Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

Multi-node Fault Classification using Machine Learning

Supervisor:
Professor Zeynep Kiziltan

Co-supervisors:
Dr. Alessio Netti
Dr. Alina Sîrbu

Presented by:
Vito Vincenzo Covella

III Session
2019-2020

Contents

Acknowledgements

Before presenting to you the work we did for this thesis, I'd like to spend some time to thank some people that have helped me through this journey.

First of all I'd like to thank my supervisor Professor Zeynep Kiziltan and my co-supervisors, Dr. Alessio Netti and Dr. Alina Sîrbu, for having guided me through the work done for the thesis and for having inspired me. Even though we were limited by the pandemic and interacted only through Skype and emails, I really liked discussing with them about the results and about how to tackle the problems we encountered; I hope the inspiration and advice they gave me help me become a better computer scientist. Dr. Alessio Netti deserves special credits also for having acquired the dataset at LRZ.

I'd also like to thank some of my friends, in particular Alessandro Lopez, Giuseppe Antonio Nanna and Patrizia Castellaneta.

A special thanks goes also to the Telegram community called "Jacksoft Labs", where we discussed in a light-hearted way about Computer Science and various other topics.

I'd also like to thank my family for having supported me during troubling and difficult times, in particular my father Nicola Covella and my mother Beatrice Catucci, my aunt Rosa Catucci and my cousins Nunzio Nanna and Ortenzia Ludovico.

Finally a special thanks goes to all the researchers that worked on the topic before me and to the Open Source community that built the libraries we used throughout the work; as Isaac Newton said, "if I have seen further it is by standing on the shoulders of giants".

Sommario

Un sistema ad alte prestazioni (HPC - High Performance Computing), cioè un sistema con molta più potenza computazionale di un generico sistema, è composto da differenti sezioni e molti nodi di computazione. In un sistema così complesso gli errori e i malfunzionamenti possono verificarsi per differenti ragioni: a causa dell' interazione fra i componenti, a causa di specifiche tecnologie usate per massimizzare il rapporto performance/costi o a causa di bugs nel software. In ogni caso, per raggiungere le performance tipiche di un sistema Exascale e garantire disponibilità e affidabilità è importante rilevare e correggere queste anomalie. In questa tesi proponiamo un metodo di classificazione degli errori basato su machine learning.

Altri ricercatori hanno già lavorato su questo campo, ma il loro lavoro si basa principalmente su modelli specifici per ogni nodo. Tuttavia, siccome le operazioni di *fault injection* e addestramento non sono scalabili quando il numero dei nodi aumenta, la creazione di modelli specifici per ogni nodo è impraticabile perché richiederebbe troppi dati e la *fault injection* diventerebbe difficile da controllare ed effettuare. Per questo motivo la nostra ricerca si basa su modelli generici singolarmente usati su vari nodi (*multi-node models*), dato che per questi modelli c'è meno sforzo operazionale nella fase di addestramento e il costo di mantenimento nel tempo è minore. Nello specifico, la nostra metodologia propone non solo tecniche di esplorazione dei metaparametri, ma si concentra anche su quanti nodi sono necessari per la fase di addestramento e quali specifici nodi siano i migliori candidati. Per queste ragioni, confrontiamo due approcci: addestramento incrementale con nodi selezionati randomicamente e addestramento incrementale con nodi che rappresentano un prescelto numero di *clusters*. In entrambi i casi il risultato è un singolo modello generico che può essere usato su differenti nodi per la *fault detection* (rilevazione degli errori). L'idea di base consiste nell'avere un singolo modello multi-node che sia abbastanza generico da poter lavorare su nuovi dati che provengono da diversi nodi.

Usando il dataset reso disponibile da LRZ, che contiene i dati di 32 nodi, mostriamo che le prestazioni di classificazione si stabilizzano quando si usa un piccolo sottoinsieme di nodi nel *training set* e che entrambi i metodi precedentemente discussi superano le performance di modelli specifici per ogni nodo qualora si usi più di un nodo nella fase di addestramento. Infine mostriamo che l'approccio basato sul *clustering* è più affidabile e stabile qualora si usino più nodi di addestramento, mentre l'approccio randomico permette di ottenere migliori risultati qualora si usi un numero minore di nodi di addestramento.

Abstract

An High Performance Computing (HPC) system, which is a system with much more computational power than general computing systems, is a complex system made up of different sections and many computing nodes. In such systems failures and malfunctions can arise for different reasons: because of the interactions among the components, because of the specific technologies used to have an high performance/cost ratio or because of bugs in the software. In any case, in order to reach *Exascale* performances and guarantee availability and reliability it is important to detect and recover from these anomalies. In this thesis we propose a fault classification method based on machine learning.

Other researchers have already worked in this field, but their work mainly relies on per-node models. However, since both fault injection and training do not scale well when the number of compute nodes increases, per-node models are impractical because they require too much data and fault injection would be hard to control. For this reason our research involves single multi-node models, since for single general models there's less operational effort for training and maintaining the model over time is easier. More specifically our methodology is focused not only on metaparameter exploration, but also on understanding how many nodes are necessary for training and which specific nodes are the best candidates. For these reasons, we compare two approaches: incremental training with nodes selected randomly and incremental training with nodes which are representative of a chosen number of clusters. In both cases the end result is a single general model that can be used on different nodes for fault detection. The idea is to have a single multi-node model that generalizes well enough to work on novel data coming from different nodes.

Using the dataset provided by LRZ, which consists of data about 32 compute nodes, we show that the classification performances stabilize when using a small subset of compute nodes as training set and both the previously discussed selection methods outperform node-specific classifiers when using more than one training node. Finally we show that the clustering approach is more reliable and stable when using more training nodes, while the random approach gives better performances when using a lower number of training nodes.

Chapter 1

Introduction

In this chapter we will introduce the work done for this thesis. More specifically, this chapter is organized as follows: Section 1.1 will introduce the area of research in which we operated. Section 1.2 will present the research motivation and the goals we wanted to achieve. Section 1.3 will list our contributions to the research field. Finally Section 1.4 will present how the thesis is structured.

1.1 Area of research

Supercomputers and High-Performance Computing (HPC) systems, which are systems with much higher performances than general computers, are widely used to drive modern scientific discoveries in many fields, such as Big Data analysis, particle physics, bioinformatics and cosmology. Scientists and engineers are trying to build Exascale (10^{18} floating point operations per second) systems exploiting the use of faster processors and massive parallelism [Wa10].

HPC systems that aim to reach Exascale performances are built using lots of cores and various technologies, such as advanced low-voltage technologies (that tend to be more prone to ageing effects) [BBC^{+}08] and dynamic voltage frequency scaling. Such technologies are used to lower the energy consumption of an increasingly high number of components while maintaining high performances. The use of an increased number of components with the technologies previously described is one of the factors that lead to increased fault rates [HE17]. Moreover in such complex systems faults and anomalies can also arise from hardware that breaks, incorrect configurations and bugs contained in the software used. Nowadays HPC systems based on the cluster architecture are also made up of multiple nodes and multiple sections (the *frontend section*, the *backend section*, the *computing nodes* and the *networking infrastructure*) [Net], meaning that the possibility of having a fault arises due to the complex interactions among these sections. Moreover economic forces push the designers to build HPC systems using commodity components aimed at mass market, which can improve the likelihood of having faults [NKB+20]. Finally HPC system users must also take into account the interaction between Open Source software, modern software and legacy software; these interactions can be another component that leads to faults and errors.

Faults and anomalies can lead to sub-optimal performances and prevent applications from making progress. They can also hinder the availability and reliability required by the systems used by the scientific community, data centers and cloud providers.

Modern HPC systems also have monitoring frameworks, such as DCDB [NMA+19], in order to store information coming from sensors which can be useful to guide the process of fault detection and analyze the performance variation of the HPC system.

1.2 Research motivation and goals

Due to to the system's complexity previously described, the amount of data stored by monitoring frameworks is huge and analyzing it manually is a daunting and impractical task. For these reasons it is important to have automated tools not only for logging but also for fault detection, in order to be able to detect the faults and correct them before they lead to failures and service disruption. This is vital to have resiliency mechanisms and to achieve Exascale performances.

The work presented in this thesis involves performing fault classification by exploiting the use of supervised machine learning techniques. This has already been done by some researchers, such as by Netti et al. in [NKB+20], but only at node-specific level. They showed successful results in fault injection, using the novel FINJ tool, and in fault classification with a random forest model on the Antarex dataset[1] and using 22 statistical features for each metric in the system. However traditional node-specific fault detection methods, such as the one mentioned before, may not scale up well in modern HPC systems, instead they can become more costly to build and maintain and thus less effective in their job since the number of computing nodes in HPC systems keeps increasing. For these reasons, instead of focusing on models tailored on specific compute nodes, as others in the research field have already tried, we focused on building a single general multi-node model. This problem can be challenging since we have to mitigate the problems that can arise from the presence of different characteristics in each computing node. Focusing on multi-node models means that training is performed on a specific set of computing nodes and testing is done on different computing nodes. The main idea is to explore different techniques that will lead to the identification of the minimal set of best

[1] `https://zenodo.org/record/2553224`

computing nodes to perform training in order to create a single general model, which will be able to provide good classification performances even on different computing nodes. Multi-node models would be ideal because they are easier to build and maintain as the number of nodes scales up and may outperform node-specific models.

For the previously mentioned reasons, new methods for *fault detection* should focus not only on accuracy but also on scalability. Thus the **multi-node** aspect of fault detection becomes very important in order to achieve peak performances in such complex systems.

After acquiring and using the dataset about the CooLMUC-3 HPC[2] system, provided by LRZ and made up of data about 32 compute nodes, we wanted to develop a methodology for finding the best metaparameters of the model and for building such multi-node models. More specifically, we wanted to develop a methodology for finding the right amount of training nodes needed to achieve good classification performances and for finding which specific nodes to use.

1.3 Contributions

After having acquired the dataset about the CooLMUC-3 HPC system provided by LRZ, we processed the data in a specific way in order to obtain the best *signatures*, that is the way to obtain the best feature vectors, relying on our previous work done during the internship [Cov20]. We then performed some initial experiments in order to compare the results with the baseline obtained in our previous work. Then we performed feature selection, in such a way to save up space without compromising the classification results, and metaparameter exploration, since we wanted to understand how normalization, shuffling, class balancing and subsampling can influence the results. Finally we designed a methodology for multi-node fault detection, that is a strategy to find out how many nodes are needed during the training phase and which specific nodes to choose. This is important since using all the nodes is not a scalable option and the memory and time requirements would be too high. Moreover choosing wisely the training nodes is important in order to have a model that generalizes well enough to work on novel data.

Indeed each compute node can be configured differently or have its own peculiar hardware. Moreover the computation in the cluster may be imbalanced, using heavily some nodes but using very little resources of some other different nodes. This can happen because of particular configurations of the job scheduler or because of many different other reasons, such as the peculiarity of the applications executed. The imbalance would surely influence the sensors' readings, which are used as metrics for building the features of the classification model. Moreover different sensors' readings may be obtained because of hardware performance variability, as

[2]`https://doku.lrz.de/display/PUBLIC/CoolMUC-3`

pointed out by Inadomi et al. [IPI+15]. Despite this challenge, building a single general multi-node model that generalizes well enough for all computing nodes is still an important research topic because of the reasons explained in Section 1.2, and it is the main scientific contribution of this thesis.

To tackle these challenges we considered running incremental experiments with nodes chosen randomly inside a pool of training nodes and incremental experiments with nodes chosen with a clustering-based approach, whose aim is to identify candidate nodes which are representative of each cluster.

To summarize, our contributions are the following ones:

- a comparison between node-specific models and single general multi-node models;
- finding the number of training nodes for achieving good performances with multi-node models;
- a comparison between two novel techniques for selecting training nodes in order to build a single general multi-node model: incremental random approach and clustering-based approach.

1.4 Thesis structure

In **Chapter 2** we will introduce some background information, a summary of related work on the topic of fault classification and what's new and original in our work.

In **Chapter 3** we will describe the dataset used throughout the thesis and we will also present the details on how the data has been acquired and how it gets processed in order to obtain the final feature vectors.

In **Chapter 4** we will introduce the methodology for multi-node fault classification.

In **Chapter 5** we will present the experiments and their results, along with the experimental setup and the hardware used to execute these experiments.

