Erratic server behavior detection using machine learning on basic monitoring metrics

Martin Adam,[a,*] **Luca Magnoni**[b] **and Dagmar Adamová**[a]

[a] *Academy of Sciences of the Czech Republic (CZ)*
[b] *CERN, European Organization for Nuclear Research (CH)*

E-mail: madam@cern.ch

With the explosion of the number of distributed applications, a new dynamic server environment emerged grouping servers into clusters, utilization of which depends on the current demand for the application. To provide reliable and smooth services it is crucial to detect and fix possible erratic behavior of individual servers in these clusters. Use of standard techniques for this purpose requires manual work and delivers sub-optimal results. Using only application agnostic monitoring metrics our machine learning based method analyzes the recent performance of the inspected server as well as the state of the rest of the cluster, thus checking not only the behavior of the single server, but the load on the whole distributed application as well. We have implemented our method in a Spark job running in the CERN MONIT infrastructure. In this contribution we present results of testing multiple machine learning algorithms and pre-processing techniques to identify the servers erratic behavior. We also discuss the challenges of deploying our new method into production.

40th International Conference on High Energy physics - ICHEP2020
July 28 - August 6, 2020
Prague, Czech Republic (virtual meeting)

*Speaker

1. Introduction

In the last few decades the amount of digitally saved data has been growing exponentially. In 2014 the world's technological capacity to store information has reached almost 5 zettabytes [1]. Handling this incredible amount of incoming data requires innovative techniques increasingly leveraging horizontal scaling; an approach utilizing many computers instead of one more powerful. Such novel approaches create additional concerns for the system administrators, particularly when it comes to noticing errors that pose a threat to the efficiency and availability of the application. Although traditional monitoring methods require lots of manual labor when applied to this problem, developers of open source monitoring system have been so far reluctant to include any advanced tools. In this project we set to explore the possibility of using machine learning to spot erratic servers within a cluster running a distributed application.

In an attempt to simplify administrators work, many applications offer a set of internal metrics describing their performance. Incorporating these metrics in the existing monitoring systems might be too time-consuming, considering that the lack of skilled administrators often leads to understaffed teams. Therefore this project avoids using application specific metrics considering only basic os-level performance metrics, which leads to a black-box approach of the servers.

In this article, we present a process of acquiring and processing a stream of raw monitoring data in the MONIT [2] infrastructure. The data is then analyzed offline by a concept application using a data processing pipeline to detect unexpected behavior considering both workload (current load on the distributed application) and workflow (usual node behavior running a specific application). In its core is a prediction model trained by machine learning. We also discuss the efficiency of such an approach, benchmark the core model and present plans for future development.

2. Monitoring Systems Overview

The primary goal of traditional monitoring systems is gathering and saving monitoring data, with presenting and especially analysing it as a secondary objective. Instead, some applications are making the footprint of the client as small as possible, such as Ganglia [3], others offer effortless extensibility, such as Munin [4]. The only common erratic behavior detection feature available is comparing a single metric to a pre-defined threshold.

A completely different approach is taken by Nagios [5], which specializes in monitoring a status of a service on a server, with collecting metrics as a secondary task. It however does not offer an easy way to correlate metrics within a server, let alone within a cluster of servers.

Academic works on anomaly detection in monitoring data are available, but custom systems are always used. Some works explore the subject from the angle of intrusion detection [6]. Others take on the challenge of predicting failures, but given the advanced difficulty of this task all the works make use of all the available data including application metrics or logs [7] making them very application specific. There are some works looking into the black box method [8], but they are usually only showcased on artificial datasets.

At CERN an extensive monitoring system has been build to gather, process and visualize metrics and logs from the 40,000 machines powering the entire scientific, administrative and computing infrastructure. This data pipeline, which handles more than 3TB of compressed data per day, is composed of several distributed services, each running on several tens of machines, providing key functionality such as Apache Flume (data collection and ingestion) [9], Apache Kafka (data transport) [10] and Apache Spark (data processing) [11]. Each of those services represents the perfect case study for our research, as an application that distributes its load evenly was needed. This project will be using the MONIT infrastructure as a platform to both collect and analyze data.

