ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Dipartimento di Informatica – Scienze e Ingegneria DISI

TESI DI LAUREA

in

Intelligent Systems M

ANOMALY DETECTION IN HPC SYSTEMS

CANDIDATO:
Kevin Leto

RELATORE:
Prof.ssa Michela Milano

CORRELATORE:
Dott. Andrea Borghesi

Anno Accademico 2018 - 2019

Sessione II

SUMMARY

INTRODUCTION

Nowadays, more and more medium and large companies need to analyse large amounts of data to extract useful information and improve their gains. These data – often called "Big Data" – can be originated in very different ways. For instance, a manufactured company can extract Big Data from one of its production lines to verify if everything is working well or discover witch part of the line can be improved. Again, a banking company can decide to analyse its clients to predict who most likely will return the granted loan and who will not.
In order to analyse Big Data, it is necessary a dedicated analysis infrastructure that provides very high performance and scalability. These systems are very expensive, especially for medium companies that cannot afford the huge infrastructure cost.

In this scenario, specific companies – called cloud service providers – are born to offer computing power as a service. Medium and large companies can buy computing power on demand so that they can make Big Data analysis without paying the infrastructure cost. Cloud service providers use sophisticated computing systems – such as HPC (High-Performance Computing) systems – that are formed by a large number of computing nodes connected each other to distribute the processing cost and obtain results as fast as possible.
Since computational power is sold as a service, HPC systems need to have high availability and reliability. Anomalies in computing nodes affect these properties, so they must be avoided or at least reduced as much as possible. Granting these properties means to find out in which nodes anomalies are present and resolve them quickly to restore the right QoS (Quality of Service).

This study has the task of detecting anomalies on HPC computing nodes in an automatic way using Machine Learning (ML) techniques using Python as the

programming language. The HPC system that we are going to analyse is called MARCONI and it is own by the Cineca non-profit consortium of Bologna. Cineca is made up of 70 Italian universities, four national research centres, and the Ministry of Universities and Research (MIUR). Cineca represents one of the most powerful supercomputing centres for scientific research in Italy.

This thesis is structured as follow:
The **first chapter** will present the HPC environment explaining what HPC systems are and it will add details about the "MARCONI" system.
The **second chapter** will introduce what is an anomaly, what does it mean in HPC systems and which techniques are available for anomaly detection.
The **third chapter** will be focussed on the collecting and pre-processing tasks of the required data for anomaly detection.
The **fourth chapter** will explain which machine learning techniques will be used to analyse the pre-processed data.
The **fifth chapter** will unfold the resulting data derived from the ML analysis.
The **sixth – and last – chapter** will summarise the whole process and list possible future developments of this project.

1 HPC SYSTEMS

A high-performance computing system is generally used by scientists and engineers for tackling problems that are impossible to resolve on standard computers. The required computing power is so high that on a personal computer they will take hours, days or even more to solve them.

1.1 ARCHITECTURE

HPC computers are typically formed by multiple standard computers – called nodes – connected together by a wired network. Their size can range between the equivalent of a few personal computers to thousands of them. Their construction cost is so high that often they are shared between different departments and institutions.

HPC systems usually comprehend different types of nodes. Each node type is specialized for a specific task because each task requires a different computational power. Login nodes are those throw which users can interact with the system. They require a small amount of power because no hard computations are done there. Instead, computing nodes are those in which the real computation is done and they require higher performance.

1.1.1 Nodes

As aforementioned, a single node can be seen as a personal computer. Its processor contains multiple cores and each core contains a Floating-Point Unit (FPU) which is responsible for performing the computation. Each node contains caches that are used as temporal storage to speed up computations. The more cores are available and the more powerful are their FPUs, the higher performance has the node.

Each node can use three types of storage: caches are small memories used to speed up the computation, private disks are larger memories (typically in range 64-254GB) used to store partial results and a shared file system used to read input and permanently store final results.

1.2 USERS AND JOB SUBMISSION

Users need to submit their computations – named jobs. Jobs are generally submitted from login nodes and executed on computing nodes; this because – as shown in **Figure 1Figure 1: General HPC system architecture** – direct access to computing nodes is typically forbidden. So, it is required a software module that takes users submission requests and allocates them on computing nodes. This module is called "scheduler" or "batch system". Through it, the user can send job submissions, check jobs status and delete submitted requests.

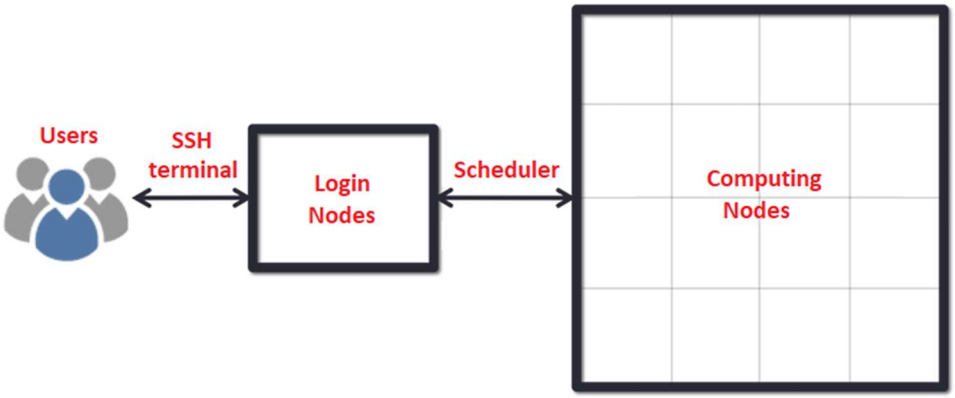

Figure 1: General HPC system architecture.

The scheduler has also the task of allocating jobs on computing nodes preventing their overload. To do that, each submitted job is queued in a specific queue based on its priority. When a computing node is available, the compatible job with the higher priority is extracted from its queue and allocated to the node. Usually, not all jobs are compatible with all nodes; this because potentially every job needs different resources that may not be available on nodes. Indeed, users can customise the requirements of their job. For instance, they can specify the number of required cores, the memory size and also the requested execution time. If this is not done the scheduler uses default options.

1.3 JOB LIFECYCLE

As shown in **Figure 2Figure 2**, the job lifecycle starts with the user submission operation that creates a job request for the scheduler. The scheduler creates the job and queues it in the right queue, so the job passes into the **Pending** state. When a compliant node is available, the scheduler allocates it to the job. In this way, the job

goes in the **Running** state and starts the computation. At the end of the computation, the job has finished its work and goes in the **Completing** state and the lifecycle is completed. If some errors occur during the process, the job is moved on the **Failed** state and its lifecycle ends in advance.

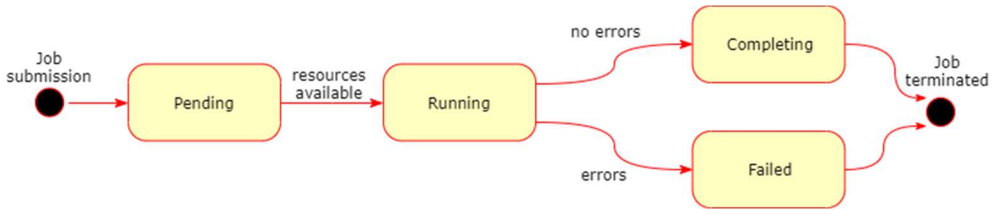

Figure 2: Job lifecycle.

1.4 THE MARCONI SYSTEM

The Marconi HPC system is the bigger supercomputer owned by the CINECA consortium. At the moment it is formed by almost 6,800 computing nodes divided into two partitions named KnightLandings and SkyLake. The KnightLandings partition owns nodes with a single 68-cores processor each, while the Skylake partition owns nodes with two 24-cores processors each. The global computational power is estimated in 20 PFlop/s, this means that MARCONI can perform $20 * 10^{15}$ (twenty thousand million million) floating-point operations per second. This leads him to be the 19th supercomputer in the world for computational power [1].

2 ANOMALY DETECTION

An anomaly is a data that does not fit the general pattern of a given dataset and anomaly detection is the process of identifying these unexpected events or behaviours.

Anomaly detection is based on two fundamental assumptions:

- The number of anomalies is small respect the dataset size that means that anomalies are rare events.
- Their features differ from normal instances significantly: looking at their features is enough to distinguish them from the rest of the dataset.

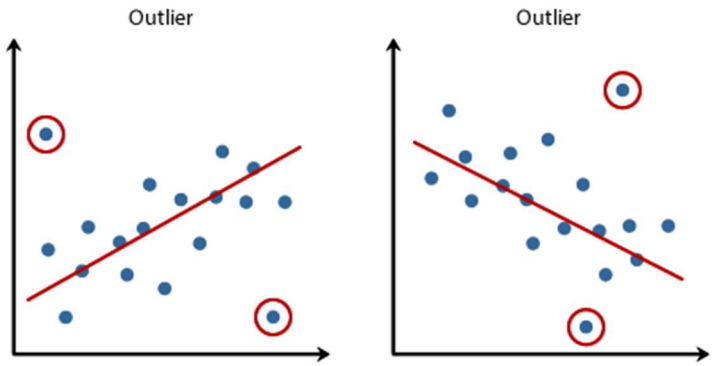

Figure 3: Outliers detection example.

2.1 MOTIVATIONS

Anomaly detection often is used before statistical data analysis to clean up the dataset from outliers. This can considerably improve the performance of the statistical model because it creates a more homogeneous dataset that can be analysed easily.

In HPC systems, the anomaly detection process has another goal. Here, anomalies mean un-normal system states, which can lead to imminent failures or breakdowns of a given component. Failures or breakdowns cause a reduction of availability and reliability, that must be avoided. Knowing in advance which components are no longer acting in their optimal states, it is possible to program maintenance activity to restore the full operating potential of the system without performance loss.

2.2 DIFFERENT APPROACHES

The anomaly detection activity can be done with different techniques. Each requires a different effort and computing cost to be used. The computing cost is based on the algorithm you want to use, while the effort is based on which data must be collected to use the selected algorithm.

A simple technique requires little efforts and small computing costs but usually leads to a large number of false positives (normal events classified as anomalies) and false negative (anomalies classified as normal events). At the contrary, a complex technique requires bigger efforts and much more computing power but obtains a very small number of misclassified instances. Let's analyse now the main techniques used for anomaly detection.

2.2.1 Statistical models

A simple technique that can be used for anomaly detection is to apply statistical models to the data to easily detect which instances do not fit the global trend or behaviour. Statistical models are divided into univariate and multivariate models.

2.2.1.1 Univariate analysis

Univariate analysis tries to detect anomalies looking on one single feature at a time. To do that, it is necessary to establish the minimum and the maximum values that each feature can assume to be considered in a good state.

An instance is considered as an anomaly, even if only one feature assumes a value outside of its range. Selecting the right boundaries is extremely important to reduce miss-classification error. It can be done using statistics on the values of the features like mean and standard deviation.

Univariate analysis requires very little computing power to be used but it has big disadvantages. Firstly, it assumes that features are independent the one from the others and this is typically not true. Secondly, computing correlations between different features is not possible; this causes the loss of important patterns in the data that are crucial to discover anomalies. Lastly, a single feature can determine an anomaly even if it is just outside of its correct range.

These disadvantages make this technique impossible to use in practice, except for specific and very limited cases.

2.2.1.2 Multivariate analysis

Multivariate analysis overcomes the limitation of the univariate. Here, outliers are seen as combined unusual scores on at least two features. If the global score is greater than a given threshold the instance is classified as an anomaly, otherwise it represents a normal behaviour.

This technique is more precise because an instance is evaluated as a whole object instead of multiple individual features. More complex patterns can be detected compared to the univariate analysis.

2.2.1.3 Univariate and multivariate analysis applied to anomaly detection

A real use case of univariate and multivariate anomaly detection can be found in the Ciocarli et al. paper [2]. They apply anomaly detection in cellular networks to discover partial and complete degradations in cell-service performance. They proposed a novel ensemble method for modelling cell behaviour that builds adaptive models to maintain the correct level of performance. The approach was tested on real cellular network data and the obtained results were better than traditional univariate and multivariate analysis.