Finally we will discuss the conclusions and possible further future developments on the topic.

In the Appendix we will add the information about the libraries and code used.

Chapter 2

Background

In this chapter we will describe the background needed to understand the rest of the thesis and introduce the related work in the field we are interested in.

The chapter is organized as follows: in Section 2.1 we will introduce a definition of HPC system, the main architectures emerged throughout the years of research and the frameworks used. In Section 2.2 we will introduce the machine learning background needed to understand the thesis and the algorithms used, along with some basic definitions of terms used throughout our work. In Section 2.3 we will explore the prior works done in the field of fault classification and emphasize what's new and original in our research work.

2.1 HPC systems

While there is no formal definition, an High-Performance Computing system (HPC), often called "supercomputer", is a computing system with a level of performance which is much higher than a general-purpose computer.[1] Indeed usually the applications executed on an HPC system require a huge number of computational and memory requirements that cannot be satisfied by using a common Personal Computer.[2] Speed, scalability and flexibility are key aspects for an HPC system. Their performance gets usually measured in FLOPS (floating-point operations per seconds) and researchers are trying to build powerful systems trying to achieve Exascale performances (1 exaFLOPS: 10^{18} FLOPS).

HPC systems are used for solving problems in a wide range of fields: from *Big Data analysis* to *bioinformatics* and, in general, to try to tackle *NP-hard* problems. Over the years, many architectures have been proposed, from *vector processors* based on SIMD to clusters.

[1]`https://en.wikipedia.org/wiki/Supercomputer`

[2]`http://www.hpcadvisorycouncil.com/pdf/Intro_to_HPC.pdf`

2.1.1 Architectures

The first systems, such as the Cray 1, were based on *vector processors* which relied on the SIMD (Single Instruction Multiple Data) paradigm [HW10]. This paradigm is based on the assumption that a single instruction is applied to a large number of arguments of the same type, that is a vector. This approach is ideal for achieving peak performances for “vectorizable” code.

However the previously mentioned kind of architecture fell out of favour with the rise of powerful RISC-based massive parallel machines [HW10]. However MPP (Massive Parallel Processors) had an high cost and low performance despite their price.

Then the trend moved to the use of SMP (symmetric multiprocessors) systems, which make use of a small number of RISC processors tightly integrated in a cluster. With this kind of architecture, there were two basic way to access the memory: S-COMA (Simple Cache-Only Memory Architecture) and ccNUMA (cache coherent Non Uniform Memory Access). This means that these systems fell into the category of SM-MIMD systems (Shared Memory - Multiple Instruction Multiple Data) [VdSD95]. Figure 2.1, taken from [VdSD95], shows how these kind of systems looked like.

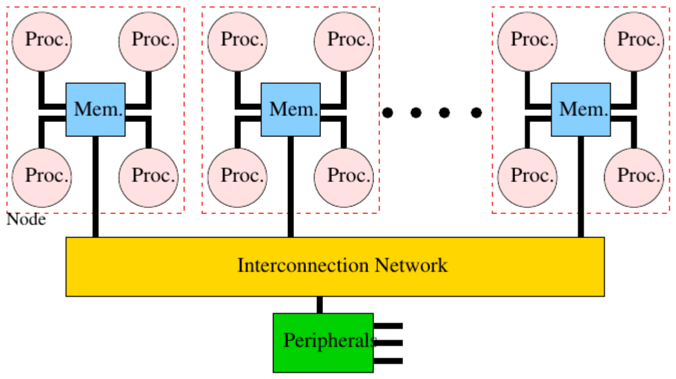


Figure 2.1: SMP SM-MIMD HPC systems.

However since the previously mentioned architecture suffers from scalability and distributed computing was difficult to use in the context of parallel performances[3], modern HPC systems have moved to an architecture based on clusters. The adoption of clusters, which are collection of workstations/PCs or servers connected by a local network, has exploded since the design of the first Beowulf cluster in 1994. This is due to the fact that for such systems there’s a (potentially) low cost for both hardware and software, but at the same time there is high possibility for the users to control and customize their systems [VdSD95].

[3]`http://www.hpcadvisorycouncil.com/pdf/Intro_to_HPC.pdf`

Clusters proved to be more affordable, cost effective and scalable than specialized systems, while maintaining a reliable architecture. Figure 2.2 shows how a typical cluster architecture looks like.

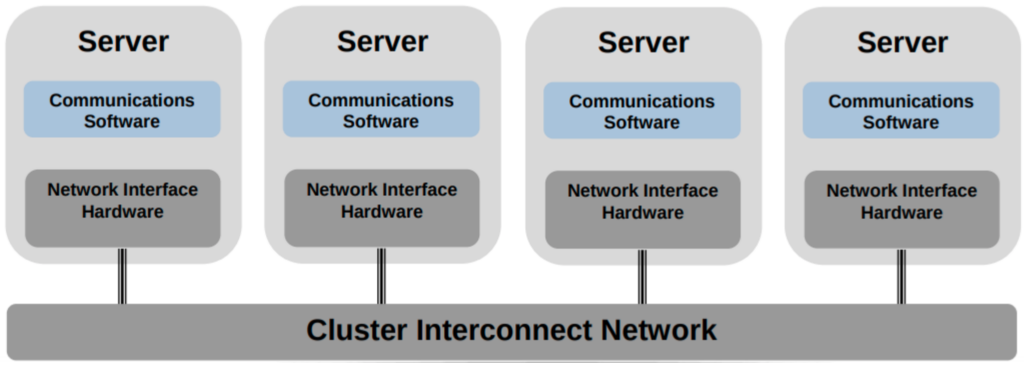

Server
Communications Software
Network Interface Hardware
Server
Communications Software
Network Interface Hardware
Server
Communications Software
Network Interface Hardware
Server
Communications Software
Network Interface Hardware
Cluster Interconnect Network

Figure 2.2: Clusters architecture, taken from `http://www.hpcadvisorycouncil.com/pdf/Intro_to_HPC.pdf`.

Modern cluster-based HPC systems try to optimize each part of their architecture: the *frontend section*, the *backend section*, the *computing nodes* and the *networking infrastructure* [Net]. The frontend section is the one the users interact with, while the backend section is responsible for the control, management of the system and job dispatching. The computing nodes are the ones that are mainly used for executing the jobs and provide the computational power of the system. The network infrastructure is also another vital component for the HPC system and during the years of research various technologies have been used to speed up the transmission of the data in clusters, from Infiniband to three-dimensional torus interconnections [Kni07, ABC+05].

Modern HPC systems based on the cluster architecture can be very huge and complex: in Bologna researchers are building a 250 Pflops (250 trillion operations per second) system, called Leonardo, equipped with 1536 servers with Intel Xeon Sapphire processors, 3456 servers equipped with Intel Xeon Ice Lake and NVIDIA Ampere architecture GPUs and using NVIDIA Mellanox HDR InfiniBand connectivity.[4] The architecture will be based on 5000 computing nodes.[5] This system, which is estimated to be built by the end of 2021, will outperform the HPC system called Marconi100, which in June 2020 was ranked in the ninth position on the global TOP500 list of the world's most powerful supercomputers.[6]

[4]`https://www.cineca.it/en/hot-topics/Leonardo-announce`

[5]`https://www.cineca.it/en/our-activities/data-center/hpc-infrastructure/leonardo`

[6]`https://www.top500.org/lists/top500/2020/06/`

2.1.2 Software and programming models

Besides benchmark applications such as the famous LINPACK [DBMS79], the applications that run on HPC systems are developed using various techniques, depending on the type of architecture that it is addressed.

Shared-memory parallel programming models can make use of OpenMP[7], while distributed-memory systems must rely on Message Passing Interface (MPI), since there's no way for a processor to address the memory assigned to another processor. However MPI can also be viewed as a *programming model* and also be used on shared-memory or hybrid systems [HW10].

2.2 Preliminaries

2.2.1 Definitions

First of all a definition of *fault* is needed: Gainaru et al. define a *fault* as an anomalous behaviour at the software or hardware level that can lead to illegal system states (*errors*) and to service interruptions (*failures*) [GC15]. In this thesis we are interested in the automatic detection of these faults via *supervised* machine learning techniques [Mit99]. Alpaydin [Alp20] defines *machine learning* as the series of programming techniques used in order to optimize a performance criterion using example data or past experience. The program, usually called *model*, optimizes its parameters during the learning phase and it can then be used to either make predictions (*predictive model*) or gain knowledge from the data (*descriptive model*), or both.

Supervised techniques make use of *labelled* data during the training phase. More specifically, each data point supplied to the learning algorithm is tagged with the class it belongs to. In more simple and precise terms, supervised learning consists in using a collection of input-output pairs in order to learn a function that predicts the output for new inputs [RN$^+$13]. As the title of the thesis suggests, we will deal with *classification*, that is automatically identifying to which category (also called *label* or *tag*) an input belongs; this is a common task solved by *supervised* learning.

Instead *unsupervised* learning [Mit99, MMC13] requires data containing only the features that describe the system state, without labels; useful properties about the data and its structure will be learned by the algorithm. *Clustering* is the automatic identification of different groups of inputs such that each group is made up of input points that share some common properties; *clustering* is a common task that can be approached using *unsupervised* techniques.

[7]`https://www.openmp.org/specifications/`

Finally *semi-supervised* techniques rely on partially labelled data.

Sometimes we will refer to the term *overfitting*: overfitting happens when the model is too complex and specialized over the peculiarities of the training data given as example. Basically it relies too much on the examples, on which it works very well, but it doesn't generalize well enough for working on new novel data.

We will also refer to the process of *feature selection*, which is the process of selecting a subset of the available features in order to save up memory space and prevent overfitting, without compromising the evaluation metric used for the classification results (in our case, the F1-score). Feature selection can be done by hand with expert knowledge (especially when the number of total features is low) or automatically via algorithms.

For training and testing we will refer to two strategies: splitting the dataset in training and testing set (for example 60% as training and 40% as testing set) and *k-fold cross validation*; this last approach is based on dividing the set of observations into k groups, or folds, of approximately equal size. The first fold is treated as a validation or test set, and the model is trained on the remaining $k - 1$ folds [JWHT13].

Some of the datasets used in the literature for this topic are obtained via *fault injection*, that is the deliberate triggering of faults and anomalies in a computing system done in order to observe the behaviour of the system in a controlled environment [HTI97].

2.2.2 Decision trees and random forests

A Decision Tree is a model based on *supervised* learning; each node in the tree represents a test on the value of one of the input attributes, while the branches from the node are labelled with the possible values. The leaves of the tree are labelled with the value returned by the learned function [RN$^+$13]. Usually the algorithms used to learn this kind of models make use of the concepts of *information gain* and *entropy* [SW98]. Russel et al. [RN$^+$13] define entropy as a measure of uncertainty of a random variable and say that the acquisition of information corresponds to the reduction of entropy.

The goal of many algorithms designed to build Decision Trees (such as ID3 [Qui86], C4.5 [Qui14], CART [BFSO84], etc.) is to design the tree in such a way to minimize the depth of the tree. The main idea is to choose each time the attribute that goes as far as possible toward giving an exact classification of the example provided.

The Random Forests are an example of *ensemble learning*: instead of training one single Decision Tree, training happens on many Decision Trees, each on a random subset of training set or a random subset of the input features, then their predictions get combined (for example using the *mode* of the classes for classification or the mean/average prediction of individual tree for regression) [Alp20, Ho95, Ho98]. Random Forests models, along with *decision tree pruning*, can mitigate the overfitting problem typical of Decision Trees.

2.2.3 Gaussian mixture models

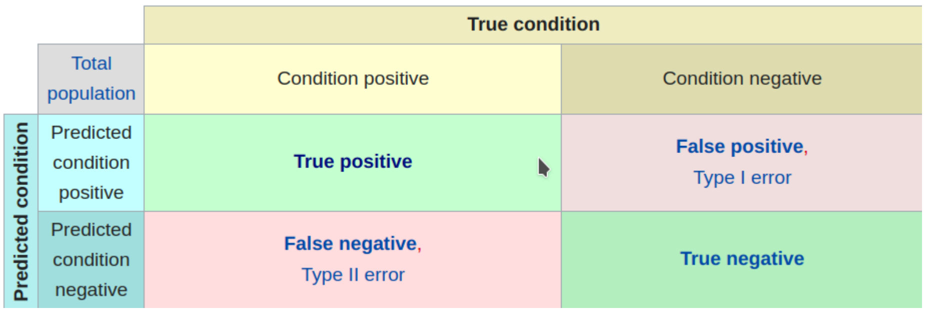
A Gaussian Mixture Model is a probabilistic model that assumes that the data given as input belong to a mixture of a finite number of Gaussian distributions with unknown parameters [PVG+11]. Thus the GMM model can be used for clustering and the most common algorithm used for optimizing the parameters is the *expectation-maximization* (EM) [YSM11, XJ96, DLR77].

