FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Martin Adam

Machine Learning in the Monitoring of Computer Clusters

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	Mgr. Martin Pilát, Ph.D.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2020

To
those who helped me,
those who will be with me to celebrate
and
those who will not,

thank you.

Title: Machine Learning in the Monitoring of Computer Clusters

Author: Martin Adam

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: With the explosion of the number of distributed applications, a new dynamic server environment emerged grouping servers into clusters, whose utilization depends on the current demand for the application. Detecting and fixing erratic server behavior is paramount for providing maximal service stability and availability. Using standard techniques to detect such behavior is yielding suboptimal results. We have collected a dataset of OS-level performance metrics from a cluster running a streaming distributed application and injected artificially created anomalies. We then selected a set of various machine learning algorithms and trained them for anomaly detection on said dataset. We evaluated the algorithms performance and proposed a system for generating notifications of possible erratic behavior, based on the analysis of the best performing algorithm.

Keywords: machine learning, system administration, anomaly detection

Contents

Introduction

In the last few decades the amount of digitally saved data has been growing exponentially [1]. In 2014 the world's technological capacity to store information has reached almost 5 zettabytes [2]. Handling this incredible amount of incoming data requires innovative techniques increasingly leveraging horizontal scaling; an approach utilizing many computers instead of one more powerful. This trend, which is only expected to grow with the predicted end of the Moores law [3], explains the explosion of the number of distributed applications.

An application is called distributed, when its' processes run on two or more computers. The main benefit of this approach is being able to add a new worker to the pool of computers running the application, when the load exceeds current capabilities. There is also a financial motivation as several average servers tend to be cheaper than one super-powerful, while simultaneously tackling the problem of a single point of failure. The disadvantages stem from additional complexity caused by not sharing one operating system. Unlike in a single host multi-process environment, the processes of a distributed application cannot rely on a shared memory address space for passing messages or data. Instead the network has to be used, which brings the additional computational overhead of serialization, communication administration and deserialization as well as a delay caused by network latency. This is only one of many factors contributing to the additional complexity of distributed applications compared to standard ones, other examples are load balancing and fault tolerance.

This new approach comes with new challenges for the operations teams maintaining the infrastructure. Most of the standard monitoring and provisioning tools were developed to work well with standard applications, leaving most of the administration support to be dealt with by the application itself. Each new application therefore has its' own approach, metrics and APIs that the system administrators have to become familiar with and incorporate in already existing systems. Considering the constant lack of skilled professionals [citation needed?] this might be a problem especially when examining multiple similar solutions to pick the best for future usage. Even after the preferred application is selected and deployed, fully understanding a wide range of metrics and their patterns adds yet another problem to be tackled by the already overworked administrator.

Many monitoring tools focus primarily on collecting and storing the monitoring data. Alarms signaling the possibility of a problem are usually based on comparing the current value to a predefined threshold, if implemented at all. Creating more sophisticated checks by aggregating or correlating different metrics is usually left to the user, often without providing a usable API. Even displaying different types of metrics for a group of servers to assist with visual inspection is not always well supported.

This work explores the possibility of using advanced data analytics to detect anomalies implying erratic behavior of nodes in a cluster running a distributed application. More specifically the objectives this work sets are

- to develop an infrastructure to collect, pre-process and store monitoring data using the MONIT infrastructure,
- evaluate multiple approaches of modeling the stored data,

- propose an automatic procedure using a model of the data to identify erratically behaving servers among the cluster running a distributed application before the error on said server can cause a fault in the application itself.

In Chapter 1 several classical monitoring systems along with a number of related works in monitoring data analysis are reviewed and discussed. Chapter 2 clearly states the problem hinted here and defines the goals as well as non-goals of this work. Chapter 3 describes creating the dataset used for teaching the anomaly detection module including the process and challenges of collecting the data and creating artificial anomalies. Chapter 4 introduces the various approaches of detecting the anomalies in the dataset and Chapter 5 evaluates their effectivity. Finally the last chapter provides conclusions and hints the direction of future work.

1. Monitoring Systems Overview

First an overview of three all-in-one monitoring systems will be provided. Then an alternative approach to creating a monitoring system will be mentioned. Finally a number of papers on analysis of monitoring data will be reviewed and discussed.

1.1 All-in-one Monitoring Systems

The following systems are examples of a complete solution covering every task from collecting data, transporting and storing them to presenting them to the user.

Ganglia is an open-source, scalable, distributed monitoring system developed for large clusters and computing grids [4]. Written in C with minimizing system requirements of the agents as a priority it became a standard for computing center monitoring. The default installation will gather usual OS metrics[1], but additional plugins for other numeric metrics can be added by the user. Data collecting is organized in a tier system, where only a couple of nodes in a cluster assemble the metrics for the other and then send the complete bundle to one central server. Clusters are displayed together on the website making spotting differently behaving nodes easier, however no alerting is available.

Munin is a monitoring tool developed with an emphasis on modularity and ease of use [5]. Expanding the pre-configured set of metrics is almost effortless making Munin perfect for unusual types of metrics. Gathered data are displayed in charts using a simple web interface. Basic alarms based on constant thresholds can be defined for individual metrics, however no advanced alarm handling infrastructure is available. The focus on simplicity meant sacrificing scalability to more than 5000 hosts per instance on the server side.

Nagios is unlike the two project mentioned earlier centered on alerting and system state [6] rather than recording system performance. In the Core installation, metrics are not stored for historical review, instead each host contains several services that change states between OK, Warning and Error. The current service state is determined and reported by a plugin running on the client. Plugins range from simple to complex depending on the service[2] and can be added by the user.

Note that the listed tools are just examples of free options available to Linux system administrators. Making a complete list of all-in-one monitoring solutions is out of scope of this work.

[1]Metrics available from the operating system describing the load on the server. Typical set will contain cpu metrics (percentage of user, system usage and idle), memory (size of memory used for application, buffers and cache and left free) as well as network and disk utilization.

[2]Service in this context can represent a simple thing, such as the current load being under a certain limit, but can also be complex (checking mount state and accessibility of predefined file systems, testing a performance of an application or checking network connectivity).

1.2 Modular Monitoring Systems

Computing centers vary in many ways (size, provided services, administrators experience), therefore creating one solution to fit them all is impossible. However the basic tasks[3] of the monitoring systems are always the same. Decoupling these tasks to independent programs/modules can provide much desired flexibility of the final system (a smaller computing center can have a simple database and larger computing center may use a distributed storage, while they both use the same daemon for collecting the data). The following systems are examples of commonly used combinations.

1.2.1 CollectD + InfluxDB + Grafana

The lightweight **CollectD** daemon was introduced in 2005 as a simple but modular solution to the problem of gathering (and storing) system and application performance metrics [7]. Widespread usage was aided by the daemon being written in C, which minimized resource requirements and made it a reasonable option even for less powerful systems. Although featuring a simple web interface for presenting results, the project was from the beginning focused mainly on collecting and secondly on storing metrics. The storage part is natively handled by the RRDTool plugin[4], which is in line with the overall simplicity and lightweightness of the project, but sub-optimal for modern visualization and other data handling standards.

Compensating for this weakness multiple plugins are available providing an interface to many commonly used storage solutions. Moreover a universal Network Write plugin could be used when the storage of choice is not specifically supported as is the case of InfluxDB — the storage back-end in this example. **InfluxDB** is an open-source time series database [8], which makes it ideal for storing monitoring data[5]. Expected features like data retention and down-sampling are easily configurable, moreover a SQL-like query language is implemented for advanced data handling. It is mentioned in this example instead of one of the storage solutions supported by plugins because it is not specifically designed for monitoring data so it might be already deployed at the computing center for other reasons. Also it is supported by the third party analytics and visualization platform Grafana.

Grafana is an open-source general purpose web dashboard supporting multiple storage back-ends [9]. It offers many different types of visualization methods as well as the ability to query the data making it a great tool for visual knowledge extraction. It also features simple threshold based alarms with various notification hooks.

This example illustrates the versatility of modern products compared to the older ones. In this case Grafana could use Graphite[6] as its' data source and the creators of InfluxDB have introduced their own take on the metric collection

[3]collecting data, transporting them to a database for archiving, serving them to the user

[4]Based on the RRDTool by Tobias Oetiker `https://oss.oetiker.ch/rrdtool/index.en.html`. RRDTool is used by all the previously mentioned monitoring tools as well.

[5]Monitoring data are single values acquired in discreet times (usually evenly spaced), thus being a prime example of a time series data stream.

[6]`graphiteapp.org`

problem. But system administrators cannot always choose, which specific solution to deploy so by employing this kind of modularity the developers secure a higher chance of their product being used. Note however, that although this example mentions state-of-the-art products, there is no support for advanced data analytics.

1.2.2 Elastic Stack

The Elastic Stack (formerly ELK Stack) is a set of open-source tools for gathering, storing and visualizing data mainly from text files [10]. Combining Elasticsearch — a real-time distributed open-source analytics and search engine [11] — with two other stand-alone projects (Logstash for collecting raw log data and transforming it in Elasticsearch compatible JSON documents and Kibana for visualization) created a widely used centralized logging solution. The components are again changeable. For example Grafana supports Elasticsearch as a data back-end, so it could be used instead of Kibana.

Collecting logs is substantially different from numeric metrics in several ways:

- Numerical metrics just need to be read and collected, log lines on top of that have to be parsed into multiple fields containing meaningful information. This process has to be repeated for each application (sometimes logging schemes can change even with different versions of the same application), since the structure of the log line and the relevant parts vary.

- Log lines in general have variable length, which may pose a challenge for transport and especially storage, whereas numerical metrics have a fixed scalar type.