2.2.2 Machine Learning techniques

Another possibility is to use Machine Learning (ML) techniques. Machine learning algorithms use computational methods to "learn" information directly from data without relying on a predetermined equation as a model. The algorithms adaptively improve their performance as the number of samples available for learning increases. Machine learning techniques are divided into different categories based on how they work. Let's now give a clear and detailed explanation of the main categories providing also some use cases in which those techniques have been applied.

2.2.2.1 Supervised learning

In supervised learning, the task of an algorithm is to find a mapping function between the input and the output. From input data, it tries to create a model that reproduces the requested output. This process is called training or training phase.

Supervised learning is commonly used for classification tasks, where new observations must be labelled as one of the possible categories.

To train the model, it is required to know the expected output for each provided input.

For each input sample, the model processes the input and generates an output. If the model output differs from the real one, the mapping function is adjusted to reproduce it perfectly. When the whole dataset is processed, the training ends.

The second phase is called "on-line" phase. Here, the model is used to predict the output on un-labelled data.

The whole process is represented in the following scheme (Figure 4).

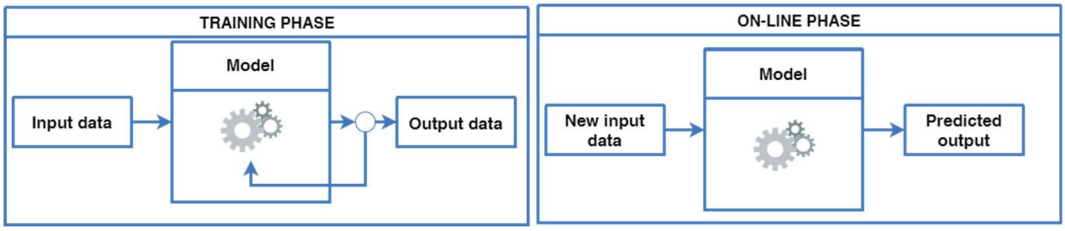

Figure 4: Supervised Machine Learning model.

This technique requires labelled data that are expensive to be obtained. Labels must be well-known and pre-defined, but the obtained mapping function is well defined and extremely precise.

2.2.2.2 Supervised approach applied to anomaly detection

A real application of supervised learning for anomaly detection can be seen in Wang et al. study [3]. They create a supervised model for fault diagnosis in power systems using sparse stacked autoencoder (SSAE) neural network, Principal Component Analysis (PCA) and Support Vector Machine (SVM). Their network was able to recognise three different classes: one for normal behaviours and two for anomalous states. SSAE network and PCA are used to extract salient features from raw data to speed up the computation and reduce the computing costs. The accuracy of the obtained model reaches 90%.

Another example is represented by Tuncer et al. approach [4]. They propose a method to diagnose performance variations in HPC systems (so to distinguish normal from anomalous behaviours). They trained several different machine learning algorithms with performance metrics extracted from HPC nodes. In the end, the algorithm that has provided the most meaningful results is the Random Forest classifier that overcomes the results obtained by previous similar works.

2.2.2.3 Unsupervised learning

In unsupervised learning, the task of an algorithm is to find unknown patterns in the data without pre-existing labels. Unsupervised learning algorithms identify similarities in the data and react based on the presence or absence of such similarities. Usually, it is used for dimensionality reduction problems and clustering tasks (**Figure 5**). Dimensionality reduction problems are those in which data has a large number of features and the algorithm selects in some way a reduced set of features (already existing but also new ones). While a clustering algorithm uses the discovered patterns in the data to split the dataset into sub-groups based on the similarities of their features.

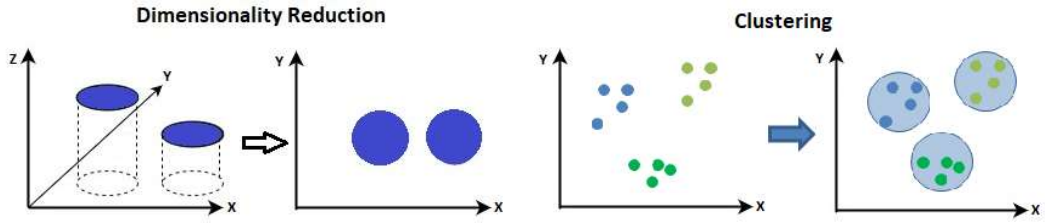

Figure 5: Dimensionality reduction and clustering techniques [5] [6].

Unsupervised learning requires unlabelled data that are simple to obtain and very interesting patterns can be discovered from them. Unfortunately, it has bad performance on specific data.

2.2.2.4 Unsupervised approach applied to anomaly detection

An extensive application of unsupervised learning applied to anomaly detection is done by Goldstein et al. [7]. In their study, 19 different unsupervised algorithms are evaluated on 10 different datasets belonging to various application domains.

Their study shows, for the first time, the strengths and the vulnerabilities of each approach. It reveals that k-nearest-neighbour (kNN) based algorithms perform better in most cases when compared to clustering algorithms, but kNN algorithms require the choice of the number of clusters (k) that can lead to very different outcomes.

A very different strategy is applied by Dani et al. [8]. They applied the k-Means clustering algorithm on log files to detect anomalies with the assumption that anomalous messages are formatted differently from normal messages. The focus of their work was exclusively on anomalies recognizable by the node itself, since it has

to generate the log messages used to identify faults. This approach significantly speeds up troubleshooting and failure analysis by clustering heterogeneous logs.

2.2.2.5 Reinforcement learning

Reinforcement learning is a machine learning technique in which an agent performs actions in the environment. For each performed action, the agent receives a reward. The reward is positive if the action is considered "good" and negative instead.
The agent's goal is to maximise the global reward and it learns how to act to increase the reward in particular situations.

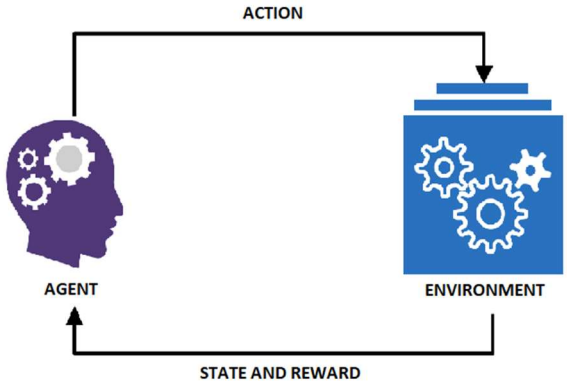

Figure 6: Reinforcement learning model [9].

Reinforcement learning is used very often to resolve planning problems in which the interaction between the agent and the environment is required.

2.2.2.6 Reinforcement learning applied to anomaly detection

Reinforcement learning techniques can be applied to anomaly detection activities. Lu et al. [10] study the development of an anomaly detection system to prevent the motor of a drone from operating at anomalous temperatures. The temperature of the motor is recorded using sensors and then an agent is trained to adjust motor speed and direction to allow the secure landing of the drone. The efficacy of this approach was confirmed by experimental results.

2.3 Our approach

The approach used in this thesis is based on machine learning techniques. In particular, data analysis will be done into two separate phases. At first, we will reduce the data dimensionality using an unsupervised technique based on autoencoders and then we will classify our data using a neural network in a supervised way.

2.3.1 Dimensionality reduction

In the unsupervised phase, a particular neural network model called autoencoder is used to decrease the data dimensionality.

Dimensionality reduction techniques are extremely helpful in machine learning when dealing with a large number of variables because they significantly improve the learning algorithm performance.

A huge problem when dealing with high-dimensional data is called "the curse of dimensionality" (Figure 7). It declares that the number of the required data samples, to achieve a given accuracy, grows exponentially with the number of dimensions.

Typically, the dataset size is fixed so to obtain a higher accuracy it is necessary to reduce the number of features.

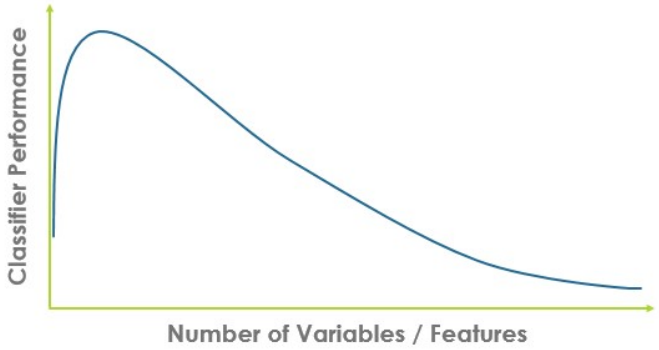

Figure 7: The curse of dimensionality chart [11].

There are two main families of algorithms used for dimensionality reduction:

- **Feature selection**: features are selected among the starting ones in a way to maintain as much as possible the original data variability.
- **Feature extraction**: features are created as combinations of the starting ones to accurately and completely describe the original data set.

2.3.1.1 Autoencoder

Autoencoders belong to the feature extraction family. They are feed-forward neural networks where the input is the same as the output. They aim to compress input data into a lower-dimensional code and then reconstruct it as output.

Autoencoders are formed by three main components (Figure 8):

- **Encoder**: a fully-connected neural network with an arbitrary number of hidden layers that starts from input data and generates the code through compression.
- **Code**: a lossy compression of the input data, also called "latent-space representation".
- **Decoder**: a fully-connected neural network, typically the mirror image of the encoder, that takes the code and generates the output.

Encoder and decoder architectures can be modelled as wish with the unique constraint to have the same dimension for the input and the output. An example is provided below (**Figure 8**).

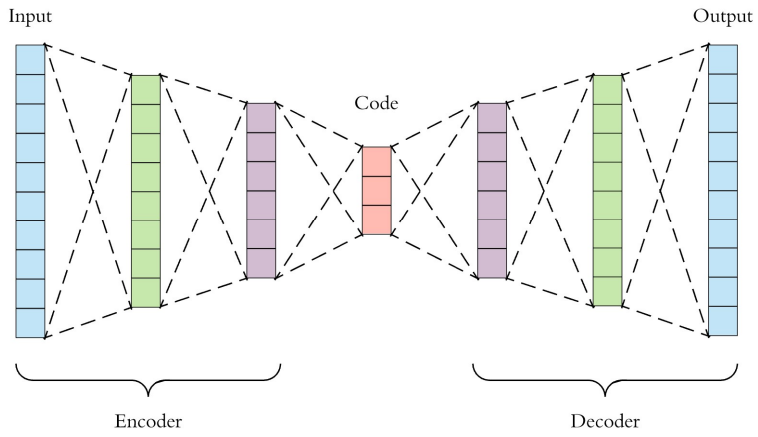

Figure 8: Autoencoder architecture [12].

Four hyper-parameters can be set and tuned in autoencoders:

- **Code size**: the number of nodes in the middle layer. A small size implies more compression but also more data loss.
- **The number of hidden layers**: a deep neural network can learn more complex features but too many layers may cause overfitting.
- **The number of neurons in hidden layers**: the number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder (stacked autoencoder). The symmetry between encoder and decoder is not mandatory but is typically present.

- **Loss function**: autoencoders are trained using back-propagation, so a loss function is needed to evaluate differences between the input and the produced output. Depending on data, the main loss functions used are the Mean Squared Error (MSE) and the binary cross-entropy.

For the autoencoder implementation, we will use the Keras python library that provides APIs to create and manage every type of neural network model, included autoencoders.

2.3.2 Data classification

In the supervised phase, compressed data (that derives from the dimensionality reduction step) are associated with numerical labels (normal/anomalous states respectively "0" and "1") and are feed to a binary classifier.
Binary classifiers are general neural networks that can distinguish individuals among the two classes (Figure 9), reproducing the associated labels as the output.

