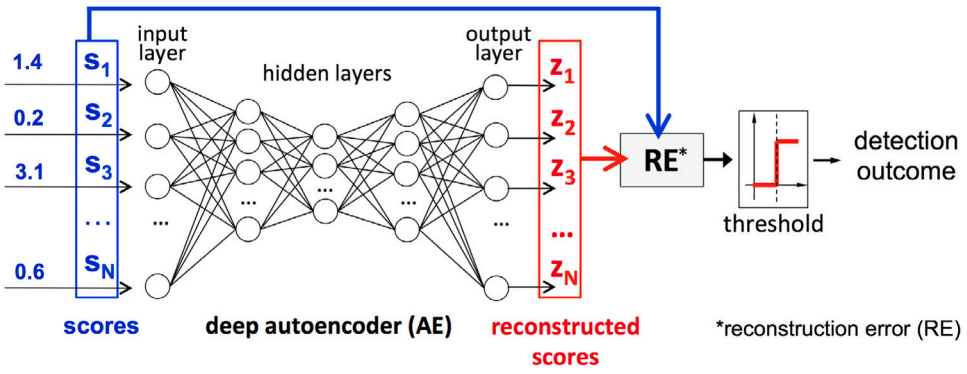


Fig. 6. Anomaly detection in AutoLog.

Fig. 2 (i.e., *scores* and *detection outcome*, respectively). More importantly, Fig. 6 details the internal organization of the anomaly detector. Scores – generated by the scoring component – are fed at the input layer of the AE: scores pass through a number of hidden layers until the output layer returns the reconstructed scores. Noteworthy, input and output layers have the same dimension N. In the context of our study, the value of N is selected equal to the number of logging entities, i.e., components emitting lines in logs as described above. The number of logging entities and their selection for each system is presented in Section 3.3.2.

The rationale underlying the use of the AE in AutoLog is that RE can be used as an *anomaly indicator* as follows. If the AE is trained using only vectors of normative scores – hence the notion of **semi-supervised learning** – it will provide (i) *low* RE (good reconstructed representation) for future normative input vectors, and (ii) *high* RE (bad reconstructed representation) for future anomalous input vectors. In this respect, the AE serves as a normative baseline: when the AE attempts to reconstruct a data point that is "outside" the norm – and thus a potential anomaly – it will experience an increase of the RE because it was never trained to reproduce anomalies.

As for many anomaly detection techniques, we apply a binary *threshold* function to RE in order to pinpoint anomalies: the input vectors that produce RE values under the threshold are deemed *normative* and those with REs above the threshold, *anomalous*. The threshold is called here **anomaly threshold**: as the specific configuration of the AE for the data in hand (e.g., in terms of layers, neurons and activation functions), the setup of a suitable threshold is an outcome of the training phase and it is addressed in Section 4.4. RE computation and comparison with the threshold represent the steps of AutoLog that produce the final detection outcome, as shown in Fig. 6.

4.3. Dataset partitioning

As discussed in Section 3.3, we conduct our experiments with datasets of scores arranged into labeled vectors; labels indicate whether a given vector is collected from normative operations or under an anomaly. As for the labels, we carry out a symbolic-numeric conversion by substituting 0 to the labels corresponding to a normative vector and 1 to the labels corresponding to anomalous vectors. Each dataset is split into three **disjoint subsets** through random sampling. It is worth noting that the sampling procedure is *without replacement*: once selected from a set, the vector is not placed back to the set it comes from. As said, a vector consists of chunk-wise scores computed from the log lines emitted by the logging entities during a sampling round. Fig. 7 shows how a given dataset is partitioned. Let NV be the cardinality of the normative vectors and AV the cardinality of the anomaly vectors. According to Fig. 7, the first cut consists in (i) separating normative from anomalous vectors, and (ii) splitting normative vectors into two disjoint subsets of cardinalities $(0.8 \cdot NV)$ and $(0.2 \cdot NV)$, respectively. At the bottom of Fig. 7, we find:

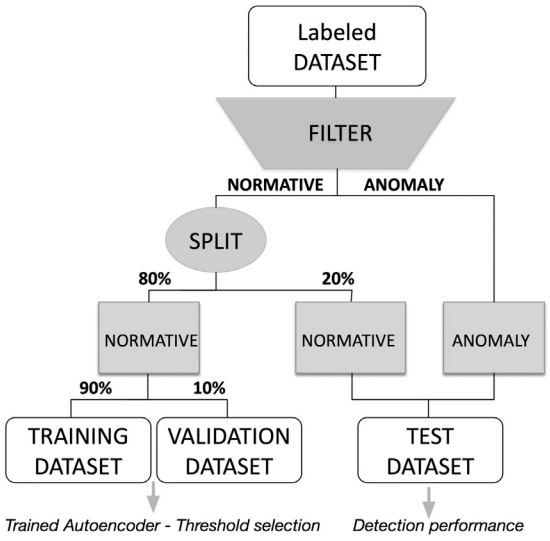


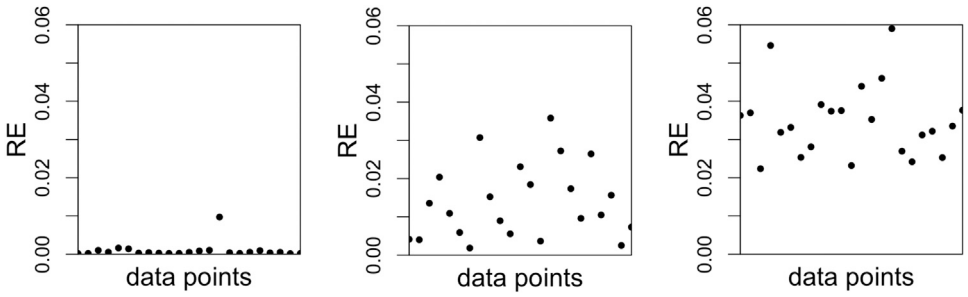
Fig. 7. Dataset partitioning.

- *Training set.* It contains only normative vectors: the cardinality of the training set is $0.9 \cdot (0.8 \cdot NV)$ of randomly selected normative vectors from the originating dataset. The set is meant for training the AE. Labels are removed.
- *Validation set.* Again, it contains only normative vectors and is used for determining the detection threshold. In particular, the cardinality of the validation set is $0.1 \cdot (0.8 \cdot NV)$. As with the training set, labels are removed.
- *Test set.* It contains both normative and anomalous vectors. The cardinality of the test set is $(0.2 \cdot NV) + AV$, i.e., 20% normative vectors – obtained at the first cut – plus all the anomalous vectors. According to the sampling procedure and the partitioning in Fig. 7, normative vectors of the test set are **held out** from training. Vectors in the test set are accompanied by the corresponding labels in order to evaluate the correctness of the predictions.

After partitioning, we obtain three non-intersecting subsets. Table 4 provides the size of each dataset for the systems in hand, and the corresponding breakdown into training, validation and test set. Given a system, the number of vectors is equal to the number of chunks divided by the number of logging entities of that system. For example, the total chunks of the industrial system (i.e., 22640) returns $\frac{22640}{16} = 1415$ total vectors, where 16 is the number of logging entities shown in Table 3. A similar computation applies to all chunks/vectors pairs in Table 4.

Table 4
Training, validation and test set size.

System	Total chunks/vectors	breakdown		
		Training	Validation	Test
Industrial	22640/1415	6928/433	768/48	14944/934
Microservices	12116/932	6552/504	728/56	4836/372
BG/L	3473548/51844	2450525/36575	272288/4064	750735/11205
Hadoop	114884/4103	70756/2527	7868/281	36260/1295



(a) Configuration 1. (b) Configuration 2. (c) Configuration 3.

Fig. 8. Analysis of RE on the validation set by configuration; Configuration 1 is the final selection for AutoLog.

Table 5
Layering structure of different AE configurations (all the layers are dense).

Configuration 1		Configuration 2		Configuration 3	
Layer	Activation	Layer	Activation	Layer	Activation
Input	–	Input	–	Input	–
Hidden 1	ReLU	Hidden 1	tanh	Hidden 1	tanh
Hidden 2	tanh	Hidden 2	sigmoid	Hidden 2	tanh
Hidden 3	ReLU	Hidden 3	sigmoid	Hidden 3	tanh
Output	ReLU	Output	tanh	Hidden 4	ReLU
				Hidden 5	sigmoid
				Output	ReLU

4.4. AE design, training and threshold selection

The design of a deep neural network, such as the AE, is based on establishing many hyperparameters that are subject to fine-tuning. As for any machine learning study, the choice of the hyperparameters is guided by experimental tests carried out by analyzing the outcome of the model – RE in our study – with respect to the **validation set**.

There are two *desirable* properties of the RE, which make it possible to usefully deploy an AE for anomaly detection in our context (i) RE ≈ 0 and (ii) small dispersion. An AE design that meets these properties is summarized by the Configuration 1 in Table 5, which is the selected configuration for the datasets in hand: its RE on the validation set of the industrial system is shown in Fig. 8(a), where it can be noted that it is $\approx$0 for all – if not one – data points. In order to provide concrete examples of less effective AE designs for the industrial system, Figs. 8(b) and 8(c) show the REs achieved on the validation set by two different configurations that fail to obtain RE $\approx$ 0 (Configuration 2 and 3 in Table 5).

As summarized by Configuration 1 in Table 5, the chosen AE is made up of five layers. These layers include N-128-64-128-N neurons, where N is the number of logging entities. Dropout layers are placed after the first hidden layer and after the second hidden layer, in order to prevent overfitting. The Rectified Linear Unit (ReLu) has been selected for the encode layer, the decode layer and the output layer, while for the bottleneck layer, i.e., Hidden 2, has been used the Hyperbolic Tangent (Tanh) activation function. Moreover, to achieve the sparsity, activity regularizer terms L1 are applied on each layer. We train the AE on training data points for 100 epochs using the RMSProp optimizer with learning rate value $lr = 0.001$. We also shuffle the training data before each epoch. It is worth pointing out that the training phase takes around 1 min in the worst case (BG/L dataset). This time has been obtained on a laptop computer without GPU acceleration, and so it can be greatly reduced using more powerful or ad hoc hardware. In particular, the experiments are conducted on a MacBook Pro with an Intel Core i5 2.6 GHz processor and 8 GB of RAM.