- Due to log lines being a more complex object, the data extracted might be visualized and explored in more ways. Spanning from simple counting the number or frequency of a certain type of log line through graphing a numerical metric extracted from the log to exploring the raw messages, the dashboard in a log consuming application has fulfill all these tasks.

The Elastic Stack is mentioned to illustrate the variety in monitoring data. The operators might need to collect not only hardware metrics, but analyze log lines as well in order to spot as many problems as possible, although that requires expert knowledge and lots of time.

1.2.3 MONIT

One of the most important scientific communities, for which handling and processing huge amounts of data became essential, is the High Energy Physics (HEP) community of the experiments at the Large Hadron Collider (LHC) at CERN[7]. The volumes of collected data are larger than any single computing center within the LHC collaboration could handle, so the concept of distributed data management was conceived. For almost two decades now, the WLCG — an international

[7]European Organization for Nuclear Research (name derived from Conseil Européen pour la Recherche Nucléaire) — European research organization that operates the largest particle physics laboratory in the world.

collaborative grid-based computer network [12] — has been essential to CERNs' scientific endeavour.

Recently, it was agreed to merge WLCG monitoring services with the internal CERN IT data centre monitoring, which were both using similar technologies but split for historical reasons. This effort resulted in the development of a Unified Monitoring Architecture (UMA) [13], which after deployment became known as MONIT. The goal was to create a modern monitoring system with many various data inputs capable of handling large data throughput, enabling the operators and experiment experts to analyze the data streams in real time as well as storing all of it for later visualization and analysis. The overall architecture can be seen in Figure 1.1, all the major components will be introduced in the following section.

Data Sources create an interface between all the various monitored environments. Most of those are controlled by the experiment collaborations and have different ways to publish their data. Several standard methods of data collection were identified and implemented including direct database connection, using Apache ActiveMQ[8], accepting logs or reading a HTTP feed. The local CERN computing center hardware metrics are gathered using CollectD. Other metrics include transfer monitoring, grid job monitoring, network monitoring and others. All the data is channeled through a set of nodes running Apache Flume[9].

Transport buffer with a retention period of 72 hours is set up using a cluster of Apache Kafka nodes [14]. The data arrive in a predefined scheme from the Flume sinks and are available for low-latency real-time operations. There are 20 Kafka nodes used for production data and a smaller cluster for QA, all of them are virtual machines (VMs) hosted on the CERN IT OpenStack[10]. In MONIT each partition has 3 replicas for fault tolerance and load balancing.

Processing of the data is done using a Apache Spark cluster [15]. Data could be either read as a stream directly from Kafka or as batches from perma-

[8] `activemq.apache.org`
[9] flume.apache.org
[10] `www.openstack.org`

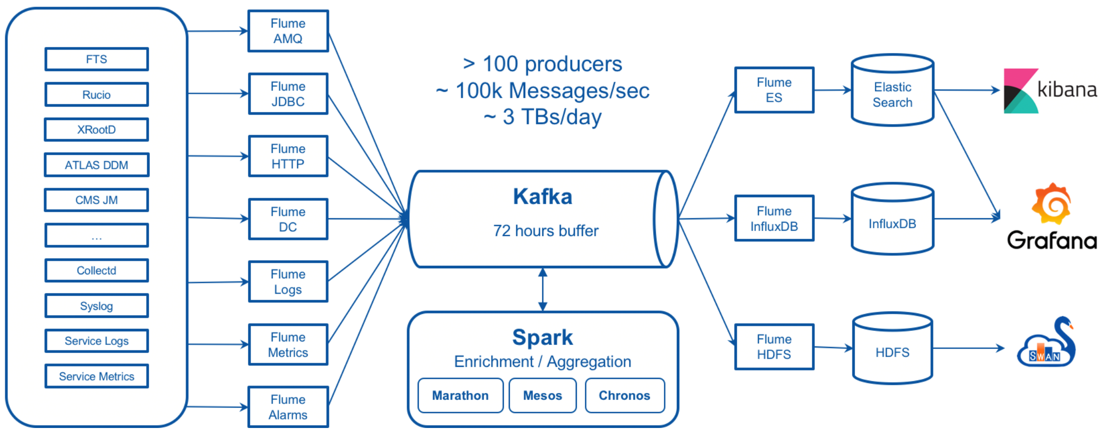


Figure 1.1: MONIT architecture

nent storage. Streaming jobs are used for data enrichment (e.g. adding site name and country to records based on producers' hostname) or aggregation (averaging multiple metrics over a time bin or combining multiple metrics from the same source into one record). Batch processing is used for compressing previously saved data, creating high-level reports or advanced offline data analytics.

Storage is divided based on the supported time window. For short term storage and log data exploration, Elasticsearch is used. Time series data is stored in InfluxDB for medium-term period in case of raw data and long-term for aggregated data based on the down-sampling policies. For long-term storage MONIT shares a HDFS cluster with other CERN IT projects.

Access is provided by multiple web applications. Grafana is used for visualizing mainly time-series data from both Elasticsearch and InfluxDB. Kibana is also used as an alternative interface for data stored in Elasticsearch. For offline and more advanced analytics, data cached in Kafka as well as those stored in all of the storage backends can be accessed by SWAN — a platform for interactive data analysis based on the Jupyter[11] technology [16] with access to the Spark cluster for heavier workloads.

MONIT is the most complex of all the systems described, mainly because the democratic nature of the scientific domain spawns many different solutions to the problems and MONIT has to accommodate all of them. Advanced data analytics are not built in, but users can leverage the opportunities the Spark cluster and SWAN provide as MONIT is more of a platform than a final solution.

1.3 Monitoring Data Analysis Works

Neither the out-of-the-box monitoring solutions, nor the modular ones provide a truly advanced data analytics tool. However many researchers have experimented with such a task usually creating an purpose built tool solely for the sake of the study. This section will provide a brief overview of some of those works focusing on machine learning applications.

1.3.1 Intrusion detection

The field of cyber security has seen a wide adoption of machine learning methods, mainly for the specific task of intrusion detection[12]. Although not entirely the same task as general monitoring anomaly detection, the similarities in the nature of its data make it a relevant comparison.

Surveys show that merely any applicable method has already been investigated [17], although others argue that identifying network intrusion is a particularly difficult problem for machine learning [18]. Recently the works have mainly been focusing on complex artificial neural network architectures [19], which thrive even in environments with extremely rare anomalies and difficult input data.

[11] `jupyter.org`

[12] An intrusion detection system discovers and identifies unauthorized use, alteration, or destruction of information systems

1.3.2 Failure prediction

Some works have taken on the task of failure prediction, a significant advancement of failure recognition. Failure prediction is the problem of assessing whether the current situation bears the risk of the system being unable to deliver its' expected service. This is a key step in designing services providing high availability. As with cyber security many approaches have been tried and tested [20], but given the advanced difficulty of the task all the works make use of all the available data including application metrics or logs [21] making them very application specific.

1.3.3 Black-box methods

Finally there is a number of works exploring data analysis on application oblivious metrics, which are arguably closest to the main subject of this work. These can be either OS metrics or metrics from a middleware on which the application itself is based as is the case of Pinpoint [22]. Pinpoint is an application-level failure detection system using both peer similarity and historical similarity to detect anomalies. It tracks the path requests travel through an application studying the interactions of components with the rest of the system. It is constrained to several middlewares used for building component based internet applications, however it is independent on the specific application. The work shows interesting results detecting high level failures with high speed, it also notes the challenges when working with application metrics.

Being constrained to system metrics only is a notable limitation, so works in this class usually focus on a single use-case. As is the case of Ganesha: Black-box fault diagnosis for MapReduce systems [23]. Ganesha uses peer similarity based on clustering to determine an erratic node. Data used for training were obtained in a experimental environment, however both the artificially injected faults and one of the three workloads were chosen to well represent a real-world scenario. The work showed that a black-box approach is viable when detecting faults with static and asymmetric manifestation[13].

1.4 Time series analysis

Because server monitoring data is a time series type dataset, other time series analysis works might also be of interest, although related less closely then works specifically analyzing computer performance. The recent trends have been to replace traditional methods with complex Neural Network models.

The Long Short Term Memory networks have been successfully used for anomaly detection on real world datasets [24], as a core model of a setup similar to the system described by this work. But even more elaborate models, leveraging convolutional layers as well as LSTMs have been explored. The MSCRED Framework (Multi-Scale Con-volutional Recurrent Encoder-Decode) introduces a complex system combining a convolutional encoder-decoder with LSTM networks [25]. That is then evaluated on a synthetic and real world dataset.

[13]Where the faulty node behaves differently from the others and the fault is not moving across the cluster

2. Problem statement and Approach

In this chapter first some essential definitions are provided, then the core hypotheses and goals as well as non-goals are stated. Let us just remind that the relevant results are discussed later in Chapter 5.

2.1 Definitions

This work is centered on detecting anomalous behavior of servers. However not any anomaly is interesting enough to be worth reporting. To better clarify the goals of this work, some of the basic terms are defined. These terms are widely used in works related to computer system reliability and dependability, but not always consistently. The definition used here is hopefully consistent with the prevailing understanding [26].

Service failure or failure for short is an event that occurs when the delivered service deviates from expectations. An alternative equivalent definition is that a failure is a misbehavior that can be observed by the user.

Error is a situation, where the systems' state deviates from the correct state. An error, that has not yet reached the services interface (has not yet caused a failure) can be detected by the system administrator via monitoring.

Fault is a root cause of an error.