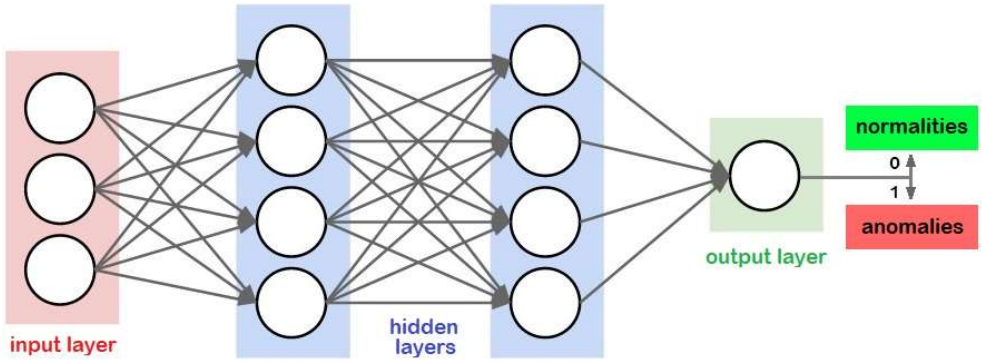

Figure 9: Binary classifier architecture [13].

As for autoencoders, the number of hidden layers and their neurons number can be adjusted by necessity. Usually, the binary cross-entropy loss function provides better results in binary classification problems.

3 DATA PRE-PROCESSING

Before analysing any type of data, it is necessary a pre-processing activity in which data is extracted, cleaned and transformed to reach a standard format that can be easily analysed. This process is quite similar to the ETL process (Extraction, Transformation and Loading) used in data mining for data warehouse purposes.
The difference between our approach and the ETL process is that we do not have a centralised database for storing pre-processed data, so they are just saved on disks.
Summarising, the steps we are going to follow will be (also shown in **Figure 10**):

1. **Extraction:** raw data are extracted from the data sources.
2. **Transformation:** raw data are transformed into a more compressed format, data sampling is aligned to 5 minutes and the resulting data are merged. Finally, a normalization step is applied to create a common scale for each feature.
3. **Store:** normalized data are permanently saved to allow future analysis.

Figure 10: Pre-processing steps: Extraction, Transformation and Store.

For this project, we have decided to analyse data coming from a single HPC node (r183c12s04) on a period of 4 weeks (from October 7th to November 4th 2019).

3.1 DATA SOURCES

Raw data are collected through a real-time monitoring framework called Examon. This infrastructure collects data coming from heterogeneous sources (both physical

sensors and software modules) and stores them in a single database. In this way, it provides uniform access to the whole data collection (Figure 11).

The primary components of the Examon infrastructure are several low-overhead software daemons running along with the computing nodes, tightly coupled with them. These daemons monitor many performance and utilization metrics and the power consumption of each node [14].

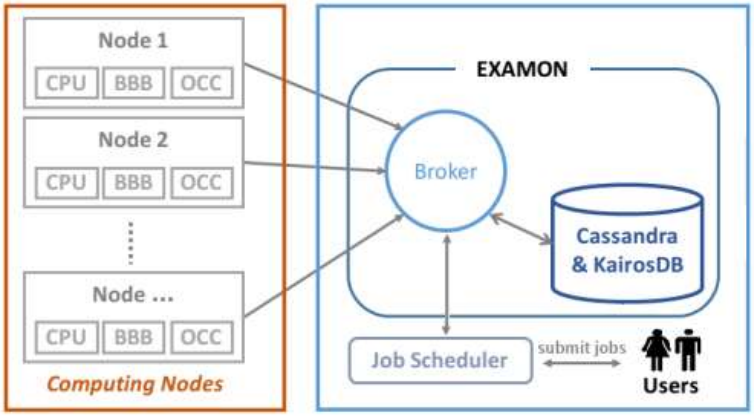

Figure 11: Examon architecture [14].

The extracted metrics are grouped into "plugins" based on their origin. Each plugin gives information about a different aspect of the node.

Let's now analyse the plugins that are used in this project:

- **Ganglia:** it provides CPU statistics (speed, frequency, idle time, …) and system load state (free memory on disk and RAM, available space in buffers, boot time, …).
- **Confluent:** it gives information about the power consumption, the temperature of each component, the disks and the communication network.
- **Nagios:** it collects data about the general state of the node. When an anomaly is manually detected, it is registered here with a numerical value that goes from 0 to 3 and a text description. The numerical score represents the urgency level for technical intervention. If it is low (0 or 1) it represents normal behaviours or small variations of them that do not affect the node performance. Instead, a higher value (2 or 3) means a critical state that must be resolved as soon as possible (for example, it is possible that a node remains disconnected from the network because a link failure occurs).

3.2 DATA EXTRACTION

The data extraction process is done through the Examon-client plugin, a software module – developed by Francesco Beneventi – that allows users to interact with Examon using a SQL-like language.

After a few setup lines (**Figure 12**), the Examon-client plugin is ready to handle requests. The user can specify different parameters to match its extraction intent. Among others, it can choose from which plugin the data must be extracted, which node has to be considered and for which time period. **Figure 13** shows an example of a request.

```
KAIROSDB_SERVER = '###.###.###.###'
KAIROSDB_PORT = '###'
USER = 'user'
PWD = 'password'

ex = Examon(KAIROSDB_SERVER, port=KAIROSDB_PORT, user=USER, password=PWD, verbose=False, proxy=True)
sq = ExamonQL(ex)
```

Figure 12: Examon-client plugin setup.

```
data = sq.SELECT('state','description') \
         .FROM('*') \
         .WHERE(plugin="nagios_pub", node="r18c12s04") \
         .TSTART("01-12-2019 08:00:00") \
         .TSTOP("01-12-2019 10:00:00") \
         .execute()
```

Figure 13: Examon-client query example.

In our context, three different extractions are made – one for each plugin. Each extraction is made with almost the same parameters, only the plugin name changes.

3.2.1 Confluent and Ganglia data extraction

For Confluent and Ganglia, the reference raw data format is shown in **Figure 14**.

These plugins provide also a lot of general information about the node like the cluster to which it belongs, the feature type and so on. But the real data is contained in the columns: *name*, *value* and *timestamp* (highlighted in red). The first contains the feature name, while the second contains the value of that feature at a given timestamp.

As you can see, each feature has a certain sampling frequency that can differ a lot from one to the other. In this case, *CPU_1_DTS* has a sampling time of 4 minutes, but others are collected every 20 seconds, 1 minute, 15 minutes or so on. Sampling time depends on the daemon configuration that collects that particular feature.

timestamp	value	name	chnl	cluster	...
2019-10-18 08:03:00.061	-10.8	CPU_1_DTS	data	marconi	
2019-10-18 08:03:00.061	Ok	CPU_1_Overtemp	data	marconi	
2019-10-18 08:03:00.061	Present	CPU_1_Status	data	marconi	
2019-10-18 08:03:00.061	84	CPU_1_Temp	data	marconi	
2019-10-18 08:07:00.006	-10.4	CPU_1_DTS	data	marconi	
2019-10-18 08:07:00.006	Ok	CPU_1_Overtemp	data	marconi	

Figure 14: Reference raw data format for Confluent and Ganglia.

3.2.2 Nagios data extraction

The same can be said for Nagios, but it has only one feature called *plugin_output*, its sampling time is 15 minutes and the useful information are contained in the columns: *timestamp*, *value* and *state*. *Value* contains a text description of the node status, while *state* contains a numerical score (from 0 to 3).

Critical states that we are interested in are those in which the state score is "2" and the text description contains one of the following keywords:

- DOWN(*),
- DOWN(*)+DRAIN,
- IDLE+DRAIN,
- ALLOCATED+DRAIN.

As mentioned in paragraph 3.1, the state score "2" represent an anomalous behaviour of the node. When a system administator checks the node state and turns it off for solving the issue, the node is marked with one of the above descriptions based on the problem type.

For our scope, a node is considered in an anomalous state if and only if its state score is "2" and the associated description is one of the above. An example of an anomalous state can be seen in **Figure 15** (highlighted in green).

timestamp	value	name	node	state
2019-10-07 00:45:00.100	C() W() O({mmgetstate-active:rdma-on:no-relevant-waiters}[/marconi,/marconi_work])	plugin_output	r183c12s04	0
2019-10-07 00:45:00.100	DOWN+DRAIN matches a critical state	plugin_output	r183c12s04	2
2019-10-07 01:00:00.014	C() W() O(/,/boot,/boot/efi,/dev,/dev/shm,/opt,/scratch_local,/tmp,/usr,/var)	plugin_output	r183c12s04	0
2019-10-07 01:00:00.014	C() W() O(/[19%],/boot[14%],/boot/efi[4%],/dev[0%],/dev/shm[0%],/opt[59%],/scratch_local[1%],/tmp[1%],/usr[24%],/var[1…	plugin_output	r183c12s04	0
2019-10-07 01:00:00.014	OK, total memory 197537984 greater than 196000000	plugin_output	r183c12s04	0
2019-10-07 01:00:00.014	SSH OK - OpenSSH_6.6.1 (protocol 2.0)	plugin_output	r183c12s04	0

Figure 15: Reference raw data format for Nagios.

3.3 DATA TRANSFORMATION

Data transformation activity aims to create a more compressed data format that can be analysed more easily. Data coming from Ganglia and Confluent follow a very articulated transformation process because they contain both categorical and numerical features that must be considered separately, while Nagios data follows a simpler process (Figure 16). In the end, data coming from each plugin is merged based on timestamps and values are normalised in the range [0, 1].

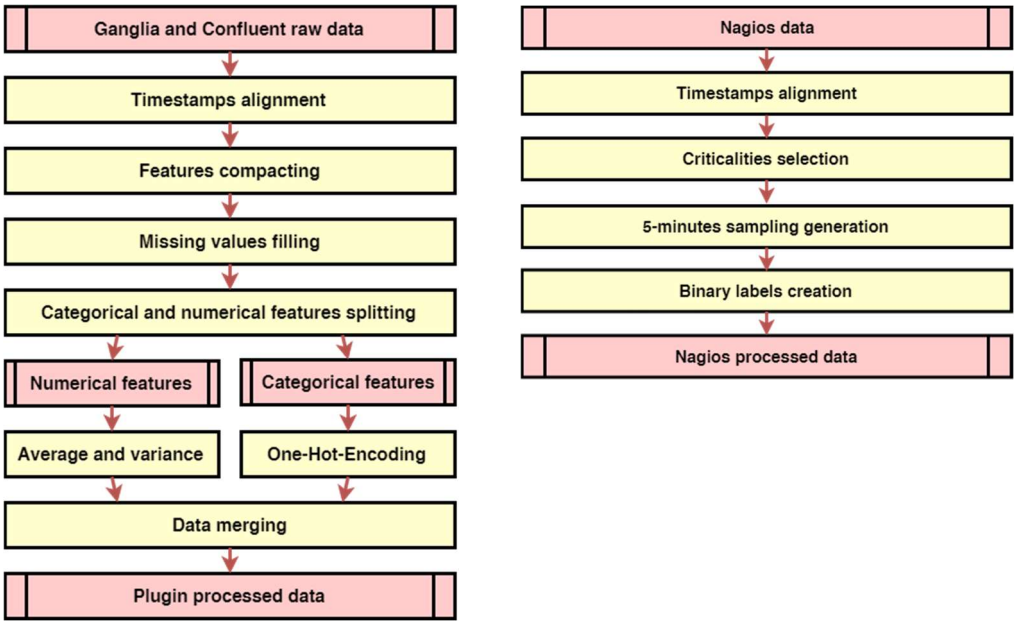

Figure 16: Ganglia, Confluent and Nagios transformation process.

3.3.1 Ganglia and Confluent process

Starting from the raw data format shown in Figure 14, the first transformation step applied is the timestamp alignment to 5 seconds and the microseconds removal. This is done to facilitate data merging because the sampling time is not perfectly

respected: fluctuations of some microseconds – or even seconds – are present in the time information associated with each data.

The second step aims to compact the data into a more flexible tabular structure in which each column represents a feature and each row represent a given instant of time. Inside each cell, there is the feature value for a given time (Figure 17).

timestamp	boottime	core_freq_avg	core_freq_max	...
2019-10-03 08:00:15	1570048000	1529	1968	...
2019-10-03 08:00:35	None	None	None	...
2019-10-03 08:00:45	None	1525	1968	...
2019-10-03 08:00:55	None	None	None	...
2019-10-03 08:01:20	1570048000	1518	1968	...