GMM models have been the preferred over K-means in various papers. For example, Patel et al. found out that Gaussian Mixture Models perform better than K-Means, because, in the context of cloud workloads, they can discover complex patterns and group them into cohesive, homogeneous components that are close representatives of real patterns within the dataset [PK20]. Moreover Ozer et al. [ONTS20] successfully used GMM models to characterize the performance of an HPC system's components and identify anomalies using sensor data.

2.2.4 Evaluation metrics

Throughout the work done for the thesis we exploited the use of these evaluation metrics: F1-score, sensitivity (also called *recall*), specificity, false positive rate and false negative rate.

Given the following *confusion matrix* in Figure 2.3, here we will define each evaluation metric; in the following equations TP stands for True Positive, TN for True Negative, FP for False Positive and FN for False Negative.[8] As can be seen, the *confusion matrix* is a table organized as follows: each row represents the instances in a predicted class, while each column represents the instances in an actual class [Pow08].



		True condition	
	Total population	Condition positive	Condition negative
Predicted condition	Predicted condition positive	**True positive**	**False positive**, Type I error
	Predicted condition negative	**False negative**, Type II error	**True negative**

Figure 2.3: Confusion Matrix.

[8]`https://en.wikipedia.org/wiki/Sensitivity_and_specificity`

Sensitivity (also called **recall** or **True Positive Rate, TPR**) is the proportion of true positives that are correctly identified by the test [AB94]. It is calculated as follows:

$$sensitivity = \frac{TP}{TP + FN} \tag{2.1}$$

Specificity (also called **True Negative Rate, TNR**) is the proportion of true negatives that are correctly identified by the test [AB94]. It is calculated as follows:

$$specificity = \frac{TN}{FP + TN} \tag{2.2}$$

Before introducing **F1-measure**, we have to talk about **precision**. **Precision** denotes the proportion of predicted positive cases that are real positives [Pow20]. Precision is calculated as follows:

$$precision = \frac{TP}{TP + FP} \tag{2.3}$$

F1-score (or **F1-measure**) is a way for combining both precision and recall, since it's their harmonic mean. It is calculated as follows:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.4}$$

The **False Positive Rate**, which can be interpreted as the probability of false alarm, is the proportion of all negatives that still yield positive test outcomes and it is calculated as follows:

$$False\ Positive\ Rate = \frac{FP}{FP + TN} \tag{2.5}$$

The **False Negative Rate**, also known as miss rate, is the proportion of positives which yield negative test outcomes with the test and it is calculated as follows:

$$False\ Negative\ Rate = \frac{FN}{TP + FN} \tag{2.6}$$

Finally the **accuracy**, which is the measure of all correctly identified classes, is calculated as follows:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.7}$$

In our work we used mainly the *F1-measure* instead of the *accuracy*, because the *F1-measure* tends to take into account how the data is distributed and is useful when there's imbalance among the classes in the data. Moreover it's a good trade-off between *precision* and *recall.*

2.3 State of the art

Previous work of other researchers

Tuncer et al. [TAZ+17] leverage resource usage and performance counter data collected during application runs by a monitoring framework instead of relying on log files. This data is then converted in statistical features, such as *percentiles*, *standard deviation*, *skewness*, *kurtosis*, *serial correlation*, *linearity* and *self-similarity*, that are able to summarize the system's state. These features are then supplied to different ML algorithms in order to classify the behaviour of the system.

Baseman et al. [BL16] use a similar approach, more specifically their work is based on a statistical technique called *classifier-adjusted density estimation*. They then use a random forest classifier in order to rank how anomalous a specific data point is.

Other scientists focused on reducing the dimension of the collected data without hindering too much the classification performances. Bodik et al. [BGF+10] used a specific representation of the system state called a *fingerprint*, which is based on quantiles related to different time epochs. Netti et al. [NTOS20] proposed a way to compute a compact representation, called a *signature*, from multi-dimensional time series data based on the *Correlation-Wise Smoothing* method. The researchers showed that this method is useful not only for having a compact representation of the system's status (that can be used by ML algorithms), but also for visualization purposes, since the signatures produced can be visualized as heatmaps.

Another important source of inspiration for the work done in this thesis is the paper written by Netti et al. [NKB+20], where they presented FINJ, a novel tool for fault injection, and performed fault injection and fault classification on the Antarex dataset[9]. In this work the researchers injected 8 faults in the system and then used various classification models to perform fault classification. More precisely they made experiments using a Random Forest (RF), Decision Tree (DT), Linear Support Vector Classifier (SVC) and a Neural Network (NN) with two hidden layers, each having 1000 neurons. The feature vectors were created via aggregation; each feature set corresponded to a 60s aggregation window and was related to a specific CPU core, while the time step between consecutive feature sets was 10s. For each metric they computed several statistical measures over the distribution of values within the aggregation window. These measures were *average*, *standard deviation*, *median*, *minimum*, *maximum*, *skewness*, *kurtosis*, and the *5th, 25th, 75th* and *95th percentiles*. The best classifier was the RF model, with an overall F-score of 0.98. This is the reason why throughout the work done for this thesis the

[9]`https://zenodo.org/record/2553224`

main model used for classification was the RF model.

As for clustering, we were inspired by the work of Ozer et al. [ONTS20], where they use Bayesian Gaussian Mixture Models in order to characterize the performances of an HPC system's components and identify anomalies using sensor data. For the CPU core-level analysis they exploited the use of instructions and cache-misses metrics. Instead for the compute node-level analysis they focused on power consumption, temperature and CPU-idle time. Finally for the rack-level analysis they used the water's inlet temperature in the racks, its return temperature and the amount of heat removed from the racks quantified in Watts. Moreover they were able to detect outliers exploiting the use of the *Mahalanobis distance.*

Another important work in the field is the one by Borghesi et al. [BBL+19], where they use a *semisupervised* technique to perform anomaly detection. More specifically their approach is based on building an autoencoder which is capable of reconstructing its input. This autoencoder is trained on non anomalous data (cpu frequency governor set to *conservative*) and the reconstruction error is used to detect anomalous states (belonging to *powersave*, *performance* and *on-demand*). The researchers also compared the performances of two different models: a node-specific model and a single general model, that is a single model used for all the computing nodes. To the best of our knowledge, this is the only prior work done for multi-node models.

As can be seen from the previous paragraphs, most of the research done in the scientific literature on this topic is based on building a model tailored for a specific computing node. Instead we wanted to focus on trying to build a single general model for all computing nodes, mitigating the problems that can arise from the presence of different characteristics in each computing node.

To the best of our knowledge, there is very little work done on building a single general multi-node model. Borghesi et al. [BBL+19] did a similar work, but they deal with performance anomalies that are based on different configurations of the CPU frequency governor rather than with faults and errors that can cause the disruption of computation. Moreover it is based on different techniques, more precisely on deep learning with *semisupervised* technique. The work by Pecchia et al. [PCKI11] deals with the issues of the tuple heuristic, used to group the error entries in the log, in multi-node computing systems and doesn't deal with fault detection using machine learning. IDÉ et al. [IK04] propose a method for anomaly detection based on multi-node computer systems relying on principal eigenvector of the eigenclusters of a graph, where each node represents a *service* and each edge represents a dependency between services. The data is then analyzed by their own novel online algorithm that doesn't use traditional machine learning or deep learning. In the work by Preuveneers et al. [PRT+18] the researchers show

how incremental updates on a machine learning model can be distributed in a federated learning environment and chained together on a distributed ledger. However the focus is on intrusion detection (malicious nodes with malicious behaviour) rather than detection of faults, errors and anomalies that can disrupt the computation.

The process that leads to the construction of a model that generalizes well for all computing nodes, using the best minimal set of computing nodes, is the main scientific contribution of this thesis.

Our previous work

The starting point for the thesis was built up from the internship, whose works began the 22nd of June 2020 and ended the 7th of September 2020 [Cov20]. During this period we used data coming from the CooLMUC-3 HPC[10] system located at the Leibniz Supercomputing Centre (LRZ) in Munich, while it was subject to fault injection. This dataset was different from the one used in the thesis, more precisely it was made up of 7 datasets with faults injected in them (instead of 32) and instead of PENNANT there was LAMPS. Moreover the datasets were acquired in separate user jobs, without collecting any sensitive user data. Instead the new dataset used for this thesis is larger in size and more representative of real systems. More details can be viewed in the final internship report [Cov20]. We executed different experiments on computing nodes belonging to different acquisition experiments in order to identify on which dataset the model performed better. Then the aim was to use various techniques of data processing and classification in order to improve the classification results and obtain a new baseline for future works. The work done during the internship was heavily inspired by the paper written by Netti et al. [NKB+20] (whose work on the Antarex dataset have been described in Section 2.3) and the results obtained have been used as the starting point for working on the this thesis.

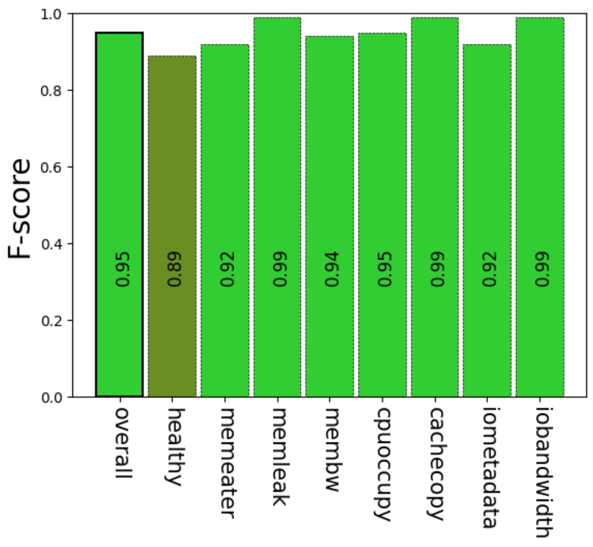
During the data acquisition phase, several HPC proxy applications were executed. These applications come from the Coral-2 suite[11] and from LINPACK [DBMS79]. More specifically, they are *kripke*, *AMG*, *Neckbone*, *LAMMPS*, *HPL*. Along with these applications, the system could also be in *idle* state. The faults, injected using the HPAS framework [AZA+19], were *memeater*, *memleak*, *membw*, *cpuoccupy*, *cachecopy*, *iometadata*, *iobandwidth*. Along with these anomalies, the systems could also be in *healthy* state.

After many experiments we concluded that the best results have been obtained by computing the horizontal aggregation of CPU-specific metrics, by computing 11 features per metric and

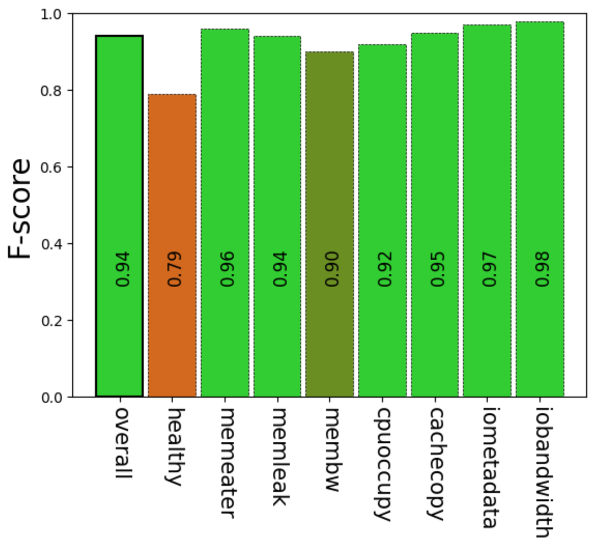
[10]`https://doku.lrz.de/display/PUBLIC/CoolMUC-3`

[11]`https://asc.llnl.gov/coral-2-benchmarks`

by selecting the optimal number of features via RFECV. Minor differences can be observed by using as labelling strategy either the mode or the most recent fault. Class balancing was also important in order to get better classification performances. Figure 2.4 shows the best results obtained at the end of the internship.



a) best results using the mode



b) best results using the most recent fault

Figure 2.4: Best results so far with different labelling strategy.