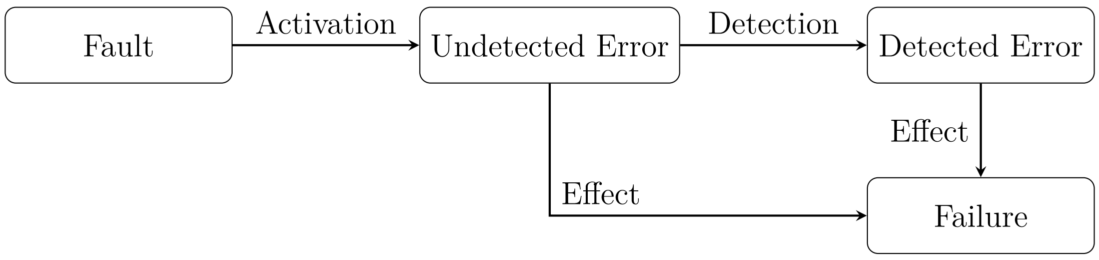


Figure 2.1: Relations between faults, errors and failures.

2.2 Hypotheses

The core hypotheses that this work is trying to prove states, that given a history of a distributed application workload and a cluster of servers running said application, it is possible to detect an erratic server by analyzing only the os-level metric data.

2.3 Goals

The goals of this work are to:

1. Gather a dataset. Create an application consuming metrics commonly available in computing centers and pre-processing them for usage as input for a machine learning module. Specifically the metrics will be handled on the CERN MONIT infrastructure, as CERNs' computing center is an example of a traditional data center.

2. Confirm core hypothesis. Train a machine learning model on the gathered dateset that is able to recognize an erratic node. The process has to be online (the model has to work on a live stream of data) so that the operator can take action before the error causes a failure. Investigate the universalness of such model.

2.4 Non-goals

While the error detection has to be done on live data, this does not mean that the anomaly has to be caught the moment it appears. The aim is to provide a useful tool for the system administrator of the application so that detectable errors are reported before they cause failures. Many applications have built in redundancy to prevent failures caused by a single node error, however combination of several minor errors might cause problems while being difficult to detect by ordinary oversight, making a automatic error detection tool beneficial for the operator.

This work is aimed on proving the hypothesis, among its' goals is not developing a wholesome production ready application.

3. Data Gathering and Aggregating

In this chapter, the process of acquiring the dataset is described. The dataset required for accomplishing this works goals has to have the following properties:

- represent recorded OS metrics from a cluster of equal servers running a load balanced distributed application for a time period long enough to enable learning of an advanced model,
- present the metrics in a way that they might be correlated for a particular server or subset of servers,
- capture numerous anomalies so the detection mechanism can be thoroughly tested.

By the time this work started, no such dataset was available. The decision was made to record our own.

3.1 Recording OS Metrics using MONIT

All of the conditions mentioned above could be fulfilled using the data in CERNs' MONIT infrastructure (described in Section 1.2.3). MONIT offers OS metrics via its collectd stream [7]. Each collectd daemon submits the metrics to the stream individually, not at the same time and in various intervals. Joining the metrics at any single point in time is therefore unfeasible. Instead each metric is averaged on its own on a common time window and the complete snapshot of the servers state is joined afterwards. See Appendix C for the data schemes of all the relevant kafka topics.

Initially 10 OS metrics were selected from the collectd data streams. These include cpu-idle and memory-used for monitoring the overall cpu and memory usage respectively; running-processes and shortterm-load for process status monitoring; connections-established, bytes-sent and bytes-received for network monitoring; cpu-iowait, disk-write, disk-read for I/O monitoring. Later, after the system was setup, the whole set of available OS metrics started being saved (see Table 3.1).

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. It is also still a reasonable delay for the anomaly detection this work is trying to achieve, since it is aiming for detecting an error before it causes a failure rather than real-time performance.

Two streaming Spark jobs implement the operations described (Figure 3.1). The first one reads all the collectd data in the MONIT infrastructure and averages only selected metrics for four clusters chosen because of their potential to be interesting for our case. The processed data are then written into new topics (one per metric type). Needing to read the whole collectd stream means the job

Name	Original	Note
bytes received	True	bytes received on the network interface
bytes sent	True	bytes sent on the network interface
connections established	True	established tcp connections
connections listen	False	local sockets in listening state
connections synrcv	False	tcp connections in a establishing state
connections synsent	False	tcp connections in a establishing state
connections closed	False	tcp connections in closed state
connections closewait	False	tcp connections to be closed
connections closing	False	tcp connections in a closing state
connections finwait1	False	tcp connections in a closing state
connections finwait2	False	tcp connections in a closing state
connections lastack	False	tcp connections in a closing state
connections timewait	False	tcp connections in a closing state
cpu idle	True	percentage of cpu idle
cpu interrupt	False	perc. cpu used for HW interrupts
cpu iowait	True	perc. cpu used for I/O operations
cpu nice	False	perc. cpu used by tasks with nice > 0
cpu softirq	False	perc. cpu used for SW interrupts
cpu steal	False	perc. cpu stolen by the hypervisor
cpu system	False	perc. cpu used by system tasks
cpu user	False	perc. cpu used by user tasks
disk read	True	bytes read from the main disk
disk write	True	bytes written from the main disk
load	True	short-term load
used memory	True	bytes of used memory
free memory	False	bytes of free memory
cached memory	False	bytes of memory used for I/O cache
slabrecl memory	False	bytes of SLAB[1] reclaimable memory
slabunrecl memory	False	bytes of SLAB un-reclaimable memory
paging processes	False	number of paging processes
running processes	True	number of running processes
blocked processes	False	number of blocked processes
sleeping processes	False	number of sleeping processes
stopped processes	False	number of stopped processes
zombie processes	False	number of zombie processes

Table 3.1: All the collected collectd metrics. The second column notes, whether the metric was used in the dataset before the metric number expansion.
[1]SLAB is a memory management mechanism for increasing efficiency of memory allocation and reuse by the Linux kernel

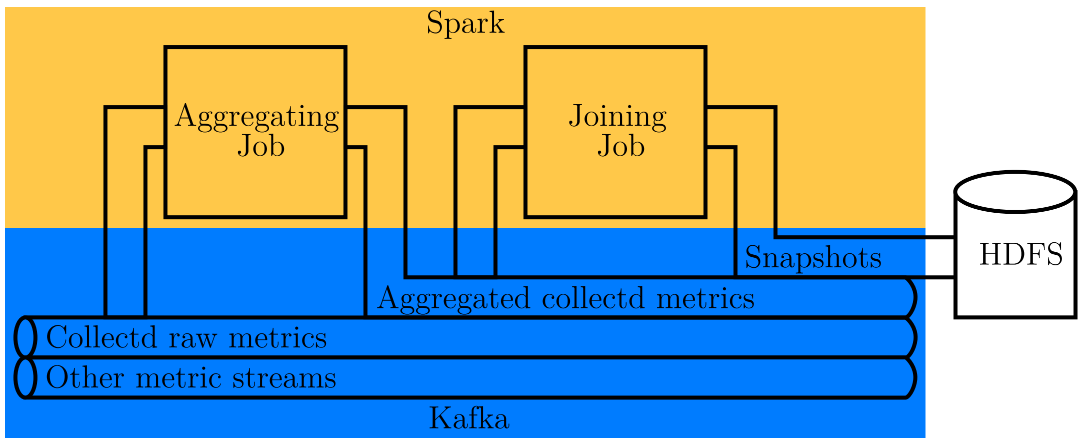


Figure 3.1: Scheme of the data collecting pipeline inside the MONIT infrastructure. Note that only parts relevant for this work are depicted. Connection between Kafka (blue) and Spark (orange) is handled by Sparks' Kafka library. Filling Kafka and writting into HDFS is done by Flume agents.

has to run in the Spark cluster mode rather than on a single server: the resource requirements for fetching, decompressing and reading 13MB/s are 25 CPU cores and over 111GB of memory. Since the whole feed is already being consumed, adding new metrics or clusters to the aggregated stream is now only a matter of re-configuring the job, the resource requirements would not grow significantly.

The second Spark job reads the processed data and joins them on hostname and data aggregation window creating a full status snapshot of the host for the specific time. It then again writes them into a separate topic. The joined data is then written by a Flume agent to HDFS for later analyses.

3.2 Creating anomalies

Gathering data from a live setup would not be sufficient for testing an anomaly detection process. Anomalies are not common enough in ordinary data, therefore artificial anomalies were crafted and injected into the data. 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 only 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.
- Memory-leak type — the memory usage gradually increases up until a point where the OOM killer[1] engages. 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]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.
- 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).

All of the anomalies were injected one or two per day on one of the nodes. The initial goal was to collect 10 instances of each anomaly. However because the available cluster was dedicated to testing configuration before deploying in production, time sequences, when the application was running ordinarily were sparse. Moreover creating anomalies caused the service to fail (although that was not the aim at the time), so several time periods have to be discarded from the usable dataset. Table 3.2 presents the final number of collected anomalies per type. Anomalies are spread over 19 days and the dataset also includes five 3 day sequences of stable production for algorithm training.

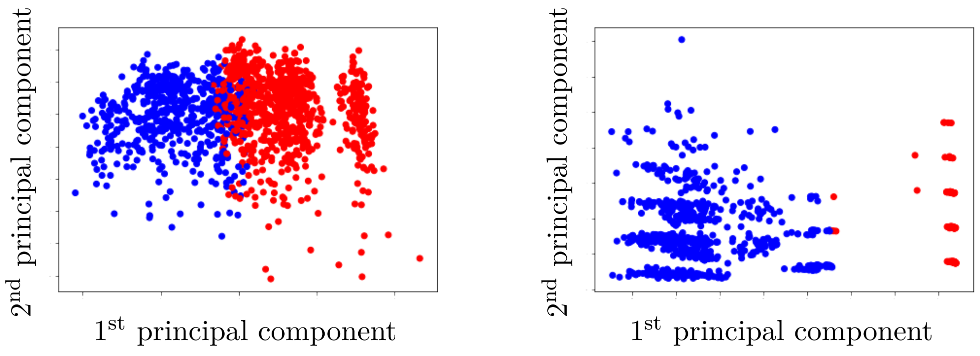
anomaly type	count
service stop	6
memory overuse	2
cpu overuse	8
combined cpu and memory overuse	10

Table 3.2: Final anomaly count per type.