We select the threshold value by using normative points, and hence the training and validation procedure is semi-supervised. As no label is required, this is by far the simplest approach for threshold selection. In particular, we adopt a *percentile* method and choose as **anomaly threshold** the 90th percentile of the RE values obtained by the AE on the validation set. Section 5.3 validates the proposed threshold selection method by analyzing receiver operating characteristic curves. Selecting the 90th percentile appears suitable for all the datasets we have considered so far. In practice, it aims to mitigate the impact of sporadic RE outliers caused by accidental anomalies in the normative logs. We are aware that selecting the 90th percentile is not a *one-fits-all* approach. For example, in those systems where anomalies are sparse and with a high false positive rate, the detection approach might further benefit from the adoption of a more sophisticated threshold selection, such as the one recently proposed in Carrington et al. (2021). Nevertheless, it should be noted that our choice is inline with many other studies in the area that rely on easy-to-explain thresholds, such as Aygun and Yavuz (2017), which adopts the mean value. Although worthy to be investigated, digging into fine-grained threshold-related aspects is beyond the perspective of the study at this stage, where we focus on the overall methodology.

We implemented our architecture with Keras[16] (Version 2.4.3) and TensorFlow[17] (Version 2.4.1). Keras is a Python library that runs on top of TensorFlow; it provides highly modularized APIs for building and training deep learning models. The core code of AutoLog encompassing the implementation of the autoencoder has been made publicly-available through GitHub.[18]

5. Experimental results

AutoLog is applied to the reference systems – industrial, microservices, BG/L and Hadoop – in order to quantify its effectiveness to

[16] https://keras.io/
[17] https://www.tensorflow.org/
[18] https://github.com/ScalingLab/AutoLog

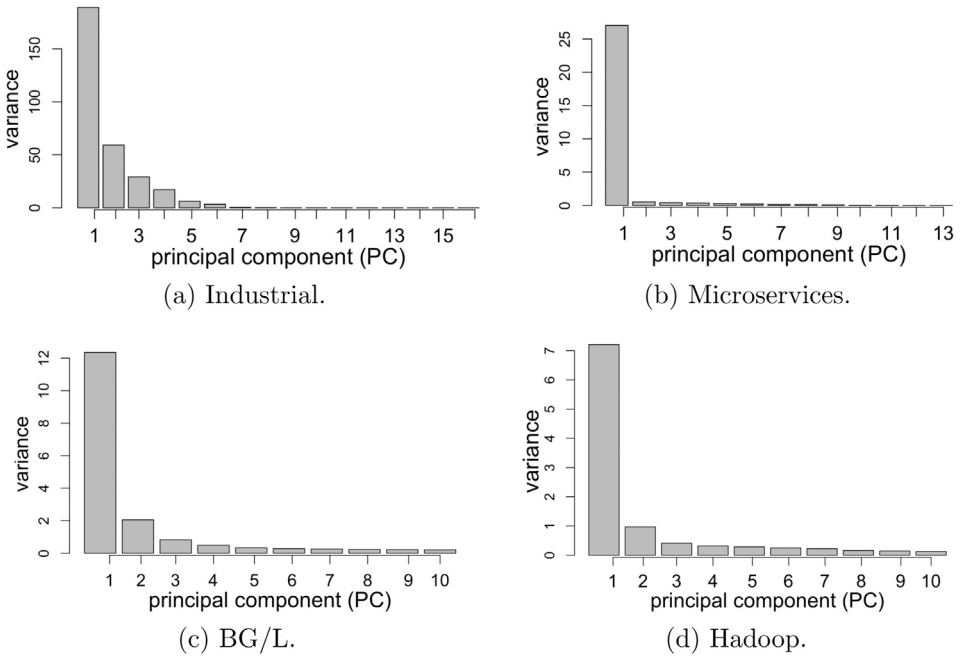


Fig. 9. Screeplots of the variance of the principal components.

discriminate normative operations from the occurrence of anomalies. In the following, we present some reflections on the challenges of PCA and clustering and discuss the results of AutoLog. A comparison study of AutoLog with a wide set of techniques is provided is Section 6.

5.1. Reflections on the use of clustering

We conduct an **exploratory data analysis** to check whether the scores obtained in face of normative conditions and anomalies tend to cluster into different groups. To this aim, the vectors of the datasets can be conveniently regarded as "points" of a Euclidean space. Nevertheless, the *dimensionality* of the datasets, i.e., 16, 13, 67 and 28 – not including the label – for each system, respectively (with each dimension corresponding to the number of logging entities, as in Table 3) prevents from obtaining human-readable visualizations for exploratory purposes. In consequence, a Principal Component Analysis (PCA) is done beforehand.

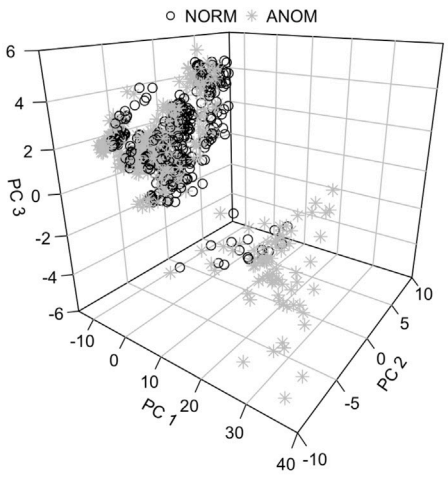
PCA is a **dimensionality reduction** technique whose objective is to find the *directions* along which a set of high-dimensional points line up "best". Points are transformed in a new coordinate space where each axis catches progressively less variance of the original data. This concept is summarized by means of a *screeplot*, which is one of the byproducts of a PCA. Fig. 9 shows the screeplots obtained for the datasets in hand. The x-axis refers to the principal components (PC) – each denoted by an integer – and the y-axis is the variance explained by the PC; it should be noted that PCs are sorted by descending variance. For the industrial system (Fig. 9(a)) the variance of the top-3 PCs accounts for 189.2, 59.1 and 29.2, respectively, out of the total variance of 304.9 (i.e., the sum across all PCs): as a consequence, the top-3 PCs catch the 91.0% of the total variance. Similarly, the total variance of the PCs for the microservices dataset is 29.1; the top-3 PCs account for 27.0, 0.52 and 0.39 variance, which sum up to 95.9% of the total. As for BG/L and Hadoop in Figs. 9(c) and 9(d), we show the top-10 components due to the large number of original dimensions: the top-3 PCs encompass 68% and 80% of the total variance for BG/L and Hadoop, respectively. Based on these results, we *narrow* the visualization to a more suitable 3D Euclidean space, where each point consists of the coordinates along the top-3 PCs.

Fig. 10(a) shows the 3D scatterplot of the industrial system dataset, where the axes are PC 1, PC 2 and PC 3, respectively. Moreover, $\circ$-*marked* points denote normative points and $*$-*marked* points depict anomalies, i.e., any of the types presented in Section 3.3. It can be noted that the points tend to group around two areas. Whilst at the bottom part of the grid we find almost only anomalous points, the top left group is strongly cluttered: in fact, normative and anomalous points are intertwined. Fig. 10(b) provides a similar perspective where anomalies are broken down by class: again, $\circ$-*marked* points denote normative points while the remaining mark types are meant to represent anomalies. It can be noted that misuse-related anomalies, i.e., *MOD* ($\diamond$ mark) and *DOS* ($\triangledown$ mark), take a clear stand from normative points; however, the real challenge for detection is the *intertwinement* of 4 different classes in the top left group, which undermines the practical usability of PCA and clustering with respect to the data in hand. This finding holds for the microservices dataset as well. Fig. 11 shows the 3D scatterplot of the top-3 PCs for the microservices dataset, i.e., *binary* view (Fig. 11(a)) and *multiclass* view (Fig. 11(b)). Again, normative and anomalous points are strongly intertwined.

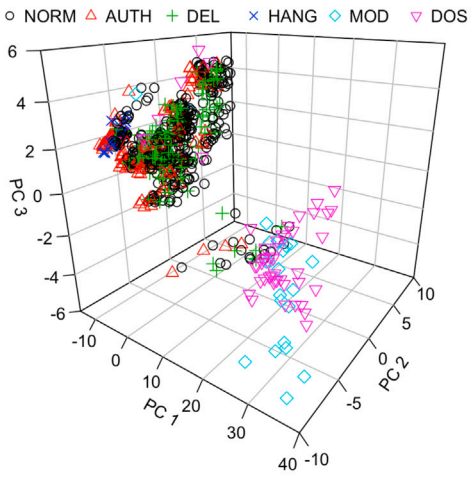
The scarce *linear separability* of the data points is a key challenge to the use of PCA and clustering as reference technique for unsupervised learning. While the industrial and microservices systems were available at a private premise, BG/L and Hadoop are common benchmarks being used by other papers in the area of log analysis. Fig. 12 indicates that the data points of BG/L and Hadoop suffer from the same intertwinement of normative and anomaly classes. For example, it is worth noting that the paper Meng et al. (2019) attempts to apply PCA to BG/L with no success for anomaly detection. Our deep autoencoding approach capitalizes on non-linear transformations of the data, which are much more suited to the datasets in hand and yield to better results.

5.2. Evaluation of AutoLog

We run the **test set** of the datasets in hand with AutoLog trained as described in Section 4.4. Since the datasets are labeled, each RE produced by AutoLog is accompanied by the label, which we use for evaluation; as said, the AE saw *no anomalies* during training. Figs. 13, 14, 15, and 16 show the REs for the industrial system, the microservices system, BG/L, and Hadoop, respectively. In particular, each data point

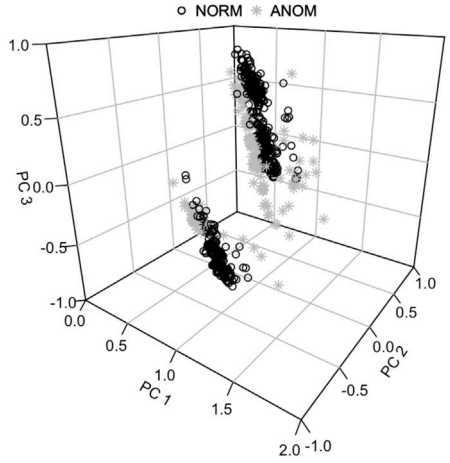


(a) Normative *vs* anomaly data points irrespective from the classes.