We showed here the final best F1-scores obtained during the internship because they will serve as a baseline and starting point for the thesis. Moreover the whole process of feature generation and model configuration will be based on this work.

Chapter 3

Dataset

In this chapter we will introduce the dataset we acquired and prepared for the experiments. More specifically, the chapter contains the following sections: in Section 3.1 we will talk about how the researchers of LRZ acquired the data needed for the research. In Section 3.2 we will describe how we processed the data in order to have the final feature vectors needed in our experiments.

3.1 Data acquisition

The HPC system

The CooLMUC-3 is made up of 148 nodes, each one of them equipped with 64 cores (4 hyperthreads per core), which operate at a nominal frequency of 1.3 GHz, and 96GB of DDR4 memory. The network layer uses Omnipath switches and the infrastructure makes use of water cooling. The system can reach 459 TFlop/s as peak performance.[1]

Experimental setup

The data has been collected from the CooLMUC-3 HPC system using DCDB [NMA$^+$19], a comprehensive monitoring system designed to be scalable and extensible. DCDB also provides a series of command line tools (libDCDB) in order to make queries on the storage backend (Apache Cassandra).[2]

The DCDB framework was configured to sample fine-granularity sensor data (for example from ProcFS and CPU *perfevents*) from 32 compute nodes of the system. The collection process took roughly 48 hours and the dataset does not contain per-CPU core performance counter,

[1] `https://doku.lrz.de/display/PUBLIC/CoolMUC-3`

[2] `https://gitlab.lrz.de/dcdb/dcdb`

because the DCDB framework was configured to compute and store the minimum, maximum, sum, median, 25th and 75th percentiles for each of them instead, resulting in 167 metrics for each node.

Benchmark applications

While acquiring the data, several HPC proxy applications coming from the Coral-2 suite[3] and from LINPACK [DBMS79] were executed. The applications and their labels are the following:

20. *Kripke*: a structured deterministic transport using wavefront algorithms, which stress both memory and network latency and bandwidth;
21. *AMG*: a parallel algebraic multigrid solver for linear systems;
22. *Nekbone*: a compute-intensive mini-app derived from the Nek5000 Navier-Stokes CFD solver;
23. *PENNANT*: a mini-app for hydrodynamics on general unstructured meshes in 2D (arbitrary polygons), which makes heavy use of indirect addressing and irregular memory access patterns;
24. *HPL*: measures performance in solving a system of linear equations.

Label 0 is used to identify the *idle* state.

Fault injection

In order to reproduce anomalous conditions the HPAS[4] framework [AZA$^+$19], which consists of a set of synthetic anomalies that reproduce common root causes of performance variations in supercomputers, has been used. In total seven fault programs are used to simulate anomalies: they use processes that run in user space and do not require any hardware or kernel modification. The programs and their labels are the following ones:

1. *memeater*;
2. *memleak*;
3. *membw*;
4. *cpuoccupy*;

[3] `https://asc.llnl.gov/coral-2-benchmarks`
[4] `https://github.com/peaclab/HPAS`

5. *cachecopy*;

6. *iometadata*;

7. *iobandwidth*

Label 0 is used to identify *healthy* states where no fault is running. The programs *cpuoccupy* and *cachecopy* are used to generate CPU faults, the programs *leak*, *memleak* and *membw* are used to produce memory faults while *iobandwidth* and *iometadata* simulate an I/O fault. More specifically *cpuoccupy* affects CPU performance, *cachecopy* emulates cache contention or cache lines to be unexpectedly evicted, *leak* allocates an array of a given size and fills it with random values, *memleak* models a memory leak, *membw* creates contention in the memory bandwidth, *iobandwidth* causes contention in the disks of the storage servers and *iometadata* stresses the metadata server.

Compared to the dataset used during the internship, the intensity of the *iobandwidth* and *iometadata* faults had to be reduced so as to avoid distrupting the cluster's operation. All fault programs are launched on a random subset of 16 up to 32 nodes, except for the *iobandwidth* and *iometadata* faults; these are launched on random subsets of 8 to 16 nodes, hence they will have roughly 50% less observations than other faults.

3.2 Data preparation

Before analyzing and using the data with Python libraries like Pandas [pdt20] and Scikit-Learn [PVG+11], we needed to code a pipeline whose purpose was to extract the data from Apache Cassandra, store it in CSV files, one CSV for each compute node, and then compute feature vectors from them. Figure 3.1 summarizes the pipeline used for data preparation and described in the following sections.

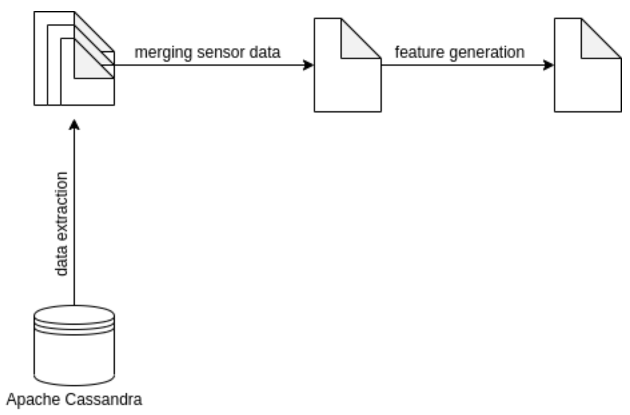


Figure 3.1: Data preparation pipeline.

Data extraction

The first step consisted in using *dcdbconfig* in order to obtain the list of all sensors stored in Apache Cassandra and then using *dcdbquery* to actually obtain the data of each sensor as a CSV. To do so, the information stored in the log file of the corresponding SLURM job was used. Both *dcdbconfig* and *dcdbquery* are taken from libDCDB, which is a part of DCDB [NMA+19].

Merging sensor data

In the second phase of the pipeline, we coded a Python script in order to merge the CSVs previously obtained in a single CSV per compute node. At this stage we introduced various checks and operations to align the timeseries data and check for monotonic columns, converting them to their delta equivalent.

Feature generation

For the third stage we coded another Python script whose purpose is to generate feature vectors for each CSV. This script accepts various command line arguments and allows the user to compute 11 (or 6, depending on the argument provided) features for each metric in the dataset and considers different durations for the aggregating window (the default value is 60 seconds). For our experiments, the indicators used are the average, standard deviation, the 25th and 75th percentiles, the sum of the changes measured within the aggregating window, the last monitored value for each metric, the minimum and maximum value, the 5th and the 95th percentiles and the median of the distribution of the values measured within the aggregation window. Moreover the user can decide which labelling strategy to use (most recent fault running in the corresponding time window or using the mode, that is the most frequent label in the corresponding time window).

Efficiency of data manipulation

The previously described scripts underwent various debugging and optimization phases in order to make them faster and more memory efficient. More specifically, the Bash script used in data extraction is designed to run different *dcdbquery* tasks in parallel in background (the number of parallel tasks can be provided as a command line argument).

The script used for merging sensor data also underwent optimizations and uses wisely thresholds to merge lists of Pandas' dataframes. Moreover we understood the various performance implications and memory footprints of some Pandas' functions, like `DataFrame.join`, `DataFrame.align` and `Pandas.merge`.

Finally, the Python script used for feature vectors generation makes use of the process-based parallelism using the multiprocessing Python package instead of of using multithreading, whose performances are limited by the GIL (Global Interpreter Lock). Moreover, to reduce the memory footprint, the script reads and computes the data from CSV previously obtained in chunks (the number of lines per chunk can be set as a command line argument), making it capable of handling large files and large amount of data.

Chapter 4

Methodology

In this chapter we will describe our methodology used during the experimental stage. More precisely, we want to show how we want to achieve our goals, that is how to perform feature selection and other metaparameters exploration and how to build multi-node models. In this last case we need to figure out how many nodes are necessary for the training phase and which specific nodes to choose. We also want to compare multi-node models against single-node models.

The chapter is organized using the following sections: in Section 4.1 we will present how to obtain models for each specific node. In Section 4.2 we will instead focus our attention on the multi-node case, explaining how we choose good training nodes and the best metaparameters.

4.1 Single-node baseline

In our experiments we want to compare how a single general multi-node model compares to models specifically tailored for each node. For this reason, it is vital to present how such node-specific models are obtained: for each node a train-test split is produced; we then train the random forest model on the training set using only a subset of the features and test it on the testing set. Then we consider the classification results across all the testing sets and plot some box plots in order to compare this kind of models with multi-node models. Indeed the results obtained from node-specific models will be used by us as a baseline during the research, since we want to see if (and which) multi-node models can outperform them.

4.2 Multi-node fault classification

Multi-node models are classifiers that can be used also for fault detection in nodes which are different from the ones used for training. As we've already said, our aim is to obtain a single multi-node model that generalizes well enough in order to be able to obtain good classification performances on various different nodes. Before solving the issues presented in the following sections, we identified the compute nodes in which the faults were more balanced, in order to use them as the training pool. Finally, unless otherwise stated, we executed multiple runs for the various experiments that will be presented in Chapter 5.

4.2.1 The number of training nodes

In order to reach our goals, one of the issues we had to tackle was how to choose the right amount of training nodes for the classifier. To deal with this issue we decided to adopt an incremental approach.

The incremental approach consists in different experiments in which each time we train a model increasing the number of training nodes by one. The training nodes are chosen randomly inside a pool of well balanced training nodes. In this way we can observe if the results get better when the number of training nodes increases; moreover we can also find out if the results stabilize when using a number of nodes above a specific threshold. This last case would be very ideal, since it would become easier to find a minimal number of training nodes for our multi-node classifier.

Choosing the right amount of training nodes is important because we want to develop a methodology that is scalable; if the number of training nodes required is too high, we would need to perform fault injection and training on an high number of training nodes, which can take effort and be time consuming. Instead if the number of training nodes is too low, the classification performances (such as the F1-measure) can be poor and not optimal.

4.2.2 The actual training nodes

The other issue we had to deal with was how to choose the best specific nodes for training; to tackle this issue we decided to compare random selection with a clustering-based approach.

Random selection consists of selecting randomly k training nodes from the training pool, which was a subset of the whole dataset since we restricted this pool to be made up of compute nodes with well balanced faults.

However random selection doesn't take into account the peculiarities of each computing node. Indeed each one of them can have peculiar sensors' readings because of an imbalance in the job scheduler or because of how a specific application affects each computing node, using heavily the resources of some computing nodes and using very little resources of other different nodes. Obviously all of this will be reflected on the features vectors, which are computed from the metrics and that will be used by the classifier. For this reason we also took into consideration a more informed strategy for choosing the training nodes, that is the clustering-based approach.

The clustering approach consists of selecting nodes which are representative of k clusters and use them as training nodes in order to have a model that generalizes better and prevents overfitting. We performed clustering using *Gaussian Mixture Models*, inspired by [ONTS20], and hand-picked the metrics using expert knowledge. We also filtered out specific faults (using only *healthy* states) and applications running on the compute nodes; the reason behind this is that we don't want the selection process to depend from fault injection, which can be impractical with an increasing number of compute nodes. We then used the minimal *Mahalanobis distance* to identify good candidate nodes. Finally we compared the results obtained with the incremental random approach, the clustering-based approach and the single-node baseline using mainly box plots about the evaluation metrics.

The algorithm used for clustering is the following one:

```
input  : Folder containing the dataset
output: candidates: Candidate nodes for each cluster
begin
    Select the faults to consider;
    Select the applications to consider;
    Select the columns/metrics to perform clustering on;
    k ← number of clusters/candidate nodes to find;
    Choose outlier threshold (optional);
    Choose to divide each compute node in half (optional);
    for each compute node of the dataset or each half of them do
        Filter out the data not needed using the faults and applications to consider;
        Compute the averages of each column;
        Add the resulting row to df;
    end
    clusteredDF ← GaussianMixtureModel(df, k);
    if Outlier threshold chosen then
        clusteredDF ← DetectOutliers(clusteredDF, MahalanobisDistance);
    end
    candidates ← FindCandidates(MahalanobisDistance, clusteredDF);
end
```

Algorithm 1: Clustering algorithm.