3.3 Initial Data Exploration

To assess whether any knowledge mining could be successful, data from one of the anomaly days were selected and several clustering algorithms were applied. Because all the data described servers running one distributed application, the opportunity of using a simple clustering algorithm for anomaly detection had to be explored. The clustering algorithms output however had to be interpreted differently than when used for clustering itself. For the k-means [27] algorithm, the distance of the examined data point from the center of the cluster it was assigned to had to be measured. If the distance exceeds a limit, the point is marked as an anomaly. The mixture of gaussians [28] algorithm enriches the clustering classification with a assignment probability of the data point for each cluster.

The number of clusters was set to two, however, we were expecting all the data points should form only one cluster with several outliers possibly scattered around. In practice, many miss-classified anomalies were observed when applying this approach to a day with stable service (see Figure 3.2a). Conversely, on a day where one of the nodes actually displayed some erratic behavior, its' data formed a separate cluster. Therefore it could not be identified using the approaches described above (see Figure 3.2b). The results however hinted, that there is some knowledge available for extraction hidden in the data.



(a) Stable day (b) Day with an erratic node

Figure 3.2: Result of a run of the K-Means clustering algorithm on PCA transformed data. Colors correspond to the clustering classification.

These initial results were presented[2] on the Computing in High Energy Physics 2018 conference in track Networks and facilities and were published in the conference proceedings [29].

kk

[2]Contribution details: `indico.cern.ch/event/587955/contributions/2937940/`

4. Anomaly Detection Module

After the dataset has been described, this chapter will introduce a variety of approaches for detecting anomalies and the next chapter will evaluate their performance. Keeping the work as universal as possible we decided not to use supervised classification on labeled anomalies, which might constraint the performance to our artificial errors. Instead a selection of unsupervised outlier detection algorithms and supervised regressors was explored. Note that none of the core algorithms was implemented in this work, all of the models, classifiers and regressors are available in their respective libraries. This also divides the models in two main categories:

- algorithms available in the Spark.ML library [30],
- algorithms available in other Python libraries.

Usage of the framework in which the data collection module is written and which the MONIT infrastructure supports well would be a great convenience. However Sparks' in-house machine learning capabilities are somewhat limited, so other models, which were not natively provided, were investigated as well.

4.1 Data Pre-processing

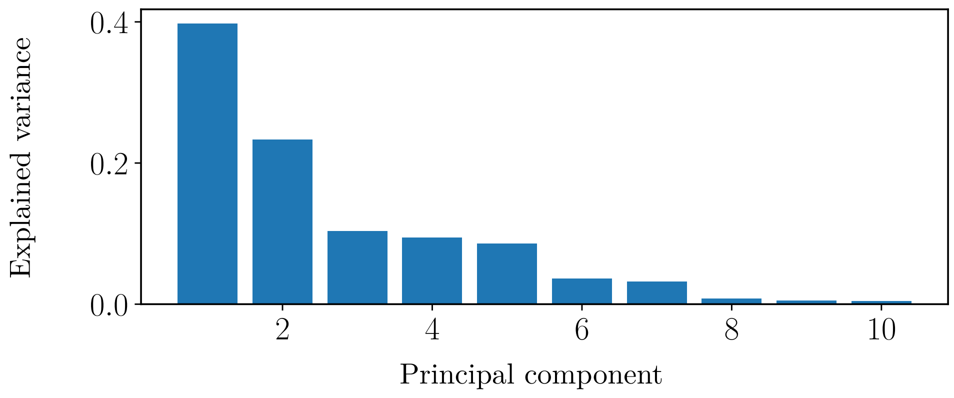
The raw captured data consist of 10 and later 35 highly varied metrics. To boost the algorithms performance, all the input data are scaled to unit variation. However 10 and especially 35 dimensions could be too much for some of the algorithms, so the Principle Component Analysis was employed [31]. Many components appeared to be highly correlated as expected[1]. The performance gain with certain algorithms was clear, although performed without the knowledge of the target variable (flagged anomalies) since that would not be available in production settings.

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 (see Figure 4.1).

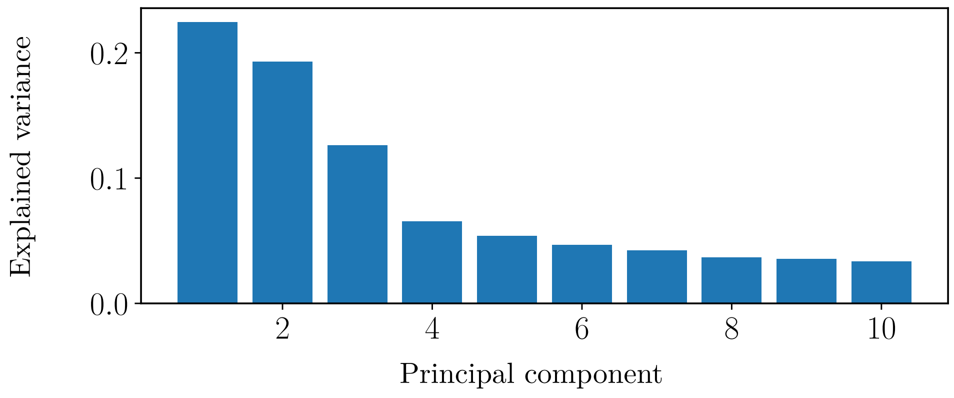
4.2 Unsupervised Learning

Anomaly detection can be approached with unsupervised learning as a problem of characterizing a dataset based solely on the regular data points. New data are then determined to be anomalies based on how well they would fit the learnt characterization. The input data points in our case would be simply the aggregated metrics per time window per hostname. When learnt on a certain time period, the algorithm will then mark the abnormal data points, thus indicating

[1]It was clear that variables such as `running-processes` and `shortterm-load` will be correlated since they describe a very similar property only in different ways. The decision was made to collect as many metrics as possible and let pre-processing algorithms such as PCA reduce the dimensionality rather than manually pre-select a set of metrics based on personal experience.



(a) Decomposition for the 10 raw metrics.



(b) Decomposition for the 35 raw metrics.

Figure 4.1: Proportions of variance explained by each principal component.

the anomalous server and time the anomaly might have occurred. Because the Spark.ML library does not offer any suitable unsupervised learning algorithms, two algorithms from the popular Scikit-learn library [32].

The **One Class Support Vector Machine** [33] is a novelty detection algorithm based on the ubiquitous SVM classifier. The one class variant, instead of characterizing a hyper-plane separating two classes, learns a frontier surrounding the ordinary data. New data are then labeled based on their location relative to that frontier.

The **Isolation Forest** [34] is an attempt to use the random forest approach for outlier detection. The algorithm is based on partitioning the space until it isolates all the input data points. Anomalies are then those samples, which are the easiest to isolate — require the least partitions to separate from the rest of the data points.

4.3 Supervised Regression

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[2]. 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 a 80 minute history of the server in question.

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

The number of collected metrics N was initially 10 and later grew to 35, which means the input vector representing 80 minutes history would be 40 and 140 items respectively. This might pose a challenge not only for the algorithms themselves, as the size of the learning set might not be big enough to reliably train with such large number of input variables, but also the interpretation of so many results would be complicated. It was therefore decided to use PCA for dimensionality reduction. Information about the rest of the cluster is added to each snapshot by joining it with per metric cluster averages, thus doubling the input vector length.

After the algorithm makes its' prediction for the next state and the true state comes in, we can consider the distance of the prediction and the truth. Extracting anomalies from prediction errors is yet another task. When testing the algorithms on a set of data from one day, we can calculate the mean and standard deviation of the errors and every prediction with error larger than the mean error plus three standard deviations can be considered a predicted anomaly.

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

4.3.1 Non-Spark Model

In classical time series analysis, primarily in economics, the ARIMAX[3] class of models is a very popular one. Extending the basic combination of an auto regressive and a moving average model (ARMA) by adding a differencing step to handle possible non-stationarity and an ability to consult other variables than the predicted one makes it an interesting possible fit for our needs.

For working with a large number of varied time series automatic selection of hyperparameters for the model is a major challenge. An implementation of a informed grid search of the parameter space is luckily provided by the `pmdarima` library [35], however all the analytics using this class of models had to be done outside the Spark framework.

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

[3]AutoRegressive Integrated Moving Average with eXogeneous input

4.3.2 Spark Models

The following models will all be implemented in Spark.ML making their application on the acquired dataset and possible future conversion to a production application uncomplicated.

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 linear and the load balancing of the investigated application works rather well, this approach yielded good base results.

The Spark machine learning library offers a decent selection of models, from which we chose 2 very different ones. 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 [36], which could model more elaborate functions. 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.