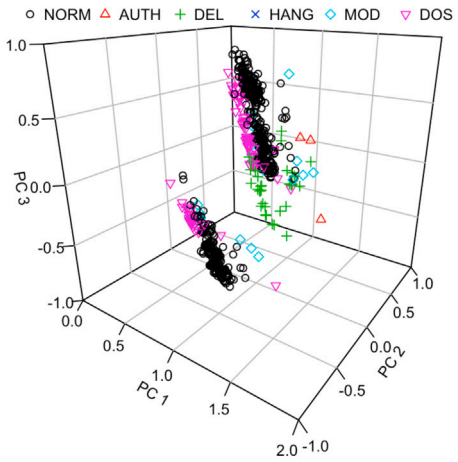


(b) Normative *vs* anomaly data points broken down by classes.

Fig. 10. Scatterplot of the dataset with respect to the top-3 principal components (industrial system).

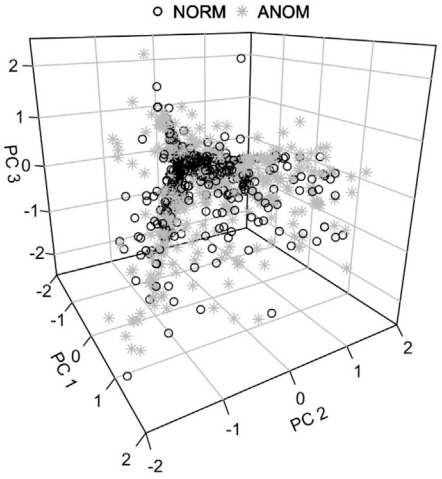


(a) Normative *vs* anomaly data points irrespective from the classes.

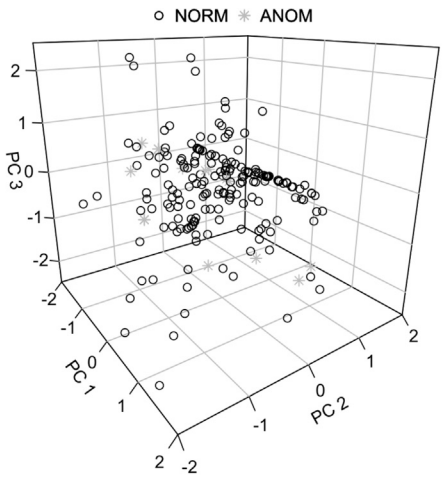


(b) Normative *vs* anomaly data points broken down by classes.

Fig. 11. Scatterplot of the dataset with respect to the top-3 principal components (microservices system).



(a) BG/L.



(b) Hadoop.

Fig. 12. Scatterplot of BG/L and Hadoop datasets with respect to the top-3 principal components.

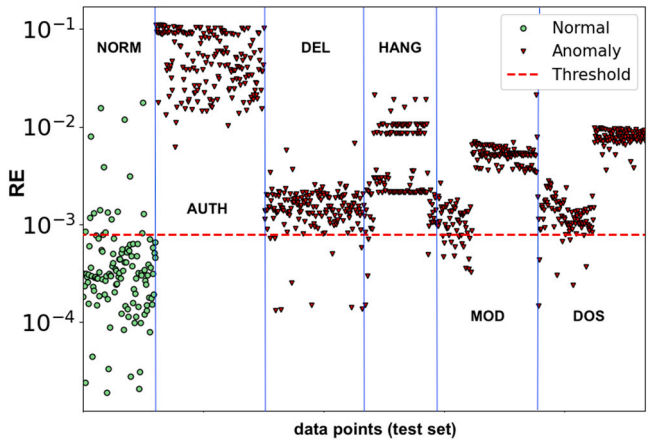


Fig. 13. AutoLog: RE of the test set for the industrial system.

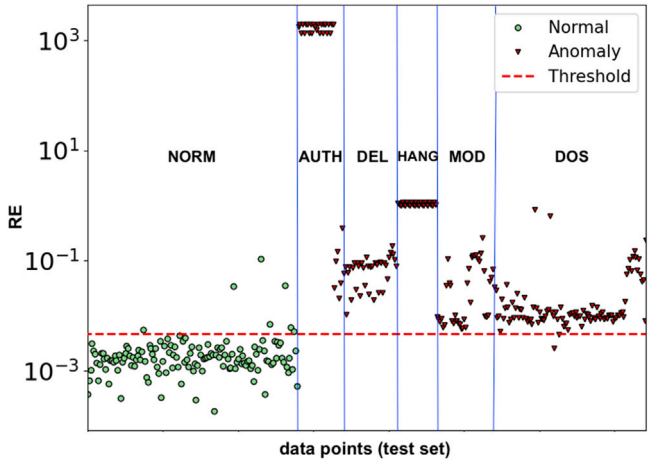


Fig. 14. AutoLog: RE of the test set for the microservices system.

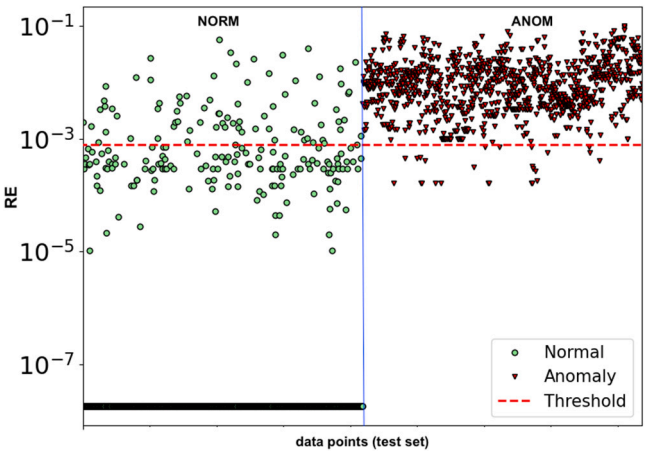


Fig. 15. AutoLog: RE of the test set for BG/L.

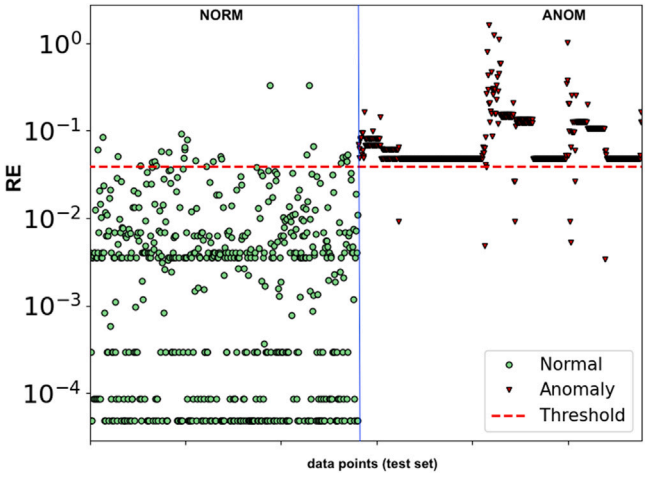


Fig. 16. AutoLog: RE of the test set for Hadoop.

Table 6
AutoLog: evaluation metrics.

System	Recall	Precision	F1 score
Industrial	0.96	0.98	0.97
Microservices	0.99	0.97	0.98
BG/L	0.98	0.93	0.95
Hadoop	0.98	0.96	0.97

is marked by either $\circ$ or $\triangledown$ for better visualization of normative and anomalous points; moreover, we superimpose vertical continue lines to delimit the data points by class, which are named according to the descriptions in Section 3.3.2. A semi-logarithmic scale (x-axis in linear scale and y-axis in log scale) is used to better visualize the RE values. The y-axis is the RE; the x-axis is the id of the points in the test set. The horizontal dashed line in Figs. 13–16 indicates the anomaly threshold, which is 0.00079, 0.00460, 0.00080 and 0.03937 for each system, respectively. Section 5.3 investigates the validity of our threshold selection approach.

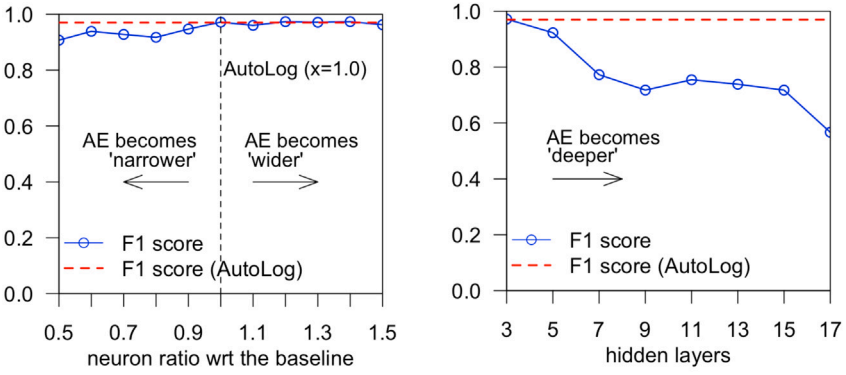
We compute the metrics of *recall* (R), *precision* (P), and *F1 score* to evaluate AutoLog. Metrics are computed as:

$$R = \frac{TP}{TP + FN} \qquad P = \frac{TP}{TP + FP} \qquad F1\ score = 2 \cdot \frac{P \cdot R}{P + R} \tag{6}$$

where True Positive (TP) and True Negative (TN) represent the points that are correctly predicted, while False Positives (FP) and False Negatives (FN) indicate misclassifications. For example, TP is the set of anomalies whose RE is higher that the threshold; similarly, TN is the set of normative points whose RE is lower that the threshold. Table 6 provides the evaluation metrics of Autolog for all the systems.

Results are notable, especially if we consider that AutoLog is based on a semi-supervised approach, which means that it does not need for anomalies at training time. According to Figs. 13–16, we observe that most of the anomalies have high RE and are well above the thresholds for all the systems. This outcome allows making some relevant considerations. For example, unlike the results obtained by means of PCA, *AUTH*, *DEL* and *HANG* anomalies of the industrial system are now easily identifiable: their RE is much higher than the typical RE of normative data points. These anomalies were strongly intertwined with normative points in the scatterplots of the dataset shown in Fig. 10. A similar consideration can be done for the microservices system. According to Fig. 11, data points grouped in two clusters – both composed by normative and anomalous points – in the Euclidean space: differently, the points appear well-separable with AutoLog, as in Fig. 14. Furthermore, Figs. 15 and 16 show that anomalies are above the threshold also for BG/L and Hadoop.

