



POLITECNICO DI TORINO

Master degree course in Computer engineering

Master Degree Thesis

Data Science for Information Security with Open Source technologies

Supervisor

Prof. Elena Baralis

Candidate

Carlo Ventrella

matricola: 240205

Internship Tutor

Volker Gundermann

September 2018

This work is subject to the Creative Commons License.

Contents

1	Introduction	5
1.1	The proposer	6
1.2	The goal	6
1.3	The use case	7
1.4	Structure of report	7
2	Log preparation	9
2.1	Best practices	10
3	Architecture	13
3.1	External components	13
3.1.1	Elasticsearch	13
3.1.2	Logstash	15
3.1.3	Kibana	16
3.2	Developed components	16
3.2.1	Log Retriever	16
3.2.2	Log Analyzer	17
4	Anomaly detection	21
4.1	Use cases	22
4.2	Methodology	22
4.3	System availability	23
4.3.1	Forecasting model	24
4.3.2	Detection model	37
4.4	File handling	41
4.5	Error monitoring	46
5	Models integration	55

6	Deployments	59
6.1	PABS	59
6.2	Sysper	59
7	Conclusions	61
7.1	Future work	62
	Bibliography	63

Chapter 1

Introduction

Every IT system produces log files, collections of significant events produced by all software modules of the system. The three main goals of loggings are:

- Accounting: the process of collecting and recording information about events
- Audit: the systematic, independent, and documented process for obtaining audit evidence and evaluating it objectively
- Monitoring: pro-actively monitoring system for information security incidents

Logs may be used for various purposes, for example to investigate the nature of a failure or of a security incident. They can help monitoring the health of the system and evaluating its performances. Since they reflect the behavior of the applications, logs can be used effectively to help business decisions.

For those reasons, servers record events produced at both operating system level and application level. Understandably, this process generates large amounts of data.

This is particularly true for medium-large sized companies, where applications can be easily distributed on many nodes, each one producing hundreds or thousands of logs every day. In this context, especially if the application handles sensitive data, logs can have a legal value; therefore a number of policies are often defined to regulate the content and the structure of the files.

Despite its obvious benefits, logs are often ignored due to their high level of verbosity, which makes it cumbersome to effectively extrapolate the desired informations. Therefore it becomes impossible for a human being to efficiently parse and understand this massive flow of informations. Thus, the task of log analysis is

a natural application of automation.

When logs are parsed and centralized, several techniques can be implemented to detect anomalies in the system utilization. An abnormal peak of failed logins needs to be notified, as well as a suspicious number of exceptions. If the application manages sensitive data, specific detective controls can be implemented to monitor the activities on those resources.

As a result, a variety of controls can be applied, some of them are generic and can monitor a variety of metrics, and others are application dependent and require the involvement of domain experts.

For those reasons it is desirable to have a system that collects and analyzes logs, that can help the mining of valuable information hidden in the logs, and that is able to detect deviations from the normal behavior of the application.

1.1 The proposer

In the European Commission, DIGIT.B.2 is the Unit responsible for creating and maintaining IT systems for managing human resources. These systems assist DG Human Resources and Security (HR), the Pay Master Office (PMO) and the European Personnel Selection Office (EPSO) in carrying out their tasks in the various fields relating to human resources management and financial rights.

1.2 The goal

To build a system prototype based only on Open Source technologies for the collection and the analysis of the logs. The prototype is meant to provide an easy to use view into the the system utilization. Objective of this project is then to extract information that can be helpful for both technical and business domains through informative and customizable visualizations. It has to automatically analyze logs in order to trigger alarms in case of anomalies and to highlight insights.

First part of this project is thus to build a robust and reliable infrastructure using open source technologies. It has to continuously retrieve logs from multiple nodes, parse them and locally store them. A system like this requires to be highly configurable. As discussed extensively under Chapter 2, users should be able to specify which type of log to feed into the pipeline and from which node. Each node may

need its own credentials and different types of logs can be downloaded with a customizable frequency.

Secondly, a module that periodically analyzes logs to detect anomalies has to be developed and integrated in the pipeline.

This system has to provide the users a way to dive into the results of the analysis, e.g. if an abnormal peak of access requests is detected, the users will want to investigate the details of the anomalies. For this reason the infrastructure has to include a tool to quickly and effectively visualize reports, to perform complex queries and build informative visualization whenever needed.

This paper will cover the infrastructure details and the algorithms implemented to perform the detective controls, as well as the issues encountered during the development.

1.3 The use case

The prototype has been realized following the needs of PABS (Post Activity Beneficiary Suite), which covers the termination of service domain of the European Communities institutions and in particular the management of inward/outward transfer of pension rights, severance grant and unemployment allowance.

From a technical architecture point of view, the PABS information system is based on Java, with its database layer implemented in Oracle. On average, the application is used by 40 users during working days, and produces about 2000 log entries per working hour.

1.4 Structure of report

This thesis is structured in the following way.

Chapter 2, describes the log preparation, the log pre-processing phase that possibly precedes the ingestion of the logs into the pipeline. One section of this chapter is dedicated to the presentation of guidelines that can help minimizing the impact of log preparation for future installation of this prototype. It is followed by chapter 3, which contains the description of the components that constitute the infrastructure. It goes through the configuration process of the chosen technologies and the development of ad-hoc modules.

Chapter 4 covers the analytical part. For each task requested to be addressed, a set of alternatives are proposed and theoretically explained. The proposed models are tested on the available datasets and results are presented. Finally, the model that better fits the data is selected.

In chapter 5 it is explained how the selected models are integrated into the pipeline. Chapter 6 summarizes the resulting prototype and Chapter 7 provides some considerations about future improvements and concludes the thesis.

Chapter 2

Log preparation

Log preparation is the task of applying changes to the structure of log files, by means of insertion, modification or deletion of fields.

As outlined in the introduction, detective controls can be more or less generic. As a matter of fact, some type of logs can be found in every web server. Access logs, error logs and syslog are some examples.

Within the same company, these types of files usually share the same structure, thus easing the ingestion process into the pipeline. Some relevant information can be extracted from those files: access logs provide the distribution of access requests against a web server, the endpoints reached by the customers of the application and the resulting status code. They record also informations about the clients, such as their IP address and their user agent. Error logs instead collect exceptions raised within the application. They are essential for the troubleshooting process and since the distribution of errors over time can be extracted, they are well suited for detective controls, useful for proactive fault handling.

Thus, the definition of detective control for this class of logs can be extended to a variety of applications. Furthermore, because of their well defined structure, they don't need any preparation. It is a natural choice to develop these detective controls in a such way that they can be easily adopted on other use cases.

On the other side, each application will produce its own set of log files. They will be domain dependent and will have different structures. For this reason, specially for large companies, a set of policies is defined to regulate the logging process. They comprise rules and best practices about the structure and content of the files. Depending on the sensitivity level of the application, different levels of policies can

be specified.

In this case the definition of detective controls is strictly routed by the needs of the business and cannot be generalized. For this project some business controls were requested, and this led to an important issue: some log files needed to be prepared. In particular, for some of them it was necessary to enrich the content of the files, by means of adding mandatory fields for the satisfaction of the detective controls. In addition, due to the useless level of verbosity and the quantity of data, some fields needed to be summarized without losing information. Another problem that can be faced during this phase is the inconsistency of the files. Entries can be duplicated in multiple logs, which increases verbosity and imposes an additional requirement in the pipeline, that is, it has to guarantee that duplicates are correctly filtered. Duplicates can compromise statistics and heavily impact disk usage. As last change, it is found convenient to introduce a transaction identifier in all business logs. This change eases the grouping of entry logs among multiple files.

Log preparation is therefore a process that requires the active involvement of the development team, responsible for the feasibility evaluation of the changes and their implementation. As for the development of this prototype, the definition and the implementation of those changes required more or less 15 working days.

Of course this process has an important drawback, that is the impossibility to use older files for the analysis task, since they can lack of mandatory information useful for the aforementioned controls. Thus, as extensively described in Chapter 4, this will reflect in important decisions for the choice of the analytical models.

2.1 Best practices

Based on this experience and on the logging guidelines provided by the proposer, this section summarizes the previous chapter by listing good practices that should be followed to minimize the impact of log preparation for both installation and analytical phases.

- Log structures have to be consistent (i.e. no multiple structures in the same file)
- Presence of event date and time
- Presence of a unique identifier of the current user

- Presence of a transaction unique identifier
- Each log file should have a reasonable minimum retention period
- Only relevant information should be recorded

Especially for application based requirements, this phase will still be needed; however it will drastically reduce its duration.

Chapter 3

Architecture

This chapter goes through the technologies that comprise this prototype. Some of them are well known open source products; others are modules ad-hoc developed to cover some gaps introduced by the requirements of the project.

The combination of these modules results in a system that achieves the followings:

- Automatically retrieves logs from multiple servers
- Stores logs in a database
- Efficiently performs complex queries in the database
- Provides informative and customizable visualizations
- Periodically performs detective controls
- Send notifications whenever anomalies are found

The entire architecture is depicted in Figure 3.1.

3.1 External components

3.1.1 Elasticsearch

“Elasticsearch is a distributed, RESTful search and analytics engine.”

It is the heart of the ELK stack, it is used to store and analyze big volumes of data in a near real-time fashion. Elasticsearch is a document oriented database,

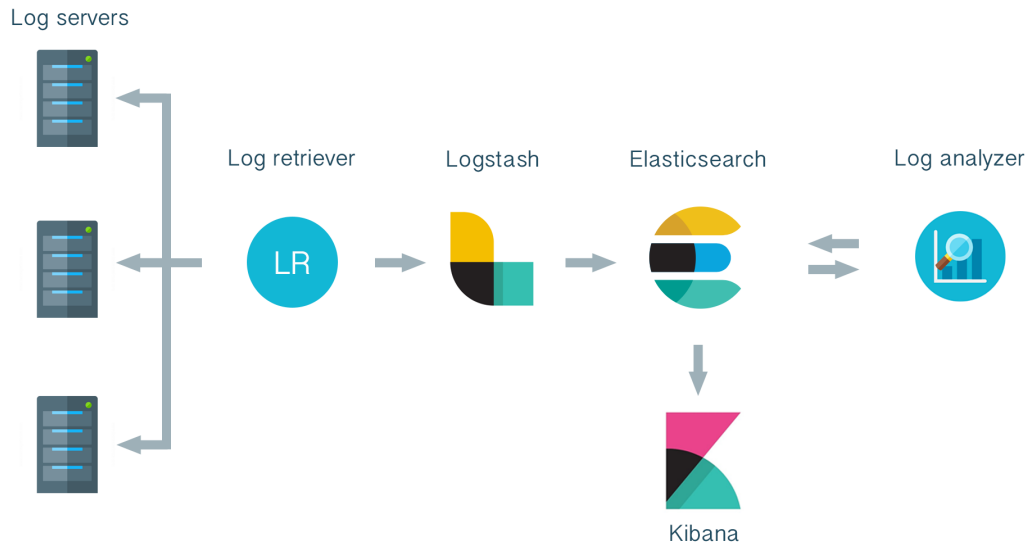


Figure 3.1: Architecture design.

hence it stores data in the form of JSON documents, each one belonging to an index.

Being Elasticsearch a full-text search engine, matching documents is not easy as for databases that deal solely with structured data. In this case, a query simply reduces to check whether a field matches a specific value. Under the hood, Elasticsearch exploits Lucene scoring strategy to return relevant documents. It consists in a combination of the *boolean model* and of the *practical scoring function*. The former simply exploits *AND*, *OR* and *NOT* operators to exclude the documents that do not contain the requested terms. The latter is a function based on the *TF-IDF* measure and is used to accurately assign a score to each document.

Elasticsearch is distributed, meaning that indices are stored in shards hosted on multiple nodes of the cluster. It is a peer to peer based system, where the coordination of the nodes is managed by a discovery module in charge of electing a master node that maintains the cluster state. When nodes join or leave the cluster, the master node reassigns the shards.

To provide reliability, each shard can have one or more replicas, however, one node only is available for this project, thus making replicas unnecessary.

Several strategies can be adopted for the definition of indices (e.g. daily indices, monthly indices etc.). In general, daily indices have the advantage that some types

of queries (e.g. deletions) are much more efficient. Moreover, since one index belongs to a shard and each shard can host up to 2^{31} documents, this can be a hard limit. On the opposite side, every index carries the overhead.

Because of the large amount of logs being fed into the system, daily indices are adopted. In particular, for each log type (e.g. access log, error log etc.) a new index is created every day. To give an idea of the amount of logs stored, it has been observed that one index collects an average of 50 000 log entries during working days.

3.1.2 Logstash

Logstash is the data processing pipeline, used to automatically ingest, filter and convert data in JSON documents. It is configured to parse different types of data structures and to feed them into a *stash*, Elasticsearch in this project.

Logs can be gathered in many ways since Logstash can be combined with a number of beats, which are lightweight data shippers, installed where data is produced.

Particular attention need to be paid to the pre-processing phase. Logstash ingests different type of logs (e.g. access logs, error logs, syslog etc.), each one with its own structure. Its crucial in this phase to carefully parse log entries, to map each field to the proper type and to send it to the right Elasticsearch index. However, Logstash comes with a number of filter plugins which ease this process.