3. Data Gathering

The MONIT infrastructure offers OS metrics via its collectd stream [12]. Collectd is a UNIX daemon used to collect and send performance metrics. The metrics are reported individually by type and also each host reports them at a different time. We therefore decided to aggregate the metrics individually over a fixed time window first and join them together later. The longest metric readout interval among these metrics is 5 minutes, so the averaging time window length was set as 20 minutes. This guarantees that in each time window we get at least 4 samples of each metric, providing a simple smoothing procedure.

Two Spark streaming jobs were deployed to the MONIT infrastructure realizing the described operations. First is consuming all the collectd data. To decompress and read 13MB/s it needs to run in cluster mode on 25 CPU cores and using over 111GB of memory. It selects only data for clusters of interest, applies the aggregation and writes the processed data back into Kafka.

The second job consumes only the processed data and therefore has a significantly smaller resource usage. As joining is a stateful operation, it still needs to keep a checkpoint saved for fault-tolerance, which can pose a challenge to the underlying storage system due to its size. Metrics are joined on hostname and time-window and the results are then written by a Flume agent to HDFS for later analyses.

3.1 Creating Anomalies

It became clear soon after we started collecting the data, that the number of naturally occurring anomalies is too small for a meaningful statistic. We selected a cluster of virtual servers running the Kafka service for MONIT in a development setting[1] and started creating our own anomalies. Alternating the already collected metrics might not produce plausible results, so creating an error on the server itself, while data was being collected and the rest of the cluster ran a normal workload, was our prefered option. Several classes of errors were thought of and implemented:

- Base line anomaly — the main application is shut down on one of the nodes in the cluster. Six were collected.
- Memory-leak type — the memory usage gradually increases up until a point where the OOM killer. Two were collected.[2]. In this case the memory allocation is stopped before the OOM killer intervenes, to avoid the situation when the main application process is killed.

[1]This cluster was used for testing changes before applying them into production. Most of the time the cluster ran with the same setting as the production cluster, but meddling with it did not pose a risk to the stability of services provided by MONIT.

[2]Out Of Memory Killer (OOM) is a process that the Linux kernel employs when the system is critically low on memory. The kernel attempts to recover memory by terminating programs so that the system can continue running

- CPU over-utilization — the CPU usage increases conflicting with the main application possibly rendering it un-operationable. Eight were collected.
- Combined — both cpu and memory usage increases. Such an anomaly might indicate a rogue program being launched, or defective load balancing in the main application (the anomalous node is overloaded). Ten were collected.

The number of anomalies varies. This is due to the fact, that the service had to run smoothly for a period of time before we started inserting anomalies for a model to be able to learn. After at least 2 days of stable and uninterrupted production, anomalies were inserted once or twice per day. In the settings provided, capturing these conditions proved to be more challenging than expected.

4. Analysing the Data

Collecting all the available collectd metrics resulted in status snapshots with 35 attributes. Input data with such high dimensionality might cause sub-optimal performance of some of the algorithms and because we were expecting a portion of the metrics to be correlated, the Principle Component Analysis was employed [13]. The number of dimensions to be used after the PCA transformation based on the variance explained by respective components was selected to be 3 or 5 for some algorithms.

Although the usual approach to anomaly detection is unsupervised learning (clustering or one-class classification algorithms), we did not see any significant results using these, therefore we omit those from this paper for the sake of length.