5. Model Evaluation

In this chapter, we explain the evaluation of the algorithms introduced in the previous chapter. The comparison needs to be done among diverse approaches, so although the standard evaluation metrics will be mentioned when assessing their respective types of algorithms, the paramount question is, whether the resulting system is able to reliably detect the artificial anomalies without many false positives. We will be using the standard precision and recall metrics, defined as follows.

predicted	actual	
	anomaly	normal
anomaly	true positive	false positive
normal	false negative	true negative

$$\text{precision} = \frac{\text{true positives}}{\text{positives}}$$

$$\text{recall} = \frac{\text{true positives}}{\text{anomaly count}}$$

5.1 Unsupervised Learning

With the novelty and outlier detection algorithms the process will be simple. Their output is directly a anomaly classification, so the sole procedure is to compare the output with the known anomalies. The hyperparameters for both algorithms were chosen using 5-fold cross validation on one random day with an anomaly and then fixed for the rest of the anomaly days.

From Table 5.1 it is clear, that the Isolation Forest algorithm yields better results than the One Class SVN. PCA pre-processing helps the OC-SVN, which is

model	precision	recall
I-forest-nopca	0.79	0.37
I-forest-pca5	0.75	0.32
I-forest-pca3	0.72	0.32
OC-SVN-nopca	0.29	0.16
OC-SVN-pca5	0.43	0.19
OC-SVN-pca3	0.40	0.21

Table 5.1: Mean precision and recall for the unsupervised algorithms. Suffix `-pca3` means the algorithms' input data were pre-processed by PCA and the first three components were used, the `-pca5` suffix denotes an algorithm trained on the first 5 components of the PCA decomposition and the `-nopca` suffix denotes an algorithm trained and applied on data without any PCA pre-processing.

more sensitive to high dimensionality, but it is not helpful in case of the Isolation Forest. When inspecting the performance on specific instances of anomalies, the performance varies dramatically (see Table 5.2 and Table 5.3 for shortened results or Appendix B for full results). Note that the days when the algorithms are performing the worst are included in the portion of the data used to set the hyper parameters. Performance variations are also not linked to the anomaly types.

date	precision	recall
2019/11/27	1 0	0.67
2019/11/30	1 0	0.58
2019/02/11	0.25	0.17
2019/02/09	0.22	0.17

Table 5.2: The two best and worst days with respect to recall with anomalies on which the Isolation Forest algorithm without PCA pre-processing was applied

date	precision	recall
2019/02/11	0.50	0.38
2019/02/19	0.57	0.31
2019/12/18	0.43	0.12
2019/02/22	0.33	0 8

Table 5.3: The two best and worst days with respect to recall with anomalies on which the OC-SVN algorithm on the first five components of PCA.

5.2 Supervised learning

As discussed in the previous chapter, the regression algorithms output the next server state based on historic ones (see Section 4.3). One of the standard regression metrics is the root mean square error (RMSE), see Table 5.4 for results on data with PCA reduced dimensions. These results suggest, that the BASELINE model would perform the best of all inspected.

In case of using the regression model for anomaly detection, the overall regression metric score arguably is not the main concern, the anomalousness of the

model	error0	error1	error2
BASELINE	0.53	0.42	0.63
ARIMA	5.40	5.30	1.90
lr	10.9	4.13	9.1
rf	8.86	16 6	12.28

Table 5.4: Mean errors per output column for the regression algorithms. *LR* represents Linear regression and *RF* the Random Forest regressor.

error is. In other words the error overall can be high while the error at the time of the anomaly is noticeably higher. What is going to be in the center of our interest therefore is the progression of the error. That can however differ greatly with the type of anomaly, with the algorithm used and with the current conditions of the cluster.

First, lets inspect the BASELINE algorithm. Its' errors are quite low, but then spike high for severe anomalies such as stopping the service (see Figure 5.1), but can also be rather unstable on other days (see Figure 5.2). The instability can be seen in other algorithms as well depending on the specific time period. Remember that the data are gathered from a live installation and apart from the artificial anomalies, no other changes were made including 12 to smoothen the raw data. Also note that both the beginning and the end of the anomaly are denoted by a spike in the error. This can be explained by the applications' attempts to reestablish stable operations by re-synchronizing the anomalous server with the rest of the cluster once the anomaly ends. This manifests itself by higher cpu and especially network usage. The regressor however has no information about the anomaly itself nor about the return to normal operations. The spike at the anomaly end is therefore expected. That is especially true for the BASELINE algorithm, since it has on information about the rest of the cluster, but we can observe similar behavior from the other models as well.

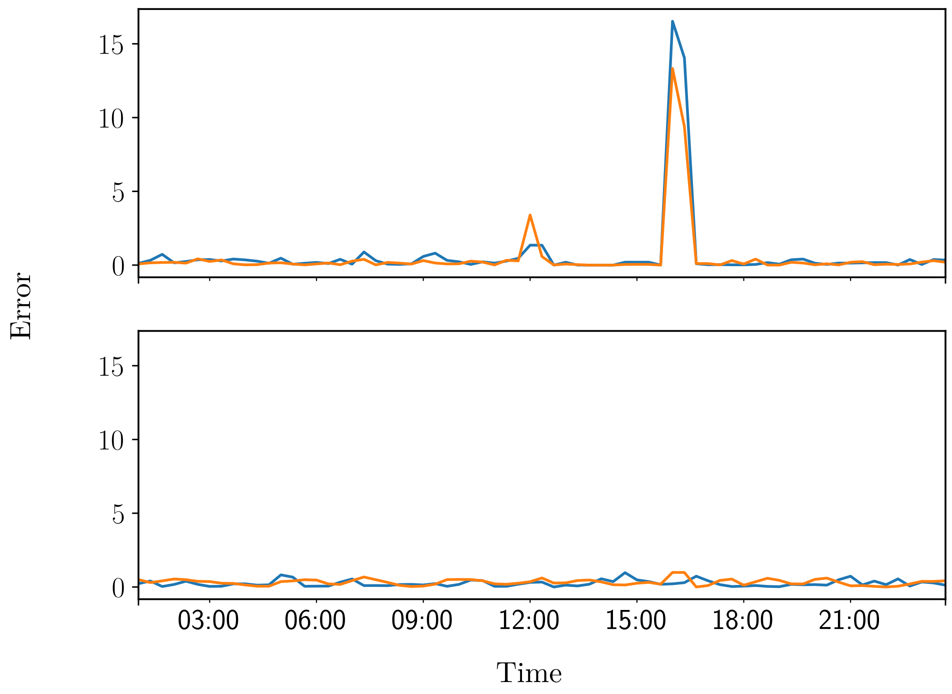


Figure 5.1: Prediction error in the first and second dimension with PCA of the BASELINE algorithm during a day with a service stop anomaly. Top chart is for the anomalous host, bottom for a random one. The anomaly started at 12:00 and ended at 16:00

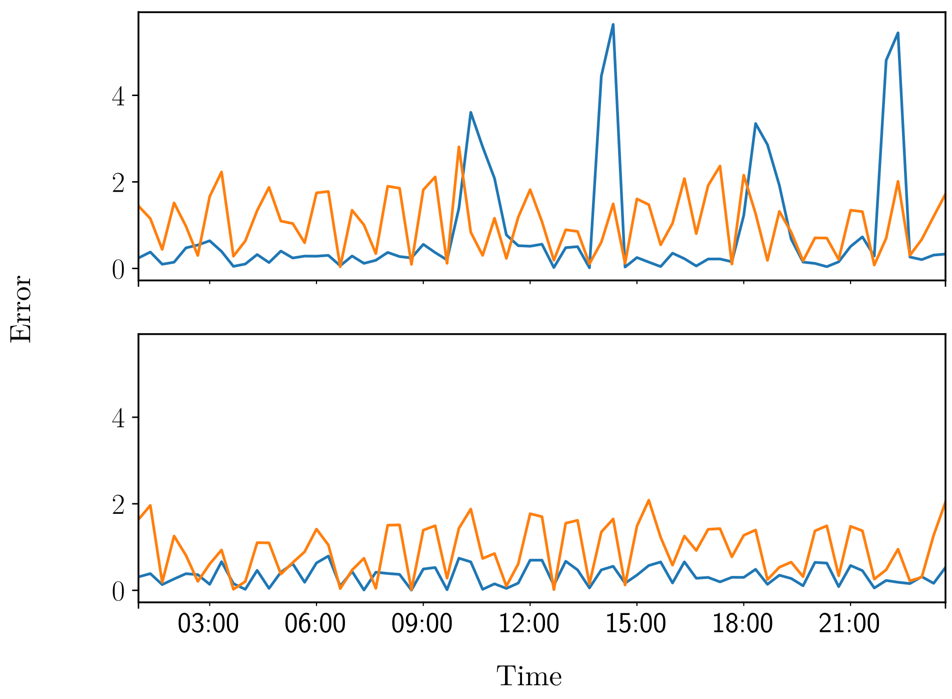


Figure 5.2: Prediction error in the first and second dimension with PCA of the BASELINE algorithm during a day with a cpu overuse anomaly. Top chart is for the anomalous host, bottom for a random one. There were two anomalies inserted that day: 10:00-14:00 and 18:00-22:00.

Similar behavior is captured by Figure 5.3, which depicts the results for the Random Forest algorithm. The start of the anomaly can again be spotted as a clear bump, but even more distinct is the spike corresponding with the end of the anomaly.