4.2.3 Metaparameters

When we were considering the methodology to follow, we also needed a way to find the best configuration for the metaparameters, which are the subset of the features, normalization, shuffling, sampling and class balancing.

We already talked about *feature selection* in Section 2.2.1. It is done to reduce the memory requirements of the process without impacting negatively on the classification performances. Without feature selection, we wouldn't be able to perform experiments on an high number of training nodes.

Normalization implies the adjustment of values that vary on different ranges to make them vary inside a common range. This process can be done in different ways, such as using Standard Score Scaling or Min-Max Scaling.[1] We chose Min-Max scaling, which is calculated using the following equation, where X' is the new scaled value, X the old value, X_{min} the minimal value that X can assume, while X_{max} is the maximum value:

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{4.1}$$

Since random forest models work on thresholds and the metrics may have different ranges, normalization may help the classifier in such a way to achieve better F1-scores.

Shuffling consists in randomly changing the order of the feature vectors; this technique may mitigate the negative impact of potential time regions in which some sources of interference might be present.

Sampling consists in choosing a subset of the feature vectors in order to save up space without compromising the results. Along with feature selection, it might be useful when we want to scale up training on an higher number of training nodes.

Class balancing is generally an operation that is used to make all the classes have almost the same number of instances in the dataset. It is used to prevent overfitting on the most represented classes in the dataset, which would result in a model with poor F1-scores for the less represented classes.

As for feature selection, we picked a random node inside the chosen pool of possible training nodes (under the assumption that any such node can be representative of all the nodes for the feature extraction process) and used a classifier (Decision Tree) to select only a subset of all the features.

Regarding the other metaparameters, we executed a single incremental random experiment (from 1 up to 16 training nodes) for each metaparameter: normalization vs non-normalization,

[1] `https://en.wikipedia.org/wiki/Normalization_(statistics)`

shuffling vs non-shuffling and finally balancing the test set vs non-balancing the test set (the training set was always balanced).

We then investigated further the effects of class balancing (both for training and testing set) and subsampling with multiple runs of incremental random experiments (again each run uses from 1 up to 16 training nodes).

Chapter 5

Experimental study

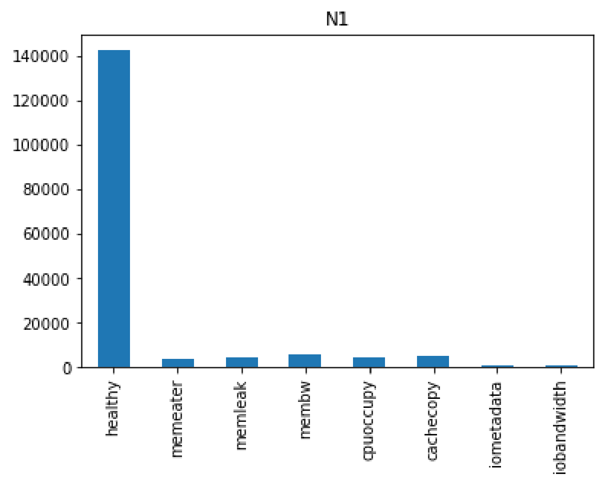
In this chapter we will present the experimental results from which we can draw interesting conclusions.

The chapter is organized as follows: in Section 5.1 we will describe the setup used for the experiments. In Section 5.2 we will show the results of some preliminary experiments. In Section 5.3 we will show the results regarding the feature extraction process. In Section 5.4 there will be the results for the process of metaparameters exploration, done in order to get the best configuration of metaparameters. Finally in Section 5.5 we will compare the results obtained using incremental random experiments and incremental clustering-based experiments.

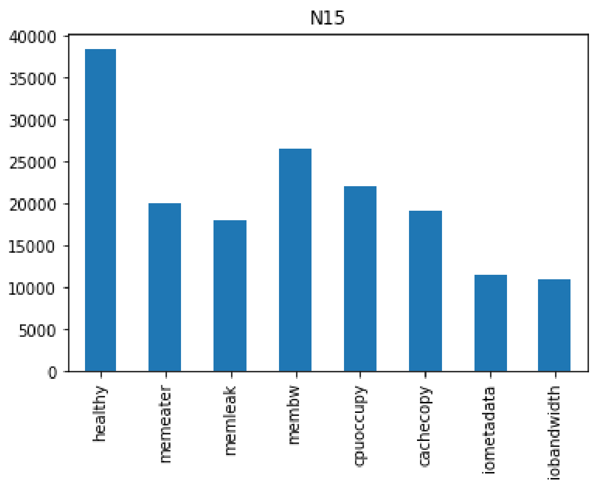
5.1 Experimental setup

First of all, we chose a naming scheme for the 32 computing nodes, which goes from N1 to N32. Then we identified using histograms the most balanced compute nodes, which were the ones between N9 and N24 (extremes included). For this reason, unless otherwise stated, the training pool for the multi-node experiments is made up of these nodes, excluding the so called "edge nodes". In the multi-node case testing is done on all the nodes, including the "edge nodes", but excluding the nodes used for training.

Figure 5.1 shows the difference in faults distribution and balancing between "edge nodes" (in this case N1) and nodes chosen for the training pool (in this case N15).



a) N1



b) N15

Figure 5.1: Histograms of faults in N1 and N15.

Feature generation and selection

As we described in Section 3.2, the script used for feature generation can compute 6 or 11 features per metric. For the following experiments we configured the script in order to use 11 features per metric with an aggregating window of 60 seconds, for a total of 1805 features. The labelling strategy used is the mode, that is the most frequent label in the corresponding time window. We decided to adopt this configuration because it resulted in best results in the experiments done during the internship.

We executed some preliminary experiments using all the available features generated. However, in order to increase the number of training nodes and avoid problems in space complexity, it was vital to perform feature selection. This process also helps to avoid overfitting the model.

We performed feature selection on node N15 using two strategies: getting an optimal number of 16 features using the RFECV algorithm[1] and selecting 100 most important features using the Decision Tree `feature_importances_` attribute. In both cases the whole dataset was balanced using the RandomUnderSampler[2] with "majority" technique (the majority class gets undersampled and its cardinality becomes equal to the one of the minority class). However when testing the effects of feature selection on the same node, we applied a 60-40 split (60% for training and 40% for testing) and we performed balancing with the same technique separately for each set. The evaluation metric used was the F1-score (the overall score is obtained via weighted avarage).

[1]`https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html`

[2]`https://imbalanced-learn.org/stable/generated/imblearn.under_sampling.RandomUnderSampler.html`

At the end, for the next experiments, we ended up using 100 most important features selected using the latter method because it was a good middle ground between selecting too few very important features and too many features. Moreover the results showed minimal differences between the two methods, as we will see in Section 5.3.

Single-node baseline

We conducted the experiments for node-specific models using only 100 most important features and splitting each node in 60% training set and 40% test set. We didn't perform any class balancing in this case. Then for each fault we considered all the results across all the nodes in order to be able to plot the box plots. The evaluation metrics computed are the F1-score, sensitivity, specificity, false positive rate and false negative rate. The overall scores are computed via weighted average.

Configuration of the model

The model used throughout the experiments is the Random Forest Classifier, with 30 estimators and max depth of 20. This was the best configuration that resulted from prior related works and from the internship. Unless otherwise stated, a `random_state` of 42 was used in order to ensure reproducibility. For the multiple runs experiments used for metaparameters exploration, the `random_state` seed was set to the time the instruction was running, in order to fully randomize each run.

Exploration of other metaparameters

In Section 4.2.3 we described the general methodology used for setting the metaparameters. Here we will describe the specific settings.

When using normalization, we normalized the metrics using Min-Max Scaling before computing the feature vectors. When using shuffling, it was performed randomly using 42 as seed for the single run experiments.

As for subsampling, the following strategy was used: the script used allows the user to specify the `--step` parameter that controls subsampling. If this parameter is set to a value $f = 1$, no subsampling will be performed on the training set. Otherwise, if $f > 1$, each time we pick up a node to be part of the training set, the algorithm picks a feature vector, skips the next $f - 1$, picks again a feature vector, and so on. For example, if $f = 3$, each time we pick up a node to be part of the training set, we pick feature vectors in lines 1, 4, 7, and so on. For a file with 166669 vectors, using $f = 3$ gives us only 55557 vectors. When subsampling was considered, we used $f = 3$. We chose this value in order not to be too "aggressive" with subsampling, but at the same time to obtain a good reduction in memory usage.

Regarding class balancing, the Imbalanced-Learn's RandomUnderSampler with "majority" technique was used. When using multiple runs of incremental random experiments we executed 10 runs, each time training from 1 up to 16 nodes.

Clustering

We performed clustering using a web app we coded by exploiting the use of the Streamlit framework[3] in such a way that the user can select the parameters and obtain the results within minutes, without executing each time a script with different command line arguments.

The clustering process follows 4 stages:

1. *Data preparation:* for each node, the data is filtered according to the faults and applications selected by the user and then aggregated computing the mean of each sensor/column that will be used for the clustering process;
2. *Clustering:* the data obtained from the previous step is used as input to the GMM model, that performs clustering;
3. *Outlier detection (optional):* outliers are detected using the Mahalanobis distance and a user-defined threshold, this stage is optional and we didn't use it in the experiments;
4. *Candidates identification:* nodes representative of each cluster are obtained detecting the node with minimal Mahalanobis distance within each cluster.

As we said in Section 4.2.2, we selected the metrics used for clustering using expert knowledge, taking inspiration from the work of [ONTS20]. More specifically, we used the *cache-misses.min*, *instructions.max* and *power*, which are related to memory, CPU and energy consumption. Moreover we filtered the data selecting only the healthy instances and the instances in which *HPL* was running. As we've already said, we didn't want the node selection process to depend from fault injection and the application used should stress all the system components equally.

We considered clustering only for nodes inside the range N9 to N24 (extremes included), because these nodes are the most balanced ones. We configured the Gaussian Mixture Model with default parameters and a `random_state` of 42 (again for reproducibility); however the number of components to retrieve (which will be the number of clusters) can be user defined.

[3] `https://www.streamlit.io/`

When searching for 4 and 5 clusters, we configured the web app in order to divide each compute node in half, considering the first half and second half of the original node as separate nodes, in order to have a larger pool of nodes to perform clustering on; indeed finding 4 or 5 clusters with only 16 nodes can be difficult or give unreliable results. The notation that we used in this case is N⟨X⟩_1 and N⟨X⟩_2 where ⟨X⟩ is the node number (for example N10_1 is the first half of node N10, while N10_2 is the second half).

The *Mahalanobis distance* formula is the following one, used via the Scipy framework:

$$D_M(\vec{x}) = \sqrt{(\vec{x} - \vec{\mu})^T S^{-1} (\vec{x} - \vec{\mu})} \tag{5.1}$$

In Equation 5.1 $\vec{\mu}$ and S are respectively the mean vector and the covariance matrix of the Gaussian distribution to which the vector $\vec{x}$ is assigned by the GMM algorithm.

Hardware and software

The hardware on which we executed the scripts is the following one:

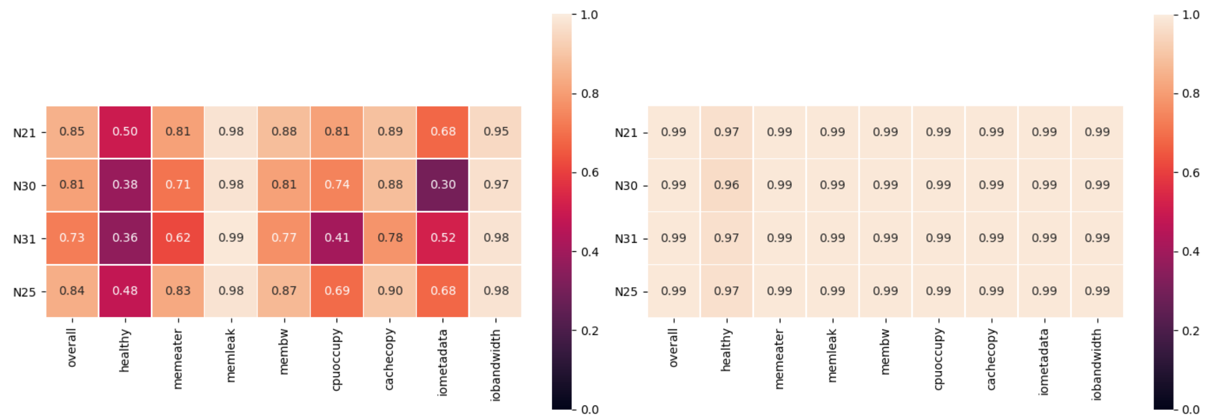
- a laptop with Intel Core i5 5300U 2.9GHz and 16GB of RAM, running Fedora 32;
- a VM instance equipped with Intel Xeon Gold 5120 CPU 2.20GHz and 16GB of RAM, running Ubuntu 16.04.5 LTS;
- a dedicated Linode instance equipped with 48 threads (AMD EPYC 7601 2.2GHz) and 96GB of RAM, running Fedora 32; this instance has been used for memory heavy workloads, such has running the RFECV algorithm.