Figure 17: Ganglia and Confluent compact format.

The third needed action is the filling of missing values because of different feature sampling frequency. Each sampling refers to the previous time slot, so it is possible to propagate a value bottom-up until reaching the previous one. The result is exposed below in Figure 18.

timestamp	boottime	core_freq_avg	core_freq_max	...
2019-10-03 08:00:15	1570048000	1529	1968	...
2019-10-03 08:00:35	1570048000	1525	1968	...
2019-10-03 08:00:45	1570048000	1525	1968	...
2019-10-03 08:00:55	1570048000	1518	1968	...
2019-10-03 08:01:20	1570048000	1518	1968	...

Figure 18: Confluent and Ganglia missing values filling.

The next step creates a uniform sampling time of 5 minutes. To combine different sampling time, it is necessary to consider categorical and numerical data separately. Categorical data is converted into numerical values using the One-Hot-Encoding (OHE) technique and then only one row every 5 minutes is selected.

The OHE algorithm takes a categorical variable and converts it into a series of binary variables, the number of binary variables is the number of distinct values of the original column.

Figure 19 shows an example in which there is a single feature called *age* that can assume two different values ("young" and "old"). The algorithm transforms the categorical column into two distinct binary variables, coding the string *young* as the ordered pair $(\mathbf{0}, \mathbf{1})$ and *old* as $(\mathbf{1}, \mathbf{0})$.

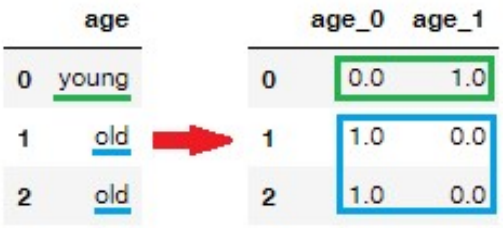

Figure 19: One-Hot-Encoding example.

Instead, numerical data are split into chunks of 5 minutes. For each chunk, a single row containing the weighted average and variance of the values is created. Average and variance are computed using the time difference in seconds as weights. An example can be seen in the image below (Figure 20).

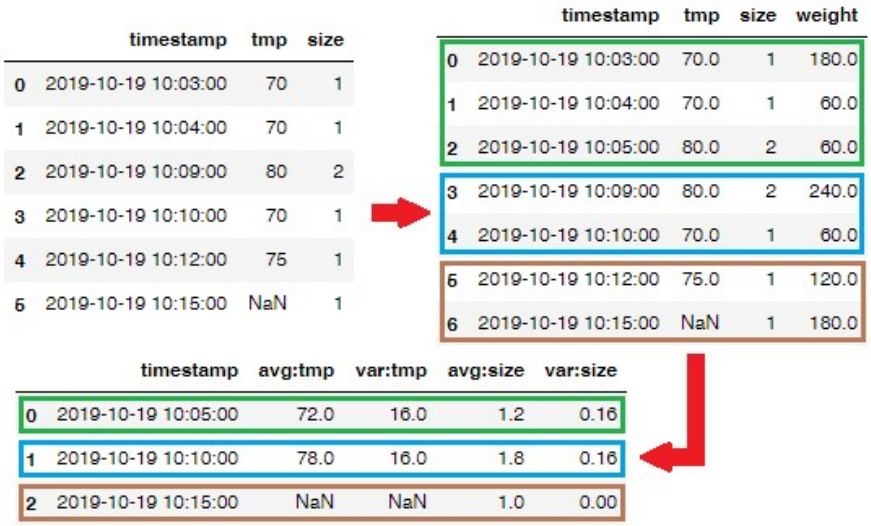

Figure 20: Average and variance evaluation.

Categorical and numerical data are then merged based on timestamps. Finally, rows with missing values and single-value columns are removed because they do not provide any useful information.

3.3.2 Nagios process

The first step is the timestamps alignment in which timestamps are pruned of seconds and microseconds. After it, criticalities are selected based on their status value and text description. As discussed in paragraph 3.2.2, a record is considered anomalous its state value is “2” and its description text matches one of the previously mentioned patterns. In **Figure 21**, there is an example of anomalous states (in red) and normal behaviours (in green).

timestamp	value	name	node	state
2019-10-08 09:00:00	DOWN+DRAIN matches a critical state	plugin_output	r183c12s04	2
2019-10-08 09:15:00	DOWN+DRAIN matches a critical state	plugin_output	r183c12s04	2
2019-10-31 13:45:00	ALLOCATED+DRAIN matches a critical state	plugin_output	r183c12s04	2
2019-11-01 21:07:00	ERROR, the command timeouted: /sbin/runuser -c '/usr/bin/ssh master01 /cinecalocal/nagios/passive/mhsc_nodehealth_a3.pl' root	plugin_output	r183c12s04	2
2019-11-02 17:45:00	C({topwaiter: monitored, PCacheMsgMgrThread: SINCE 603.7923 sec.}[/marconi,/marconi_work]) W() O()	plugin_output	r183c12s04	2

Figure 21: Nagios criticalities example.

Suddenly the sampling time is aligned to 5 minutes. Nagios sampling time is 15 minutes, so each sample is replicated 3 times. In this way, each sample covers exactly 5 minutes. Finally, each time slot is marked with a label equal to “1” if the associated state is critical. Otherwise, the label is set to “0”.

timestamp	value	state
2019-10-08 08:45:00	DOWN+DRAIN matches a critical state	2
2019-11-02 17:45:00	C({topwaiter: monitored, PCacheMsgMgrThread: SINCE 603.7923 sec.}[/marconi,/marconi_work]) W() O()	2

timestamp	label
2019-10-08 00:35:00	1
2019-10-08 00:40:00	1
2019-10-08 00:45:00	1

timestamp	label
2019-11-02 17:35:00	0
2019-11-02 17:40:00	0
2019-11-02 17:45:00	0

Figure 22: Nagios label creation example.

3.3.3 Merging and normalising results

Now that data coming from each plugin are processed, a merging step is applied to obtain a single features table. Finally, data are normalised in the range [0, 1].

Normalization is a technique often applied as part of data preparation for machine learning. The goal of normalization is to change the values of numeric columns in the dataset to a common scale, without distorting differences in the ranges of values.

It is required only when features have different ranges because this may cause the prevalence of one feature over the others.

3.4 DATA SAVING

Normalised data are then saved on disks in CSV format to allow future analysis.

4 ARCHITECTURE OF THE APPROACH

As forehead mentioned in chapter 2.3, data analysis is done in two distinct steps. At first, a lossy compression algorithm is applied to Ganglia and Confluent data to reduce their dimensionality and then a binary classifier is trained using compressed features and Nagios data as labels.

When machine learning techniques are applied, the dataset must always be divided into three subsets:

1. **Training set**: used to train the machine learning model (weights updating).
2. **Validation set**: used to evaluate model performances during the training phase. Usually, it is extracted directly from the training set.
3. **Test set**: used to evaluate the performances of the obtained model after its training.

Figure 23 shows the complete architecture of our approach.

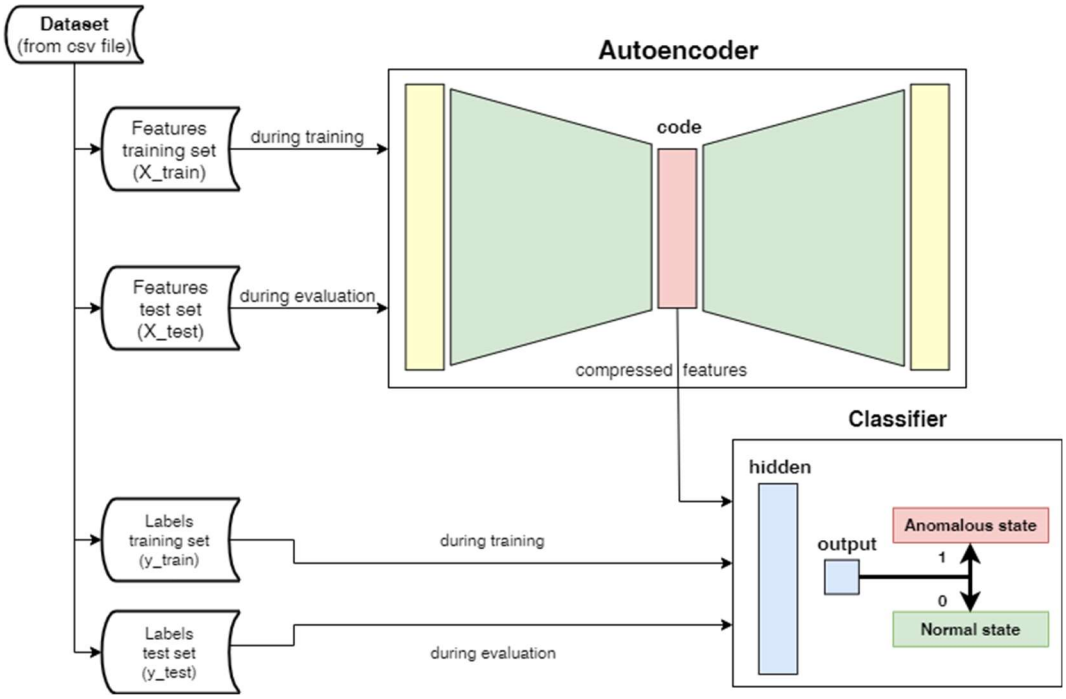

Figure 23: Complete architecture of our approach.

4.1 PERIOD AND NODE SELECTION

This experiment has been realised selecting a single node of MARCONI (r183c12s04) in 4 weeks: October 10th 2019 → November 4th 2019.

The period selection has been performed taking into account that the Confluent plugin is part of the Examon monitoring framework from a few months and complete data are available since September 2019. Afterwards, the selection has been realised looking at the number of anomalies. At the moment of the extraction, October was the most anomalous month.

The choice of the node has been made aiming for a good trade-off between the number of available features and the percentage of anomalous states.

Nodes with a high number of features but with a very limited volume of detected anomalies (like nodes r033c01s01 and r033c01s07) have been discarded.

The selected node has 99 available features (63 numerical and 36 categorical) and the percentage of anomalies in the period is 5.7% (Figure 24).

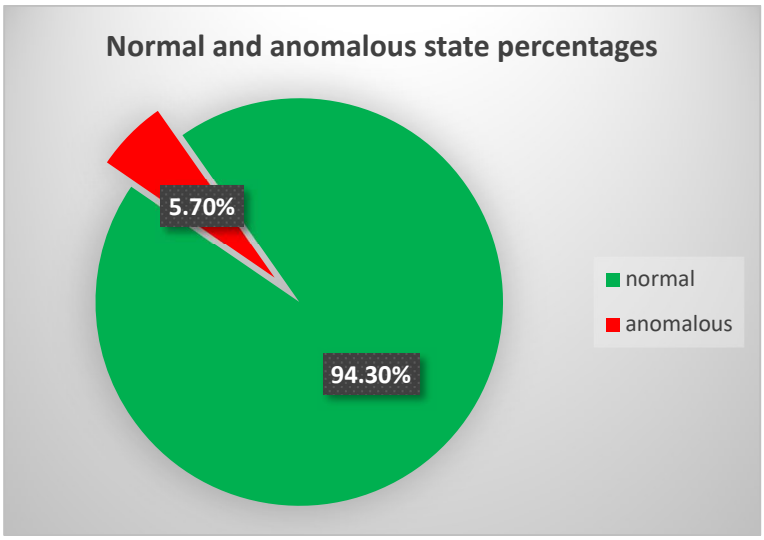

Figure 24: Percentage of normal and faulty behaviours in the period 07/10 - 03/11 2019.

It is important to note that MARCONI system is currently in production state, so this analysis must be done without reducing the system availability.

In this scenario, the number of anomalous states is very limited and anomaly injection strategies - that typically are used to increase their number - cannot be used in systems that are in production.