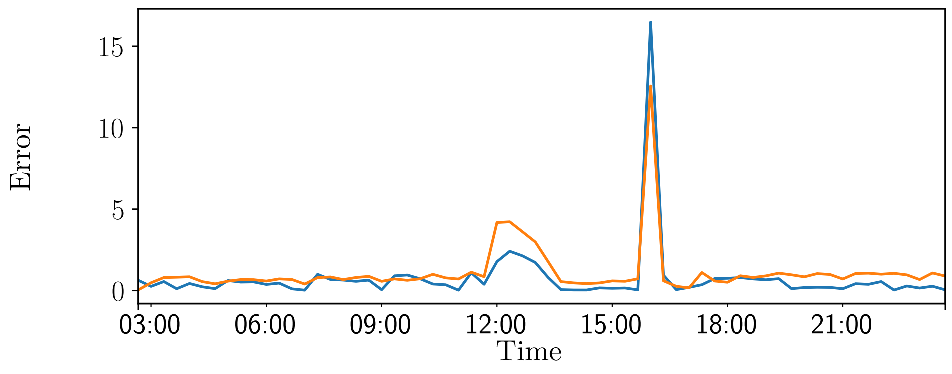


Figure 5.3: Prediction error in the first and second dimension with PCA of the Random Forest algorithm during a day with a service stop anomaly. The anomaly started at 12:00 and ended at 16:00.

5.2.1 Extracting Anomaly Information from Prediction Error

Applying the regression model is only the first step in the anomaly detection engine. Next is to extract the anomalies from the prediction errors. If we assume the errors to be normally distributed, we can then calculate the mean and standard deviation of the prediction error and all the predictions with error further from the mean than $3 * std\_dev$ can be treated as an anomaly prediction.

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

After applying this three-sigma rule on the data shown in Figure 5.3, specifically to error in the first dimension, we can see that both the beginning and the end are marked as an anomaly (see Table 5.5). Also some other servers statuses from the same time as the anomaly end are marked as well. Note that the standard classification metrics do not score this result particularly well with a precision of 0.71 and recall of 0.38.

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 5.5: 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.

For the anomaly system, dealing with the false positives, although they are understandable, poses a critical challenge. In order to be of any value to the system administrator, it needs to separate the actual anomaly, especially its' start, from its' side effects. False positive notifications have the potential to clutter the operators' message feed and make the whole system more of a burden than an asset. For completeness sake Table 5.6 presents the average recall and precision per model and error column on the PCA pre-processed data.

5.2.2 Filtering Anomaly Notification feed

To improve the unsatisfactory results, a simple filtering mechanism was conceived, mainly with the aim of filtering false positives while maintaining the recall. Table 5.5 would suggest, that if there indeed 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. Unfortunately recall also dropped. To counter act that, we decided to

model	error0	error1	error2	sum error	avg error
BASELINE	0.14	0.21	0.11	0.18	0.18
ARIMA	0.14	0.38	0.18	0.25	0.25
lr	0.14	0.24	0.15	0.20	0.20
rf	0.28	0.29	0.16	0.38	0.38

model	error0	error1	error2	sum error	avg error
BASELINE	0.28	0.63	0.29	0.53	0.53
ARIMA	0.12	0.33	0.21	0.27	0.27
lr	0.43	0.71	0.54	0.71	0.71
rf	0.37	0.72	0.48	0.81	0.81

Table 5.6: Recall (top) and precision (bottom) for the regression models. Anomaly classifications are received by a prediction error being too high, thus each table column describes classification retrieved by using predictions for separate input dimensions. Then a sum and average are computed per data sample and the same anomaly classification mechanism is deployed. The last two columns describe the recall received from such aggregated errors.

lower the $3 * std\_dev$ threshold for an error to become an anomaly to only a single standard deviation tolerance. With that, we still received no false positives with a significant improvement of the recall. The whole process of producing anomaly notification is described in high level in Algorithm 1.

Algorithm 1 Create Anomaly Notifications Using a Regression Model on Offline Data

```
  anomaly_free_data ⟵ read_from_database()
  model ⟵ train_model(anomaly_free_data)
  input_data ⟵ read_from_database()
  processed_data ⟵ model.apply(input_data)
  prediction_errors ⟵ calculate_prediction_errors(processed_data)
  anomaly_candidates ⟵ filter(filtering_rule, prediction_errors)
  anomaly_notifications ⟵ find_consecutive_candidates(anomaly_candidates)
  return anomaly_notifications
```

Table 5.7 illustrates the performance of the algorithms with the notification filtering heuristic in place. To enrich the results, the values in the table are the number of time windows of lag the algorithm needed to detect the anomaly (0 means detected directly at start, NaN represents an anomaly without detection). Since all the anomalies are 4 hours or 12 time windows long, value of 12 in the table means the algorithm detected only the end of the anomaly, which essentially means a failure. The same notification filtering heuristic can be also applied to the unsupervised algorithms input (see Table B.7 in Appendix B), however the results are inferior to those of the supervised learning algorithms.

anomaly id	lr	rf	BASELINE	ARIMA	rf-nopca
0	0	0	12	0	0
1	0	0	12	0	0
2	0	0	12	0	0
3	0	NaN	12	4	0
4	12	NaN	12	0	0
5	12	NaN	12	7	0
6	0	NaN	12	4	0
7	1	NaN	0	NaN	NaN
8	2	NaN	NaN	10	9
9	NaN	NaN	NaN	NaN	NaN
10	NaN	0	NaN	NaN	0
11	NaN	3	NaN	NaN	0
12	12	3	1	NaN	NaN
13	NaN	2	0	NaN	NaN
14	NaN	NaN	NaN	NaN	NaN
15	NaN	0	NaN	NaN	NaN
16	NaN	3	NaN	NaN	0
17	0	0	0	NaN	6
18	0	0	0	8	7
19	0	0	0	NaN	7
20	0	0	0	NaN	6
21	0	0	0	6	7
22	0	0	0	6	6
23	0	0	0	5	6
24	0	0	0	4	7
25	12	0	0	2	6
26	12	1	0	3	6

Table 5.7: Anomaly detection with the notifications filtering heuristic. Values are the lag of time windows the algorithm needed to detect the anomaly, NaN is a non-detection.

Note that the success rate is not even among the anomalies. Explanation can be found in the anomaly type:

- ids 0-2 are service stop anomalies in a very stable service period,
- ids 3-6 are service stop anomalies with a virtualization host problem affecting the service stability,
- ids 7,8 are memory overuse anomalies,
- ids 9-16 are cpu overuse anomalies,
- ids 17-26 are combined cpu and memory overuse anomalies.

With a recall of 0.7 and precision of 1, the Random Forest based system gives the best results of all the tested models with input composed of the first three principal components of the PCA. Those results however can be improved still, be leaving out the PCA based dimensionality reduction.

As was the case with the unsupervised algorithms, some regression algorithms do not perform best with a high number of input dimensions. The Random Forest regressor model, similarly to the Isolation Forest model, usually does not suffer by the curse of high dimensionality. In our case however, not reducing the input dimensionality implies also many more dimensions to predict to obtain prediction error. Moreover predicting many dimensions spawned the need for many regression models, because regression algorithms tend to have only one dimensional output. This enforced compromising on the training procedure. Specifically, unlike in the case of the three dimensional space after applying PCA, the hyperparameters for the regressors could not be selected by cross-validation each time a model is trained on new data, as the computation power required by such task would be excessive. Instead the hyperparameters were fixed after running cross-validation once on a single set of training data[1].

The improvement in the recall can best be examined by comparing to the previously best algorithm (see Table 5.7), precision was again 1 – no false positives. Note the higher score for anomalies 17-26, the combined over-use anomalies. As was the case for the rest of the over-use anomalies, the over-usage was scaled gradually, hence the late detection.

model	recall
rf-nopca	0.78
rf	0.70
lr	0.56
ARIMA	0.56
BASELINE	0.48

Table 5.8: Summary results for the whole system with notification filtering and $1 * \mathit{std\_dev}$ as the anomaly classification prediction error threshold. Notifying on the anomaly at the time of its' end is not counted here as valid notification.

When evaluating the error per dimension, it is clear that prediction of some of the metrics brought little benefit in comparison to others (see Table 5.9 for full results.). Those results however may vary with the specific application and artificial anomalies.

In general, anomalies affecting only one of the metrics are more challenging than the more complex ones. Summarizing the performance of the whole system with the notification filtering heuristic is Table 5.8.

[1]Hyperparameters were set to maxBins = 50, maxDepth = 3, numTrees=333

model	rf-nopca precision	rf-nopca recall
cpu idle	1.00	0.85
running processes	1.00	0.73
load	0.99	0.68
cpu system	0.64	0.46
cached memory	0.63	0.44
avg error	0.64	0.39
sum error	0.64	0.39
cpu softirq	0.48	0.27
connections lastack	0.28	0.27
connections listen	0.28	0.27
connections finwait2	0.34	0.26
bytes received	0.31	0.21
connections established	0.24	0.18
blocked processes	0.28	0.14
cpu user	0.32	0.13
sleeping processes	0.36	0.13
slabrecl memory	0.24	0.12
slabunrecl memory	0.22	0.11
connections closed	0.44	0.09
bytes sent	0.09	0.07
connections closing	0.37	0.06
zombie processes	0.13	0.03
disk read	0.05	0.02
free memory	0.02	0.01
connections timewait	0.07	0.01
connections synrcv	0.00	0.01
connections closewait	0.01	0.00
connections finwait1	0.00	0.00
disk write	0.01	0.00
connections synsent	0.00	0.00
paging processes	0.00	0.00
cpu steal	0.00	0.00
stopped processes	0.00	0.00
used memory	0.00	0.00
cpu iowait	0.00	0.00
cpu interrupt	0.00	0.00
cpu nice	0.00	0.00

Table 5.9: Precision and recall per input column for the Random Forest regressor without PCA preprocessing. Anomaly classifications are received by a prediction error being too high, thus each table row describes classification retrieved by using predictions for separate input dimensions. Then a sum and average are computed per data sample and the same anomaly classification mechanism is deployed.

Conclusion

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.