The libraries and code used for the experiments are specified in the Appendix.

5.2 Initial experiments

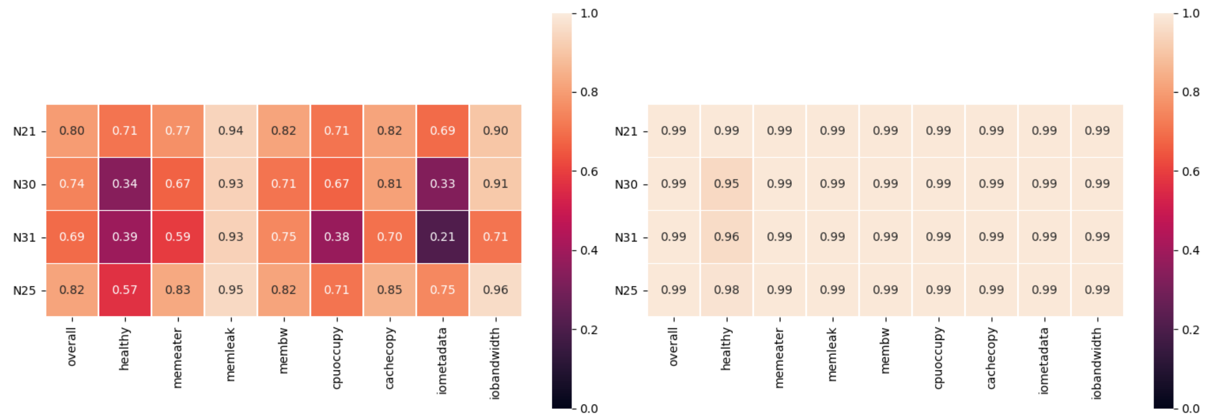
In order to see if we could obtain results similar to our previous work [Cov20], we executed some preliminary experiments with 5 fold cross validation on 4 (out of 32) random nodes. We considered shuffled and non-shuffled data (shuffling is performed setting the `shuffle` parameter of `KFold` to True, meaning that shuffling happens before splitting the data into batches) and we also considered the *last* labelling strategy and the *mode* labelling strategy. We applied class balancing using the “majority” technique each time to both training and testing folds.

The results that show the F1-scores are in Figure 5.2 and Figure 5.3.



a) without shuffling b) with shuffling

Figure 5.2: 5-fold cross validation, mode labelling.



a) without shuffling b) with shuffling

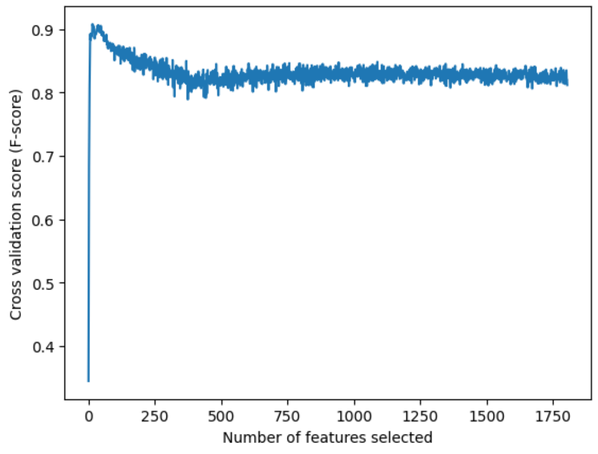
Figure 5.3: 5-fold cross validation, last labelling.

In all the cases, shuffling seems to mitigate the negative impact of potential time regions in which some sources of interference might be present. However the difference between shuffling the data and disabling shuffling is so high that we wanted to investigate further its effect during the metaparameters exploration.

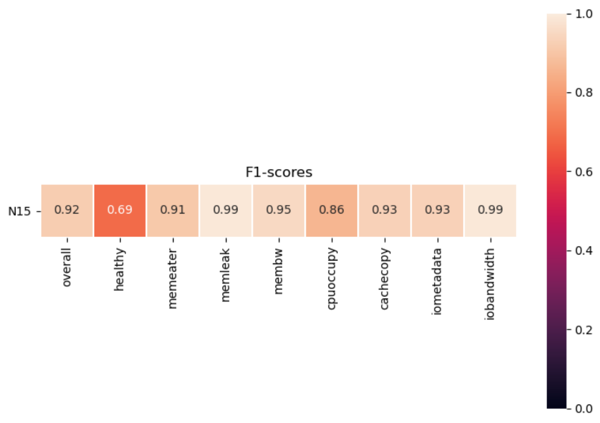
Comparing the heatmaps for the two labelling strategies without shuffling, we can see that with the *last* labelling strategy the classification performances of some faults generally drop a little for some nodes, while the *healthy* state shows very little improvement. For this reason the next experiments will be focused on using the *mode* labelling strategy, which also proved to be better in prior related works done during the internship [Cov20].

5.3 Feature selection

We performed feature selection using the setup explained in the appropriate subsection of Section 5.1. The results, showing the F1-scores, are presented in Figure 5.4 and Figure 5.5.

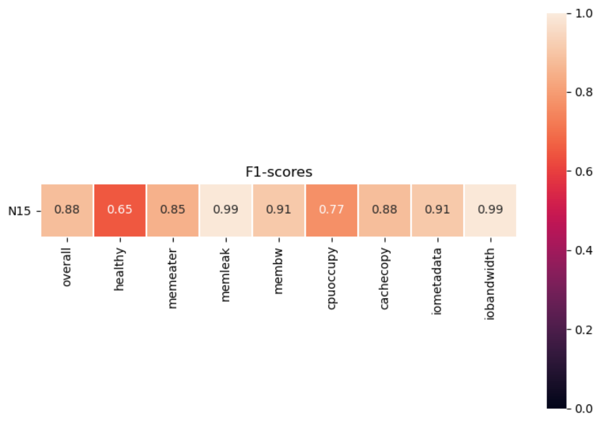


a) overall F1-score changes when the number of features changes

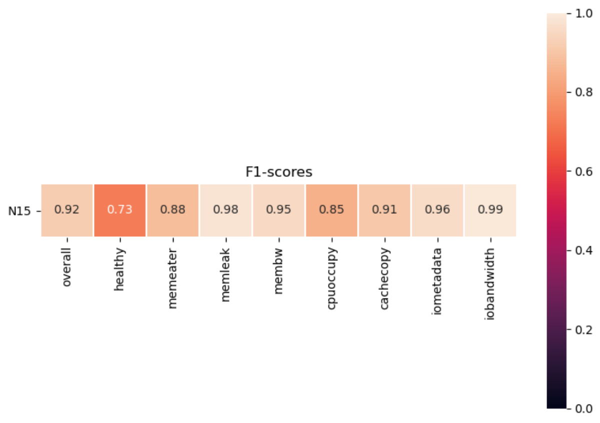


b) results with RF model on a 60-40 split

Figure 5.4: Results of RFECV on N15.



a) with all the features



b) with 100 features

Figure 5.5: F1-scores on a 60-40 split on node N15.

As can be seen from the figures, selecting 100 features clearly outperforms the case in which all the features are used. Moreover between using only the optimal 16 features retrieved by using RFECV and the 100 most important features obtained via DT there's minimal difference, with the *healthy* state that improves a little when using 100 features; for the other faults the differences are inconsistent, sometimes they improve with 100 features, sometimes they are worse, but the difference is very minimal.

As we said, for these reasons, we decided that 100 most important features is a good middle ground for our research.

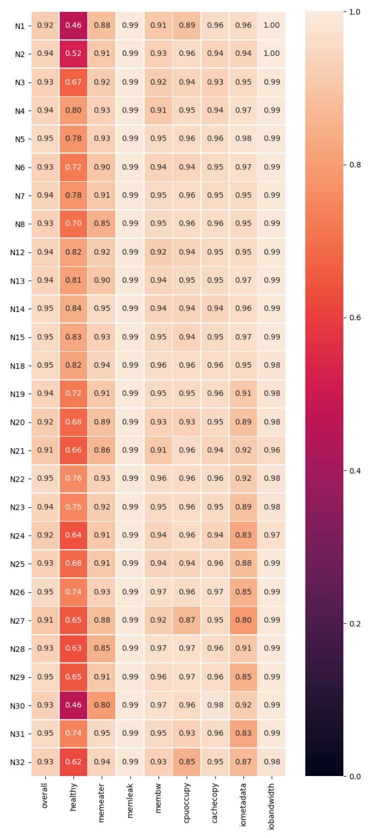
5.4 Other metaparameters and the number of training nodes

In this section we will show the results regarding the metaparameters: shuffling, normalization, class balancing and subsampling. The methodology has already been described in Section 4.2.3 while the experimental setup can be retrieved from Section 5.1. In these experiments, along the way, we increase each time the number of training nodes; this helps us see how the results change when the training nodes are increased and if there is some sort of stabilization after a certain threshold.

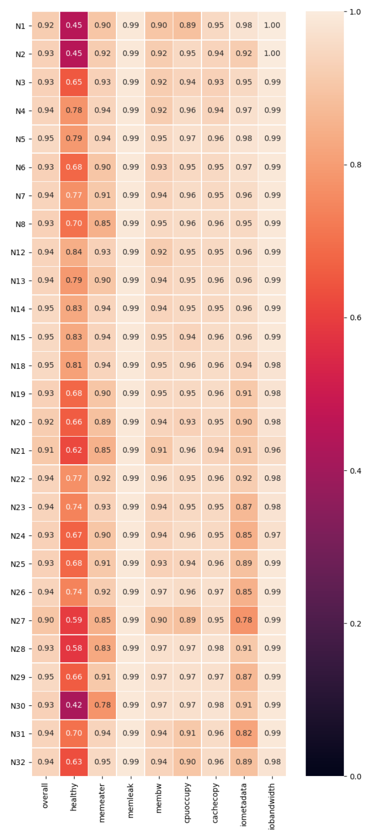
Shuffling

In this experiment we fixed class balancing (on both training and testing set), we used non-normalized data and evaluated the impact of training data shuffling.

In Figure 5.6 you can see the comparisons of the F1-scores between shuffling disabled and shuffling enabled when 5 nodes are used as training set. Even though the results with a different number of training nodes are similar, we decided to investigate further what happens to *healthy* and *memeater* in Figure 5.7 and Figure 5.8 by using box plots.

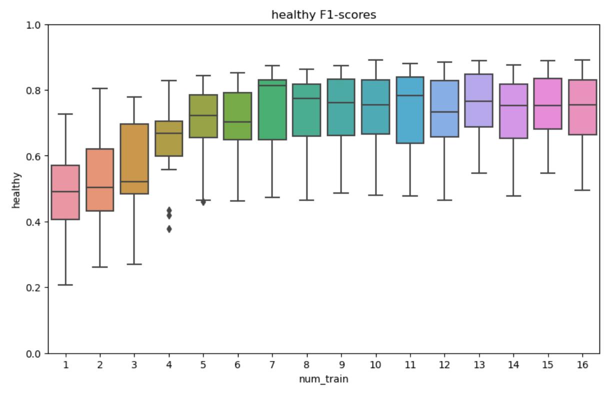


a) without shuffling

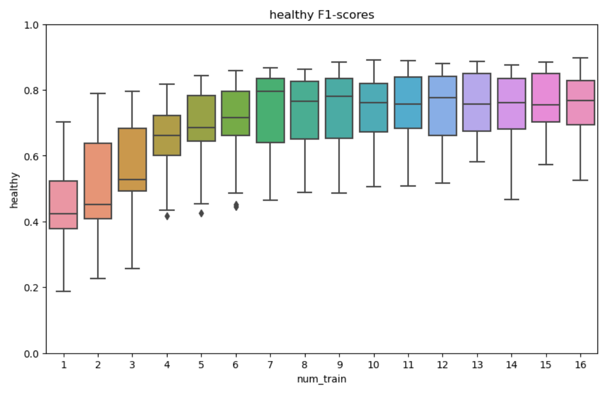


b) with shuffling enabled

Figure 5.6: F1-scores on the test nodes when 5 nodes are used as training nodes.