Our approach is general and it can be applied to any node of the HPC system. In this research, we will focus mainly on node r183c12s04, but additional tests will be done with data coming from other nodes to prove the generality of the approach.

4.2 DATA PREPARATION

Machine learning models can be trained using two different modalities:

- **online mode**, data which are provided to the model are extracted in real-time from the system. As soon as new data are available from the system, they are fed to the model to improve the training process. The training set grows over time.
- **offline mode,** all data which are provided to the model are already available before the starting of the training phase. Typically after the extraction process, data are stored in files (like CSV) to be easily readable when needed. The training set size is fixed and does not change over time.

Machine learning models that we are going to use are trained in offline mode as the extraction process ends before the training phase.

Pre-processed data are read from the CSV file in which they were stored at the end of the pre-processing phase. Then features and labels are extracted from the dataset and used to create the training and the test sets (**Figure 25**).

```
dataset = pd.read_csv('final_data/dataset.csv')
input_data = dataset.drop(labels=['timestamp','label'], axis=1).astype('float64').values
labels = dataset.loc[:,'label'].astype(int).values
input_dim = input_data.shape[1]
```

```
X_train, X_test, y_train, y_test = train_test_split(input_data, labels, test_size=0.10, shuffle=True, random_state=21)
```

Figure 25: Data preparation - python code.

The dataset split is done just after data shuffling to ensure randomness in the split procedure. Features and labels are split in a coupled way to maintain data relations. If a feature sample is placed in a given set, also its label goes in the same one.

The *test_size* parameter tells to the splitting algorithm which percentage of samples must be placed in the test set.

Now data are ready to be fed into a machine-learning algorithm.

4.3 DIMENSIONALITY REDUCTION WITH AUTOENCODER

Our autoencoder model has 9 hidden layers: encoder and decoder have 4 hidden layers each and between them, there is the code layer.

The number of neurons in the input and output layers is equal to the number of the features (124). Encoder layers have respectively 100, 80, 60 and 40 neurons each. The decoder has the same layers of the encoder but in reverse order. While the code layer has only 20 neurons. The autoencoder architecture can be seen in Figure 26.

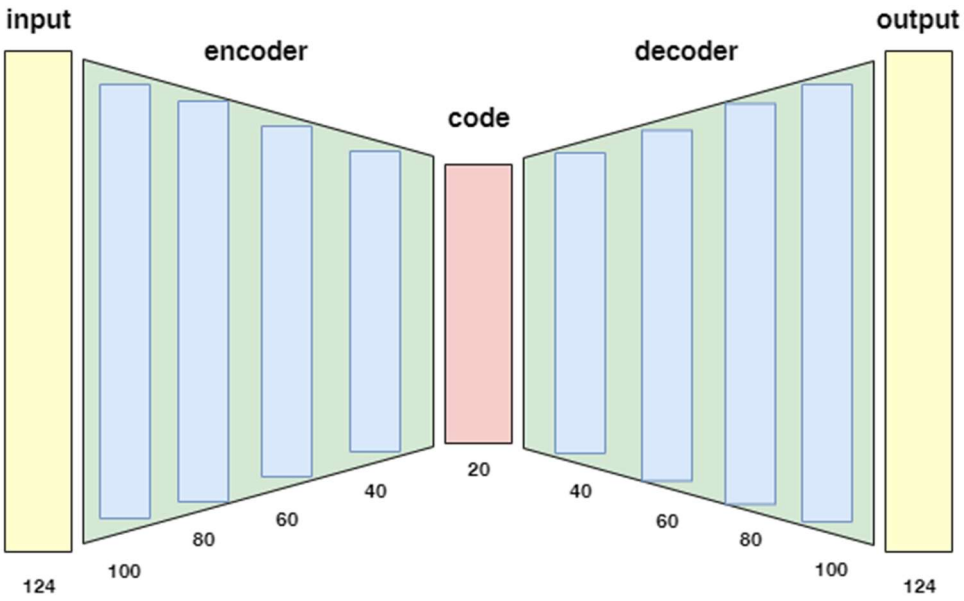

Figure 26: Graphical representation of the autoencoder.

The implementation of the autoencoder model is done throw the Keras library and it can be seen in Figure 27. The model is created starting from the Sequential class that allows you to add layers one after the others.

The input layer is automatically added before the first layer specifying the input dimension with the *input_dim* parameter. So, 9 hidden layers are added to the model. Each of them is a fully-connected layer (called "Dense" layers in Keras) with the ReLU (Rectified Linear Unit) activation function. Also the output layer is fully-connected but it has Sigmoid as the activation function.

In the end, the autoencoder must be compiled selecting the optimizer and the loss function to be used.

```
autoencoder = Sequential()
autoencoder.add(Dense(100, activation='relu', input_dim=input_dim, name='encoder_1'))
autoencoder.add(Dense(80, activation='relu', name="encoder_2"))
autoencoder.add(Dense(60, activation='relu', name="encoder_3"))
autoencoder.add(Dense(40, activation='relu', name="encoder_4"))
autoencoder.add(Dense(20, activation='relu', name='code'))
autoencoder.add(Dense(40, activation='relu', name="decoder_1"))
autoencoder.add(Dense(60, activation='relu', name="decoder_2"))
autoencoder.add(Dense(80, activation='relu', name="decoder_3"))
autoencoder.add(Dense(100, activation='relu', name="decoder_4"))
autoencoder.add(Dense(input_dim, activation='sigmoid', name='output'))
autoencoder.compile(optimizer='adam', loss="mean_squared_error")
print(autoencoder.summary())
```

Figure 27: Autoencoder model - python code.

The model can be trained using the "fit" method (Figure 28). We want autoencoder to reproduce its input as the output, so *X_train* is provided both as input and expected output. The batch size is set to 75 samples, the number of epochs is 200 and the validation split is 10% of the whole training set.

The loss value in red is the loss calculated on the training samples, while the green one is calculated on the validation split. The optimiser tries to reduce the training loss. If also the validation loss decreases the learning process is working well. Typically, the validation loss remains a little bigger than the training one.

```
autoencoder.fit(x=X_train, y=X_train, epochs=200, validation_split=0.10, batch_size=75)

Train on 6332 samples, validate on 704 samples
Epoch 1/200
6332/6332 [==============================] - 3s - loss: 0.0568 - val_loss: 0.0238
Epoch 2/200
6332/6332 [==============================] - 0s - loss: 0.0225 - val loss: 0.0172
...
Epoch 199/200
6332/6332 [==============================] - 0s - loss: 6.7941e-04 - val_loss: 8.2335e-04
Epoch 200/200
6332/6332 [==============================] - 0s - loss: 6.8070e-04 - val_loss: 8.4909e-04
```

Figure 28: Autoencoder training - python code.

The training and validation losses can be analysed over epochs to discover their descending trends. As exposed in Figure 29, the validation loss remains slightly higher than the training one.

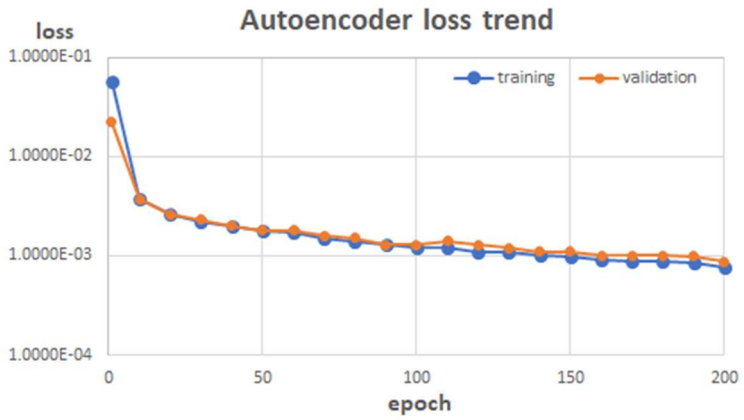

Figure 29: Autoencoder loss trend.

Finally, the autoencoder must be evaluated on the test set. The output of the evaluate method is showed in Figure 30 and represents the average loss applied to the set.

```
loss = autoencoder.evaluate(X_test, X_test)
print("loss: {:.4e}".format(loss))
```

```
782/782 [==============================] - 0s
loss: 7.2880e-04
```

Figure 30: Autoencoder evaluation on the test set - python code.

The final loss is quite similar to the one computed on the training data, so the autoencoder has learnt how to compress correctly the input features into a lower-dimensional space.

4.4 CLASSIFICATION WITH NEURAL NETWORK

At this point, the encoder part of the autoencoder can be used to successfully compress input data. The classification activity can be done over the compressed features, decreasing the training time and the computational costs.

The classifier model is built starting from the pre-trained encoder and code layers. Then a dense layer – with ReLU activation function – is added to the model. The output layer is formed by a single neuron with Sigmoid activation that is able to classify samples into two distinct classes (anomalous/normal states).

The classifier architecture can be seen below in Figure 31.

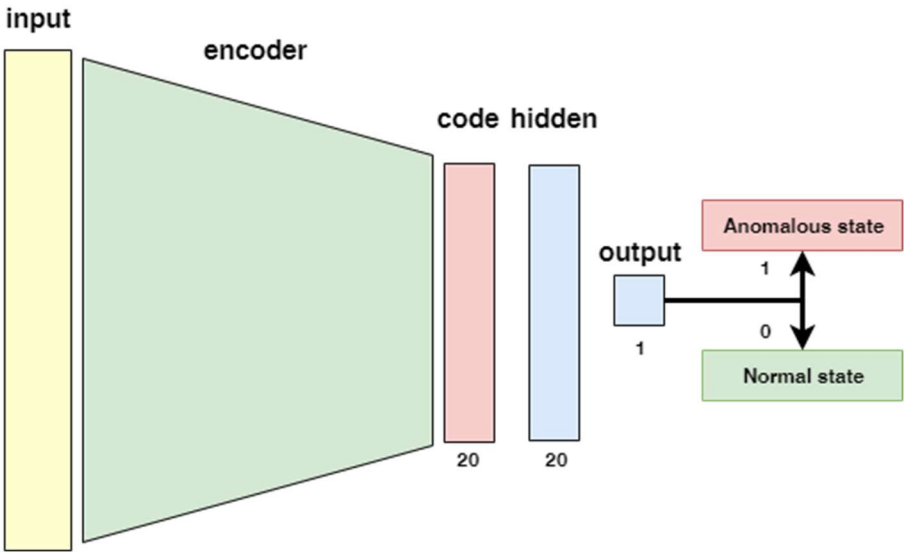

Figure 31: Graphical representation of the classifier.

The encoder and code layers are added to the classifier blocking their weights. In this way, during the training the weights update is performed only for the new layers preserving the encoder knowledge.

As the possible classes are only two, the selected loss function is the binary cross-entropy.

```
encoder = Model(inputs=autoencoder.get_layer("encoder_1").input, outputs=autoencoder.get_layer("code").output)

classifier = Sequential()
classifier.add(encoder)
classifier.add(Dense(20, activation='relu', name="hidden"))
classifier.add(Dense(1, activation='sigmoid', name="output"))

classifier.layers[0].name="encoder"
classifier.layers[0].trainable=False

classifier.compile(optimizer='adam', loss="binary_crossentropy", metrics=['accuracy'])
print(classifier.summary())
```

Figure 32: Classifier model - python code.

The classifier is trained similarly to the autoencoder but using features data as input and labels as output.

5 RESULTS

5.1 CLASSIFIER EVALUATION

The classifier is evaluated on the test set and it provides similar results: 99.62% accuracy with a mean loss of 1.8651e-02.

```
loss, accuracy = classifier.evaluate(X_test, y_test)
print("loss: {:.4e}".format(loss))
print("accuracy: {:2.4f}%".format(accuracy*100))
```

```
736/782 [============================>..] - ETA: 0s
loss: 1.8651e-02
accuracy: 99.62%
```

Figure 33: Classifier evaluation - python code.