The discussion in Section 4.4 (where we presented the selection of the hyperparameters of the autoencoder and how they affect the RE) is supplemented here by means of the assessment of anomaly detection performance. As for Configuration 2 and 3 reported in Table 5 — strongly different from AutoLog in terms of both activation functions (Configuration 2), and number of layers and activation functions (Configuration 3)— they achieve R = 0.89, P = 0.97, F1 score = 0.93 and R



(a) Fixed number of hidden layers and decreasing/increasing number of neurons.

(b) Increasing number of hidden layers.

Fig. 17. Sensitivity of the F1 score with respect to (wrt) the baseline AutoLog autoencoder (AE).

= 0.20, P = 0.85 and F1 score = 0.32, respectively. These detection figures are unsatisfactory when compared to AutoLog reported in Table 6: both the configurations fail to render RE ≈ 0 on the validation set – one of the key design principles of AutoLog – as shown in Section 4.4.

We perform an additional analysis by assessing the sensitivity of the F1 score while *narrowing*, *widening* and *deepening* the AutoLog autoencoder. Results are summarized in Fig. 17 for the industrial system dataset. As for Fig. 17(a), we test different autoencoders with 3 hidden layers. The x-axis is the **neuron ratio** with respect to (wrt) the selected number of neurons for AutoLog at the hidden layers, i.e., (128, 64, 128): given a value of x in Fig. 17(a), the number of neurons of the autoencoder under test is given by (128, 64, 128) $\cdot$ x. For example, x = 0.5 returns (64, 32, 64) neurons, i.e., a "narrower" autoencoder compared to AutoLog; similarly, at x = 1.5 we obtain (192, 96, 192), i.e., a "wider" autoencoder. It is worth noting that x = 1.0 corresponds to AutoLog itself, whose F1 score for the industrial system is represented by a dashed horizontal line. Fig. 17(a) indicates that the F1 score achieved by the autoencoders flattens at x = 1.0, i.e., the number of neurons we selected for AutoLog; *widening* further the autoencoder does not improve over AutoLog. On the other hand, Fig. 17(b) shows how the F1 score varies when we increase the number of hidden layers. As indicated by the x-axis, analysis is done by step 2 because we inject two hidden layers – one at the encoder and one at the decoder, according to Section 4 – for each test; moreover, the number of neurons for the new layers is set between 64 and 128. Noteworthy, the leftmost data point, i.e., 3 hidden layers, corresponds to AutoLog. Fig. 17(b) indicates that, with respect to the problem addressed and data in hand, deepening the autoencoder does not necessarily improve anomaly detection.

5.3. Validation of the anomaly threshold

Choosing the optimal anomaly threshold is a critical task, especially if performed automatically (Liu et al., 2015). As explained in Section 4, our threshold selection is based only on the REs of normative data points in the validation set; notwithstanding, the threshold is able to discriminate fairly well normative from anomalous points. In order to validate our threshold selection method, we compare it with the "best-fit" threshold selection that can be done on the test sets when considering both the RE and the *ground truth*, i.e., the knowledge of the class of the points to be classified. The availability of the ground truth is the typical assumption of supervised techniques.

We analyze the Receiver Operating Characteristic (**ROC**) curve for our model. It can be considered as a diagnostic plot, which evaluates the effectiveness of the classification at various threshold settings. The ROC curve shows how much the model is capable of distinguishing between classes at different operating points. The area under the ROC curve (**AUC**) provides a numeric score to summarize the performance of a model with a value between 0.5 (no-skill) and 1.0 (perfect skill). The higher the AUC value, the better the model at predicting the classes. The ROC curve is plotted with **True Positive Rate** (y-axis) against the **False Positive Rate** (x-axis). A diagonal line on the plot from the bottom-left to top-right indicates the "curve" for a *no-skill* model, and a point in the top left area of the plot indicates a model with *perfect skill*. The performance of each classifier is represented by a point of the ROC curve and, as the threshold changes, the location of the point changes. The point that allows the best performance is in the top left area of the plot. From the analysis of the ROC curves corresponding to our model for the datasets in hand (Fig. 18) it is possible to note a number of points – and therefore potential thresholds – in the top-left area of the plots.

Our goal is to locate the threshold with the optimal balance between false positive and true positive rates. There are several techniques that allow to achieve this goal (Du, Vong, Pun, Wong & Ip, 2017). In particular, for our analysis we have selected the **Geometric Mean or G-Mean** score. It is a metric that tries to find a balance between the **Sensitivity** and the **Specificity**, and computed as follows:

$$G\text{-}Mean = sqrt(Sensitivity \cdot Specificity) \tag{7}$$

where:

$$Sensitivity = TruePositiveRate \tag{8}$$

$$Specificity = (1 - FalsePositiveRate) \tag{9}$$

The approach we have pursued is to assess all the thresholds obtained from the ROC analysis and select the one with the highest G-Mean score. The ROC curves in Figs. 18(a)–18(d) highlight the optimal points (•-*marked* points in the top left area of the plots). For a direct evaluation we report in Table 7 the thresholds produced by the proposed percentile selection method and the best-fit thresholds obtained with the ROC curve method for all the datasets. It is worth noting that the threshold values obtained with the two methods are very close. The two approaches lead to very similar results, and so that our choice – semi-supervised and based on a much simpler procedure – is perfectly reasonable.

6. Comparative study

In this section we compare AutoLog with the following techniques: **isolation forest**, **one-class SVM**, **decision tree**, **vanilla autoencoder** and **variational autoencoder**. These techniques are widely used in

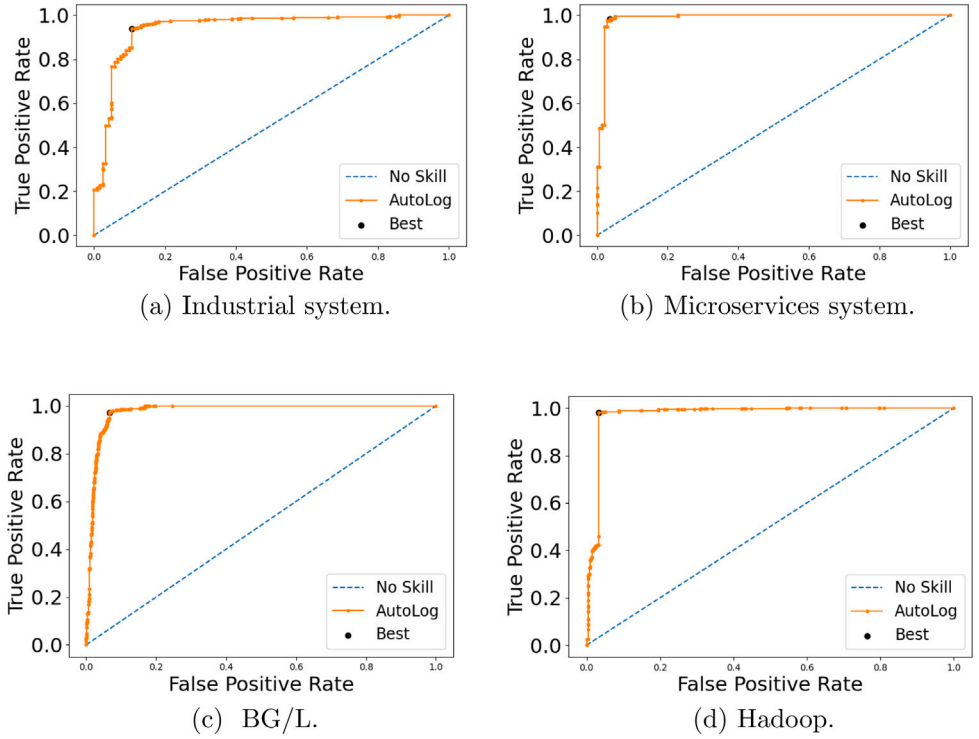


Fig. 18. AutoLog: ROC curves.

Table 7
Anomaly threshold values.

System	Percentile method (AutoLog)	ROC curve method (supervised scenario)
Industrial	0.00079	0.00086
Microservices	0.00460	0.00564
BG/L	0.00080	0.00100
Hadoop	0.03937	0.04821

Table 8
Isolation forest: evaluation metrics.

System	Recall	Precision	F1 score
Industrial	0.53	0.87	0.66
Microservices	0.85	0.87	0.86
BG/L	0.64	0.78	0.70
Hadoop	0.53	0.71	0.61

different research fields for detecting anomalies from data and are applied to the datasets in hand. We implement the isolation forest and one-class classifier by means of `scikit-learn`.[19] The decision tree is implemented with WEKA[20]; both the vanilla and variational autoencoder are based on Keras. Given a dataset, for all the techniques, if not the decision tree, we use the same training and test conditions of AutoLog: (i) 80% normative vectors – random selection without replacement from the originating dataset – for training and, (ii) remaining 20% normative vectors *plus* all the anomaly vectors for testing and computing the evaluation metrics. As for the decision tree, we use both normative and anomalous vectors for training and testing, as described in Section 6.3. In the following sections we briefly introduce each technique and then present recall, precision and F1 score achieved for each system.

6.1. Isolation forest

Isolation forest (or *iForest*) is an anomaly detection technique based on the assumption that anomalous data points are rare and far from the centers of normal clusters (Liu, Ting, & Zhou, 2008). Many anomaly detection techniques rely on the construction of *normal profiles* by defining how "normal" points look like: in consequence, anomalies are those points that do not conform to the defined "normal" profile. Different from this general approach, an isolation forest aims to directly target anomalies.

The isolation forest technique generates **partitions** on a given dataset by randomly selecting a feature from the given set and a split value between the minimum and maximum values of the selected feature. Anomalous points are likely to require fewer partitions to be isolated compared to the so defined "normal" data points in the dataset. Partitioning is represented by a **tree structure**: anomalies will be the points with a shorter path length within the tree, i.e., number of partitions required to isolate the points. As with other anomaly detection methods, also in the case of isolation forest is defined an *anomaly score*:

$$s(x,n) = 2^{-\frac{E(h(x))}{c(n)}} \tag{10}$$

let x the point and n the number of external nodes (with no child), $h(x)$ is the path length of the point x and $c(n)$ is the average path length of unsuccessful search in the tree. Each point is assigned an anomaly score value.