Duplicates prevention

Its worth to mention that for many reasons (e.g. bugs in application, log file corrupted etc.) the same log line can be fed into Logstash multiple times.

To prevent Elasticsearch storing duplicates, the *Fingerprint Filter Plugin* is exploited to compute a digest of the log line which is used as document as unique identifier. The plugin provides a number of hash functions that can generate the digest, such as MD5-HMAC, HMAC-SHA1, HMAC-SHA2, MURMUR3.

The choice of the algorithm is rooted by their collision resistance. Because of the *Birthday Paradox*, a hash function that produces a N bit output will require on average $2^{N/2}$ input before a collision is produced. If Elasticsearch receives a document with the same ID of another document present in the index, the latter is overwritten.

In the context of this project the number of logs per index (i.e. per day) is about 50 000 (on average). It means that to prevent collisions, a hash of $N = 2^{N/2} > 50000$ is required.

The MURMUR3 implementation of fingerprint plugin outputs hash of 32 bit, thus producing a collision after 2^{16} inputs, that is 65 536. Since it is close to 50 000 and considering that the latter is an average, a greater number of bits is preferred.

MD5 and SHA2 produce a hash of 128 bit and 256 bit, respectively. Both satisfy the requirement, so the MD5 version is used as a compromise between collision resistance and speed.

3.1.3 Kibana

“A Picture’s Worth a Thousand Log Lines”

Kibana is a web interface which runs on top of Elasticsearch. It allows to build informative dashboards and visualizations to easily perform complex queries. Its intuitiveness makes it a powerful tool to inspect the results of the analysis, to deeply understand the causes of an anomaly, but also to answer business related questions.

As a result, in the context of this project Kibana is used for two main purposes:

- To visualize the results of the analysis. A number of dashboards has been created to provide an exhaustive view of the analysis performed on the logs.
- To provide a tool to perform custom queries to address specific tasks. As a matter of fact, Kibana can be easily distributed to different actors involved in the application (e.g. developer team, business level, contractors etc.)

3.2 Developed components

3.2.1 Log Retriever

In a context where logs are produced on many nodes, beats are the ideal solution because they can be installed on each node and automatically ship logs through the network to the logstash instance. However, due to IT infrastructure policies, it was not possible to install software on the nodes, therefore a custom solution developed in python has been implemented to periodically pull new logs.

Since each analyzed IT systems stores its logs on different servers, in different locations and with its own credentials, the log retriever has been designed to be

highly configurable.

Application design

A configuration file is used to instruct the log retriever, it contains the configuration of each node, which comprises the following:

- Base URL
- Login credentials
- Proxy credentials
- List of log to download
- Download frequency

Log Retriever exploits the configuration file to schedule one job per node. Each job periodically performs an HTTP request against each log files that is configured to be monitored. By comparing the last-modified header of the remote version with the one stored locally, it downloads only updated files. Files are downloaded in a folder constantly monitored by Logstash, and eventually extracted.

To prevent disk saturation, a garbage collector periodically removes logs that have already been ingested by logstash.

3.2.2 Log Analyzer

Substantial efforts have been devoted to the development of the anomaly detector module, a python application used to analyze logs to detect anomalies and eventually produce notifications.

The idea behind this module is to periodically query Elasticsearch for new logs and to feed them in a number of analytical models exploited to detect different type of anomalies.

With reproducibility in mind, the software has been designed to be used by several IT systems, each with its own requirements to be met: as extensively explained in the followings chapters, some type of anomalies are application independent, others are suited for the specific use case.

For this reason, the application required high modularity to allow analytical models to be shared among requirements and to ease the integration of ad-hoc detective controls.

Application design

The first part of the application is a module, called *loader*, which parses the configuration and schedules a number of *jobs*. One *job* is in charge of performing analysis on a topic (e.g. exceptions, access logs, user activities etc.) by exploiting one or more techniques.

Each job can be scheduled with a different frequency and is started with a set of parameters needed to configure it properly. It will retrieve data from a specific Elasticsearch index, and will save the result in another one. Moreover, the models presented in the following chapters require parameters that depend on the data that is analyzed.

These parameters are collected in sections of the configuration file, which contains a total of $N+1$ sections, where N is the number of jobs. The additional section is used for parameters that are common to all the jobs, such as the log path, log rotation details, log level and so on.

On application startup, the loader parses the configuration file, and schedules the jobs with the specified frequency and provides them with their set of configured parameters.

The structure of a job is designed to provide a common way of performing multiple analyses. As presented in the following chapters, all the models requires a first phase where historical values are elaborated, models are trained and statistics are computed, and a second phase where new data is analyzed for anomalies. Statistics and models coefficients produced during the training phase are stored locally. In particular, every job is provided with a folder where models store their data.

Once models are trained, they are ready to analyze new data. The details of the configuration phase, such as how often it occurs, and how many data historical data it requires, depends on the adopted model. The details of each model will be extensively depicted in the following chapter.

When this first phase is over, every time a job is run it queries elasticsearch for all the data not yet analyzed. This is accomplished by locally storing the timestamp of the last analyzed log. New logs are then analyzed and the results (e.g. anomaly score, forecast, etc.) are uploaded in elasticsearch.

Whenever anomalies are found, an email is sent to the list of configured recipients. The objective of the email is to provide the user of the application a high level description of the anomaly that will allow him to evaluate the gravity of the situation and further investigate it.

Chapter 4

Anomaly detection

Anomaly detection is the process that leads to the identification of observations that are not compliant with the expected pattern. As Dunning and Friedma highlight [2], it is the science of *spotting the unusual, catching the fraud, of discovering the strange activity*. They are not necessarily the rare items, but can be also be unexpected bursts of activities. In the context of web application, an example can be a sudden peak of failed logins, an abnormal number of exceptions thrown, or an unusual number of downloads. Application of anomaly detection techniques ranges over multiple domains, such as intrusion detection, fraud detection, system health monitoring, medical and public health.

As pointed out in [1] , anomalies can be classified in three categories:

- Point anomalies: single observations that are far off from the rest. An example can be the amount spent by a customer in a single day, it can be a warning signal for a credit card fraud.
- Contextual anomalies: also called conditional anomalies, are observations that are unexpected with respect to the actual context. For example, a certain number of visits on a web site can be reasonable during normal days, but anomalous during holidays.
- Collective anomalies: observations that collectively represents an unexpected situation. Single instances may not be novelties by themselves, but the combination makes them anomalous. a high number of downloads for a specific resource may not be anomalous, but if this behavior affects in the same time multiple resources it can be the cause of data exfiltration.

4.1 Use cases

Several anomaly detection techniques have been studied and implemented to monitor different metrics of the system utilization.

Number of access requests over time

Access logs record HTTP requests performed against a web server. Each line provides important information, such as the client IP, the requested resource, the status code returned and many more. The number of access requests over time is an interesting metric monitored to track server availability.

In this sense, anomalies are sudden peaks or drops of the number of requests in multiple time frames (e.g. 15 min, hour, day). A drop might indicate a system failure, a peak can reveal an intrusion. These situations have to be detected and notified.

Exceptions

The analysis of error logs is vital for the troubleshooting process. Java based applications produce exceptions in form of stack traces: on one side they collect detailed information needed to address the root cause, on the other side this level of verbosity makes error logs difficult to parse. This is particularly true when the application is a medium-large one, used by hundreds of users.

The implemented technique has to help the diagnosis process; it has to provide a way to highlight unusual exceptions and to detect sudden bursts.

Application specific metrics

While the previous two use cases can be generalized to almost every application, many metrics can be defined to monitor aspects that depend on the business logic. Those metrics are particularly important because they may help business decisions. For this prototype, one anomaly detection technique has been implemented to monitor who have access to sensitive data.

4.2 Methodology

The idea behind an anomaly detection technique is to exploit the normal behavior in order to identify non-normal situation. To cite [2], *"anomaly detection is*

based on the fundamental concept of modeling what is normal in order to discover what is not".

For all use cases unlabeled dataset are collected and used for the analysis. This kind of analysis is referred to as *unsupervised*, because there's no indication of what is "normal" or "abnormal". Unsupervised anomaly detection techniques assume that the majority of the observations collected in the dataset describe the normal behavior. Because of the lack of labeled events, the operational team helped the process of evaluating the models by means of pointing out true anomalies and false positives.

4.3 System availability

The technique analyzed in this chapter provides a way to detect anomalies on time series. In particular, the number of access requests over time is considered as a time series. A web application employed in a working environment is characterized by a pattern with well defined properties. Indeed, as pointed out in the figure 4.1, daily and weekly seasonalities are easily recognizable. A high number of access requests is present from 8.00 am to 18-19 pm during working days (Monday to Friday), and a near-zero level otherwise (non working hours, weekends and holidays).

Access requests performed against PABS are used to evaluate the prediction accuracy of the model. The dataset comprised 5 months of data, with a granularity of 15 minutes.

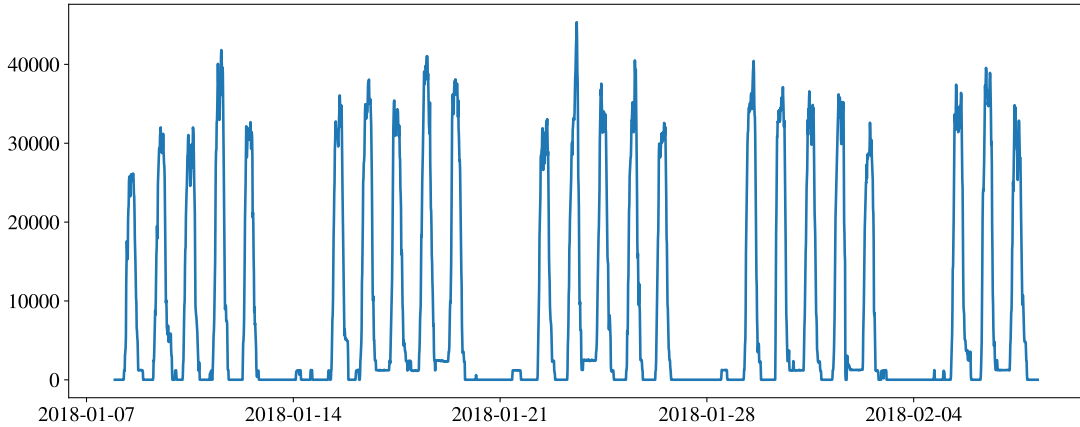


Figure 4.1: Number of access requests performed against PABS, collected every 15 min from January to February 2018

Despite the strong seasonalities, the previous graph highlights also discontinuities, mostly during late hours, which depends on many factors: it is possible to recognize the impact of releases after work and scheduled jobs during the week-ends. As expected, holidays dramatically affect the number of access requests, as well as the days close to them.

The technique

A common approach to detect anomalies in time series is to use a model trained on the normal behavior and that predicts future observations. The evaluation of the prediction error is then needed to classify the observation. The idea behind this approach is that the forecasts will reflect the pattern learned, that is presumed to be normal. An observation that is far off from the forecast is likely to be anomalous. The implementation of such technique can be decomposed in two tasks:

- The definition of a model that is able to learn the properties of the time series (e.g. seasonalities) and reflects them in the forecasts. This is called the *prediction phase*.
- The definition of an anomaly detection technique that exploits residual to classify the observation. This is called the *classification phase*.

Several combinations of prediction and classification strategies can be adopted to achieve this task. Related works describing different variants of the same method are presented in [5], [6] and [7].

A second approach that proved to be effective on some forecasting tasks employs VAE(variational auto-encoder) [8], that is a bayesian deep neural network based on an encoder-decoder architecture. An application of VAE for time series prediction is proposed in [9]. Unfortunately, due to time limitations, this alternative was not evaluated.

4.3.1 Forecasting model

Forecasting is the process of making predictions by exploiting historical data. It is a central task for a wide range of activities: it is applied in finance, meteorology, politics, and in a number of business domains. Despite its importance, the production of accurate forecasts has always been a challenging task. For this reason, many different models are available: each one can be more or less suitable for a specific time series, thus requiring experience and deep domain knowledge.

The forecaster used for an anomaly detector is required to produce high quality forecasts in the short term period. Since the detection of anomalies relies on the study of the residuals, the lower the accuracy, the higher the chance to ignore anomalies.

The forecaster has also to incorporate the characteristics of the time series, such as trend and seasonalities. For example, if the time series presents a well-defined weekly seasonality but the model produces forecasts that rely only on few historical observations without taking the seasonality into account, the accuracy will be very high but the deviation from the seasonality would not be noticed.

The selection of the model for this use case followed a number of tests and comparisons of some well known algorithms. Each approach is described in the following chapters, where a theoretical description is proposed, as well as the results on the given use case.

Holt Winters

The first analyzed model is Holt Winters, a very common approach when it comes to forecasts seasonal time series. It belongs to the class of the exponential smoothing techniques, also known as *triple exponential smoothing*.

The idea behind exponential smoothing methods is to assign exponentially decreasing weights to historical observations. In particular, Holt Winters method applies the smoothing across seasons, e.g. if the series presents a 7-day seasonality, the seasonal component of a Sunday would be exponentially smoothed with the previous Sundays.