The "*predict*" method of the binary classifier produces as output the probability that a given sample belongs to the anomalous class (Figure 34). As probability goes in the range [0,1], we need a threshold value to distinguish positive and negative results.

```
predicted_probabilities = classifier.predict(X_test)
```

```
def assign_class(predicted_probabilities, threshold=0.5):
    classes = []
    for i in range(len(predicted_probabilities)):
        if(predicted_probabilities[i] >= threshold):
            classes.append(1)
        else:
            classes.append(0)
    return np.asarray(classes)
```

```
classes = assign_class(predicted_probabilities)
```

Figure 34: Predict classes on test data - python code.

In our case, the threshold is set to 0.5 to split fairly the output domain.

In some domains, false negatives and false positives have different costs. For example, in the medical domain a false negative result is worse than a false positive one: the falsity of the second is detected by the results of other exams, while a false negative result could lead to discovering the disease too late.

So, the threshold must be adjusted properly to obtain the desirable balance between false negative and false positive (Figure 35).

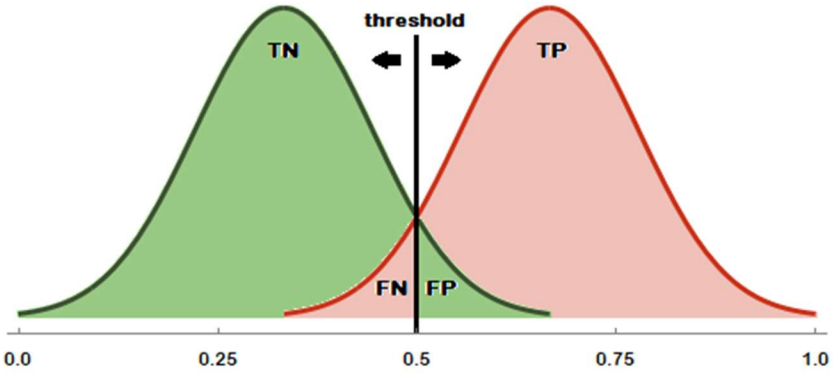

Figure 35: Threshold balancing.

As shown in Figure 36, a 0.5 threshold gives us: 35 true positives, 745 true negatives, 1 false negative and 1 false positive.

Predicted Value (predicted by the test)		Actual Value (as confirmed by experiment) positives	Actual Value (as confirmed by experiment) negatives
	positives	**TP** 35	**FP** 1
	negatives	**FN** 1	**TN** 745

Figure 36: Predicted vs Actual class values.

Plotting the real probability distribution of both classes (normal in green and anomalous in red) you get the diagram displayed below (Figure 37).

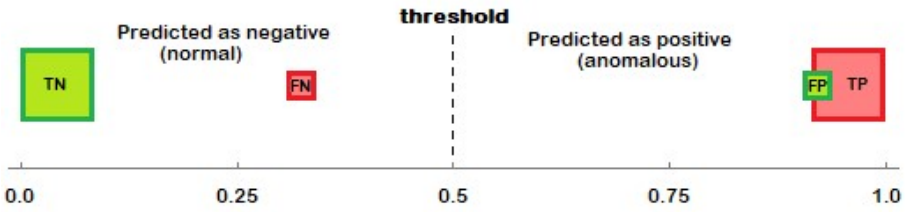

Figure 37: Real probability distribution.

It shows clearly that increasing the threshold does not increase the accuracy of predictions, but decreasing it toward a value near 0.25 leads to correctly detect all the anomalies. So, a threshold of 0.25 is sufficiently far from the true negatives

cluster. It is unlikely that it will significantly degrade the recognition of true negatives, but it can improve a lot the detection of anomalies.

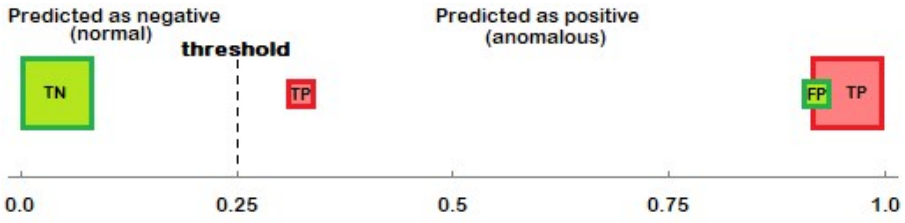

Figure 38: Threshold adjusting to match our intent.

From these results, it is possible to evaluate the main performance metrics to obtain a good description of the learned model.

Four main metrics usually are calculated to evaluate the test results:

- **Accuracy**: the ratio between correctly predicted observation and total observations. High accuracy means that the model has learnt well.
- **Precision**: the ratio between correctly predicted positive observations and the total predicted positive observations. High precision relates to the low false-positive rate and it helps when the costs of false positives are high.
- **Recall**: the ratio between correctly predicted positive observations and all observations in the actual class. High recall means a low false-negative rate and it helps when the cost of false negatives is high.
- **F1 score**: weighted average of Precision and Recall. A good score means that there are low false positives and low false negatives, so the model is correctly identifying real threats and it is not disturbed by false alarms.

Equations and associated results are shown below:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{36 + 745}{36 + 745 + 1 + 0} = \frac{511}{512} = 0.998$$

$$Precision = \frac{TP}{TP + FP} = \frac{36}{36 + 1} = \frac{36}{37} = 0.973$$

$$Recall = \frac{TP}{TP + FN} = \frac{36}{36 + 0} = \frac{36}{36} = 1$$

$$F1\ score = 2 * \frac{Precision * Recall}{Precision + Recall} = 2 * \frac{0.973 * 1}{0.973 + 1} = 2 * \frac{0.973}{1.973} = 0.986$$

The accuracy is very good. A value of 0.998 tells us that almost all observations have been correctly predicted. The precision is good too because the number of false-

negative predictions is pretty low. The recall value is the maximum possible (1.0), it means that no false negatives have been recognised. Finally, the F1 score is similarly high that means that the number of false positives and false negatives are low and almost the same.

5.2 TECHNICAL DETAILS

The data extraction process and the machine learning models training have been executed over another CINECA HPC system (called DAVIDE) to speed up the computation.

The data extraction has been performed through the DAVIDE batch system. The extraction period has been split into time frames and each one has been entrusted to a different batch job.

Finally, jobs results have been merged and stored into a single CSV file.

Even the models training has been executed on DAVIDE but here the batch system has not been used and computations have been done through the Jupyter Notebook web tool. As DAVIDE has no browser or graphical interface, the Notebook was connected to DAVIDE by an ssh tunnel.

To appreciate the scalability of this approach, here below are specified some technical details about the process:

- **Dataset size**: ~7800 samples of 124 features each (13.8MB).
- **Nodes used for computation:** 1 single node of DAVIDE (with 16 cores and 1 NVDIA Tesla P100 GPU).
- **Extraction and pre-processing time:** ~5m per 1 day of data.
- **Autoencoder training time**: ~4m 36s (over the node GPU).
- **Autoencoder inference time**: ~0.11ms per instance (over the node GPU).
- **Classifier training time**: ~2m 24s (over the node GPU).
- **Classifier inference time**: ~0.13ms per instance (over the node GPU).

5.3 EVALUATION OF OTHER NODES

To prove the generality of our approach, we have also trained and tested the models with data coming from 9 other nodes. The considered nodes are: r183c09s01,

r183c09s03, r183c09s04, r183c10s01, r183c10s02, r183c10s03, r183c11s01, r183c11s02 and r183c11s03.

The extraction period is reduced from 4 weeks to 2 (October 21st → November 4th 2019) to speed up the extraction activity.

The following figures (from **Figure 39** to **Figure 47**) summarise the prediction trends over the test set of each node.

As you can see, the accuracy is greater than 98% in every node, so the number of correctly detected instances is very high. Precision is greater than 83% in 7 cases over 9, so the number of false positives is good. Similarly, Recall is greater than 85% in 7 cases over 9, so the number of false negatives is acceptable. Finally, F1_Score is greater than 83% in 7 cases over 9 that is nearly good.

These performances are worse than the ones computed for the node r183c12s04. This is likely caused by the very low number of anomalies in the training and test sets (~75 in each dataset).

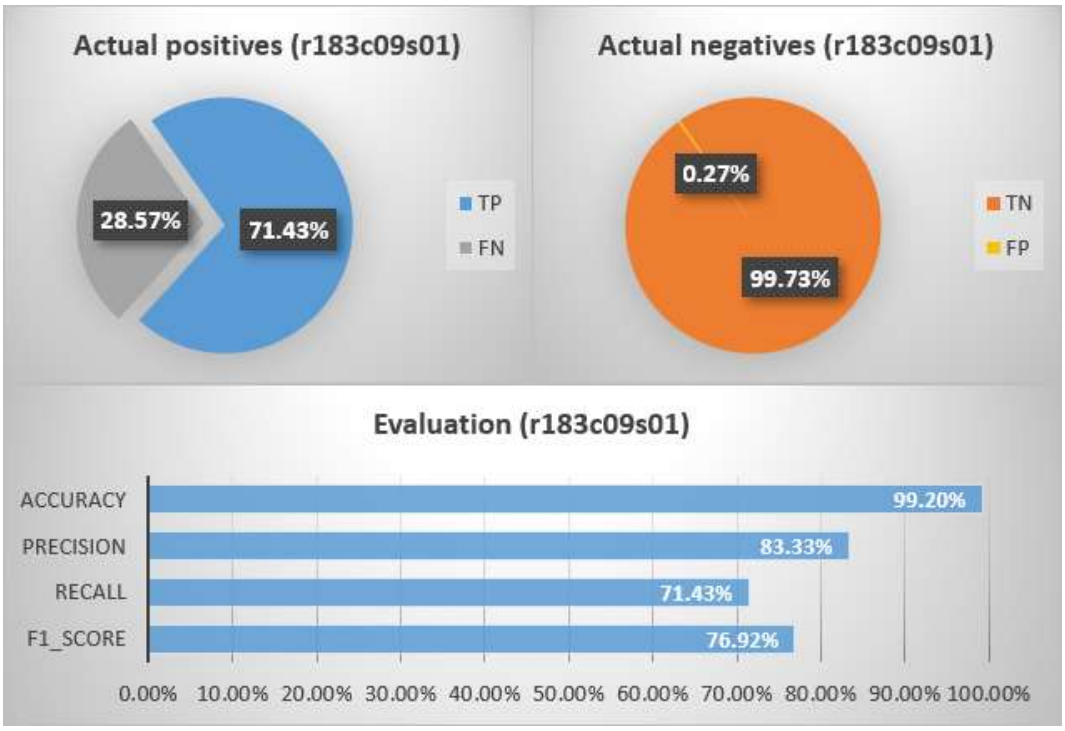

Figure 39: Evaluation of the node r183c09s01.

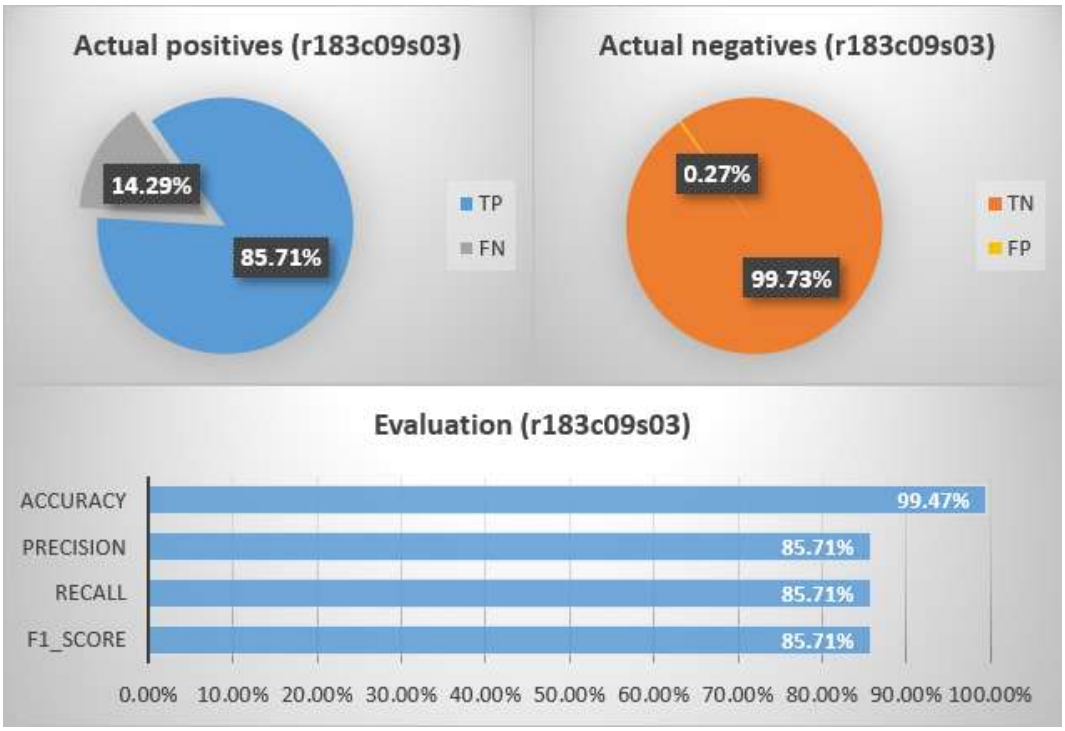

Figure 40: Evaluation of the node r183c09s03.