4.1 Supervised learning

Although using supervised classification on labeled anomalies was agreed not to be explored in this work, supervised learning could be employed in a different way. With time series data, a regression task is hidden in predicting the next data point based on one or several historic ones. To represent the relevant history, input data are several metric vectors concatenated together, each representing a server status snapshot in time. The output would then be vector describing the next status snapshot[3]. The dimensionality of such input data can grow dramatically when trying to capture longer periods of time or without using any additional dimensionality reduction techniques. With this in mind, the decision was made to set the number of historical snapshots n to 4. With the 20 minute time window that gives the algorithm knowledge about an 80 minutes history of the server in question.

$$X_t = (x_t^1, x_t^2, x_t^3, ...x_t^N)$$
$$f(X_{t-n}, X_{t-(n-1)}, ...X_{t-1}) = \tilde{X}_t$$

To provide information about the rest of the cluster, each status snapshot was extended by the clusters average at that time window. When using the first three dimensions of PCA, this gave us a input vector with 24 attributes.

[3]All of the used classifiers support only a one dimensional output, therefore each data dimension had its own prediction model

As a baseline to be able to benchmark the performance of more complex algorithms, we use the simplest future prediction possible: repeating the last seen state. Because the nature of our data is mostly stable, this approach yielded reasonably good base results.

To consider the standard approach to time series the ARIMAX[4] class of models was also evaluated.

Mainly two algorithms supported by the SparkML library [14] were employed. These would least complicate the future conversion to a production application. For initial code development and testing the **Linear Regression** model was used for its simplicity and speed. Then the same approach was applied to train a **Random Forest** model [15], which could model more elaborate functions and was the only available ensemble method supporting full parallelization. Hyper parameters for both models were selected using a grid search with 3-fold cross validation as the evaluating technique. The evaluation metric for the cross validation was root mean squared error, a standard regression evaluation metric.

The algorithms prediction was compared to the actual next status and a prediction error is computed.

$$error = \sqrt{(\tilde{X}_t - X_t)^2}$$

Information about the anomalousness of a specific data point was then extracted by comparing the prediction error to the average prediction error during that day.

$$is\_anomaly(d) = \begin{cases} True & \text{if } error(d) > mean\_error + 3 * std\_dev \\ False & \text{otherwise} \end{cases}$$

The results after applying this rule were not satisfactory, with the best being the Random Forest with a precision and recall of 0.81 and 0.38 respectively[5]. When reviewing these results, we inspected the progression of the prediction error. The beginning of the anomaly was usually denoted with a peak, but so was the end. Moreover most of the false positives (for the more advanced algorithms) came from data points representing non-anomalous servers in the same time the anomaly ended (see Table 1). Both the end peak and the false positives might be explained by the application at the anomalous host synchronizing itself with the rest of the cluster: the anomalous host has to consume all the data that arrived during the time its service was not operational and the data has to be streamed from other members of the cluster.

Table 1 would suggest, that if there is an anomaly, there should be more than one high errors in a row. Scanning for two notifications in a row lowered the number of notifications significantly, but most importantly filtered all of the false positives. To counter act the drop of recall, the $3 * std\_dev$ threshold was relaxed to a single standard deviation. The final results are listed in Table 2. Note that the results can be improved by not applying the PCA dimensionality reduction. This however resulted in the need to train 35 models, because the algorithms implemented in SparkML support only a single dimensional output. If the full hyper parameter selection would to be done, the

[4]AutoRegressive Integrated Moving Average with eXogeneous input

[5]precision = $\frac{\text{true positives}}{\text{positives}}$; recall = $\frac{\text{true positives}}{\text{anomaly count}}$

pred_time	hostname	anomaly
2019-02-09 12:00:00	anomalous.cern.ch	True
2019-02-09 12:20:00	anomalous.cern.ch	True
2019-02-09 12:40:00	anomalous.cern.ch	True
2019-02-09 13:00:00	anomalous.cern.ch	True
2019-02-09 16:00:00	host01.cern.ch	False
2019-02-09 16:00:00	host09.cern.ch	False
2019-02-09 16:00:00	anomalous.cern.ch	True

Table 1: All anomalies marked by applying the three-sigma rule to the first dimension error of the Random Forest regressor on a day with a service stop anomaly. The anomaly started at 12:00 and ended at 16:00.

computational demand would be too high. Instead the hyperparameters were fixed after running cross-validation once on a single set of training data.

model	recall
Random Forest - nopca	0.78
Random Forest	0.70
Linear Regression	0.56
ARIMAX	0.56
BASELINE	0.48

Table 2: Summary results for the whole system with notification filtering and $1 * std\_dev$ as the anomaly classification prediction error threshold. Notifying on the anomaly at the time of its end is not counted as valid notification.