Holt Winters comes with two different variants, which fit two main types of seasonality. As stated in [3], the *additive* method is preferred when the seasonal variations are roughly constant through the series, while the *multiplicative* method is preferred when the seasonal variations are changing proportional to the level of the series.

$$l_x = \alpha(y_x - s_{x-L}) + (1 - \alpha)(l_{x-1} + b_{x-1}) \quad (4.1)$$

$$b_x = \beta(l_x - l_{x-1}) + (1 - \beta)b_{x-1} \quad (4.2)$$

$$s_x = \gamma(y_x - l_x) + (1 - \gamma)s_{x-L} \quad (4.3)$$

$$\hat{y}_{s_x} = l_x + mb_x + s_{x-L+1+(m-1)modL} \quad (4.4)$$

Equations 4.1,4.2,4.3 describe respectively the level, also referred to as intercept or baseline, the trend and the seasonality. Each of the three components introduces a smoothing parameters (i.e. α , β and γ), that has to be tuned to properly fit the time series. Equation 4.4 represents the forecast equation, that is a linear combination of level, trend and seasonal components.

Another important aspect is the definition of initial values. The initial trend, described by equation 4.5, is estimated simply averaging trend averages across seasons.

$$b_0 = \frac{1}{L} \left(\frac{(y_{L+1} - y_1)}{L} + \frac{(y_{L+2} - y_2)}{L} + \dots + \frac{(y_{L+L} - y_L)}{L} \right) \quad (4.5)$$

More challenging is the definition of the initial seasonal components, usually defined as in equations 4.6, where N represents the number of periods within a season (e.g. 7 for a weekly seasonality) and L the number of cycles available in the training set.

$$s_i = \sum_{j=1}^L \frac{y_{(j-1)L+i}}{A_j}, \quad i = 1, 2, \dots, N \quad (4.6)$$

where A_j is the average over each period of the season, defined as following.

$$A_j = \frac{\sum_{i=1}^N y_i}{N}, \quad j = 1, 2, \dots, L \quad (4.7)$$

Implementation The model presented so far is implemented in Statsmodel, a Python library that provides an extensive set of statistical models, estimators and functions which ease the data exploration process.

Holt Winters has been tested with different numbers of time steps used to produce the next forecast. Given that one observation is computed every 15 minutes, a history of *96 time steps* (one day of data) is expected to incorporate the daily pattern, but a history of at least *672 time steps* (one week of data) is required to learn the weekly pattern, although this can be computationally expensive.

Prophet

The second model analyzed for this project is an additive model call Prophet, developed by Facebook and released as Open Source, key aspect of this project. As outlined in [4], the idea behind this procedure is to provide a model with interpretable parameters and that can be intuitively adjusted by analysts with domain knowledge about the time series.

The model takes into account three components: trend, seasonality and holidays.

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t \quad (4.8)$$

The trend component is $g(t)$, while $s(t)$ describes the seasonality, $h(t)$ the impact of holidays and ϵ_t represents the informations that are not taken into account by the model. This last component is assumed to be normally distributed. Since the use cases do not present any trend, this component has not been covered.

To address multi-seasonality, $s(t)$ is modeled as a Fourier series.

$$s(t) = \sum_{n=1}^N \left(a_n \cos\left(\frac{2\pi nt}{P}\right) + b_n \sin\left(\frac{2\pi nt}{P}\right) \right) \quad (4.9)$$

Here P is the period (e.g. 7 for weekly seasonality) and N is a parameter that has to be specified, that imposes the upper bound to the number of coefficients a_n and b_n that have to be estimated. As outlined in [4], a high value of N is more suited for seasonalities that change frequently, but increases the risk of overfitting.

Holidays instead depends on many factors, such as the country or the lunar phases. However, their impact is assumed to be similar every year. This is the reason why Prophet accept a custom list of holidays uniquely identified. The holiday components are modeled as follows.

$$h(t) = Z(t)\kappa \quad (4.10)$$

where κ is a parameter assigned to each holidays and $Z(t)$, that is a matrix representing whether time t matches an holiday.

Implementation Prophet has been released in both Python and R versions, both available on Conda. In line with the rest of the project, the former was tested. Of course, since less than 1 year of data is available, the holidays component cannot

be exploited.

Several variants of the model have been tested for the tuning of the parameters. Since the time series do not present any trend, the relevant parameters are the Fourier orders, previously named N , for the daily and weekly seasonal components. In the default configuration, those are set to 3 and 4.

Deep Learning

Deep Learning is a branch of machine learning that exploits deep artificial neural networks (ANN) to attempt the modeling of complex patterns through multiple non linear transformations. With respect to simple neural networks, the process of *stacking* several layers facilitates the learning of hierarchical complex structures.

Recurrent neural network Among several architectures, Recurrent Neural Networks (RNN) is one of the most popular deep neural network, designed to model sequential data, such as text, speech and time series. They proved to be very effective for both natural language processing (NLP) and forecasting tasks. The reason for this success is that, unlike other architectures, RNN do not consider observations as independent of each other. On the contrary, each output depends on the informations learned for the previous observations.

At a given time t , the RNN analyzes one observation at a time, x_t . It receives as input x_t and the hidden state of the previous time step, s_{t-1} . It updates the hidden state s_t and produces an output h_t , that depends on the current input value and on the hidden state that maintains a memory of the previous time steps. The number of time steps to "remember" has to be defined. The intuition behind this state transfer is to allow past observations to impact the current outcome. In non-recurrent architectures, the output strictly depends on the current input.

For each time step, hidden state and output are calculated as follows.

$$s_t = \sigma(Ux_t + Ws_{t-1} + b_s) \quad (4.11)$$

$$h_t = softmax(Vs_t + b_h) \quad (4.12)$$

Here, U is the weight matrix between the input layer and the hidden layers, W and b_s are the weight matrix and bias that represent the transition between two time steps, V and b_h are the weight matrix and the bias for hidden to output transition. The activation function usually exploited are the sigmoid (4.13) for the

hidden state, and the softmax (4.14) for the output. The softmax is widely used as activation function of the output layer because it maps the outputs of the hidden layer in a K-dimensional vector, where each lies between 0 and 1, and all the entries sum up to 1.

$$\sigma(x) = \frac{1}{1 + \exp^{-x}} \quad (4.13)$$

$$softmax_k = \frac{\exp^{a_k}}{\sum_{k'}^K \exp^{a_{k'}}} \quad k = 1, 2, \dots, K \quad (4.14)$$

It has been theoretically proven in [10] that the limitation of this architecture lies in the difficulty for a time-step to be affected by observations occurred far in the past. This is due to two well-known problems: *vanishing gradients problem* and *exploding gradients problem*. These problems affects the back propagation phase of many-layered networks trained using gradient based optimization techniques, thus including recurrent neural networks.

During back propagation, weights are updated proportionally to the gradient of the error function with respect to the weights. It has been proven in [10] that for recurrent neural networks, error scaling factors with magnitude lower than 1 are likely to lead to smaller and smaller updates for the earlier layers, thus preventing them to improve in an acceptable time. On the other hand, error scaling factors with magnitude greater than 1 are likely to produce large updates that will prevent the training phase to converge. As a result, since in RNN earlier layers represent time step far in the past, only the weights related to immediately preceding time steps will be updated and will impact the outcome of current observation.

This long-term dependency has been solved by new and more complex architectures, such as Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU).

LSTM Long Short Term memory [11] are explicitly designed to solve the long dependency problem by introducing *gates* in the single cell. Gates are structures used to regulate the process of adding or removing information from the cell state, a matrix that carries relevant information through the time steps of the sequence.

The first block is the *forget gate* which, by looking at the input x_t and at the previous output h_{t-1} , produces a multiplier for each entry in the cell state. The intuition behind this gate is to provide a way to decide how much of the previous

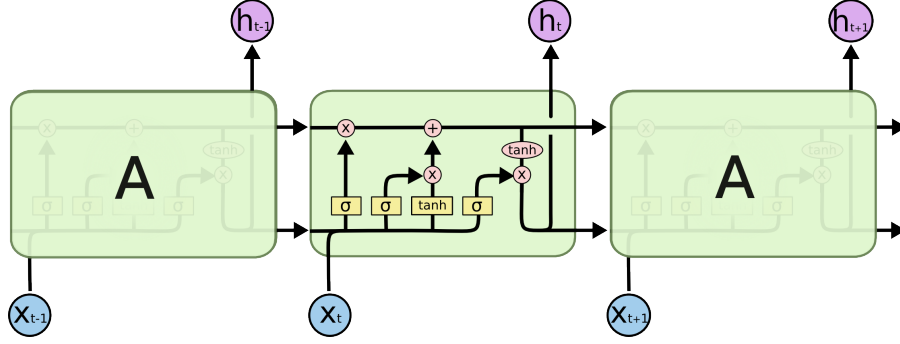


Figure 4.2: The repeating module in an LSTM contains four interacting layers. From Understanding LSTM Networks [12]

state has to be carried to next step. A multiplier close to 0 will cause that information to be forgotten, while a value close to 1 will maintain it. The first version of LSTM presented in [11] did not comprise the forget layer, introduced later in [13] and then adopted in all the implementations. The forget layer is meant to prevent the unbounded growth of the cell state, which may cause the vanishing gradient problem and the disuse of the cell state in the computation of the output, thus removing the memory functionality of the LSTM.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_t) \quad (4.15)$$

New information is added to state cell by the *input gate layer*. As for the forget layer, it takes as input the current observation x_t and at the previous output h_{t-1} . These inputs are divided into two layers; the first one decides which information will be updated, the second creates a list of new candidates that could be added to the cell state. The multiplication of these two outputs represents how much to update each state value.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (4.16)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (4.17)$$

The outputs of input and forget layers finally updates the cell state for the next time step.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \quad (4.18)$$

Finally, the *output gate* exploits the current observation x_t and the previous output h_{t-1} to compute what will flow out of the LSTM. It's result is then combined with the cell state C_t to produce the final output.

$$o_t = \sigma(W_h \cdot [h_{t-1}, x_t] + b_h) \quad (4.19)$$

$$h_t = o_t * \tanh(C_t) \quad (4.20)$$

Pre-processing phase Before the training phase, the training set is standardize in order to scale the features within $[0,1]$. This transformation avoids the network to converge to local optima and to have features which dominates over others when the domain of some features are larger than others. Among several available techniques, MinMax scaling is adopted. A standard scaler which maps values to a distribution with zero mean and unit variance was also tested without visible improvements.

Once the estimator has been fit on the training set, it is used to transform validation and test sets.

Network setup The models tested for this use case took advantage of the memory functionality provided by the LSTM network presented so far to learn the pattern of the time series.

The basic model consists of an LSTM cell followed by a fully connected layer. The LSTM cell is fed with a sequence of *look_back * n_features* observations. Here *look_back* is the number of time steps in the past, each one composed by *n_features*.

The default time series consists of only one feature, that is, the number of access log performed in one hour. As it will be presented in the results, it was interesting to notice the impact of two exogenous features, i.e. day of week and hour, on the ability of the network to learn the seasonalities.

Finally the outputs of the cell flow in a fully connected layer with softmax activation, designed with *look_ahead* nodes depending on how many time steps are going to be forecasted.

Several values for those parameters have been combined to find the model that better fit the use case. However, in addition to this simple model, some enhancements have been tested to improve the prediction accuracy.

Stacked LSTM The success of deep learning is due to the ability to extract complex feature representation of the current time series. As presented in [14] for a language modeling application, deep recurrent neural networks exploits different layers to learn hierarchical structures and easier comprehend long term dependencies with respect to common RNN. On the other hand, and as results will show, stacking more layers considerably increase the training time, as well as the computational loads.

The benefits of stacking layers have been evaluated on the use case.

Dropout One of the major issues with neural networks is the lack of generalization, a phenomenon called overfitting. As a result, the network reaches high results on the training data but performs poorly with held-out test data. The network adapts itself to minimize the error function on the training data, thus perfectly adapting to it. For this reason, the training phase aims to find a good trade-off between generalization and model complexity.

To overcome this problem, a regularization technique named Dropout [15] is often included in deep learning models. The idea behind this technique consists in randomly masking network units during the training phase, thus introducing a penalty to the loss function. Given a chosen probability p , on each presentation of the training set, each unit is ignored with a probability of $1 - p$. To cite [15], another way to view the dropout procedure is as a very efficient way of performing model averaging with neural networks.

A slightly different version of dropout has been developed for recurrent neural networks, named *recurrent dropout* [17]. In the standard version presented so far, dropout masks are applied solely to the input or the output of the cell. Recurrent dropout however has the same dropout mask applied to every recurrent connection.

The impact of recurrent dropout has been tested on the use case.

Implementation The experimentation of LSTM on the use case exploited Keras, a Python framework built on top of other deep learning libraries like Tensorflow and Theano. The former was used in this project. With respect to Tensorflow, Keras provides a high level API to design, implement and deploy complex architectures. On the other hand, Tensorflow is a lower level library which allows a much higher level of customization.

Keras provides a number of standard deep learning models, indeed, it was straightforward to build a model that comprised the building blocks needed to

easily test various combinations, i.e. LSTM cells, dropout, and fully connected layers.

The first test is performed with a simple architecture, which consisted of one LSTM cell and one dense layer of one node. Subsequently, deeper networks have been tested with more LSTM cells and dense layers. Finally, the benefit of increasing the number of nodes in the last dense layer and the introduction of exogenous variables (i.e. day of the week and hour) have been investigated.