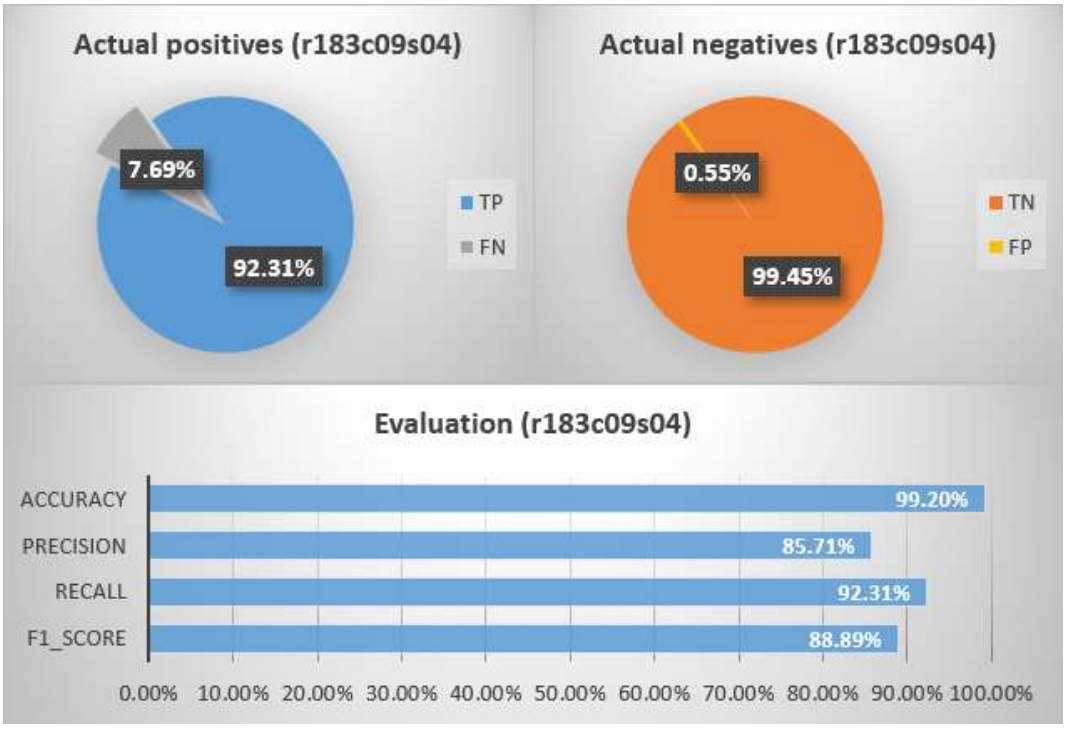

Figure 41: Evaluation of the node r183c09s04.

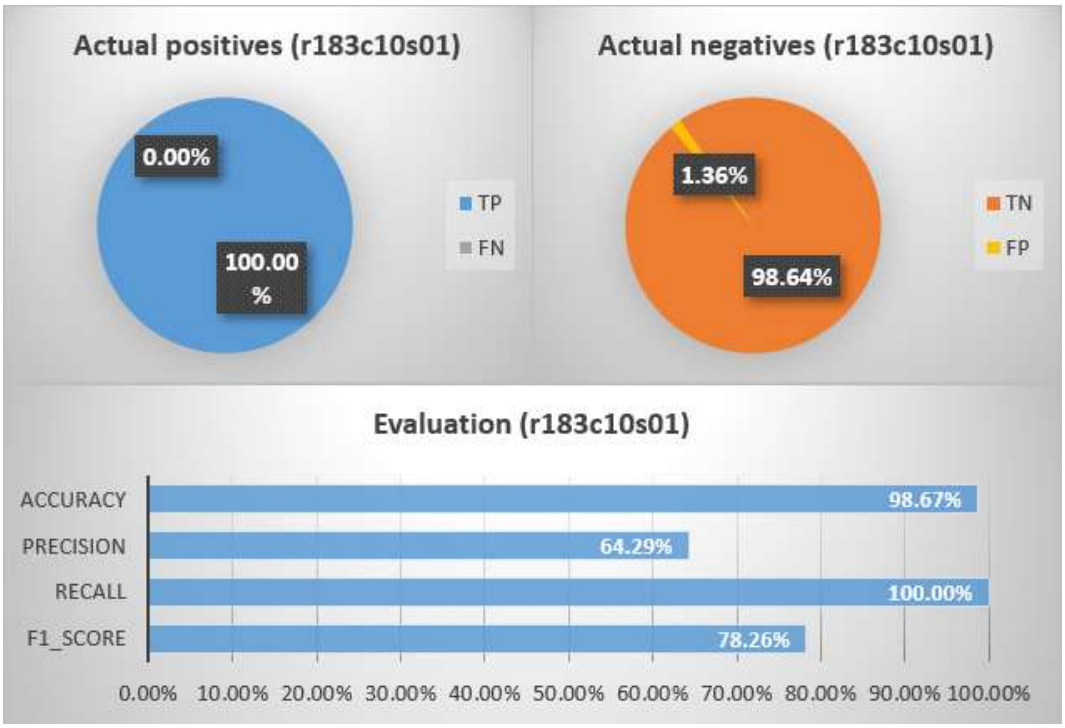

Figure 42: Evaluation of the node r183c10s01.

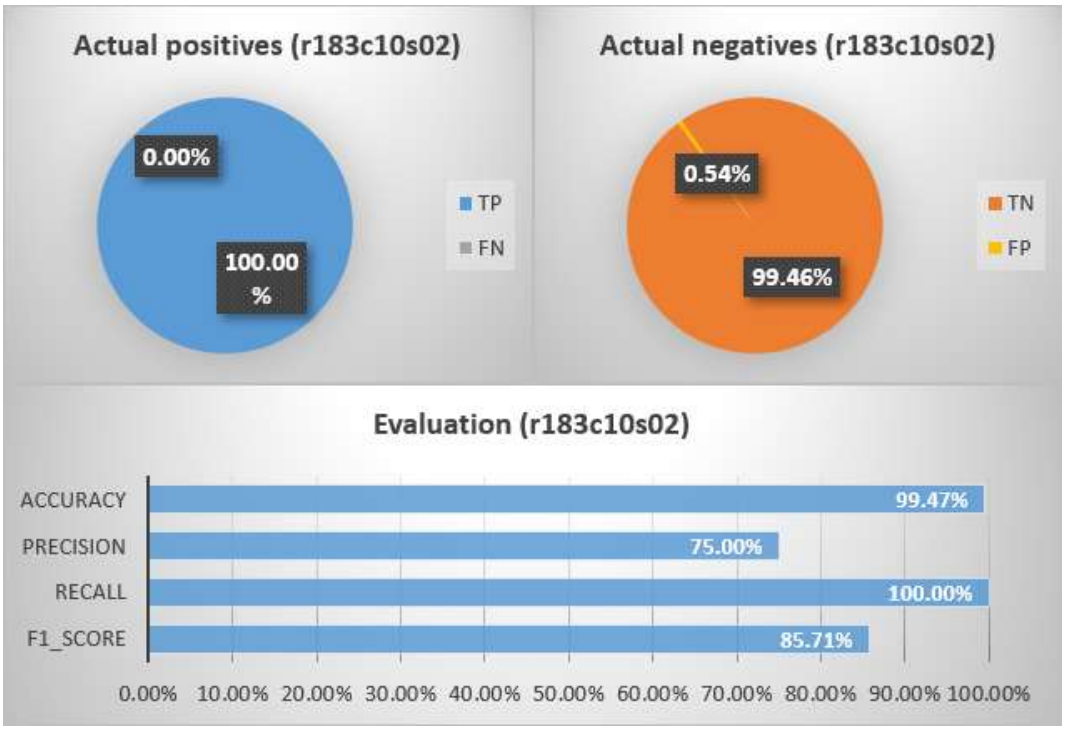

Figure 43: Evaluation of the node r183c10s02.

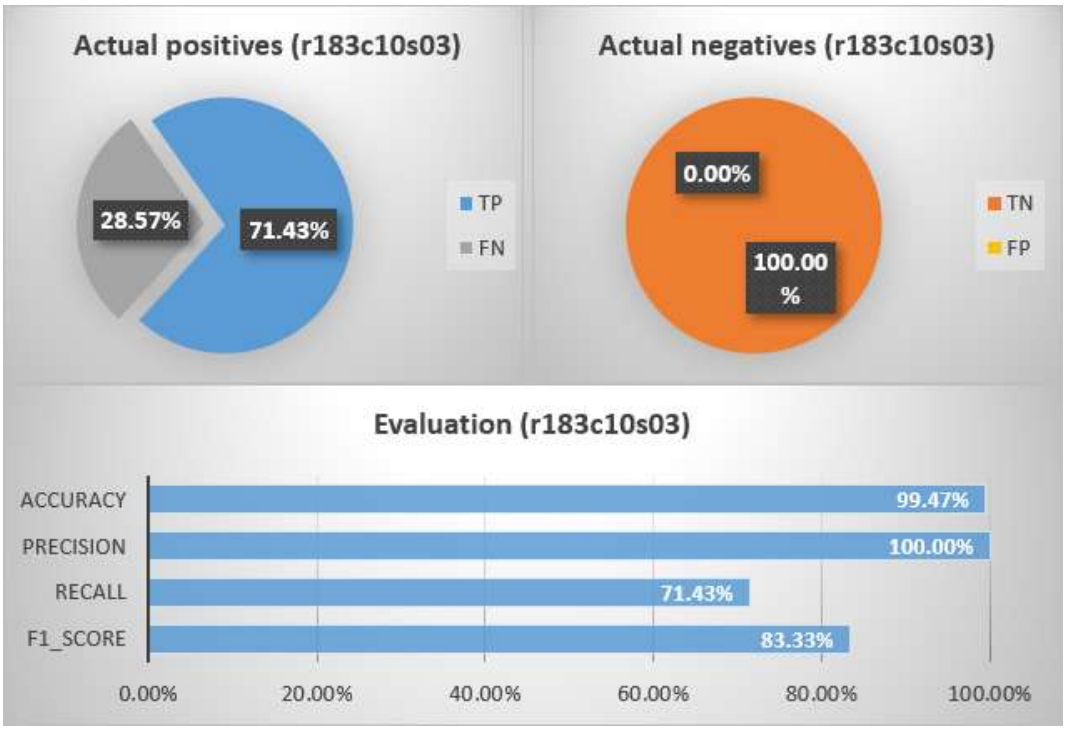

Figure 44: Evaluation of the node r183c10s03.