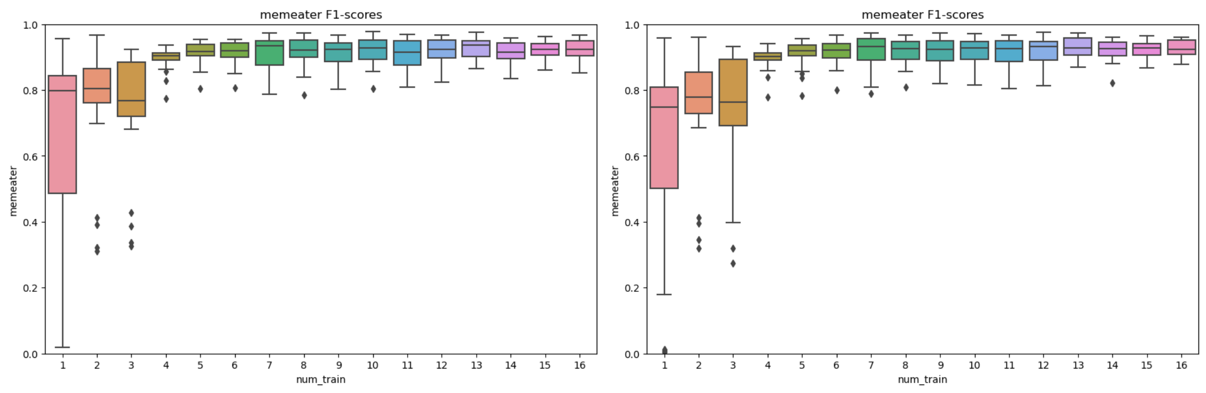


a) without shuffling



b) with shuffling enabled

Figure 5.7: Healthy scores as we increase the number of training nodes.



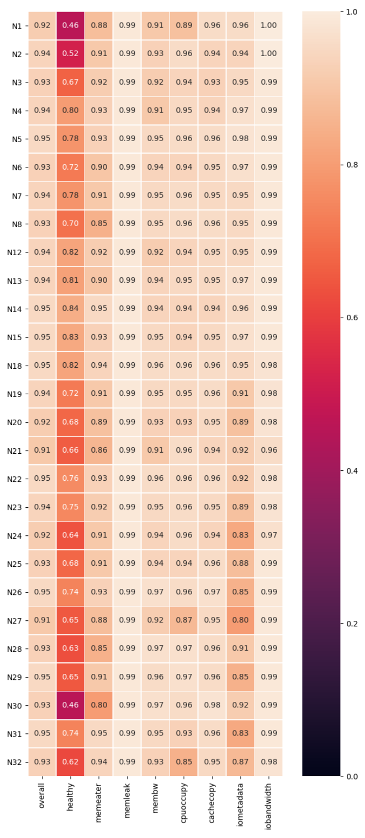
a) without shuffling b) with shuffling enabled

Figure 5.8: memeater scores as we increase the number of training nodes.

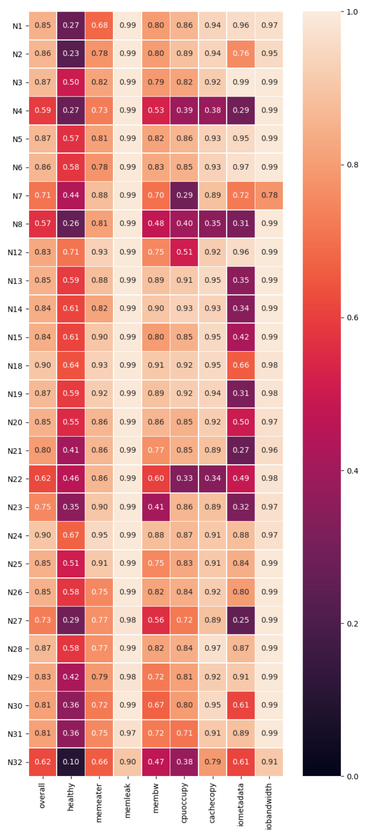
As can be seen from all the figures above, there isn't much difference in the results obtained in the two different configurations. However the boxplots in Figure 5.7 and Figure 5.8 suggest that generally the median scores of the healthy and memeater faults are a little higher when shuffling is disabled. For these reasons we disabled shuffling the training data in the following experiments.

Normalization

In this experiment balancing is performed on both training and testing set, training data shuffling is disabled and we compare the results obtained using non-normalized data vs normalized data. In Figure 5.9 we can clearly see that normalization makes us lose performances on most of the faults (results with different number of training nodes are similar). For this reason, the following experiments will be performed on non-normalized data.



a) without normalization

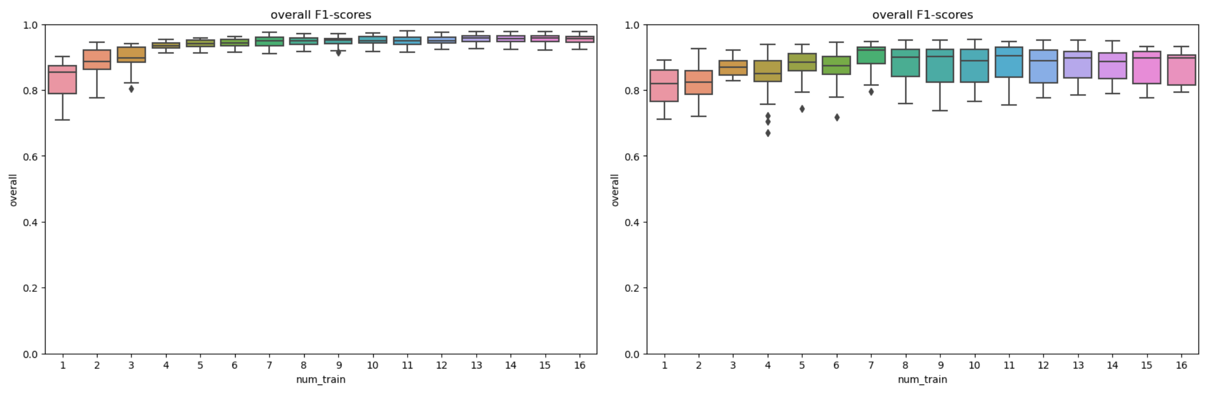


b) with normalization

Figure 5.9: F1-scores on the test nodes when 5 nodes are used as training nodes.

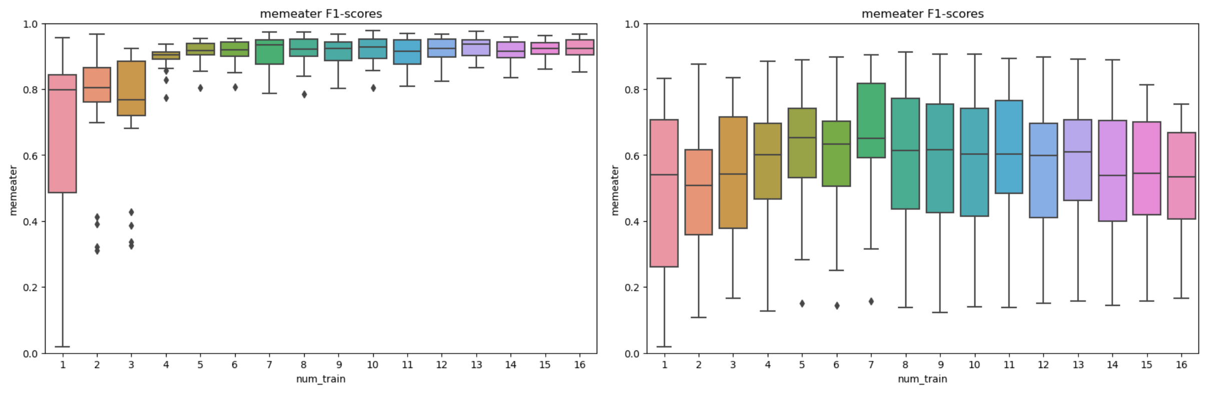
Test set balancing

In this experiment we wanted to see the impact of testing set class balancing on the final scores. The data used is non-normalized data and training set shuffling is disabled. Training set balancing is always enabled. In general balancing the testing node gives us better overall scores and better scores for *memeater*, while in this case the *healthy* scores are a little lower. However lower *healthy* scores do not seem to impact much on the overall results. You can see the details in Figures 5.10, 5.11 and 5.12.



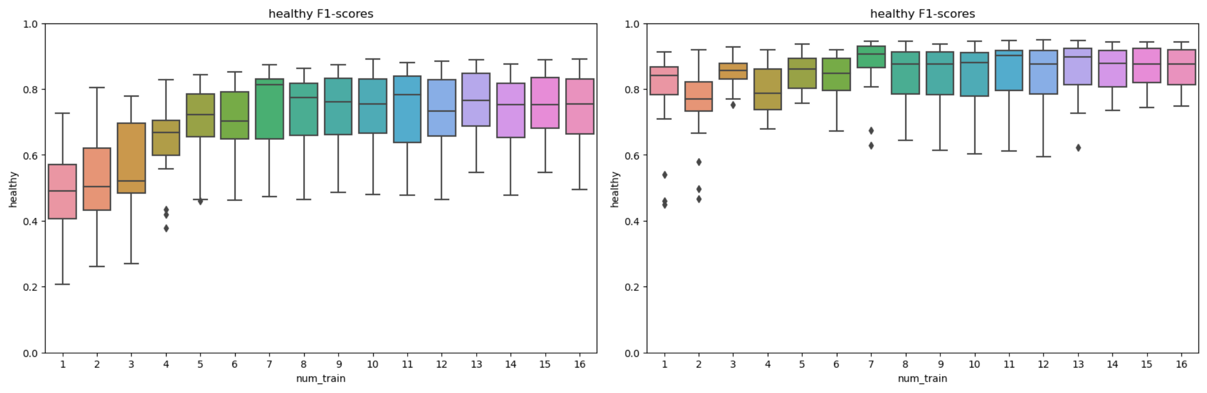
a) with balancing enabled b) without balancing

Figure 5.10: overall scores as we increase the number of training nodes.



a) with balancing enabled b) without balancing

Figure 5.11: memeater scores as we increase the number of training nodes.



a) with balancing enabled b) without balancing

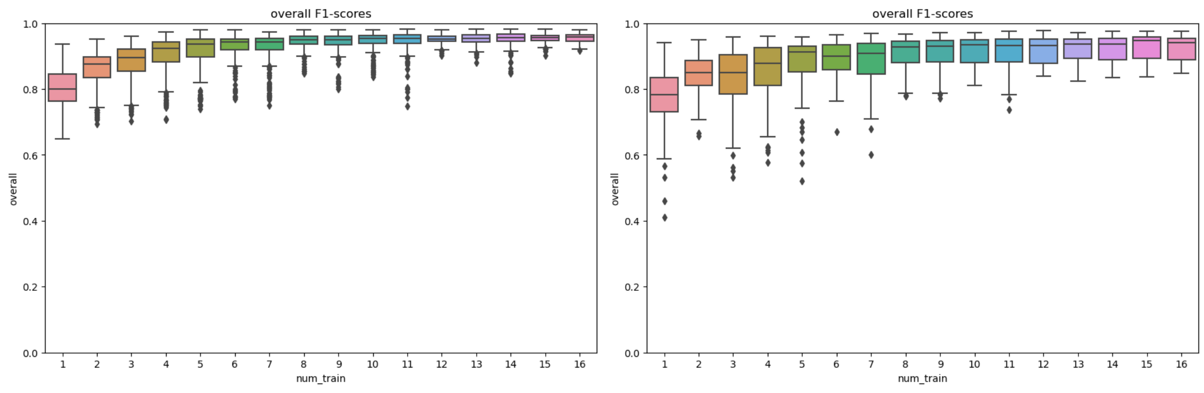
Figure 5.12: healthy scores as we increase the number of training nodes.