For our analysis we select the number of *base estimators* – number of trees in the forest – equal to 100. The value is selected after a series of tuning experiments. Table 8 shows the evaluation metrics for the datasets in hand. The best results of the isolation forest are obtained with the microservices system, although none of the metrics is higher than 0.90 in any system.

[19] https://scikit-learn.org/stable/
[20] https://www.cs.waikato.ac.nz/ml/weka/

Table 9
One-class SVM: evaluation metrics.

System	Recall	Precision	F1 score
Industrial	0.59	0.88	0.71
Microservices	0.84	0.87	0.85
BG/L	0.92	0.93	0.92
Hadoop	0.68	0.78	0.73

6.2. One-class SVM

Support Vector Machine (SVM) is a supervised technique frequently used in classification problems (Pisner & Schnyer, 2020). In general, it leverages **hyperplanes** in multi-dimensional space in order to separate classes of points. The distance between the nearest points is known as the *margin*. The core idea of SVM is to find a Maximum Marginal Hyperplane (MMH) that best separates the points into classes. Given labeled training data, the technique produces an optimal hyperplane that allows to classify new points.

Although SVM is typically used to solve binary classification problems, it can also be leveraged for **one-class problems**, where the training data belong to one class. In this context, the model is trained to learn what is "normal" so that it can identify whether new data points belong to the "normal" class or not. The technique captures the density of the majority class and classifies data points on the extremes of the density function as outliers. This alternative SVM is referred to as **one-class SVM** (Schölkopf, Platt, Shawe-Taylor, Smola, & Williamson, 2001).

In order to apply the one-class SVM technique to the datasets in hand, we leverage `scikit-learn`. The main difference with a standard SVM is that a one-class SVM is fitted in an unsupervised manner and does not provide the typical hyperparameters for tuning the margin. In particular, it provides a hyperparameter *nu* that controls the sensitivity of the technique and should be tuned to the approximate ratio of outliers in the data. For our experiments we set $nu = 0.05$, which we found to be a suitable value for the reference datasets after tuning. Once fitted, the model is used to identify anomalies by classifying the points of the test set as inliers and outliers. Table 9 shows the results obtained for the datasets, where it can be noted that one-class SVM achieves the best results with BG/L (i.e., recall and precision equal to 0.92 and 0.93, respectively).

6.3. Decision tree

Decision trees are typically used for the capability to infer explicable rules from data. Given an input data point to be classified, decision is made by traversing a tree-like structure starting from the root node. Each node is characterized by a *predicate* meant to be tested on the input data point: based on the outcomes of the tests, decision moves down through the tree until a *leaf* is reached. In a classification problem leaves are associated to the classes the input points are expected to belong. It should be noted that the decision tree is a *supervised* classification technique. At training time it requires both normative and anomalous data points, and the availability of the labels in order to infer the decision predicates. This is a key limitation in applying decision trees to many real-life scenarios because training data will likely not cover all the potential anomalies that may occur in production. Analysis is done with the *J48* tree implementation available with WEKA. For each dataset, we train and test the decision tree by means of a *K fold cross validation* approach. In consequence, the dataset is split into $K = 10$ folds f_i ($1 \leq i \leq 10$), beforehand; for each fold f_i we train the decision tree with 9 folds f_j ($j \neq i$) and test it with the "held-out" fold f_i.

Hyperparameters selection. The outcome of a decision tree may depend on the specific hyperparameters selected. Among the others, *minNumObj* is the hyperparameter that regulates the number of leaves that will be created when learning the tree: the lower the value, the higher the number of leaves and classification paths, thus causing overfitting. We perform a **sensitivity analysis** of the decision tree by varying *minNumObj*.

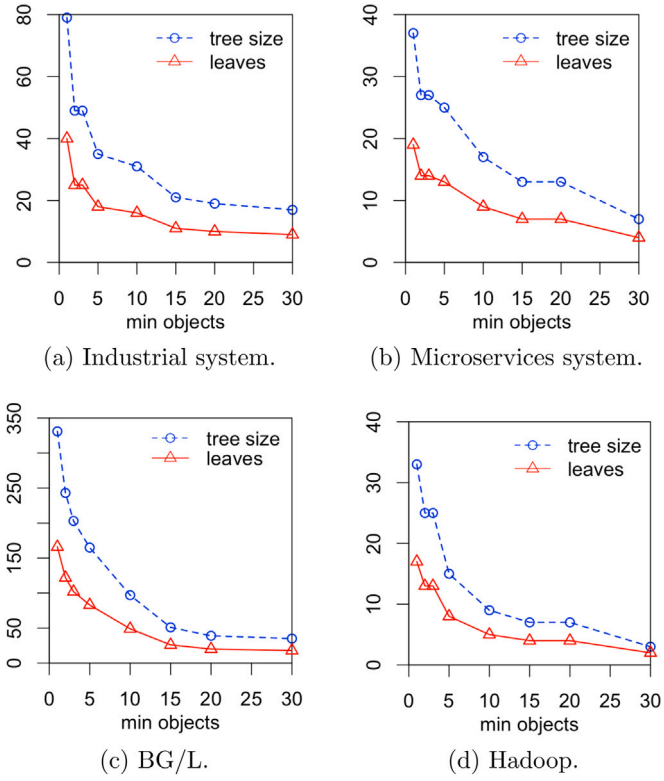


Fig. 19. Sensitivity analysis of tree size and number of leaves of the decision tree with respect to the *minNumObj* parameter.

Fig. 19 shows the sensitivity of tree-related figures – namely *tree size* and *number of leaves* – with respect to *minNumObj* for the systems. The parameter *minNumObj* is varied between 1 and 30; for each value of *minNumObj* we learn and test the decision tree by means of the *K fold* approach as explained above. We note that both *tree size* and *number of leaves* decrease as *minNumObj* increases. For example, in the industrial system dataset (Fig. 19(a)) the number of leaves is 40 when $minNumObj = 1$ and then it starts flattening at $minNumObj = 5$; similarly, the number of leaves drops from 19 to 9 when *minNumObj* increases from 1 to 10 in the microservices system (Fig. 19(b)). This issue is much more exacerbated in the BG/L dataset, which accounts for a larger number of attributes than the industrial and the microservices system. As shown in Fig. 19(c), at $minNumObj = 1$ leaves and tree size are equal to 166 and 331; these figures drop sharply until they stabilize at 18 and 35, respectively, at $minNumObj = 30$. As for Hadoop in Fig. 19(d), both tree size and number of leaves flatten at $minNumObj = 10$. A large number of leaves indicates very *specialized* decision paths that tend to overfit the data and – in turn – to bias the evaluation metrics.

Fig. 20 shows the evaluation metrics of the decision tree, i.e., F1 score, recall and precision, with respect to *minNumObj*. It can be noted that the metrics are strongly sensitive to changes of *minNumObj* for all the systems in hand. For example, in the industrial system the recall drops from 0.960 to 0.945 when *minNumObj* increases from 5 to 10 as shown in Fig. 20(a); similar considerations can be done for the microservices system and BG/L. Based on Fig. 19, it can be reasonably stated that a "stable" tree model – where *tree size* and *number of leaves* stop changing sharply – is given by $minNumObj=10$. Whilst much of the work in the area overlooks this aspect, a stable tree provides a *non-overfitted* reflection of the data. Values of recall, precision and F1 score

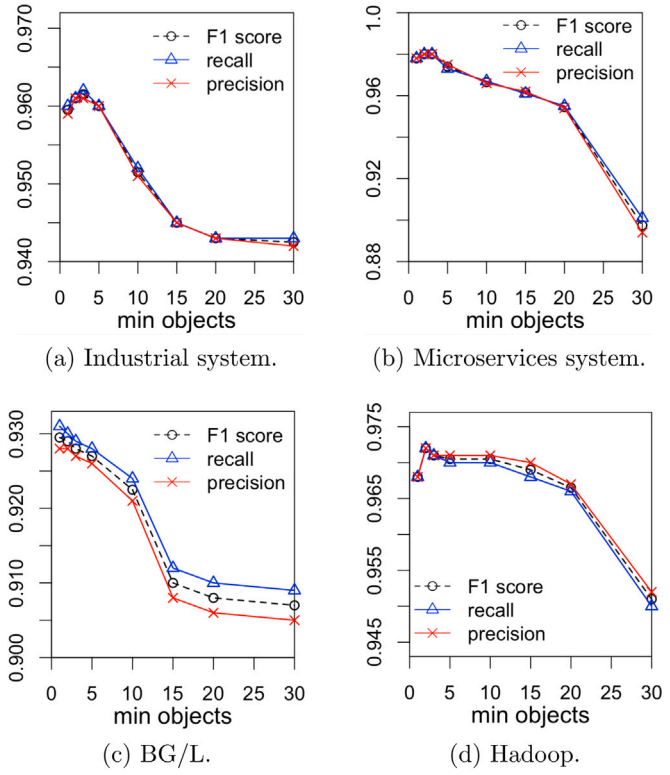


Fig. 20. Sensitivity analysis of F1 score, recall, precision of the decision tree with respect to the *minNumObj* parameter.

Table 10
Decision tree: evaluation metrics.

System	Recall	Precision	F1 score
Industrial	0.95	0.95	0.95
Microservices	0.96	0.96	0.96
BG/L	0.92	0.92	0.92
Hadoop	0.97	0.97	0.97

Table 11
Vanilla autoencoder: evaluation metrics.

System	Recall	Precision	F1 score
Industrial	0.48	0.97	0.64
Microservices	0.65	0.98	0.78
BG/L	0.81	0.91	0.86
Hadoop	0.98	0.95	0.97

obtained for all the systems with the decision tree at *minNumObj*=10 are shown in Table 10. In all the cases, if not BG/L, recall, precision and F1 score are higher than 0.95; as for BG/L, the value of the metrics is 0.92.

6.4. Vanilla AE

A *vanilla autoencoder* (AE) is an autoencoder that consists of only one hidden layer between the input and the output layer (Skansi, 2018). Typically, the input layer has the same dimension as the output layer since the autoencoder attempts to reconstruct the input data point. Vanilla AEs have been used to address anomaly detection tasks. In our experiment we consider a vanilla AE with a hidden layer of 64 neurons; therefore, it includes N-64-N neurons, where N is the number of logging entities for a given dataset. All other training hyperparameters are the same as AutoLog because they return the best result also for the vanilla AE. Table 11 shows the results obtained for the datasets.