A set of two unsupervised novelty and outlier detection algorithms was selected along with a set of three various supervised regression algorithms. The unsupervised algorithms directly flagged the anomalies, while the regression algorithms predicted the future system state based on its' short history and the aggregated status of the rest of the cluster. The predictions were then compared to the actual state and the error was used to make the anomaly detection. The algorithms were trained and their respective performance was evaluated both by standard metrics, and by the usability for the final user. To improve the later a simple heuristic was thought of, eradicating the false positives while keeping a high recall rate.

The final system with a Random Forest regressor, PCA dimensionality reduction and anomaly notification filtering heuristic yielded a recall of 0.7 with a 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.

The initial results were presented[2] on the Computing in High Energy Physics 2018 (CHEP2018) conference in track Networks and facilities and were published in the conference proceedings [29]. On CHEP2019 an update was given[3] and the full results are to be published in the conference proceedings.

Future Work

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.

[2]Contribution details: `indico.cern.ch/event/587955/contributions/2937940/`
[3]Contribution details: `indico.cern.ch/event/773049/contributions/3473855/`

Bibliography

[1] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, 2011.

[2] Martin Hilbert. *Information Quantity*, pages 1–4. Springer International Publishing, Cham, 2017.

[3] M. Mitchell Waldrop. The chips are down for moore's law. *Nature*, 530(7589):144–147, 2016-2-9.

[4] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[5] Patricia Jung. Munin-the raven reports. *Linux J.*, 2009(180), April 2009.

[6] Nagios, the industry standard in it infrastructure monitoring. `www.nagios.org/projects/nagios-core/`. [Online, Accessed: 2019-03-06].

[7] Collectd, the system statistics collection daemon. `www.collectd.org`. [Online, Accessed: 2019-03-06].

[8] Influxdb, open source time series database. `www.influxdata.com/products/influxdb-overview/`. [Online, Accessed: 2019-05-29].

[9] Grafana. `www.grafana.com/grafana`. [Online, Accessed: 2019-05-30].

[10] Elk stack: Elasticsearch, logstash, kibana. `www.elastic.co/elk-stack`. [Online, Accessed: 2019-06-02].

[11] Clinton Gormley and Zachary Tong. *Elasticsearch: The definitive guide: A distributed real-time search and analytics engine.* " O'Reilly Media, Inc.", 2015.

[12] Ian Bird. Computing for the large hadron collider. *Annual Review of Nuclear and Particle Science*, 61(1):99–118, 2011.

[13] A Aimar, A Aguado Corman, P Andrade, S Belov, J Delgado Fernandez, B Garrido Bear, M Georgiou, E Karavakis, L Magnoni, R Rama Ballesteros, H Riahi, J Rodriguez Martinez, P Saiz, and D Zolnai. Unified monitoring architecture for it and grid services. *Journal of Physics: Conference Series*, 898(9):092033, 2017.

[14] Nishant Garg. *Apache Kafka.* Packt Publishing, 2013. ISBN: 978-1782167938.

[15] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[16] Danilo Piparo, Enric Tejedor, Pere Mato, Luca Mascetti, Jakub Moscicki, and Massimo Lamanna. Swan: A service for interactive analysis in the cloud. *Future Generation Computer Systems*, 78:1071 – 1078, 2018.

[17] Anna L Buczak and Erhan Guven. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications Surveys & Tutorials*, 18(2):1153–1176, 2015.

[18] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE symposium on security and privacy*, pages 305–316. IEEE, 2010.

[19] Kalyan Veeramachaneni, Ignacio Arnaldo, Vamsi Korrapati, Constantinos Bassias, and Ke Li. Aiˆ 2: training a big data machine to defend. In *2016 IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS)*, pages 49–54. IEEE, 2016.

[20] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42, 03 2010.

[21] Xiaohui Gu and Haixun Wang. Online anomaly prediction for robust cluster systems. pages 1000 – 1011, 05 2009.

[22] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks*, 16(5):1027–1041, Sep. 2005.

[23] Xinghao Pan, Jiaqi Tan, Soila Kavulya, Rajeev G, and Priya Narasimhan. Ganesha: Black-box fault diagnosis for mapreduce systems. 01 2008.

[24] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. Long short term memory networks for anomaly detection in time series. In *Proceedings*, page 89. Presses universitaires de Louvain, 2015.

[25] Chuxu Zhang, Dongjin Song, Yuncong Chen, Xinyang Feng, Cristian Lumezanu, Wei Cheng, Jingchao Ni, Bo Zong, Haifeng Chen, and Nitesh V Chawla. A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1409–1416, 2019.

[26] Jean claude Laprie and Brian Randell. Fundamental concepts of computer systems dependability. In *Proc. of the Workshop on Robot Dep. , Seoul, Korea*, pages 21–22, 2001.

[27] Stuart P. Lloyd. Least squares quantization in pcm's. *IEEE Transactions on Information Theory*, 28:129–136, 03 1982.

[28] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Section 16.1. gaussian mixture models and k-means clustering. In *Numerical recipes*, pages –. Cambridge University Press, Cambridge, 1st ed. edition, c1996.

[29] Martin Adam, Luca Magnoni, Martin Pilát, and Dagmar Adamová. Detection of erratic behavior in load balanced clusters of servers using a machine learning based method. In *EPJ Web of Conferences*, volume 214, page 08030. EDP Sciences, 2019.

[30] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, January 2016.

[31] Hervé Abdi and Lynne J. Williams. Principal component analysis. *WIREs Comput. Stat.*, 2(4):433–459, July 2010.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[33] Bernhard Schölkopf, Robert C Williamson, Alex J Smola, John Shawe-Taylor, and John C Platt. Support vector method for novelty detection. In *Advances in neural information processing systems*, pages 582–588, 2000.

[34] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. IEEE, 2008.

[35] Taylor G. Smith et al. pmdarima: Arima estimators for Python. `www.alkaline-ml.com/pmdarima`. [Online; accessed 29/12/2019].

[36] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

List of Figures

List of Tables

List of Abbreviations

VM – Virtual Machine. A software program that exhibits the behavior of a separate computer (running an operating system, applications and programs like a separate computer).

JSON – JavaScript Object Notation, a standardized file format

OS – Operating System

QA – Quality Assurance

CERN European Organization for Nuclear Research (name derived from Conseil Européen pour la Recherche Nucléaire) – European research organization that operates the largest particle physics laboratory in the world.

LHC Large Hadron Collider – the world's largest and most powerful particle collider built by CERN

WLCG The Worldwide LHC Computing Grid – global collaboration of more than 170 computing centers in 42 countries, linking up national and international grid infrastructures

PCA Principal Component Analysis algorithm [31], used in this work for dimensionality reduction.

RF Random Forest, used in this work to denote the Random Forest based regression algorithm

LR Linear Regression

A. Attachments

This appendix describes the contents of the attached archive.

A.1 Data Collection Spark Jobs

The `/Data-Acquisition/` directory contains two directories, each with a whole build-able project of one of the Spark Streaming jobs collecting and aggregating the data for the dataset. Each directory contains a `README.md` file with basic description and instruction on how to build the project, however the jobs are purposely built to fit the MONIT infrastructure, so no further run or deployment manual is included.

The `/Data-Acquisition/spark-collectd-agg-job` contains code for the first job, which reads all the collectd data from the MONIT infrastructure and aggregates it.

The `/Data-Acquisition/spark-collectd-join-job` contains code for the second job, which reads the first jobs output and creates server status snapshots.

A.2 Data Analysis

The `/Data-Analysis/` directory contains Jupyter[1] notebooks used to analyze the data collected by the streaming jobs and stored in HDFS. In the subdirectory `/Data-Analysis/results/`, some intermediate results in csv format are stored. Please consult the attached `/Data-Analysis/README.txt` file for further instructions.

A.3 Raw Dataset

The `/Dataset/` directory contains the raw data downloaded directly from HDFS. The sub-directory structure partitions the data by date it was taken. The data itself is stored in compressed `json` format to be read directly by a Spark application.

[1]`https://jupyter.org/`

B. Unsupervised Algorithms Results

This attachment provides the unabbreviated results of the unsupervised algorithms.

date	true positives	false positives	precision	recall
2019/02/09	2	7	0.22	0.17
2019/02/10	5	4	0.56	0.42
2019/02/11	2	6	0.25	0.17
2019/02/19	6	1	0.86	0.50
2019/02/20	5	1	0.83	0.42
2019/02/21	6	1	0.86	0.50
2019/02/22	6	1	0.86	0.50
2019/06/05	5	3	0.62	0.42
2019/06/06	4	3	0.57	0.33
2019/11/27	8	0	1.00	0.67
2019/11/28	8	0	1.00	0.33
2019/11/29	8	0	1.00	0.33
2019/11/30	7	0	1.00	0.58
2019/12/01	8	0	1.00	0.33
2019/12/16	5	2	0.71	0.21
2019/12/17	7	1	0.88	0.29
2019/12/18	8	0	1.00	0.33
2019/12/19	7	1	0.88	0.29
2019/12/20	8	0	1.00	0.33

Table B.1: Full results for the I-forest-nopca model.

date	true positives	false positives	precision	recall
2019/02/09	1	8	0.11	0.08
2019/02/10	1	8	0.11	0.08
2019/02/11	2	6	0.25	0.15
2019/02/19	4	3	0.57	0.31
2019/02/20	5	2	0.71	0.38
2019/02/21	5	2	0.71	0.38
2019/02/22	5	2	0.71	0.38
2019/06/05	7	1	0.88	0.54
2019/06/06	7	0	1.00	0.54
2019/11/27	8	0	1.00	0.62
2019/11/28	8	0	1.00	0.31
2019/11/29	7	1	0.88	0.27
2019/11/30	7	0	1.00	0.54
2019/12/01	8	0	1.00	0.31
2019/12/16	7	0	1.00	0.27
2019/12/17	7	1	0.88	0.27
2019/12/18	5	3	0.62	0.19
2019/12/19	6	2	0.75	0.23
2019/12/20	8	0	1.00	0.31