Along with the design of the network, the parameters described in the previous section required to be tuned. In particular, the number of layers and nodes for LSTM and dense layers, the number of time steps used to produce the forecasts and the number of outputs. For all the configurations, the batch size was set to 256 and the learning rate to 0.001. Moreover, a recurrent dropout of 0.3 was introduced between two layers.

Results

The predictive models described so far have been applied on the use case and the comparison is presented in this section in order to explain the motivations that lead to chose the selected one. An important consideration regards the meaning of best model.

Naturally, the predictive accuracy is the most important metric of comparison. A number of measures have been defined in literature to evaluate the forecast accuracy, some of them a scaled dependent (e.g. MAE, MSE, RMSE), others are not (e.g. MAPE).

Given a set of forecast errors e_t , defined as the difference between the observed value, o_t , and the forecast, f_t , the following table lists some of the described forecast measures.

MAE (Mean Absolute Error)	$mean(e_t)$
MSE (Mean Square Error)	$mean(e_t^2)$
RMSE (Root Mean Square Error)	$\sqrt{mean(e_t^2)}$
MAPE (Mean Absolute Percentage Error)	$mean(\frac{100e_t}{o_t})$

Percentage measures, such as MAPE, they are usually recommended because they are simple and easy to understand and they are not scale dependent. It is

immediately visible that the drawback of MAPE is the possibility to be infinite for $o_t = 0$. On the other hand, scale dependent measures are useful when comparing different methods on the same dataset. Historically, RMSE is preferred to MSE because it's in the same scale of the data. However, both of them are more sensitive to outliers, which is why MAE is preferred.

It is important to notice that a model that produces forecasts with the best accuracy might not be the right one. As results will show, some of the models output predictions relying only on few observations preceding the time step of interest, without incorporating the seasonalities of the time series. This behavior is visible during holidays: a model that incorporates the seasonalities is expected to produce forecasts that mimic a normal working day. On the other hand, ignoring seasonalities causes the model to miss changes which might indicate non-normal situations such as the failure of the system or an unusual overload during weekends.

Tables 4.1, 4.2 and 4.3 shows the results of the tested models.

ID	Architecture	MAE	RMSE
HW_1	look_back(96)	6351	9622.90
HW_2	look_back(672)	3420.45	6538.10

Table 4.1: Holt Winters results

ID	Architecture	MAE	RMSE
PH_1	weekly_order(3), daily_order(4) (default)	7084.33	8860.10
PH_2	weekly_order(3), daily_order(3)	7002.10	8872.44
PH_3	weekly_order(5), daily_order(5)	7421.05	9929.56
PH_4	weekly_order(10), daily_order(10)	6651.40	8366.00
PH_5	weekly_order(15), daily_order(15)	6797.20	8393.25
PH_6	weekly_order(20), daily_order(20)	6797.30	8431.50

Table 4.2: Prophet results

Despite the high accuracies obtained by the neural networks DL_1 and DL_2 and DL_3, it was noticed that, as previously suggested, the seasonal components were not incorporated into the models. Figure 4.5 (a) and (b) compare the forecasts for models DL_1 and DL_5, clearly describing this issue.

ID	Architecture	MAE	RMSE
DL_1	lstm(32), dense(1)	880.90	1911.70
DL_2	lstm(128), lstm(64), dense(1)	878.70	1782.85
DL_3	lstm(128), lstm(64), dense(3)	667.15	1435.50
DL_4	lstm(128), lstm(64), dense(24), dense(3)	1012.20	2019.67
DL_5	lstm(128), lstm(64), dense(3), exogenous variables	1122.90	2196.87

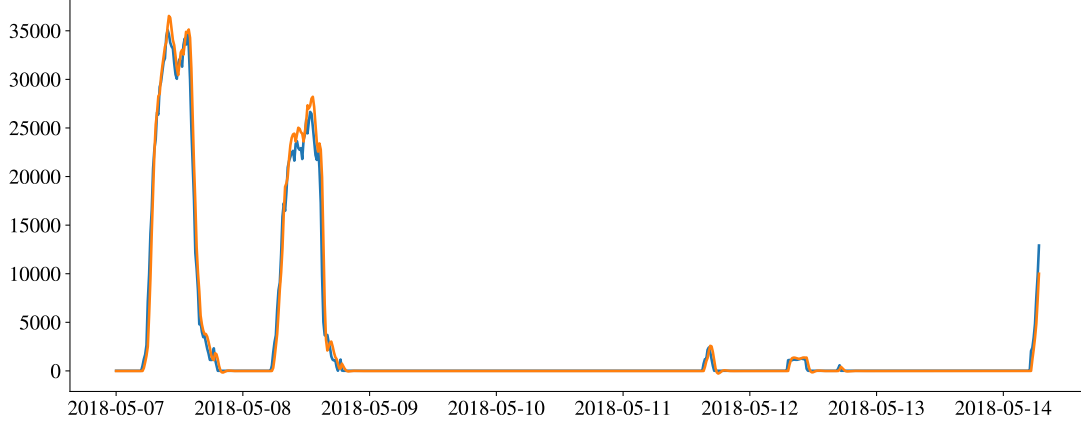
Table 4.3: LSTM results

Although the first model is visibly more accurate, it totally ignores the characteristics of the time series, which leads to miss important anomalies. For these reasons models DL_4 and DL_5 are preferred. The second benefit brought from these models is that they are more general with respect to the others. It was experimented that because of the high number of peaks and drops that characterize the analyzed time series, especially during working hours, accurate models are more prone to detect false positives.

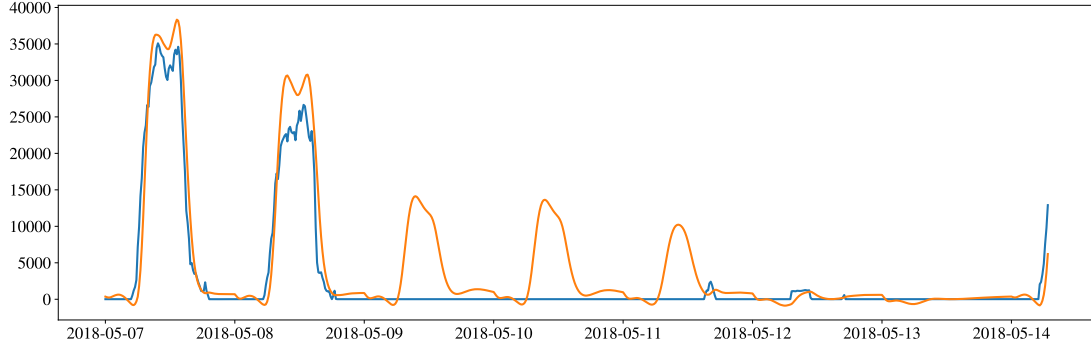
A second aspect to take into account for the choice of the algorithm is the impact on system performances, in terms of time needed for the training and prediction phases. However, as noticed during experimentations, all models required an acceptable time to predict a value. All neural network based architectures required less than 30 minutes to train and less than 30 seconds to predict. Holt Winters model and all prophet models do not have a training phase, but required more time to output a forecast: in particular both Holt Winters with *look_back* = 672 and Prophet with *N* = 20 required more than 30 min.

It is important to notice that although these models effectively detect short-term anomalies, it is not reliable for longer periods: workdays that are particularly overloaded or, on the contrary, under loaded cannot be detected. The same consideration applies to longer time windows, such as weeks. From an availability point of view they are certainly less interesting, because abnormal behaviors usually lasts from few minutes up to some hours. However, there are few cases where an overloading can be a symptom of a failure. For example this is the case of malformed releases that for many reasons can slow down the system. A situation like this might not be immediately visible but a longer period analysis can detect this change.

For this reason, a daily monitoring is included in the access log analyzer. Because



(a)



(b)

Figure 4.3: Comparison between DL_1 and DL_5, where the blue line represents the real number of requests and the orange one the predictions. The latter shows a lower predictive accuracy but effectively incorporate the weekly seasonality.

of the simplicity of the time series, only Prophet and Holt Winters have been tested, and their results are presented in table 4.4: those results are obtained by predicting 7 days given the previous 2 months (60 observations).

Architecture	MAE	RMSE
Holt Winters	142714	281575
Prophet	150852	282215

Table 4.4: Holt Winter’s and Prohet’s performances on daily access logs

It is evident that no one outperformed the other. Even from a computational point of view, both need less than 1 minute to produce 1 week of forecasts.

Although Holt Winters performed slightly better, Prophet was chosen. In general, it is found to be easier to configure and tune for a wide range of time series. Hence, integrating prophet procedure in the prototype makes it reusable for future purposes. Figure 4.5 shows the predictions from 31st of March to the 4th of May.

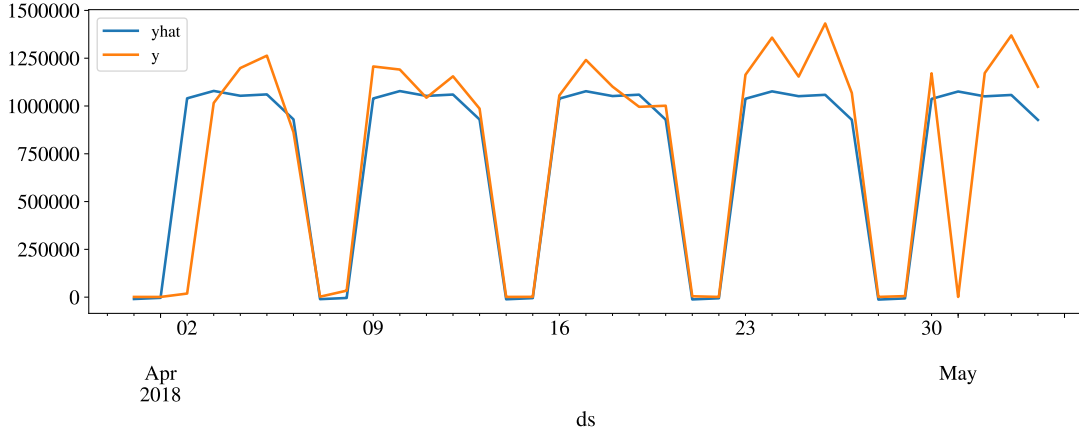


Figure 4.5: Daily predictions with Prophet.

To summarize, two models are chosen to monitor the system availability. An LSTM based architecture for a short monitoring (15 min), and a model based on Prophet for a longer monitoring (12 or 24 hours).

4.3.2 Detection model

Once a forecast is output by the predictive model, it is important to define a measure of how much the observed behavior can deviate from the forecast to be classified as an anomaly.

The forecast error, also referred to as residual, is defined as the difference between the observation received at time t and the forecast produced at time $t - 1$.

$$e_i = y_i - \hat{y}_i \quad (4.21)$$

The prediction errors from validation data are modeled using a Gaussian distribution.

$$e_i \sim N(\mu, \sigma^2) \quad (4.22)$$

The parameters, mean and variance, are computed using maximum likelihood estimation (MLE). The main assumption is that residuals follows a Gaussian distribution.

When a new residual is computed, the log probability density is calculated and used as anomaly score. This score represents the likelihood of a residual to come from the modeled Gaussian distribution. It can be seen as an anomaly score, the lower it is, the higher the chance of the observation of being an anomaly. To actually discriminate observations, a threshold has to be defined and tuned by the users of the application.

A variant to this approach is to fit residuals on different Gaussian distributions. The idea behind this consideration comes from the analysis of the residuals. As shown in figure 4.1, the time series presents low variance during non working hours (i.e. nights, holidays and weekends), and high variance during working activities. This lead to an important consideration, confirmed by the results of the analysis: if residuals of both working hours and non working hours are modeled by the same Gaussian, a residual with a magnitude that is "non-normal" for a non-working hour, would be ignored because of the higher magnitude of the residuals that characterize working hours, which contribute to model the distribution.

The best way to handle this problem would be to fit the residuals on each hour of the week, which leads to $24 * 7$ Gaussians. The main reason why this is not a viable route lies in the necessity of more data. For each hour of the week a reasonable amount of residuals would be needed, not available in this case.

The compromise is to fit the residuals on two Gaussian, which would fit working and non working hours. In this case non working hours would group together the hours in the range [20:00, 7:00], weekends and holidays.

Results Figure 4.6 represents the overall distribution of residuals. On the other side, figures 4.7 (a) and (b) compare the impact of modeling the residuals on two Gaussians.