Table 12
Variational autoencoder: evaluation metrics.

System	Recall	Precision	F1 score
Industrial	0.71	0.97	0.82
Microservices	0.76	0.94	0.84
BG/L	0.99	0.93	0.96
Hadoop	0.84	0.89	0.87

We observe that – applied to our datasets – the vanilla AE tends to achieve a high precision (e.g., 0.98 for the microservices system). On the other hand, recall is much lower; one notable exception is Hadoop where the recall is 0.98.

6.5. Variational AE

A *variational autoencoder* (VAE) is similar to AEs as structure, but tries to obtain a better – continuous and not scattered – latent space by adopting a probabilistic distribution of each attribute in latent space (Kingma & Welling, 2019). The encoder stage of a VAE generates two vectors: a vector of means (μ) and a vector of variances (Σ); a vector z is then generated using the distribution $N(\mu_i, \Sigma_i)$, which will identify the points of the latent space. The addition of the *Kullback–Leibler* (KL) divergence (van Erven & Harremos, 2014) to the cost function makes it possible to obtain a distribution as close as possible to a reference one, typically a Gaussian. The samples obtained from this distribution are fed to a conventional decoder network.

For our VAE analysis, we select a latent layer of 2 neurons, while the encoder and the decoder layers have 256 neurons after tuning on the validation sets; as for the remaining training hyperparameters, they are the same of AutoLog because they returned the best results also for the VAE after tuning. Table 12 shows the results obtained for the reference systems. The best results are obtained with BG/L, where the VAE returns 0.99 recall and 0.93 precision. Overall, the precision of the VAE tends to be high across the systems – with the only exception of Hadoop – while recall is generally low.

6.6. Comparison of the results

Barplots in Figs. 21–24 show the evaluation metrics for all the techniques and systems addressed by our study. For each system, bars are grouped by evaluation metric in order to compare the different techniques; for each group of bars, the rightmost *cross-patterned* bar refers to the metrics of AutoLog presented in Table 6. Results reveal a mixture of findings, depending on the system and the technique. Isolation forest, one-class SVM and vanilla AE tend to achieve the lowest recall for all the systems; exceptions include (i) BG/L, where the recall of one-class SVM is 0.92 – thus equal to the decision tree – and (ii) Hadoop, where the vanilla AE performs as well as AutoLog. As said, the precision of vanilla AE is notably high for all the systems; however, its recall ranges between 0.48 (industrial system) and 0.98 (Hadoop), which makes it among the worst performing in terms of F1 score for all the systems if not Hadoop. As for the variational AE, it performs reasonably well only in the case of BG/L. AutoLog achieves among the highest values of the metrics for all the systems with few sporadic exceptions, as for example (i) the microservices systems, where the precision of AutoLog (0.97) is similar to the vanilla AE (0.98), or (ii) BG/L, where the F1 score of AutoLog (0.95) is similar to the variational AE (0.96). However, it should be noted that in spite of these sporadic exceptions, none of the techniques is able to fit all the systems when compared to AutoLog. Most notably, AutoLog is strongly competitive when compared to the decision tree, i.e., a typical supervised technique, that – different from AutoLog – requires both normative and anomalous data points for training.

It is worth noting that, while the industrial and microservices systems were available at a private premise, BG/L and Hadoop are

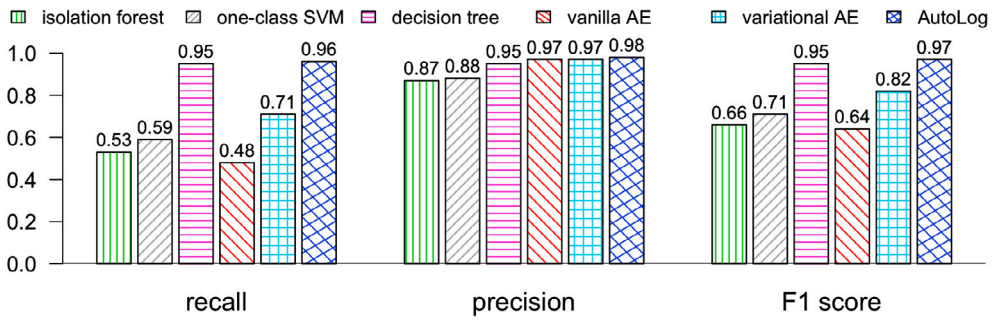


Fig. 21. Industrial system.

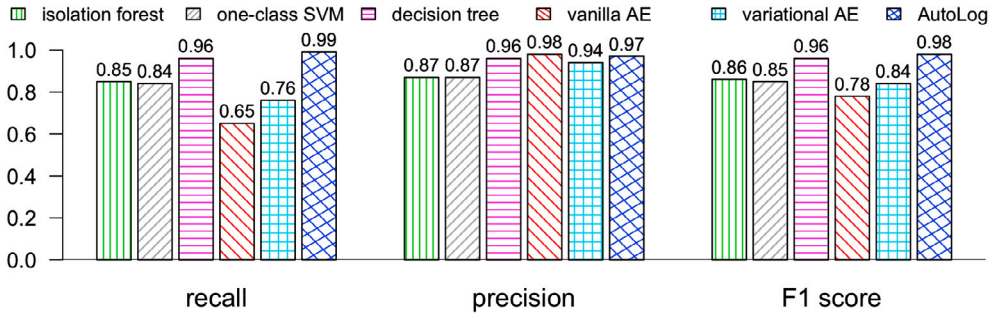


Fig. 22. Microservices system.

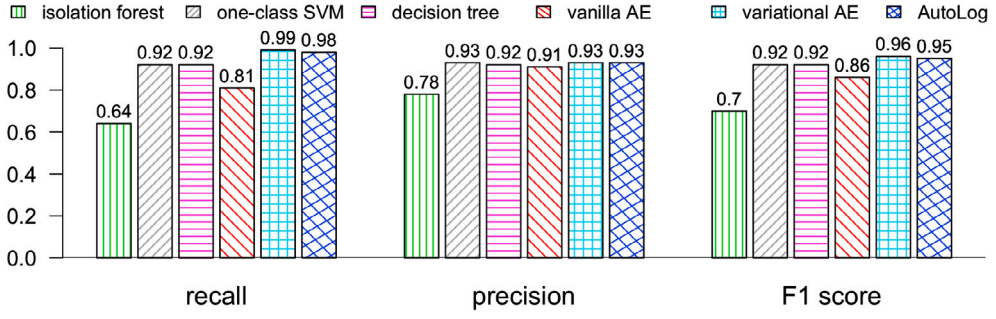


Fig. 23. BG/L.

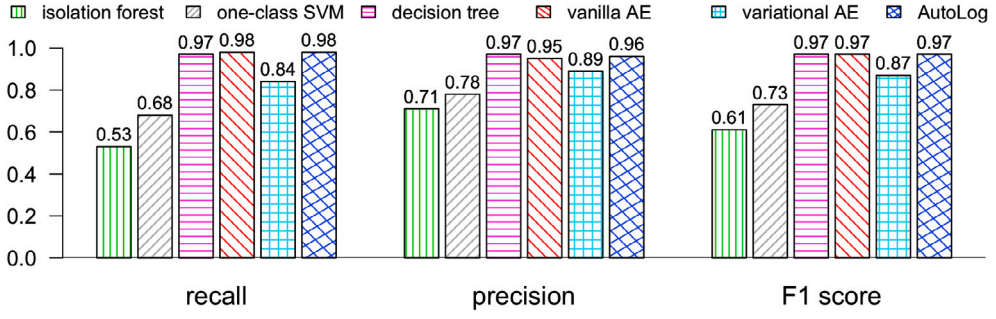


Fig. 24. Hadoop.

public datasets and widely-used benchmarks in log-based anomaly detection. In consequence, we can compare the metrics of AutoLog with other similar proposals in the literature. Authors in Meng et al. (2019) propose a method – called LogAnomaly – and assess several state-of-the-art anomaly detection methods on the same BG/L dataset used in our paper, such as LogCluster and DeepLog. LogAnomaly is based on *template2Vec*, i.e., a template representation method to extract semantic and syntax information from log templates inspired by word embedding. According to the figures reported in Meng et al. (2019) the F1 score of AutoLog in BG/L (0.95) is higher than DeepLog (0.93) and LogCluster (0.57) and similar to LogAnomaly (0.96). Unlike LogAnomaly, which achieves 0.94 recall and 0.97 precision, AutoLog achieves 0.98 recall and 0.93 precision; on the other hand, the precision of AutoLog is higher than DeepLog (0.90). As for the Hadoop dataset, the F1 score of AutoLog (0.97) is equal to the value achieved on the same dataset by the recent OneLog approach (Hashemi & Mäntylä, 2021). Overall, AutoLog is inline with other anomaly detection techniques assessed with the reference BG/L and Hadoop datasets.

Coming to the comparison with approaches implying sequential embedding, the work Guo et al. (2021) applies BERT to the BG/L log and achieves 0.92 recall and 0.89 precision: both the figures are much lower than AutoLog applied to the same dataset. As for other related approaches using BERT with BG/L, it is worth mentioning Hirakawa, Tominaga, and Nakatoh (2020) and Li, Chen, Jing, He, and Yu (2020). The former uses BERT-Base as a transformer encoder and its F1 score on BG/L is 0.89, thus lower than AutoLog: more important, a conventional LSTM autoencoder was demonstrated to achieve a much lower F1 score equals to 0.85 (Hirakawa et al., 2020). The latter proposes SwissLog (Li et al., 2020), which is applied in conjunction with LSTM, Bidirectional LSTM (BiLSTM) and Attention-based BiLSTM. The F1 score of these three variants on BG/L is 0.95, 0.96 and 0.99, respectively: the F1 score of AutoLog is in line with the first two variants, although lower than the Attention-based BiLSTM. Nevertheless, it must be noted that SwissLog is conceived and tested with a much more narrow fault model, consisting of those faults resulting in log sequence order changes and log time interval changes: different from SwissLog, AutoLog is not constrained by specific types of faults.