Table B.2: Full results for the I-forest-pca5 model.

date	true positives	false positives	precision	recall
2019/02/09	0	9	0.00	0.00
2019/02/10	0	9	0.00	0.00
2019/02/11	0	8	0.00	0.00
2019/02/19	4	3	0.57	0.33
2019/02/20	4	3	0.57	0.33
2019/02/21	4	2	0.67	0.33
2019/02/22	5	2	0.71	0.42
2019/06/05	8	0	1.00	0.67
2019/06/06	6	1	0.86	0.50
2019/11/27	7	0	1.00	0.58
2019/11/28	8	0	1.00	0.33
2019/11/29	7	1	0.88	0.29
2019/11/30	7	0	1.00	0.58
2019/12/01	8	0	1.00	0.33
2019/12/16	7	0	1.00	0.29
2019/12/17	6	2	0.75	0.25
2019/12/18	6	2	0.75	0.25
2019/12/19	7	1	0.88	0.29
2019/12/20	7	0	1.00	0.29

Table B.3: Full results for the I-forest model.

date	true positives	false positives	precision	recall
2019/02/09	2	7	0.22	0.17
2019/02/10	2	8	0.20	0.17
2019/02/11	5	6	0.45	0.42
2019/02/19	3	4	0.43	0.25
2019/02/20	2	3	0.40	0.17
2019/02/21	2	4	0.33	0.17
2019/02/22	5	3	0.62	0.42
2019/06/05	2	3	0.40	0.17
2019/06/06	2	5	0.29	0.17
2019/11/27	3	6	0.33	0.25
2019/11/28	1	5	0.17	0.04
2019/11/29	1	7	0.12	0.04
2019/11/30	1	5	0.17	0.08
2019/12/01	3	4	0.43	0.12
2019/12/16	2	6	0.25	0.08
2019/12/17	4	5	0.44	0.17
2019/12/18	0	6	0.00	0.00
2019/12/19	1	8	0.11	0.04
2019/12/20	1	7	0.12	0.04

Table B.4: Full results for the OC-SVN-nopca model.

date	true positives	false positives	precision	recall
2019/02/09	2	5	0.29	0.15
2019/02/10	3	5	0.38	0.23
2019/02/11	5	5	0.50	0.38
2019/02/19	4	3	0.57	0.31
2019/02/20	4	2	0.67	0.31
2019/02/21	4	3	0.57	0.31
2019/02/22	1	2	0.33	0.08
2019/06/05	2	7	0.22	0.15
2019/06/06	3	5	0.38	0.23
2019/11/27	3	6	0.33	0.23
2019/11/28	3	4	0.43	0.12
2019/11/29	3	6	0.33	0.12
2019/11/30	2	4	0.33	0.15
2019/12/01	3	5	0.38	0.12
2019/12/16	3	5	0.38	0.12
2019/12/17	4	5	0.44	0.15
2019/12/18	3	4	0.43	0.12
2019/12/19	4	3	0.57	0.15
2019/12/20	4	3	0.57	0.15

Table B.5: Full results for the OC-SVN-pca5 model.

date	true positives	false positives	precision	recall
2019/02/09	1	6	0.14	0.08
2019/02/10	4	5	0.44	0.33
2019/02/11	1	8	0.11	0.08
2019/02/19	3	4	0.43	0.25
2019/02/20	10	3	0.77	0.83
2019/02/21	3	4	0.43	0.25
2019/02/22	4	3	0.57	0.33
2019/06/05	2	5	0.29	0.17
2019/06/06	2	7	0.22	0.17
2019/11/27	3	4	0.43	0.25
2019/11/28	3	4	0.43	0.12
2019/11/29	4	8	0.33	0.17
2019/11/30	3	4	0.43	0.25
2019/12/01	4	5	0.44	0.17
2019/12/16	3	4	0.43	0.12
2019/12/17	3	4	0.43	0.12
2019/12/18	3	4	0.43	0.12
2019/12/19	3	6	0.33	0.12
2019/12/20	3	4	0.43	0.12

Table B.6: Full results for the OC-SVN model.

anomaly id	I-forest-nopca	I-forest-pca5	OC-SVN-pca5	OC-SVN
0	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	6.0	6.0	4.0	4.0
4	NaN	NaN	8.0	1.0
5	NaN	8.0	6.0	6.0
6	0.0	NaN	NaN	6.0
7	0.0	0.0	NaN	NaN
8	NaN	0.0	NaN	NaN
9	2.0	0.0	NaN	NaN
10	2.0	NaN	NaN	NaN
11	6.0	6.0	NaN	NaN
12	0.0	NaN	NaN	NaN
13	1.0	NaN	NaN	NaN
14	1.0	1.0	NaN	4.0
15	4.0	NaN	NaN	NaN
16	3.0	3.0	NaN	NaN
17	NaN	1.0	NaN	3.0
18	8.0	5.0	NaN	NaN
19	0.0	0.0	NaN	NaN
20	NaN	NaN	NaN	NaN
21	NaN	NaN	NaN	NaN
22	NaN	3.0	NaN	NaN
23	0.0	0.0	1.0	1.0
24	NaN	NaN	NaN	NaN
25	7.0	0.0	3.0	NaN
26	NaN	NaN	NaN	NaN

Table B.7: Anomaly detection with the notifications filtering heuristic. Values are the lag of time windows the algorithm needed to detect the anomaly, NaN is a non-detection. With a recall of 0.55 and no false positives, the best algorithm is the Isolation Forest without any PCA pre-processing.

C. Raw Data Schemes

This attachment provides examples of data handled by the Spark streaming jobs.

First, the scheme of the raw collectd stream:

```
{
  "metadata": {
    "availability_zone": "cern-geneva-b",
    "submitter_environment": "production",
    "type": "cpu",
    "toplevel_hostgroup": "monitoring",
    "event_timestamp": 1578144918000,
    "version": "3.0",
    "timestamp_format": "yyyy-MM-dd",
    "submitter_hostgroup": "monitoring/kafkax",
    "type_prefix": "raw",
    "producer": "collectd",
    "_id": "c14174f0-c157-431e-f6c4-740750cb228c",
    "timestamp": 1578144918627
  },
  "data": {
    "time": 1578144918.605,
    "interval": 60,
    "host": "monit-kafkax-003.cern.ch",
    "plugin": "cpu",
    "plugin_instance": "",
    "type": "percent",
    "type_instance": "nice",
    "value": 0.120220690839613,
    "dstype": "gauge"
  }
}
```

Second, the scheme of the `adam_agg_METRIC` topic:

```
{
  "metadata": {
    "producer": "adam",
    "type_prefix": "agg",
    "type": "cpu",
    "version": "003",
    "timestamp": 1526548777000,
    "event_timestamp": 1526548972000,
    "submitter_environment": "production",
    "toplevel_hostgroup": "monitoring",
    "availability_zone": "cern-geneva-c",
    "submitter_hostgroup": "monitoring/flume/dcsource"
  },
  "data": {
    "host": "monit-dcsource-005.cern.ch",
    "window": {
      "start": "2018-05-17T09:00:00.000Z",
      "end": "2018-05-17T09:20:00.000Z"
    },
    "value": 17.635935492350388,
    "plugin": "cpu",
    "dstype": "gauge",
    "interval": 60,
    "plugin_instance": "",
    "time": 1526547637.952,
    "type": "percent",
    "type_instance": "idle",
    "value_instance": ""
  }
}
```

Finaly, the scheme of the joined `adam_agg_os` topic:

```
{
  "metadata": {
    "producer": "adam",
    "type_prefix": "agg",
    "type": "os",
    "version": "001",
    "timestamp": 1577895402000,
    "event_timestamp": 1577896530000,
    "submitter_environment": "production",
    "toplevel_hostgroup": "monitoring",
    "availability_zone": "cern-geneva-c",
    "submitter_hostgroup": "monitoring/kafkax"
  },
  "data": {
    "host": "monit-kafkax-003.cern.ch",
    "window": {
      "end": "2020-01-01T16:20:00.000Z",
      "start": "2020-01-01T16:00:00.000Z"
    },
    "cpu_idle": 30.462260570062956,
    "cpu_iowait": 2.1812271837450967,
    "cpu_softirq": 3.5391269724934715,
    "cpu_user": 48.605475713788586,
    "cpu_nice": 0.13508588341582906,
    "cpu_system": 11.160662822483916,
    "cpu_interrupt": 0,
    "cpu_steal": 3.916160854010129,
    "used_memory": 7355500748.8,
    "free_memory": 163472588.8,
    "cached_memory": 7190465536,
    "slabrecl_memory": 401223475.2,
    "slabunrecl_memory": 92390809.6,
    "running_processes": 0.75,
    "sleeping_processes": 161.5,
    "stopped_processes": 0,
    "blocked_processes": 0,
    "zombie_processes": 0,
    "paging_processes": 0,
    "connections_established": 2341,
    "connections_finwait2": 21.25,
    "connections_finwait1": 0.25,
    "connections_synrcv": 0,
    "connections_closing": 0,
    "connections_lastack": 0,
    "connections_closewait": 2.75,
    "connections_synsent": 0,
    "connections_closed": 0,
    "connections_timewait": 3.5,
    "connections_listen": 18,
    "load": 1.1688125,
    "bytes_sent": 108987147.76704645,
    "bytes_received": 90456066.98804408,
    "disk_write": 22022261.978565954,
    "disk_read": 8888375.96737196
  }
}
```