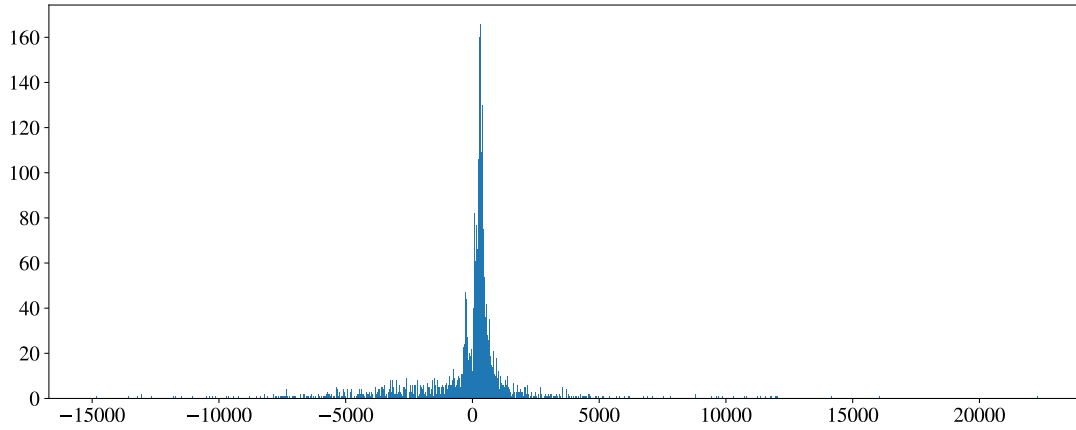


Figure 4.6: Distribution of residuals

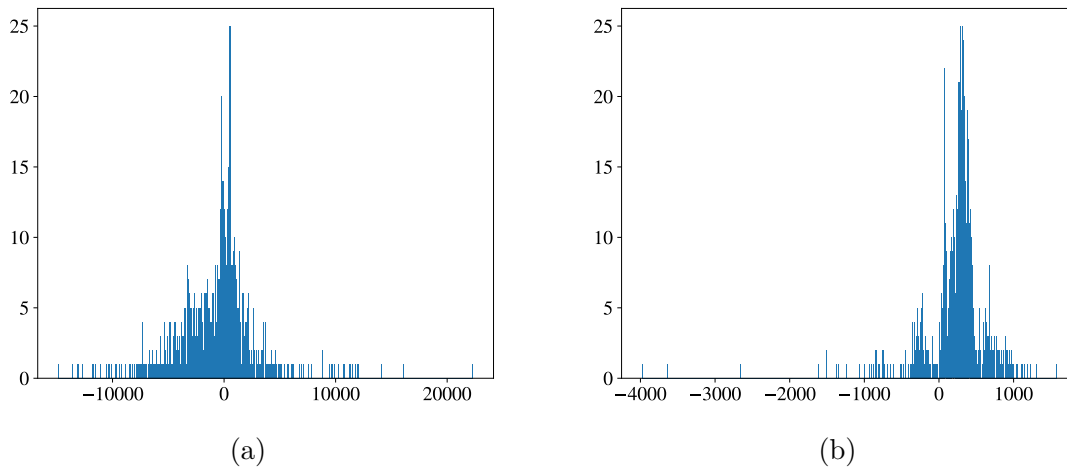
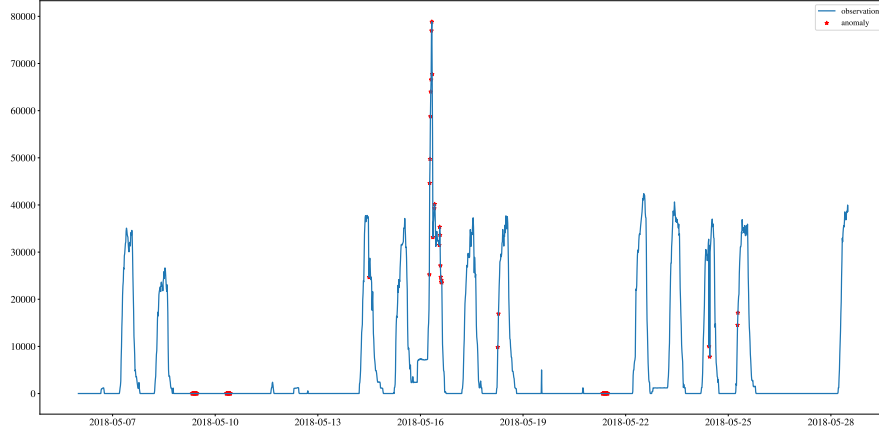
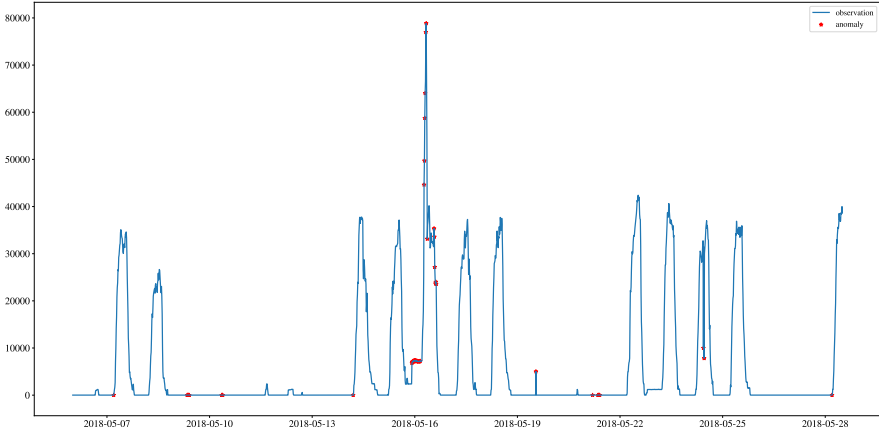


Figure 4.7: Distribution of residuals split in (a) working hour and (b) non working hours

Although the three distributions are not Gaussian-like, the separation of working and non-working residuals provides better results. It is possible to notice that values in Fig. 4.7 (b) ranges in the interval $[-1000, 1000]$, with some negative outliers. It is much thinner than the interval of Fig. 4.7 (a), where values lies in the interval $[-10000, 10000]$. Given the characteristic of the time series, this was an expected behavior.



(a)



(b)

Figure 4.8: Application of the detection model on the test set with one (a) and two (b) Gaussians.

Figures 4.8 (a) and (b) reports the results of the two variations of the detection model on the test set. Both models detect anomalies during 9th, 10th and 21st of May, which are European Commission holidays in 2018. Both models failed in detecting anomalies on 11th of May, which is also a holiday. This failure can be explained by considering the predictions in Fig. 4.5 (b), where it's possible to see that the magnitude of the forecasts decreases after the first day of holiday. As anticipated, anomalies occurred during non-working hours are not detected in the first model. Two examples can be found in the night between the 15th and the 16th and the two small peaks on Saturday 20th and Sunday 21st. Both models present few false positives: the first one detects four anomalies in the mornings of the 19th

and 25th, the second one detects three anomalies in the boundary between working and non working hours the 7th, 14th and 28th of May.

Despite the necessity of defining the working interval and the drawback of having some false positives coinciding with the boundaries, the ability of recognizing anomalies during non working hours is preferred. For this reason, the model based on two Gaussians, for working and non-working hours is preferred.

Zeroes handling When computing the residual, it may happen that the real observation is zero. It can be explained by two main reasons: the number of access requests is truly zero, which is uncommon for this use case, but it can rarely happen at release time. Or it can be a problem of the pipeline, meaning that logs are not being fed into the Elasticsearch due to a failure in the pipeline or a network issue.

Because of the impossibility to automatically distinguish between these two cases, an alert is automatically sent in order to notify the users about this issue.

4.4 File handling

As depicted in the introduction, while some checks are application independent, others are implemented to satisfy the needs of the application. As an example, the access to sensitive data was chosen as a showcase. In particular, PABS handles dossiers of commission employee, which can be considered as sensitive data. One dossier is handled by one user, but for several reasons one user can access to files handled by others, for example he can have administrative roles that allows him to check the works of other users. However, the number of time a user access to files handled by others needs to be monitored. Because of the many reasons that can motivate this behavior, anomalies are not supposed to be notified. Instead, this is provided as a tool that can be used to investigate specific cases.

The following pictures describe this metric for four random users.

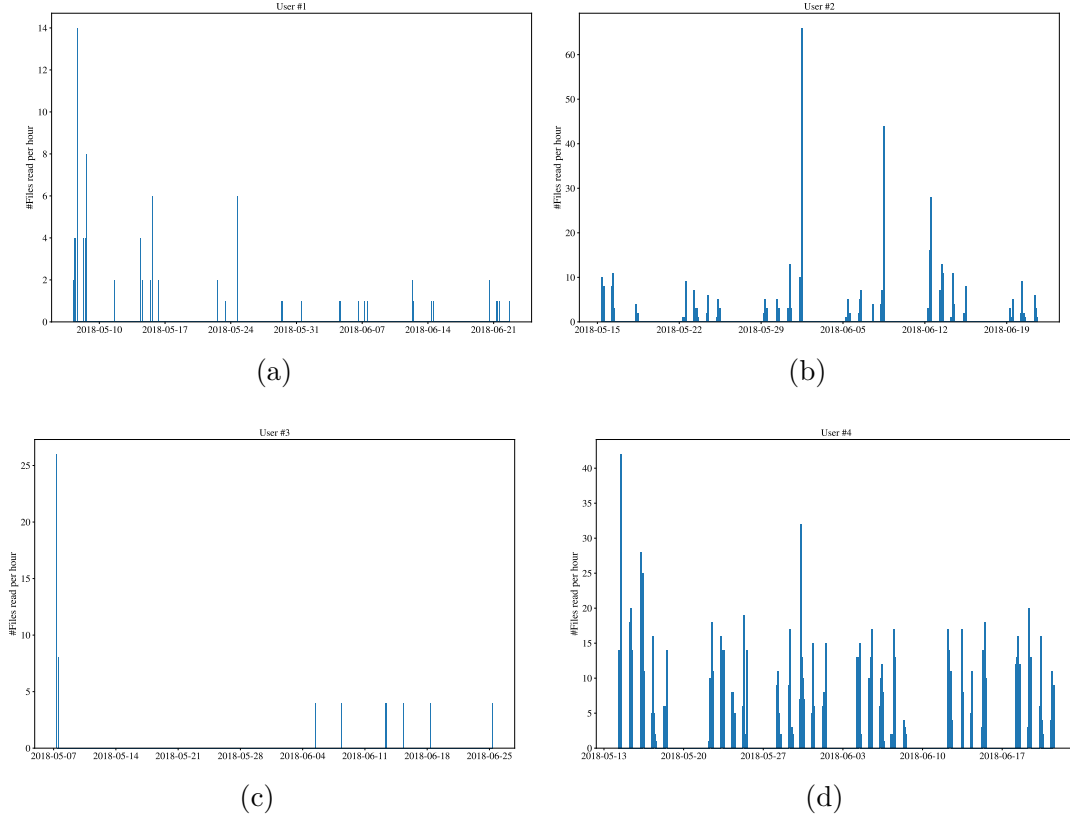


Figure 4.9: Number of accesses per hour to files owned by other users. Fig. (a) and (c) do not respect any recognizable pattern, Fig. (b) and (d) follow a weekly seasonality.

It is clear that while for some users the series follows the working time, for others the file access can be considered as a rare event. Moreover, less than two months of data is available, which reduces even further if we consider days off, weekends and holidays. For these reasons, this analysis simply aims to identify hours when the number of access to files owned by others is anomalous with respect to the normal behavior of a specific user, thus reducing to a simple outlier detection task.

Tukey's boxplot is a simple technique to pinpoint outliers; it is based on the calculation of the two following thresholds that discriminate "normal" values from anomalies.

$$T1 = Q1 - 1.5IQR \quad (4.23)$$

$$T2 = Q3 + 1.5IQR \quad (4.24)$$

Here, $Q1$ and $Q3$ represent the first and the third quartile, while IQR is called interquartile range and is defined as $IQR = Q3 - Q1$. Despite its simplicity of implementation and visualization, it assumes that the data follows a normal distribution. The skewness-adjusted boxplot modifies this interval by taking into account the skewness of the distribution.

$$T1 = 1.5IQR e^{3MC}, T2 = 1.5IQR e^{-4MC} \quad \text{if } MC > 0 \quad (4.25)$$

$$T1 = 1.5IQR e^{4MC}, T2 = 1.5IQR e^{-3MC} \quad \text{if } MC \leq 0 \quad (4.26)$$

Here MC is the *medcouple*, which measures the skewness of the distribution. It is worth to notice that for symmetric distributions MC is zero, thus reducing the formula to the standard Tukeys boxplot technique.