5. Conclusions and Future Work

This project set to explore the possibility of using advanced data analytics to detect anomalies implying erratic behavior of nodes in a cluster running a distributed application. We first built an infrastructure to collect, pre-process and store os-level monitoring data using the MONIT infrastructure at CERN. With continuous data gathering in place, we selected a cluster of ten nodes running Kafka, a distributed streaming application. We then began altering the server performance by running other programs or stopping the Kafka application service to induce anomalous behavior.

After testing multiple different approaches, best results were yielded by employing the Random Forest algorithm to predict the state of a server based on its short history augmented by the clusters average. To improve the usability for the final user a simple heuristic eradicating the false positives while keeping a high recall rate was employed. The final system with a Random Forest regressor, PCA dimensionality reduction and anomaly notification filtering heuristic yielded a recall of 0.7 and precision of 100%, with all the notifications were issued shortly after the anomaly start. Recall could be improved still to 0.78 by using a model trained on data without reduced dimensionality, which however increases the complexity and also results in a delay in detecting certain types of anomalies.

Work on this project will continue by developing a production version. This will pose many challenges, including making the process of extraction of anomalies from prediction errors online. Also the problems of automatic retraining of the models have to be solved. Those involve the problem of detecting days, when the whole application is experiencing difficulties and excluding those from the training set and others.

References

[1] M. Hilbert, *Information quantity*, in *Encyclopedia of Big Data*, (Cham), pp. 1–4, Springer International Publishing (2017), DOI.

[2] A. Aimar, A.A. Corman, P. Andrade, S. Belov, J.D. Fernandez, B.G. Bear et al., *Unified monitoring architecture for it and grid services*, *Journal of Physics: Conference Series* **898** (2017) 092033.

[3] M.L. Massie, B.N. Chun and D.E. Culler, *The ganglia distributed monitoring system: design, implementation, and experience*, *Parallel Computing* **30** (2004) 817 .

[4] P. Jung, *Munin-the raven reports*, *Linux J.* **2009** (2009) .

[5] "Nagios, the industry standard in it infrastructure monitoring." `www.nagios.org/projects/nagios-core/`.

[6] A.L. Buczak and E. Guven, *A survey of data mining and machine learning methods for cyber security intrusion detection*, *IEEE Communications Surveys & Tutorials* **18** (2015) 1153.

[7] X. Gu and H. Wang, *Online anomaly prediction for robust cluster systems*, pp. 1000 – 1011, 05, 2009, DOI.

[8] X. Pan, J. Tan, S. Kavulya, R. G and P. Narasimhan, *Ganesha: Black-box fault diagnosis for mapreduce systems*, Tech. Rep. (2008).

[9] "Distributed service for collecting, aggregating and moving log data, "Apache Flume" [software]."

[10] N. Garg, *Apache Kafka*, Packt Publishing (2013).

[11] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave et al., *Apache spark: A unified engine for big data processing*, *Commun. ACM* **59** (2016) 56.

[12] "Collectd, the system statistics collection daemon." `www.collectd.org`.

[13] H. Abdi and L.J. Williams, *Principal component analysis*, *WIREs Comput. Stat.* **2** (2010) 433.

[14] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu et al., *Mllib: Machine learning in apache spark*, *J. Mach. Learn. Res.* **17** (2016) 1235.

[15] L. Breiman, *Random forests*, *Machine Learning* **45** (2001) 5.