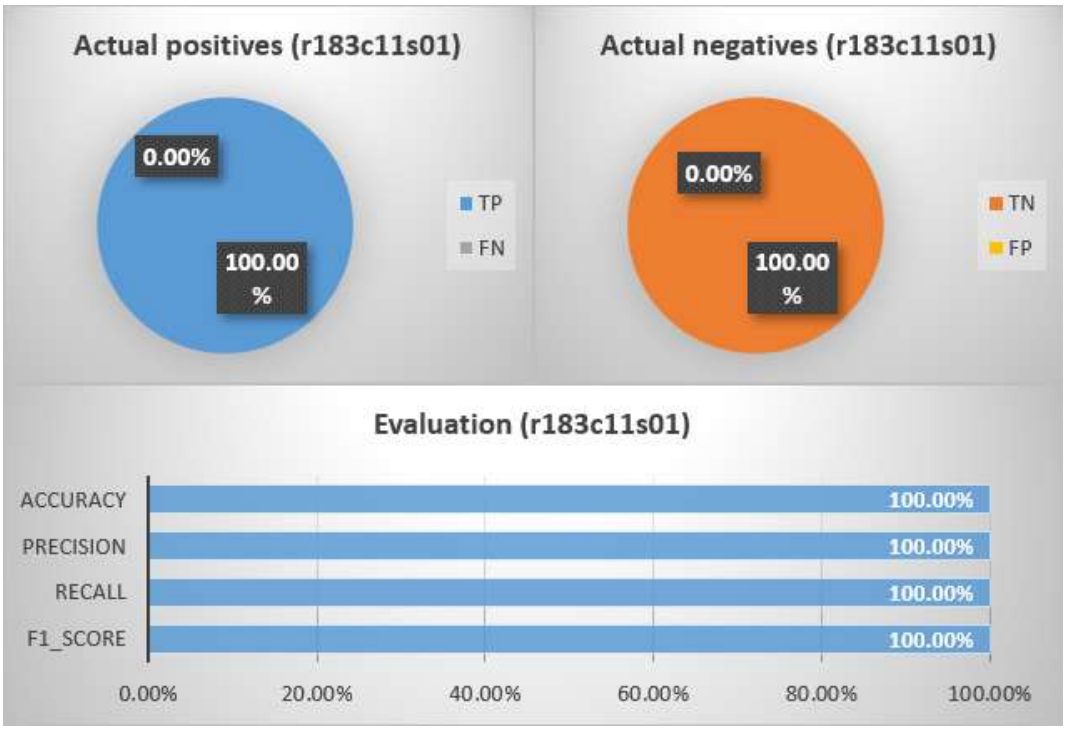

Figure 45: Evaluation of the node r183c11s01.

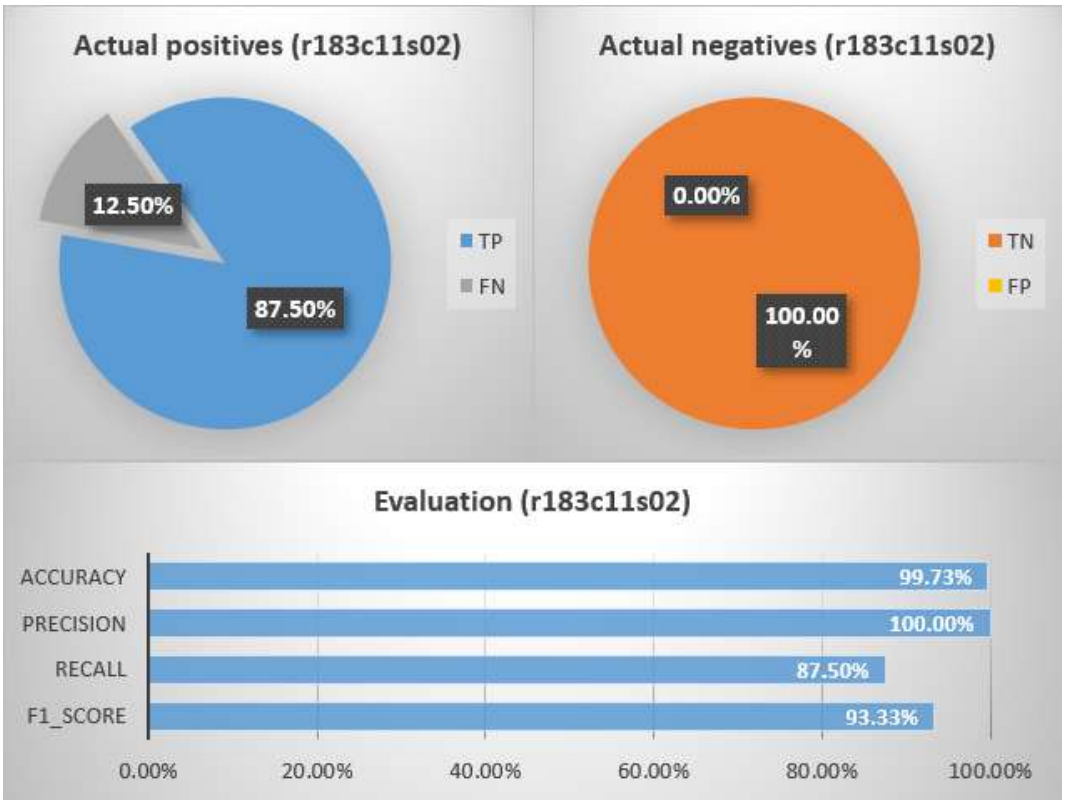

Figure 46: Evaluation of the node r183c11s02.

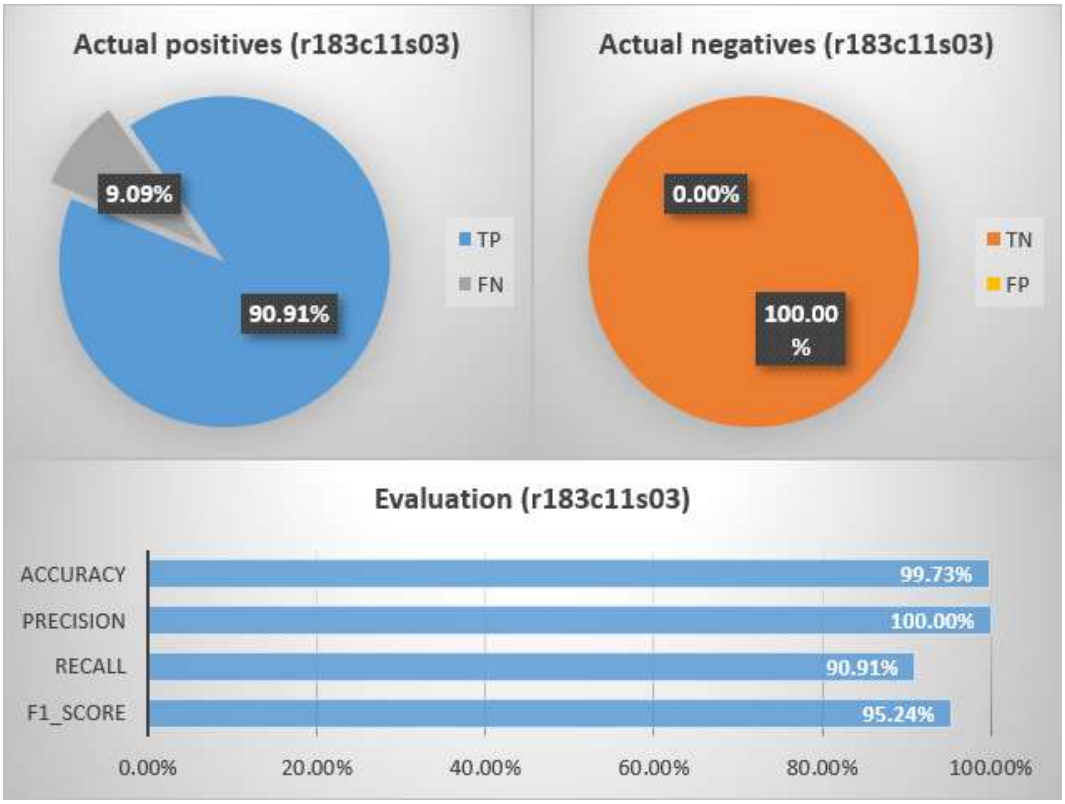

Figure 47: Evaluation of the node r183c11s03.

6 CONCLUSIONS AND FUTURE WORKS

The anomaly detection activity can significantly improve the overall performance of supercomputers. It can reduce the number of system breakdowns and adapt the maintenance schedule to the system health, granting the right quality of service.

The combined usage of different machine learning techniques has proved to be a successful approach to the anomaly detection problem. In particular, unsupervised and supervised algorithms can be mixed to take full advantage of their benefits and obtain better results.

In this thesis, the anomaly detection activity has been performed referring mainly to a single node of the MARCONI supercomputer (node "r183c12s04"). The process has been done in two different steps. At first, the data dimensionality has been reduced by employing autoencoders. Then a neural network classifier has been applied to distinguish between normal and anomalous node states.

The results obtained are extremely promising, but the real-world limitations – like the low number of anomalous state samples or the usage of data related to a single computing node – may have resulted in slightly biased outcomes.

So, more tests are needed to examine different periods, nodes and anomaly types. Tests have proved that our model is general enough to detect anomalies in other nodes, but it must be trained more intensively to obtain higher results.

In the future, the model can be generalised to detect anomalies in each type of node without re-training it. Besides, predictive strategies can be employed to diagnose incoming anomalies exploiting the temporal relationship that exists between the collected data. Finally, future models can learn to distinguish not only normal and anomalous states, but also identify the various types of anomalies.

BIBLIOGRAPHY

[1] TOP500, "TOP500 List - November 2019," November 2019. [Online]. Available: https://www.top500.org/list/2019/11/?page=1.

[2] G. F. Ciocarlie, S. Novaczki and H. Sanneck, "Detecting Anomalies in Cellular Networks Using an Ensemble Method," *Proceedings of the 9th International Conference on Network and Service,* 2013.

[3] W. Yixing, L. Meiqin, B. Zhejing and Z. Senlin, "Stacked sparse autoencoder with PCA and SVM for data-based line trip fault diagnosis in power systems," *Neural Computing and Applications,* 2018.

[4] T. Ozan, A. Emre, Z. Yijia, T. Ata, B. Jim, L. Vitus J, E. Manuel, A. Ayse and L. J. Vitus, "Online diagnosis of performance variation in hpc systems," *IEEE Transactions on Parallel and Distributed Systems,* 2018.

[5] S. Freytag, "Workshop: Dimension reduction with R," [Online]. Available: http://rpubs.com/Saskia/520216.

[6] A. Howaida, "What are the X and Y axes of Clustering Plots?," [Online]. Available: https://stats.stackexchange.com/questions/253926/what-are-the-x-and-y-axes-of-clustering-plots.

[7] G. Markus and U. Seiichi , "A Comparative Evaluation of Unsupervised Anomaly Detection Algorithms for Multivariate Data," *PloS one,* 2016.

[8] M. Cherif Dani, H. Doreau and S. Alt, "K-means application for anomaly detection and log classification in hpc.," *International*

Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, 2017.

[9] D. Lee, "Reinforcement Learning, Part 1: A Brief Introduction," [Online]. Available: https://medium.com/ai%C2%B3-theory-practice-business/reinforcement-learning-part-1-a-brief-introduction-a53a849771cf.

[10] H. Lu, Y. Li, S. Mu and D. Wang, "Motor Anomaly Detection for Unmanned Aerial Vehicles Using Reinforcement Learning," *IEEE Internet of Things Journal,* 2018.

[11] M. Ved, "Feature selection and feature extraction in machine learning: an overview," [Online]. Available: https://medium.com/@mehulved1503/feature-selection-and-feature-extraction-in-machine-learning-an-overview-57891c595e96.

[12] A. Dertat, "Applied deep leraning - Part 3: Autoencoders," [Online]. Available: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798.

[13] "Neural networks," [Online]. Available: http://cs231n.github.io/neural-networks-1/.

[14] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano and L. Benini, "A Semisupervised Autoencoder-based Aapproach for Anomaly Detection in High Performance Computing Systems".

[15] "Introduction to High-Performance Computing," [Online]. Available: https://epcced.github.io/hpc-intro/.

[16] "HPC User Guide," CINECA, [Online]. Available: https://wiki.u-gov.it/confluence/display/SCAIUS/HPC+User+Guide.