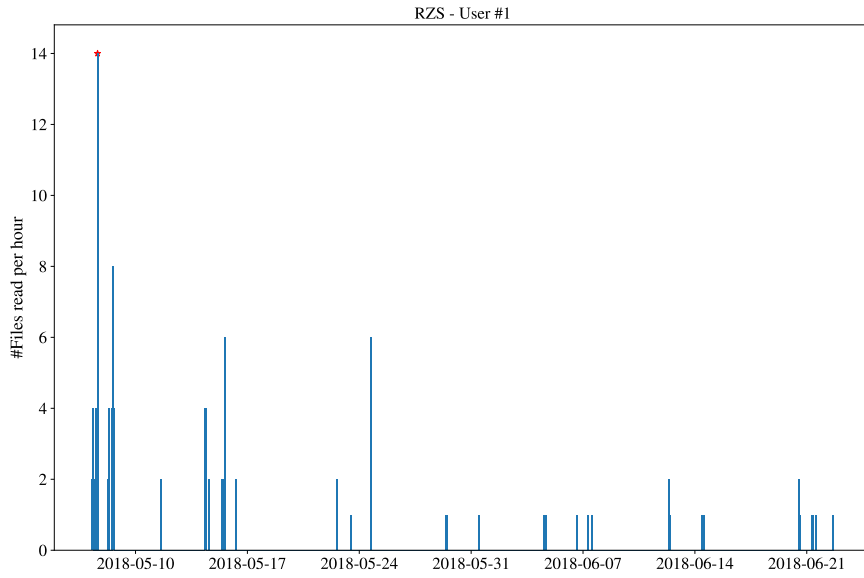
Another technique commonly employed for this task is the Z-Score approach, which aims to assign a score to each element of the sequence, based on simple statistics such as mean and variance. The discrimination between normal observations and outliers is then performed by a threshold. However, as pointed out in [16], mean and variance are not robust statistics, meaning that they are strongly affected by outliers. In the same paper, several alternative techniques that overcome this problem are illustrated. Indeed, median and MAD are presented to be less affected by outliers. The robust z-score is defined as follows.

$$zscore = \frac{(x - median)}{(1.253MeanAD)} \quad \text{if } MAD = 0 \quad (4.27)$$

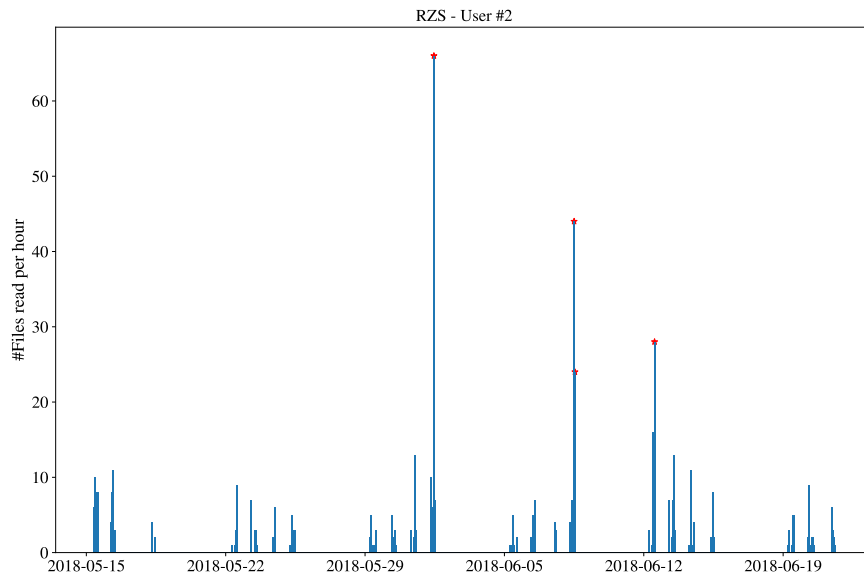
$$zscore = \frac{(x - median)}{(1.486MAD)} \quad \text{if } MAD \neq 0 \quad (4.28)$$

MAD is the median absolute deviation while $MeanAD$ is the mean absolute deviation.

Because of the significantly low number of data, Tukeys boxplot cannot be exploited: with this methodology the data is supposed to fit a Gaussian-like distribution, in both symmetric and skewed variants. On the contrary, Robust Z-Score doesn't make this assumption, which makes it suitable for the available dataset. Figure 4.10 shows the result of the Robust Z-Score with $cutoff = 3.5$.

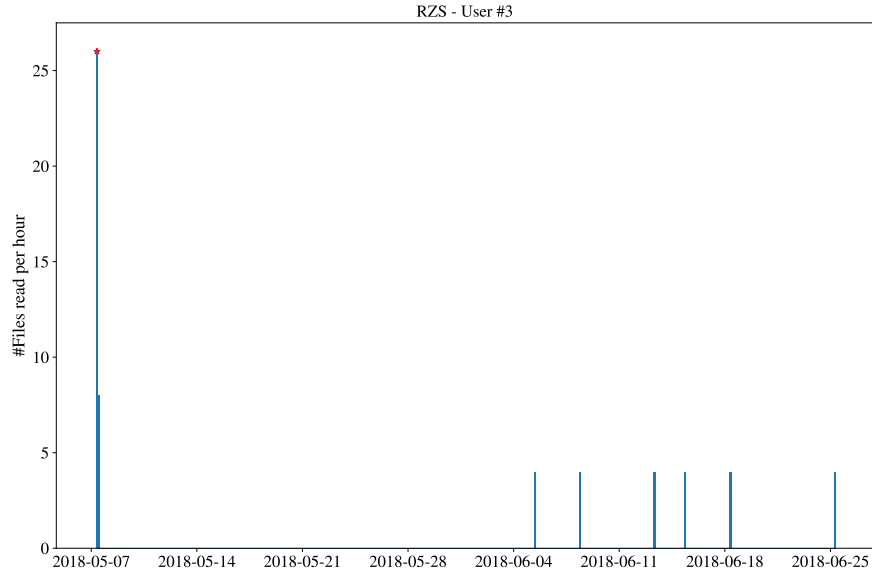


(a)

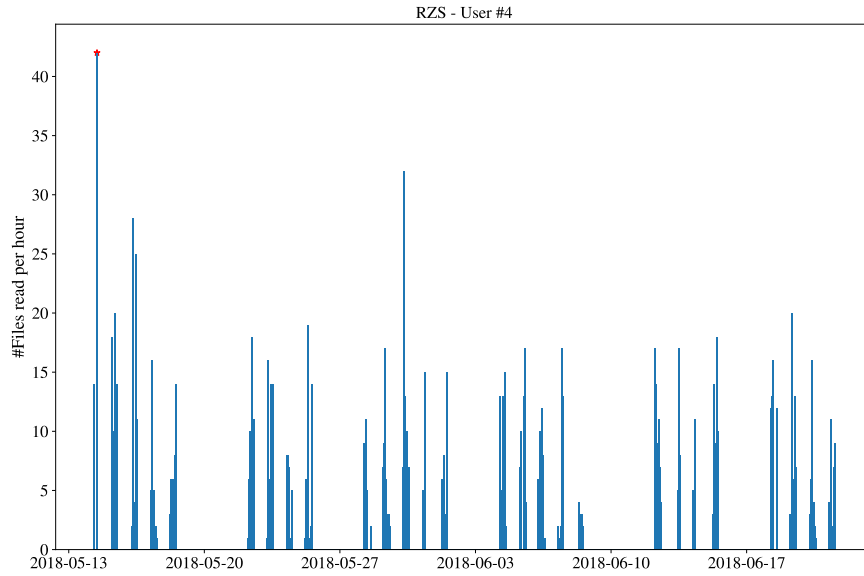


(b)

Figure 4.10: Robust Z-Score with $cutoff = 3.5$ applied to users introduced in Fig. 4.9; anomalous hours are annotated with red asterisks. This figure shows how this method pinpoints "extreme" anomalies.



(c)



(d)

Figure 4.10: Robust Z-Score with $cutoff = 3.5$ applied to users introduced in Fig. 4.9; anomalous hours are annotated with red asterisks. This figure shows how this method pinpoints "extreme" anomalies.

4.5 Error monitoring

Error logs are produced by every kind of applications. They record exceptions and error messages, and are usually highly verbose. This is particularly true for Java based applications, where entry logs consist of Java stack traces which report the chain of methods to address its origin. Despite the benefits of this high level of detail that undoubtedly helps the troubleshooting, it can be hard, specially for a medium-large application, to effectively point out new exceptions and abnormal patterns.

The number of exceptions thrown in a given time frame (e.g. one hour) is a first simple indicator that the system health needs to be deeply investigated. As for access logs, the number of exceptions reflects the utilization of the system, which translates in weekly and daily seasonality. Many exceptions are also thrown by scheduled jobs, which run independently from the number of active users. Unlike access log, predictive models did not perform efficiently for this time series for two main reasons: first of all, less than two months of data is available, secondly, the pattern is highly jagged. As will be further discussed in the result section, simpler outlier techniques have been employed to achieve this task.

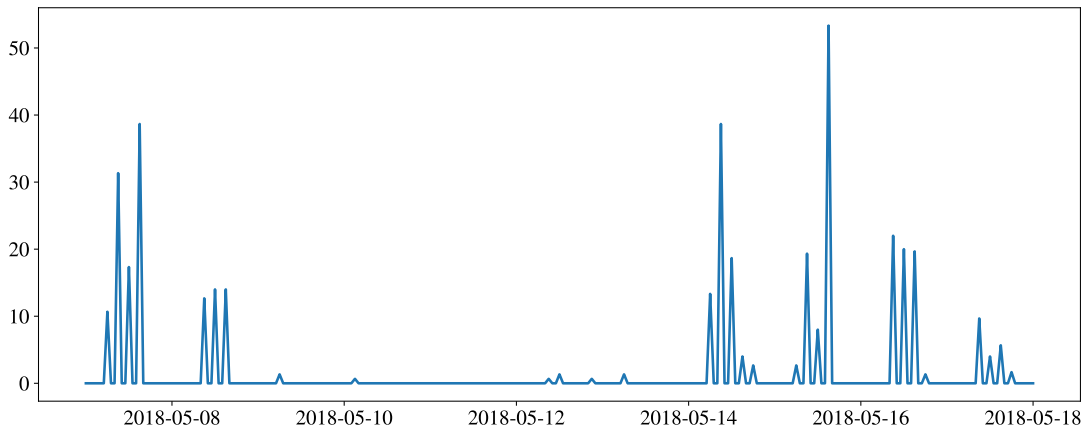


Figure 4.11: Number of exceptions thrown every hour min from 8 May 2018 to 18 May 2018

However, albeit the overall number of exceptions thrown can give a general idea of the system health, it does not help the troubleshooting process. Indeed, it can be more interesting to identify unusual patterns among specific types of exceptions.

Usually, every error log is different from the others, even though they are produced by the same line of code. This is due to the message parameters injected in the messages to better describe the cause of the error. For this reason, the idea is to find a procedure that is able to group together *similar* exceptions, and then applying detective controls on each group. It is a compromise between analyzing the exceptions separately or all together.

The idea behind the clustering of error messages is to group together *similar* exceptions. It is expected to have stack traces raised from the same location grouped in the same cluster, even though the message is quite different. The following two examples represent two cases where the values are substituted with placeholders. Even in these cases, the specializations of these two patterns should belong to the same cluster.

[Example 1]

```
error sending mail from = %email to = %email cc: %email
subject: tft-in %name
```

[Example 2]

```
getdocumenttemplatevo(%d, %language,): there are %d or more
document template for this language!!
```

Another case of interest is when different type of exceptions (e.g. `NullPointerException`, `ClassCastException`) are raised from the same class: they will share the same package and class name, but will be thrown from a different line. In the following example, two stack traces with different exception type and line number should belong to the same cluster.

```
org.springframework.dao.DataIntegrityViolationException
[...] at org.springframework[...](%d)
org.springframework.dao.DuplicateKeyException
[...] at org.springframework[...](%d)
```

Ideally the number of new clusters should reduce after few weeks, and the number of unlabeled exceptions should remain constant. Practically, since the software evolves over time, it's unlikely to zeroes these metrics. Furthermore, as it will be presented in the results, the majority of the unlabeled exceptions are due to message parameters which vary from message to message, which increasing the distance between similar exceptions (e.g. numbers, dates).

Clustering algorithm The procedure implemented to analyze clusters of exceptions is defined as follow. During the training phase, messages are fed into a clustering algorithm, which will assign a label to every entry. The label -1 indicates that the message does not belong to any known cluster. The resulting sets, *clustered_set* and *unclustered_set* are stored in Elasticsearch.

Then, every hour, when new exceptions are thrown, they are tried to be assigned to previously discovered clusters. Some of them, ideally the majority, is successfully assigned, others will be unlabeled. At this point, newly unlabeled exceptions are fed into the clustering algorithm together with previously unlabeled exceptions. At this point, *clustered_set* and *unclustered_set* are merged together with the new results, in particular:

- New clusters are appended to *clustered_set*
- New discarded exceptions are appended to *unclustered_set*
- Previously discarded exceptions that now have a cluster are removed from *unclustered_set*

DBSCAN The described procedure employs DBSCAN [18] to group together similar exceptions, which is a density based clustering algorithm.

Given a set of points in a space, DBSCAN classifies points as

- *Core points*: x is a core point if it is distant less than ϵ from at least $minP$ points. In this case, x and p are said to be *directly reachable*.
- *Reachable points*: x is reachable from p if there's a path of directly reachable core points between them. Since the points in the path must be core points, reachability is not a symmetric propriety, because a non core point can be reachable by a core point, but not the vice versa.
- *Density connected points*: x and p are density connected if there's a point q such that x and p are reachable from it.
- *Outliers*: x is classified as outlier if it is not reachable by any other point the space.

A cluster comprises all the points that are mutually density connected, and all the points that are reachable from any point in the cluster.

The parameters of the algorithm are ϵ and minP , which have to be tuned. Moreover, a distance measure has to be chosen depending on the data. Since the algorithm is going to cluster exceptions, that is text data, Levenshtein distance [19] is used. It is a measure that computes the similarity between two strings by taking into account the number of deletions, insertions and substitutions required to transform one string into the other one. For two strings a and b (of length $|a|$ and $|b|$ respectively), the Levenshtein distance $\mathit{lev}_{a,b}(|a|, |b|)$ is defined as follows.

$$\mathit{lev}_{a,b}(i, j) = \max(i, j) \quad \text{if } \min(i, j) = 0 \quad (4.29)$$

$$\mathit{lev}_{a,b}(i, j) = \min \begin{cases} \mathit{lev}_{a,b}(i-1, j) + 1 \\ \mathit{lev}_{a,b}(i, j-1) + 1 & \text{otherwise} \\ \mathit{lev}_{a,b}(i-1, j-1) + 1_{a_i \neq b_j} \end{cases} \quad (4.30)$$

Here, $1_{a_i \neq b_j}$ is the indicator function, which is equal to 0 when $a_i \neq b_j$ and 1 otherwise. The main drawback of this distance is that to produce the similarity score it has to compare all the characters of the two strings.