7. Limitations and threats to validity

As for many existing anomaly detection techniques, AutoLog relies on the availability of normative logs; moreover, AutoLog does not need to know anomalies at training time. Whilst our approach is feasible in practice, we are aware that **assembling normative logs** is a complex matter and may depend on the specific system in hand. For example, both the industrial and microservices systems were deployed in a dedicated LAN environment along with client applications supplied by the vendor and representative benchmarks, which makes us confident that logs actually reflected normative conditions. In practice, normative logs might accidentally account for anomalous events, which are hard to be filtered out before training. In this respect, the 90th percentile approach – used to determine the detection threshold of AutoLog – aims to mitigate the impact of measurement outliers due to accidental anomalies in the normative logs. It is worth noting that in BG/L we deal with spontaneous anomalies with much less confidence on the genuineness of the normative logs. In this respect, the F1 score obtained by AutoLog for BG/L is lower than the industrial and microservices systems, although inline with other anomaly detection techniques assessed with the same dataset.

At the current stage of deployment, AutoLog is a "partial" fit for the detection of **slow attacks**, whose activity may span multiple chunks. In fact, while AutoLog can detect the activity occurring within individual chunks, it misses the ability to "connect the dots" across multiple chunks collected at different times. A slow attack may cause AutoLog to generate multiple alerts, which are supposed to be reconciled later on. On the other hand, AutoLog can detect the activity affecting multiple chunks collected at the same time.

Regarding the **anomaly detection** technique implemented by AutoLog, semi-supervised training tends to obtain from the AE high-quality reconstructions of normative inputs, in such a way that anomalies can be identified by a higher reconstruction error. However, recent studies point out that sometimes autoencoders can provide good reconstructions also for outliers, which are misclassified as being in-distribution. This is obviously detrimental for anomaly detection. Possible solutions tend to improve the anomaly detection capabilities of a simple autoencoder design by resorting to *autoencoder ensembles* (randomly connected autoencoders with different structures and connection densities, Chen, Sathe, Aggarwal, & Turaga, 2017), discriminator networks with a direct anomaly score (Tong, Wolf, & Krishnaswamyt, 2020), or the use of the Grubbs and the PauTa criterion to identify the reconstruction errors corresponding to the outliers based on the traditional threshold method (Wan, Guo, Zhang, Guo, & Liu, 2019). In our experiments, which involved the use of very heterogeneous data sources, the problem did not arise – or was very limited in size – as documented by the good detection performance obtained though our AE. However, possible outlier misclassification is an issue to be considered for our future work to optimize the current design.

As for any data-driven study, there may be concerns regarding the validity and generalizability of the results. We discuss them based on the four aspects of validity listed in Wohlin et al. (2000).

Construct validity. The study builds around the intuition that numeric scores computed from system logs can be used for detecting anomalies of computer systems. This *construct* has been investigated in the context of four systems: an industrial system in the transportation domain, a microservices-based installation implementing a standard multimedia architecture adopted by large telcos, up to publicly-available system logs – common benchmarks in the literature – from a BG/L supercomputer and a Hadoop cluster. The industrial and microservices systems are run with representative load generators. The BG/L log is a public dataset and accounts for hardware and software errors observed over 215 days of operations at LLNL; Hadoop data contain different types of service failures. The study is supported by extensive experimentation leveraging widely-consolidated statistical methods, deep learning framework and evaluation metrics.

Internal validity. The results and key findings of this paper are based on direct measurement experiments, where we analyze systems logs from the reference systems obtained under a mixture of simulated and spontaneous normative operations and anomalies. For example, simulated anomalies in the industrial and microservices systems consist of bruteforce authentication attempts, tampering and misuse, which are inspired by widely-accepted taxonomies in the area. We rely on well-founded log analysis methods for extracting quantitative scores from logs. Noteworthy, AutoLog – based solely on the knowledge of the normative data points – is not biased by the anomalies in hand. The use of such diverse system logs and anomalies aims to mitigate internal validity threats.

Conclusion validity. Conclusions have been inferred by assessing four independent system logs and the sensitivity of key results with respect to the experimental choices. For example, we analyze the impact of different autoencoder configurations on the reconstruction error; similarly, the decision tree is assessed by varying a critical model parameter, such as *minNumObj*. Experiments are complemented by a preliminary discussion on PCA and clustering to gain insights into the challenges existing in our domain. More importantly, we compare AutoLog with a wide set of techniques including isolation forest, one-class SVM, decision trees, vanilla autoencoder and variational autoencoder. We present an extensive discussion of the results. The key findings of the study are consistent across the datasets, which provides a reasonable level of confidence on the analysis.

External validity. The steps of our analysis can be applied to other systems. Nowadays, logs are ubiquitously emitted by almost any system and there exists a wide range of tools for storing and handling logs, which make our approach definitively feasible in practice. In fact, in this paper we successfully ported the experiments across four independent systems – emitting heterogeneous logs – to mitigate external validity threats. Our analysis approach does not interfere with system operations: as only information from logs is used, the approach is inherently *non intrusive*. We are confident that the experimental details provided in the paper would support the replication of our study by future researchers and practitioners.

8. Conclusion and future work

This paper proposed AutoLog, a novel approach for anomaly detection based on deep autoencoding of system logs. AutoLog aims to overcome the challenges of existing log management technologies for the analysis of built-in and proprietary logs files, which lack standard formats. More importantly, AutoLog capitalizes on semi-supervised learning, which does not require anomalies during training. This is potentially valuable to complement current technologies that rely on

pre-established specifications of anomalies. We have conducted an extensive experimentation in the context of four systems and compared AutoLog with a wide set of techniques. The recall of AutoLog ranged between 0.96 and 0.99, while its precision was within 0.93 and 0.98, depending on the system. AutoLog is strongly competitive if compared to a typical supervised technique, such as decision trees; as for BG/L and Hadoop – used as benchmarks by related papers – AutoLog is inline with other log-based anomaly detection techniques available in the literature.

In the future, we will extend our analysis to further systems as well as to existing security datasets, in order to understand the limitations of AutoLog and potential mitigations. At the time being, there are several **open challenges** in anomaly detection that reflect on AutoLog. For example, the availability and construction of normative baselines is a complex matter in log analysis, given the variability of real-life workload and systems. Moreover, log analysis is a "moving" target: software upgrades or changes to the configurations may alter meaning and character of the logs. As for the detection technique used by AutoLog, one key challenge is the threshold selection technique and how it is affected by accidental anomalies intertwined with the normative data. In this respect, future research will investigate some countermeasures to mitigate these issues as well as other threshold selection methods. Other important research avenues pertain to potential strategies to partition logs into chunks, such as considering overlapping chunks, and the explicative power of the autoencoder. Future research will also investigate the actions attackers might take to evade detection and the detection of slow attacks. From a technical standpoint, we will address the implementation of accessory components for AutoLog, such a re-training mechanism, and the use of plugins to extend traditional SIEM's capabilities with the proposed anomaly detection method.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Adiga, N. R., et al. (2002). An overview of the BlueGene/L supercomputer. In *Proc. conference on supercomputing* (pp. 1–22). IEEE.

Almotiri, J., Elleithy, K., & Elleithy, A. (2017). Comparison of autoencoder and principal component analysis followed by neural network for e-learning using handwritten recognition. In *Proc. long island systems, applications and technology conference* (pp. 1–5). IEEE.

Aygun, R. C., & Yavuz, A. G. (2017). Network anomaly detection with stochastically improved autoencoder based models. In *Proc. international conference on cyber security and cloud computing* (pp. 193–198). IEEE.

Bertero, C., Roy, M., Sauvanaud, C., & Tredan, G. (2017). Experience report: Log mining using natural language processing and application to anomaly detection. In *Proc. international symposium on software reliability engineering* (pp. 351–360). IEEE.

Bhatt, S., Manadhata, P. K., & Zomlot, L. (2014). The operational role of security information and event management systems. *IEEE Security & Privacy*, *12*(5), 35–41.

Campos, J. R., Vieira, M., & Costa, E. (2018). Exploratory study of machine learning techniques for supporting failure prediction. In *Proc. European dependable computing conference* (pp. 9–16). IEEE.

Carrington, A. M., Manuel, D. G., Fieguth, P. W., Ramsay, T., Osmani, V., Wernly, B., et al. (2021). Deep ROC analysis and AUC as balanced average accuracy to improve model selection, understanding and interpretation. arXiv:2103.11357.

Catillo, M., Rak, M., & Villano, U. (2020). 2L-ZED-IDS: A two-level anomaly detector for multiple attack classes. In *Advances in intelligent systems and computing, Proc. web, artificial intelligence and network applications* (pp. 687–696). Springer.

Chen, J., Sathe, S., Aggarwal, C., & Turaga, D. (2017). Outlier detection with autoencoder ensembles. In *Proc. SIAM international conference on data mining* (pp. 90–98). SIAM.

Cinque, M., Cotroneo, D., Della Corte, R., & Pecchia, A. (2016). Characterizing direct monitoring techniques in software systems. *IEEE Transactions on Reliability*, *65*(4), 1665–1681.

Cinque, M., Cotroneo, D., & Pecchia, A. (2018). Challenges and directions in security information and event management (SIEM). In *Proc. international symposium on software reliability engineering workshops* (pp. 95–99). IEEE.

Du, M., Li, F., Zheng, G., & Srikumar, V. (2017). DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *Proc. conference on computer and communications security* (pp. 1285–1298). ACM.

Du, J., Vong, C., Pun, C., Wong, P., & Ip, W. (2017). Post-boosting of classification boundary for imbalanced data using geometric mean. *Neural Networks*, *96*, 101–114.

van Erven, T., & Harremos, P. (2014). Rényi divergence and Kullback-Leibler divergence. *IEEE Transactions on Information Theory*, *60*(7), 3797–3820.

Fahimeh, F., & Heikkonen, J. (2018). A deep auto-encoder based approach for intrusion detection system. In *Proc. international conference on advanced communications technology* (pp. 178–183). IEEE.

Farshchi, M., Schneider, J. -G., Weber, I., & Grundy, J. (2018). Metric selection and anomaly detection for cloud operations using log and metric correlation analysis. *Journal of Systems and Software*, *137*, 531–549.

Farzad, A., & Gulliver, T. A. (2020). Unsupervised log message anomaly detection. *ICT Express*, *6*(3), 229–237.

Farzad, A., & Gulliver, T. A. (2021). Log message anomaly detection and classification using auto-B/LSTM and auto-GRU. arXiv:1911.08744.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Guo, H., Yuan, S., & Wu, X. (2021). LogBERT: Log anomaly detection via BERT. arXiv:2103.04475.