[Input1]

```
getdocumenttemplatevo(23, ita, ): there are 2 or more
document template for this language!!
```

[Input2]

```
getdocumenttemplatevo(455, eng,): there are 2 or more
document template for this language!!
```

```
Levenshtein(Input1,Input2) = 6
```

Since stack traces are very likely to be more than one hundred characters long, this dramatically slows down the clustering procedure. Moreover, it is now clear the influence of message parameters: numbers, ids, timestamps increase the distance of two messages.

To overcome the first problem, a shorter message is considered instead of the full stack trace. With the help of the operational team, the interesting portion of the

stack trace is found to be the concatenation of the *exception type* and the *origin* where the exception was thrown. The origin do not consider the line number, in this way two exceptions raised in the same class are more likely to be assigned to the same cluster since the line number wouldn't increase the distance among them. An example of the resulting field is the following.

```
java.lang.NullPointerException\\
at eu.europa.ec.digit.xxx.setMailAsTreated
```

However, not all exceptions are java stack traces. As of the studied dataset, the 45% of the exceptions are a simple error message. The main motivation is that those errors are produced by different modules of the web application, which are not all Java based. This kind of messages does not suffer from the length problem.

	Error length (avg)
Non stack-traces	128
Stack-traces	15600

On the other hand, they present noisy parameters that for the former motivation are handled as follows.

- Numbers are replaced with the symbol %d.
- Emails are replaced with the symbol %email.
- Identifiers are replaced with the symbol %id: in the monitored application ids are in the form *ID-numbers*.

Those substitutions do not cover all the possibilities but only the most frequent ones. An example of the the resulting field is the following.

```
error on automatic pdf conversion callback for %email,
error cause: no matching request exists for response
jobtoken %d;%d;%d and status [success]
```

Anomaly detection on clusters The described procedure has been applied to 45 days of data, divided in 30 days of *training set* and 15 days of *test set*.

The procedure was repeated several times to tune ϵ . Instead, the minimum number of exceptions to create a cluster (*minP*) is chosen to be 2.

Eps	Clusters	Unclustered exceptions	Exceptions per cluster (avg)	Patterns per cluster (avg)	Patterns per cluster (max)
5	49	15	23.5	1.55	15
10	48	14	24	1.60	14
20	44	13	26	1.75	14
30	42	8	27	1.80	14
40	37	6	30	2.05	14
50	33	3	34	2.30	14

Naturally, increasing ϵ reduces both the number of clusters and the number of exceptions that do not belong to any group. However, since this parameter tunes the entropy of the messages within the cluster, the average number of representatives per cluster increases accordingly. For this reason, big values of ϵ (i.e. 40 and 30) have been discarded because the resulting clusters grouped together too different exceptions. The inspection of the clusters lead to the choice of $\epsilon = 20$.

Figure 4.12 depicts the number of new clusters and unlabeled exception from the training, up 10 days.

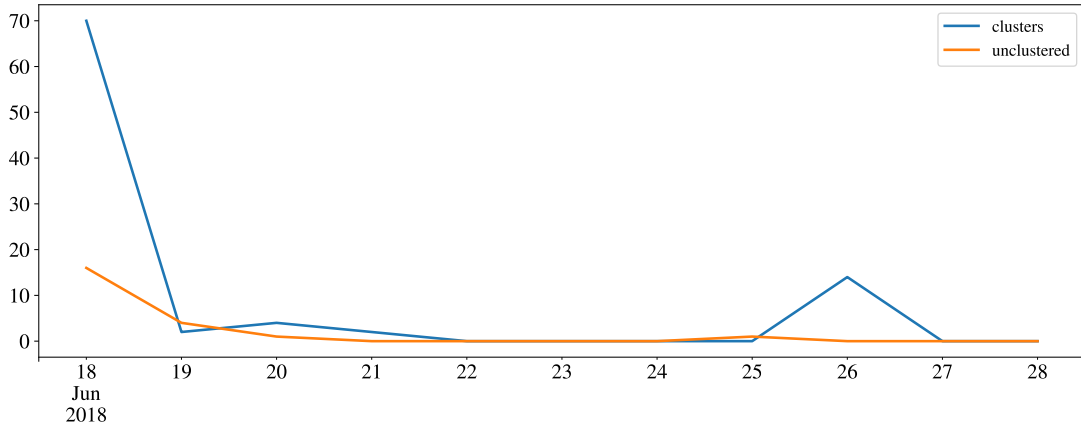


Figure 4.12: Number of new clusters and unclustered exceptions starting from the training date

By analyzing the occurrences of each cluster over time, it's possible to notice that some of them report a regular pattern, but the majority doesn't. For this reason, it was not possible to effectively employ anomaly detection analysis for time series. Furthermore, while some clusters recur daily, others tends to repeat less frequently (e.g. weeks).

Because of this variety and of the low amount of data, simple outlier detection techniques presented in the previous section were exploited to detect hours with an anomalous number of occurrences within the same cluster. However, Since those techniques take into account the sole magnitude of the number of exceptions and do not correlate it with the time, they are meant to detect point anomalies, but not contextual ones.

To apply those kind of techniques, statistics have to be computed for every cluster. This is accomplished by re-computing the statistics every time a new exception is assigned to a previously discovered cluster, and every time a new cluster is discovered.

Figure 4.13 show the results of Robust Z-Score applied on the total amount of exceptions raised in one hour. The default threshold of 3.5 defined in [16] is used. The same approach is used to highlight anomalies per cluster. Results are shown in 4.14. It's worth to notice that while the analysis detects extreme counts only, the second one is more fine grained: for instance, anomalies triggered in *cluster#7* are ignored in Fig. 4.13.

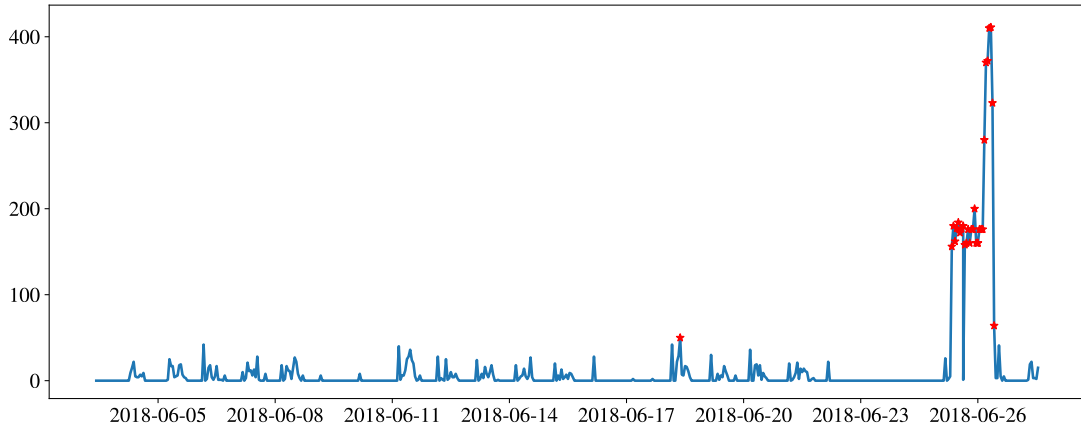
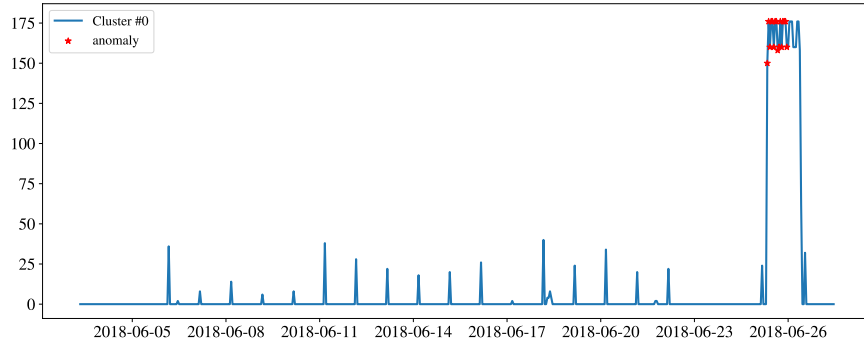
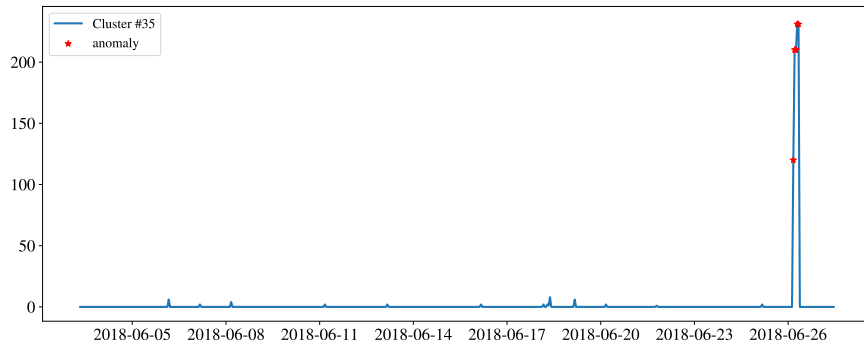


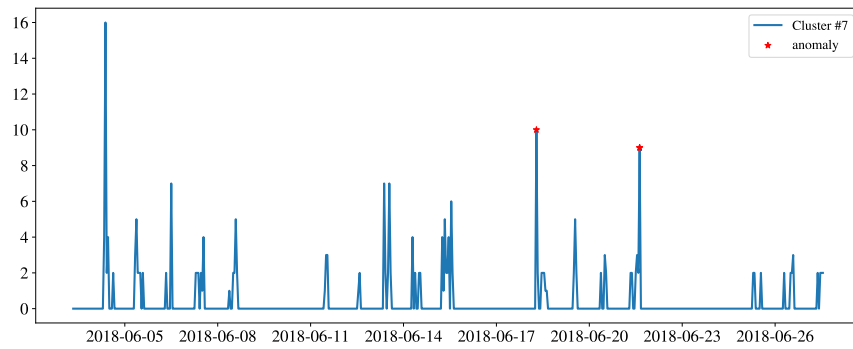
Figure 4.13: Results of Rbust Z-Score on the overall number of exceptions. Anomalous hours are highlighted with red asterisks.



(a)



(b)



(c)

Figure 4.14: Results of robust z-score applied on clusters 0 (a), 35 (b), 7(c)

Chapter 5

Models integration

After having experimented several models and having selected the best, they were deployed in the log analyzer.

Access logs For the reasons explained in section 4.3, 15-minutes analysis and 24-hours analysis are performed by combining a recurrent neural network based architecture for the former and a more lightweight regressive model for the latter.

The first time the log analyzer is executed, the neural network is trained. The training makes use of the preceding 4 months of data, which are split in training, validation and test set following the 60-20-20 rule. The upper bound of 4 months was selected because more data did not prove to improve the performances of the model. With these settings, the selected architecture requires more or less 20 minutes of training, after which the model and the weights are stored locally. During this phase, the training set is used to actually train the network, the validation set is used to evaluate the progress of the training to prevent overfitting, and the test set is used to compute the Gaussian distributions for the detection model.

At inference time, every 15 minutes a new forecast is produced by taking into account the previous *look_back* time steps and the PD of the residual is computed as anomaly score, which is finally compared to the defined threshold. Since the components of the time series do not change over the year, new data is incorporated into the model in an *offline mode*. Every M months the previous model expires and a new model is trained including new data. Here M is a parameter specified by configuration, that is set to 12 by default. By looking at the available 6 months of data, it was possible to notice that the components of the series remained static, thus not requiring any training.

On the contrary, the regressive model used for longer term anomalies does not need to be trained. Every 24 hours the daily aggregation of the previous 60 days of data is used to predict the forecast for the next day. As for the 15-minute analysis, the PD of the residual is compared with a defined threshold.

The following pictures are snapshots taken from Kibana, which show the view available to the end user.