Hansen, J. P., & Siewiorek, D. P. (1992). Models for time coalescence in event logs. In *Proc. international symposium on fault-tolerant computing* (pp. 221–227). IEEE.

Hashemi, S., & Mäntylä, M. (2021). OneLog: Towards end-to-end training in software log anomaly detection. arXiv:2104.07324.

Hawkins, S., He, H., Williams, G., & Baxter, R. (2002). Outlier detection using replicator neural networks. In *Proc. international conference on data warehousing and knowledge discovery* (pp. 170–180). Springer.

He, P., Zhu, J., He, S., Li, J., & Lyu, M. R. (2016). An evaluation study on log parsing and its use in log mining. In *Proc. international conference on dependable systems and networks* (pp. 654–661). IEEE.

He, S., Zhu, J., He, P., & Lyu, M. R. (2016). Experience report: System log analysis for anomaly detection. In *Proc. international symposium on software reliability engineering* (pp. 207–218). IEEE.

Hinton, G. E., Osindero, S., & Teh, Y. -W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, *18*(7), 1527–1554.

Hirakawa, R., Tominaga, K., & Nakatoh, Y. (2020). Software log anomaly detection through one class clustering of transformer encoder representation. In C. Stephanidis, & M. Antona (Eds.), *Proc. HCI international - posters* (pp. 655–661). Springer.

Holzinger, A., Hörtenhuber, M., Mayer, C., Bachler, M., Wassertheurer, S., Pinho, A. J., et al. (2014). On entropy-based data mining. In A. Holzinger, I. Jurisica (Eds.), *Interactive knowledge discovery and data mining in biomedical informatics: State-of-the-art and future challenges* (pp. 209–226). Springer.

Kim, S., Jo, W., & Shon, T. (2020). APAD: Autoencoder-based payload anomaly detection for industrial IoE. *Applied Soft Computing*, *88*, Article 106017.

Kingma, D. P., & Welling, M. (2019). An introduction to variational autoencoders. *Foundations and Trends in Machine Learning*, *12*(4), 307–392.

Li, X., Chen, P., Jing, L., He, Z., & Yu, G. (2020). SwissLog: Robust and unified deep learning based log anomaly detection for diverse faults. In *Proc. international symposium on software reliability engineering* (pp. 92–103). IEEE.

Lin, Q., Zhang, H., Lou, J. -G., Zhang, Y., & Chen, X. (2016). Log clustering based problem identification for online service systems. In *Proc. international conference on software engineering companion* (pp. 102–111). IEEE.

Liu, J., Song, K., Feng, M., Yan, Y., Tu, Z., & Zhu, L. (2021). Semi-supervised anomaly detection with dual prototypes autoencoder for industrial surface inspection. *Optics and Lasers in Engineering*, *136*, Article 106324.

Liu, F. T., Ting, K. M., & Zhou, Z. (2008). Isolation forest. In *Proc. international conference on data mining* (pp. 413–422). IEEE.

Liu, D., Zhao, Y., Xu, H., Sun, Y., Pei, D., Luo, J., et al. (2015). Opprentice: Towards practical and automatic anomaly detection through machine learning. In *Proc. internet measurement conference* (pp. 211–224). ACM.

Lu, S., Rao, B., Wei, X., Tak, B., Wang, L., & Wang, L. (2017). Log-based abnormal task detection and root cause analysis for spark. In *Proc. international conference on web services* (pp. 389–396). IEEE.

Macià-Fernández, G., Camacho, J., Magàn-Carriòn, R., García-Teodoro, P., & Theròn, R. (2018). UGR'16: A new dataset for the evaluation of cyclostationarity-based network IDSs. *Computers & Security*, *73*, 411–424.

Meng, W., Liu, Y., Zhu, Y., Zhang, S., Pei, D., Liu, Y., et al. (2019). LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *Proc. international joint conf. on artificial intelligence* (pp. 4739–4745). International Joint Conferences on Artificial Intelligence Organization.

Miller, D. R., Harris, S., Harper, A., VanDyke, S., & Blask, C. (2010). *Security information and event management (SIEM) implementation*. McGraw-Hill Education.

Nguyen, Q. P., Lim, K. W., Divakaran, D. M., Low, K. H., & Chan, M. C. (2019). GEE: A gradient-based explainable variational autoencoder for network anomaly detection. In *Proc. conference on communications and network security* (pp. 91–99). IEEE.

Oliner, A., Ganapathi, A., & Xu, W. (2012). Advances and challenges in log analysis. *Communications of the ACM*, *55*(2), 55–61.

Oliner, A., & Stearley, J. (2007). What supercomputers say: A study of five system logs. In *Proc. international conference on dependable systems and networks* (pp. 575–584). IEEE.

Oprea, A., Li, Z., Yen, T., Chin, S. H., & Alrwais, S. (2015). Detection of early-stage enterprise infection by mining large-scale log data. In *Proc. international conference on dependable systems and networks* (pp. 45–56). IEEE.

Pang, G., Shen, C., Cao, L., & Hengel, A. V. D. (2021). Deep learning for anomaly detection: A review. *ACM Computing Surveys*, *54*(2), 1–38.

Pisner, D. A., & Schnyer, D. M. (2020). Chapter 6 - Support vector machine. In *Machine learning* (pp. 101–121). Academic Press.

Qian, Y., Ying, S., & Wang, B. (2020). Anomaly detection in distributed systems via variational autoencoders. In *Proc. international conference on systems, man, and cybernetics* (pp. 2822–2829). IEEE.

Quan, X., Wenyin, L., & Qiu, B. (2011). Term weighting schemes for question categorization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *33*(5), 1009–1021.

Ruff, L., Vandermeulen, R. A., Görnitz, N., Binder, A., Müller, E., Müller, K. -R., et al. (2020). Deep semi-supervised anomaly detection. In *Proc. international conference on learning representations*.

Ruiu, D. (1999). Cautionary tales: Stealth coordinated attack how to. http://www.ouah.org/stealthhowto.html.

Sakurada, M., & Yairi, T. (2014). Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Proc. workshop on machine learning for sensory data analysis* (pp. 4–11). ACM.

Salton, G., & Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, *24*(5), 513–523.

Schölkopf, B., Platt, J. C., Shawe-Taylor, J. C., Smola, A. J., & Williamson, R. C. (2001). Estimating the support of a high-dimensional distribution. *Neural Computing*, *13*(7), 1443–1471.

Shone, N., Ngoc, T. N., Phai, V. D., & Shi, Q. (2018). A deep learning approach to network intrusion detection. *IEEE Transactions on Emerging Topics in Computational Intelligence*, *2*(1), 41–50.

Skansi, S. (2018). Autoencoders. In *Introduction to deep learning: From logical calculus to artificial intelligence* (pp. 153–163). Springer International Publishing.

Song, C., Liu, F., Huang, Y., Wang, L., & Tan, T. (2013). Auto-encoder based data clustering. In *Progress in pattern recognition, image analysis, computer vision, and applications* (pp. 117–124). Springer.

Stearley, J., & Oliner, A. J. (2008). Bad words: Finding faults in Spirit's syslogs. In *Proc. international symposium on cluster computing and the grid* (pp. 765–770). IEEE.

Su, Y., Zhao, Y., Niu, C., Liu, R., Sun, W., & Pei, D. (2019). Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In *Proc. international conference on knowledge discovery & data mining* (pp. 2828–2837). ACM.

Svacina, J., Raffety, J., Woodahl, C., Stone, B., Cerny, T., Bures, M., et al. (2020). On vulnerability and security log analysis: A systematic literature review on recent trends. In *Proc. international conference on research in adaptive and convergent systems* (pp. 175–180). ACM.

Tong, A., Wolf, G., & Krishnaswamyt, S. (2020). Fixing bias in reconstruction-based anomaly detection with Lipschitz discriminators. In *Proc. international workshop on machine learning for signal processing* (pp. 1–6). IEEE.

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., & Manzagol, P. A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, *11*, 3371–3408.

Wadekar, A., Gupta, T., Vijan, R., & Kazi, F. (2019). Hybrid cae-vae for unsupervised anomaly detection in log file systems. In *Proc. international conference on computing, communication and networking technologies* (pp. 1–7). IEEE.

Wan, F., Guo, G., Zhang, C., Guo, Q., & Liu, J. (2019). Outlier detection for monitoring data using stacked autoencoder. *IEEE Access*, *7*, 173827–173837.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2000). *Experimentation in software engineering: An introduction*. Kluwer Academic.

Xu, W., Huang, L., Fox, A., Patterson, D. A., & Jordan, M. I. (2008). Mining console logs for large-scale system problem detection. In *Proc. tackling computer systems problems with machine learning techniques* (p. 4). USENIX.

Yadav, R. B., Kumar, P. S., & Dhavale, S. V. (2020). A survey on log anomaly detection using deep learning. In *Proc. international conference on reliability, infocom technologies and optimization (trends and future directions)* (pp. 1215–1220). IEEE.

Yang, R., Qu, D., Gao, Y., Qian, Y., & Tang, Y. (2019). nLSALog: An anomaly detection framework for log sequence in security management. *IEEE Access*, *7*, 181152–181164.

Yuan, Y., Srikant Adhatarao, S., Lin, M., Yuan, Y., Liu, Z., & Fu, X. (2020). ADA: Adaptive deep log anomaly detector. In *Proc. INFOCOM - conference on computer communications* (pp. 2449–2458). IEEE.

Zhang, Y., Lu, Z., & Wang, S. (2021). Unsupervised feature selection via transformed auto-encoder. *Knowledge-Based Systems*, *215*, Article 106748.

Zhang, K., Xu, J., Min, M. R., Jiang, G., Pelechrinis, K., & Zhang, H. (2016). Automated IT system failure prediction: A deep learning approach. In *Proc. international conference on big data* (pp. 1291–1300). IEEE.

Zhao, Y., Hao, K., Tang, X., Chen, L., & Wei, B. (2021). A conditional variational autoencoder based self-transferred algorithm for imbalanced classification. *Knowledge-Based Systems*, *218*, Article 106756.

Zoppi, T., Ceccarelli, A., & Bondavalli, A. (2016). Context-awareness to improve anomaly detection in dynamic service oriented architectures. In A. Skavhaug, J. Guiochet, & F. Bitsch (Eds.), *Computer safety, reliability, and security* (pp. 145–158). Springer.