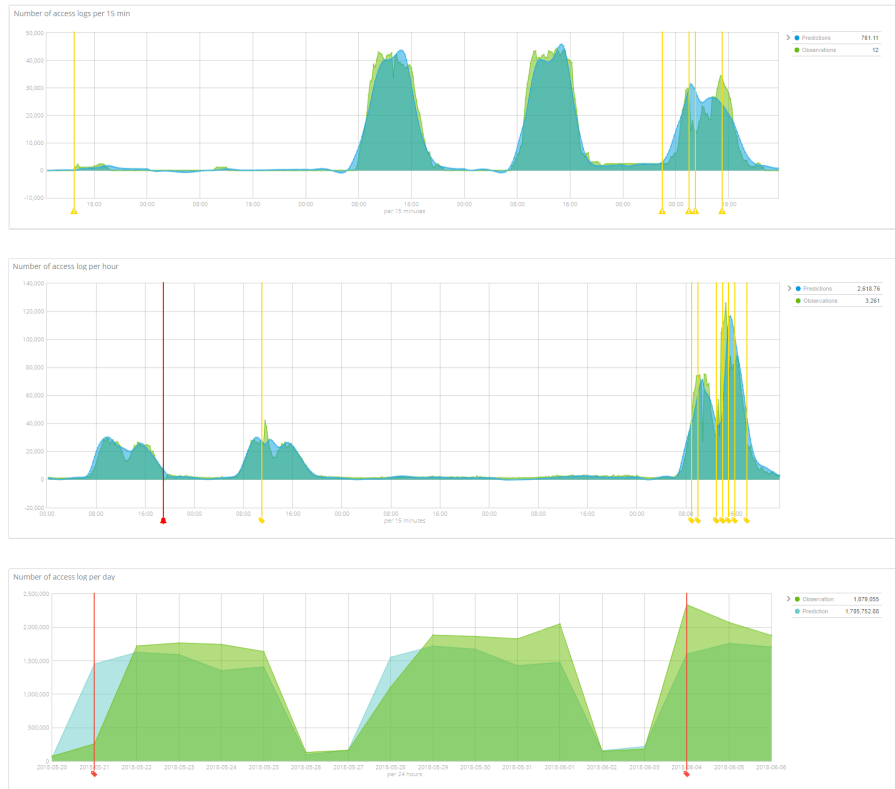


Figure 5.1: Figures (a) and (b) show the results of the 15-min analysis. Fig. (a) highlights a weak anomaly on Saturday, followed by four anomalies on Wednesday, when the requests pattern was clearly unusual. Fig. (b) shows a first (red) anomaly on Thursday evening, which matches a release process, followed by a high peak on Friday and a high number of anomalies on Monday. A posterior analysis revealed that some errors in the process release of Thursday led the application to be heavily loaded on Monday. The Friday peak coincides with the notification of employee promotions. Fig. (c) shows the 24-hours analysis. The system reported a low number of requests due to a Belgian national holiday (Whit Monday) and an overloaded Wednesday, which was probably a false alarm.

Error logs The module that monitors error logs retrieves the exceptions thrown in the previous hour and feed them in the two models described in section 4.5. The first one simply computes the Robust Z-Score related to the number of exceptions received, the second one applies the clustering procedure described in section 4.5 in order to assign or create new clusters and then computes the Robust Z-Score for each group of exceptions. In particular, first of all previous clusters are downloaded from Elasticsearch for the assignment of new data. Then, if some entries among new data are still unassigned, they are given as input of the clustering algorithm along with unclustered exceptions that are also downloaded. As mentioned in section 4.5, a minimum of 20 observations was used as a rule of thumb to compute statistics. For this reason, it is possible that a cluster was not analyzed for anomalies because of its few occurrences. However, when the assignment procedure is over, it might happen that the number of occurrences of a cluster is now above the threshold. For this reason, if a new exception is assigned to a previously discovered cluster, the statistics for that cluster are (re)computed and stored locally.

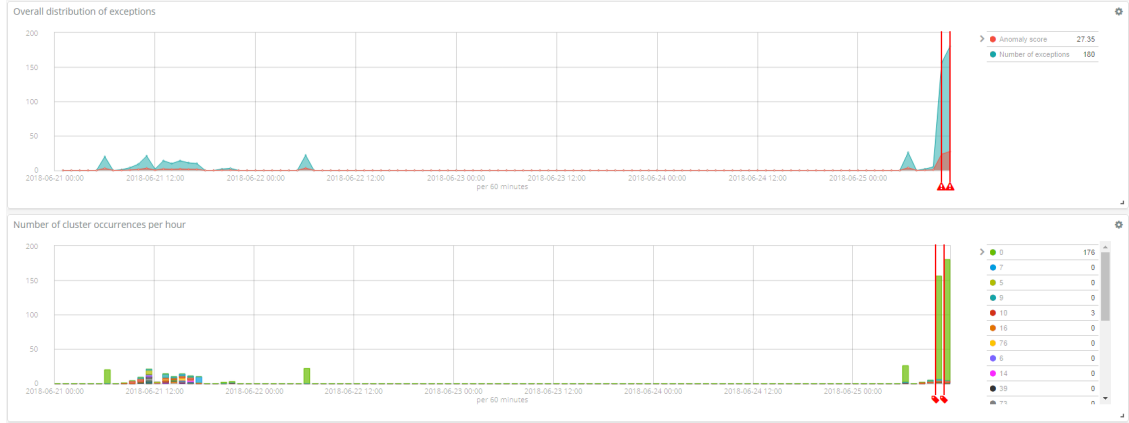


Figure 5.2: Fig. (a) represents the overall number of exceptions, which reports two days particularly full of exceptions. For the same period, Fig. (b) shows how exceptions are split in clusters. In particular, this approach helped to isolate the *cluster#0* as the cause of the anomalies.


File handling File handling analysis reflects the approach adopted for the error log analyzer. Every hour, it retrieves the number of accessed files for each active user. Of these files, only the ones not belonging to him are considered. As per error logs, the Robust Z-Score is computed for each aggregation, and for each involved users statistics are recomputed and stored locally.

Notifications As presented in the application design chapter, every model described so far is integrated with the notification module, which is in charge of providing a meaningful description of the situation. The following picture is a sample sent after the detection of an unusual number of exceptions.

Anomaly Notification

automated-notifications@[REDACTED]

Sent: Tue 26/06/2018 12:12

To:  VENTRELLA Carlo (DIGIT)

Retention Policy: [REDACTED]

1 anomalies found.

- Cluster n. 35 registered 231 exceptions thrown in the hour 2018-06-26T08:00:00+00:00. (anomaly score: 103.94222933708123)

Threshold used: 4.0

Figure 5.3: Email sample which notifies an unexpected number of exceptions for *cluster#35*. The anomaly score along with the threshold used are included in order to provide a measure of the seriousness of the anomaly.

Chapter 6

Deployments

6.1 PABS

The prototype described so far has been deployed on PABS. The entire pipeline (ELK, Log Analyzer and Log Retriever) has been installed on a single virtual machine, characterized by 16GB of RAM, 100 GB of storage and a Red Hat distribution. The pipeline was configured to look for new logs every 5 minutes, while different frequencies were defined for the different types of detective controls. Access logs are monitored every 30 minutes for short term anomalies and 24 hours otherwise; error logs are checked every hour while 24 hours were defined for file access monitoring.

6.2 Sysper

During the last part of the stage, the prototype has been installed to monitor SYSPER (Système de gestion du Personnel), which is the information system used by all the employee of the European Commission, integrating all human resources managements functions. For this reason, it has proven to be a good way to test the performances of the prototype on a larger system. Indeed, it counted 46000 access logs per hour.

For this use case, system availability is the only aspect to be monitored, by means of 15-min and 24-hours checks on the access logs. As of PABS, a VM with same characteristics hosts the prototype. The installation on the new information system required one working day, needed to install and configure the whole pipeline.

Chapter 7

Conclusions

The project concerned the development of a logging monitoring system capable of managing high quantities of data and to perform security checks in order to detect potential anomalies to help both the business and the operational teams.

The ELK stack has been installed and configured to handle the data pipeline, from the ingestion to the presentation for the end user. In addition to the canonical modules comprised in the stack (Elasticsearch, Kibana and Logstash) some custom modules have been developed to fill some gaps introduced by the requirements.

Once the logs are available in the local data base, anomaly detection techniques have been employed to monitor the system availability, to help the troubleshooting process, and to observe sensitive dossiers.

The application availability is monitored through the implementation of two anomaly detection techniques, which monitor the number of access requests within two different windows, 15 minutes and 24 hours. The former employs an architecture which relies on long-short term memory (LSTM), the latter makes use of Prophet. Both are used as predictive models, whose residuals are then modeled on Gaussian distributions to produce the anomaly score. Thresholds are defined by the users of the application to discriminate between normal and non-normal anomalies.

The troubleshooting process involved the hourly analysis of error logs. First of all, a simple technique based on Robust Z-Score was employed to monitor the number of exceptions thrown. A more fine grained approach based on DBSCAN with Levenshtein distance was applied to group exceptions together in order to ease both the analysis and the proactive fault handling. As for the overall number of exceptions, Robust Z-Score is used as an anomaly score. Even in this case, a threshold

is defined by the users.

A simple outlier detection technique is also used to monitor the hourly number of time a user accesses dossiers not assigned to him.

The logs and the results of the analysis are then made available to the end user through Kibana, by means of dashboards and visualizations. A notification system is also included in order to provide a meaningful report of the anomalies being found.

7.1 Future work

The main aspect which needs to be further improved concerns the deployment in production, which was not covered due to time limitations. On one side there's the ELK stack, which is easy to install and configure. On the other side, the log analyzer requires a high level of customization which depends on the logs being monitored.

At the time of writing, the configuration of this module is entirely done via config files, which is neither intuitive nor error safe. An interesting improvement would be to deploy the log analyzer as a Kibana plugin, which would offer a customization page inside the Kibana environment.

In order to be in line with the Open Source philosophy of the project, an additional step would be to deploy the Log Retriever as a Beat, the official Logstash plugin in charge of shipping data from the machines to Logstash or, eventually, directly to Elasticsearch. This would introduce a feature that at the time of writing is not covered by the available beats and would benefit of the support from the community supporting the ELK stack.

Another area of improvement would be the management of holidays in the anomaly detection technique for system availability. At the time of writing, holidays are not considered, thus producing false positives because of the low system utilization with respect to standard working days. By taking into consideration the EC calendar of the passed and next holidays, this behavior can be refined.

Bibliography

- [1] Varun Chandola, Arindam Banerjee, and Vipin Kumar. *Anomaly Detection : A Survey* ACM Comput. Surv. 41, 3, Article 15 (July 2009).
- [2] Ted Dunning and Ellen Friedman *Practical Machine Learning - A New Look at Anomaly Detection* O'Reilly Media, Inc., 1st edition, 2014.
- [3] Hyndman, Rob J., and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2014.
- [4] Sean J. Taylor and Benjamin Letham. *Forecasting at Scale* Facebook, Menlo Park, California, United States, 2017
- [5] M. Gupta, J. Gao, C. C. Aggarwal, and J. Han *Outlier Detection for Temporal Data: A Survey* IEEE Transactions on Knowledge and Data Engineering, vol. 26, no. 9, pp. 2250-2267, Sept 2014. doi: 10.1109/TKDE.2013.184
- [6] D. J. Hill and B. S. Minsker *Anomaly Detection in Streaming Environmental Sensor Data: A Data-Driven Modeling Approach* Environmental Modelling & Software, vol. 25, no. 9, pp. 1014-1022, 2010
- [7] R. S. Tsay, D. Pena, and A. E. Pankratz *Outliers in Multivariate Time Series* Biometrika, vol. 87, no. 4, pp. 789-804, 2000
- [8] Jinwon An and Sungzoon Cho. *Variational Autoencoder based Anomaly Detection using Reconstruction Probability*. Technical Report. SNU Data Mining Center. 118 pages, 2015
- [9] Haowen Xu and Wenxiao Chen and Nengwen Zhao and Zeyan Li and Jiahao Bu and Zhihan Li and Ying Liu and Youjian Zhao and Dan Pei and Yang Feng and Jie Chen and Zhaogang Wang and Honglin Qiao *Unsupervised Anomaly Detection via Variational Auto-Encoder for Seasonal* arXiv:1802.03903
- [10] Sepp Hochreiter, Joshua Bengio, Paolo Frasconi and Jurgen Schmidhuber *Gradient Flow in Recurrent Nets: the difficulty of training Long-Term Dependencies*
- [11] Hochreiter and Schmidhuber *Long Short Term Memory* Neural computation, vol. 9, no. 8, pp. 1735-1780, 1997
- [12] Christopher Olah *Understanding LSTM Networks* Olah's blog. Accessed 14 August 2018 <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

- [13] F. A. Gers, J. Schmidhuber and F. Cummins *Learning to forget: Continual Prediction with LSTM* Neural computation, vol. 12, no. 10, pp. 2451-2471, 2000.
- [14] Hermans, Michiel and Schrauwen, Benjamin *Training and Analyzing Deep Recurrent Neural Networks* Advances in Neural Information Processing Systems 26, pp. 190-198, 2013
- [15] Hinton, Srivastava, Krizhevsky, Sutskever, and Salakhutdinov *Improving neural networks by preventing co-adaptation of feature detectors* arXiv:1207.0580
- [16] Rousseeuw and Hubert *Anomaly Detection by Robust Statistics* arXiv:1707.09752v2
- [17] Yarın Gal and Zoubin Ghahramani *A Theoretically Grounded Application of Dropout in Recurrent Neural Networks* arXiv:1512.05287
- [18] Martin Ester, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu *A density-based algorithm for discovering clusters in large spatial databases with noise* AAAI Press. pp. 226-231. CiteSeerX 10.1.1.121.9220
- [19] Levenshtein, V. I. *Binary codes capable of correcting deletions, insertions, and reversals* Soviet Physics Doklady, Vol. 10, p.707
- [20] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The L^AT_EX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.