Since in the real world both testing data and training data may be not balanced and we don't want to hide the effects of imbalance from the results of our research (indeed balancing may artificially influence the evaluation metrics), we decided to investigate further on class balancing in the following experiments.

Overall class balancing

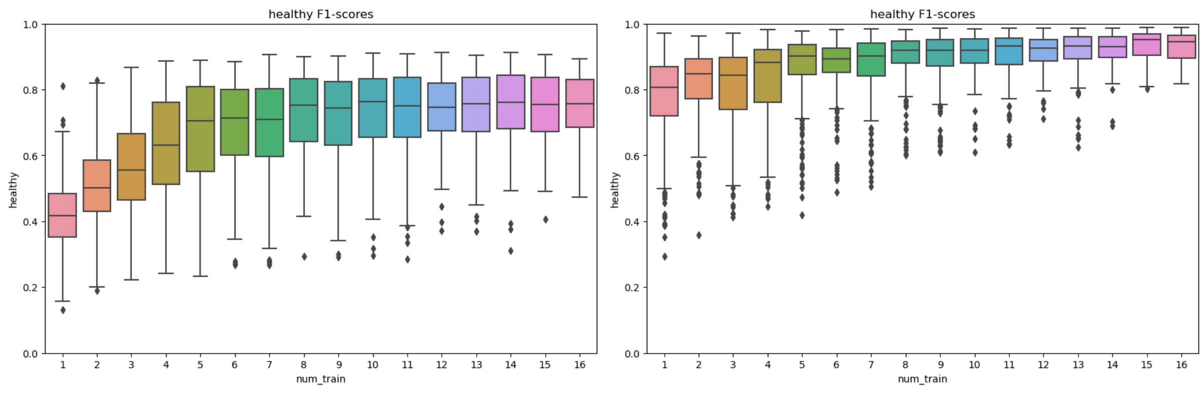
As described in Section 4.2.3 and Section 5.1, for these experiments we executed 10 runs each time and the seeds are not fixed but based on the time the instruction is executed, in order to have full randomization and have results that are statistically valid. Shuffling and normalization are both disabled. We wanted to compare what happens when balancing is enabled or disabled on both training and testing set.

In the following figures, which will show the F1-scores for each fault, part a) and will be about balancing enabled (on both training and testing set). Instead part b) will be about balancing disabled (on both training and testing).



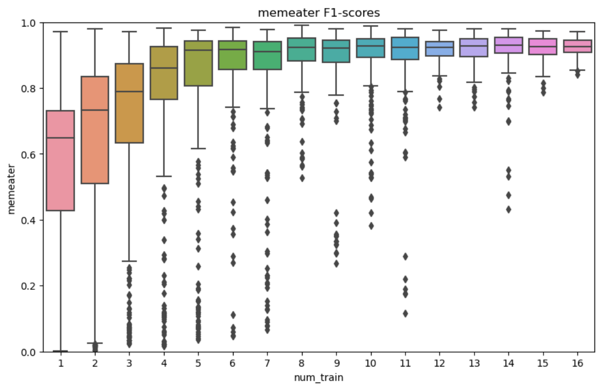
a) both balanced b) both unbalanced

Figure 5.13: overall scores as we increase the number of training nodes.

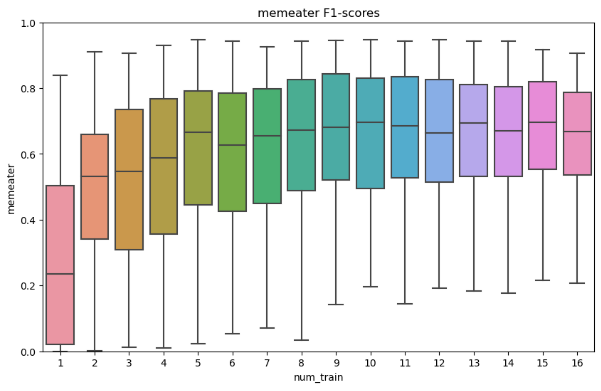


a) both balanced b) both unbalanced

Figure 5.14: healthy scores as we increase the number of training nodes.

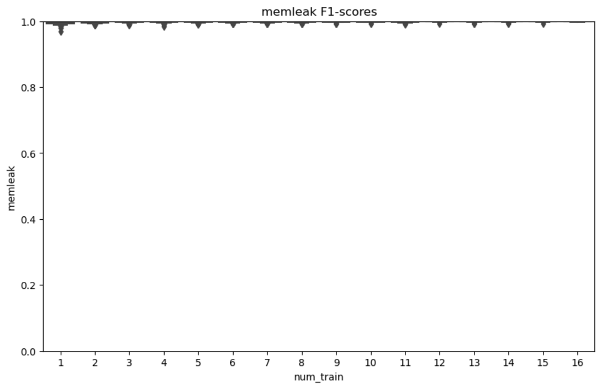


a) both balanced

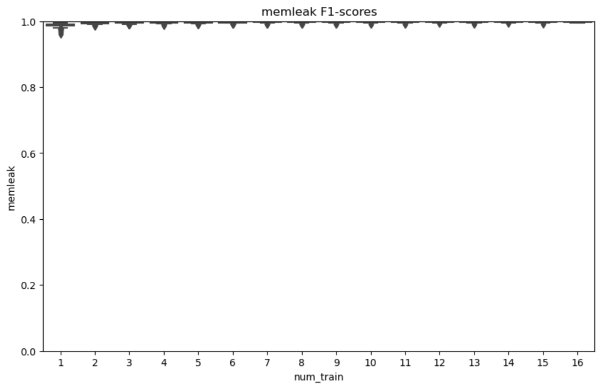


b) both unbalanced

Figure 5.15: memeater scores as we increase the number of training nodes.

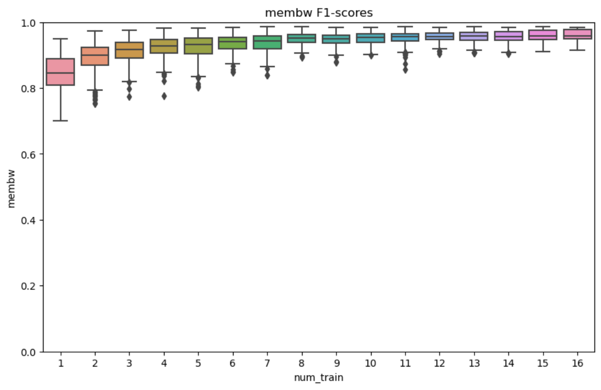


a) both balanced

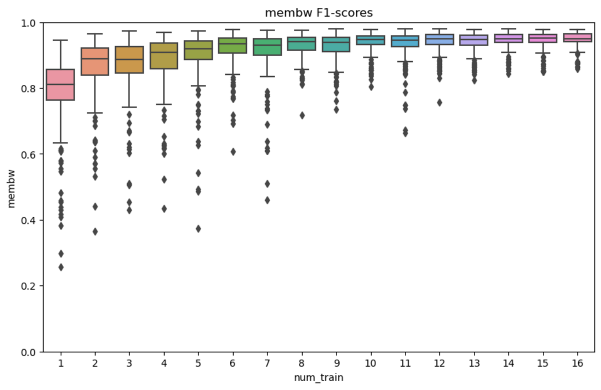


b) both unbalanced

Figure 5.16: memleak scores as we increase the number of training nodes.

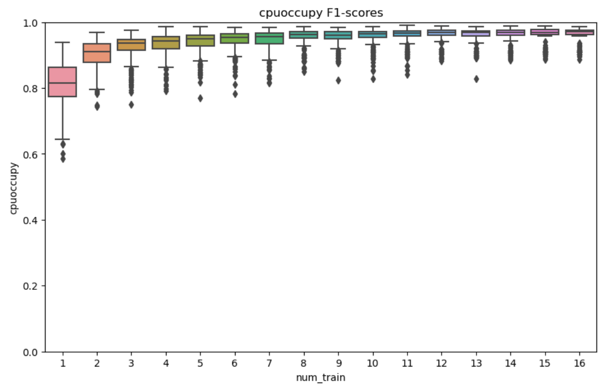


a) both balanced

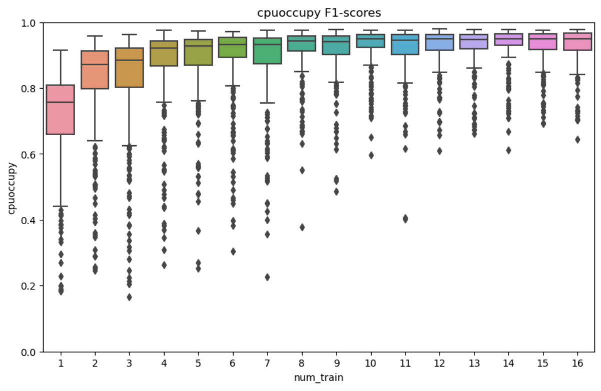


b) both unbalanced

Figure 5.17: membw scores as we increase the number of training nodes.

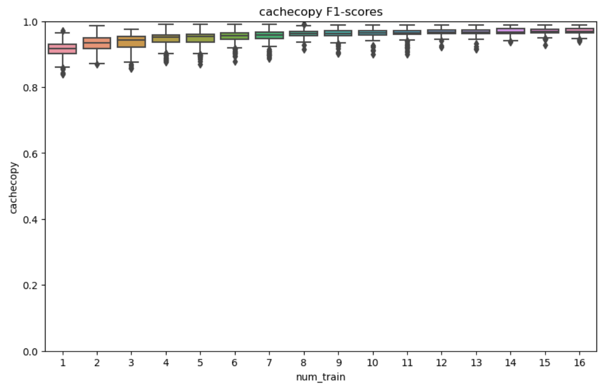


a) both balanced

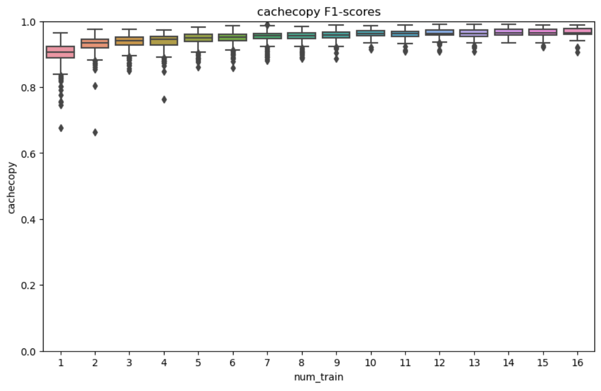


b) both unbalanced

Figure 5.18: cpuoccupy scores as we increase the number of training nodes.

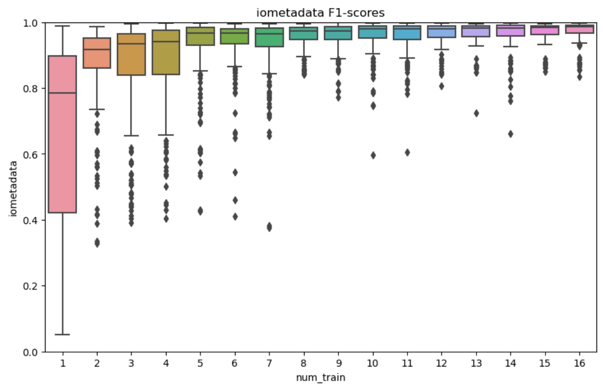


a) both balanced

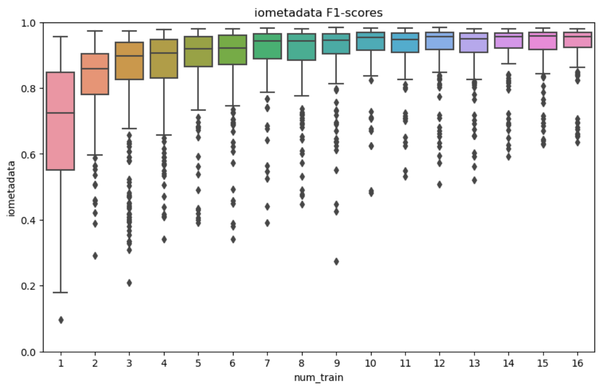


b) both unbalanced

Figure 5.19: cachecopy scores as we increase the number of training nodes.

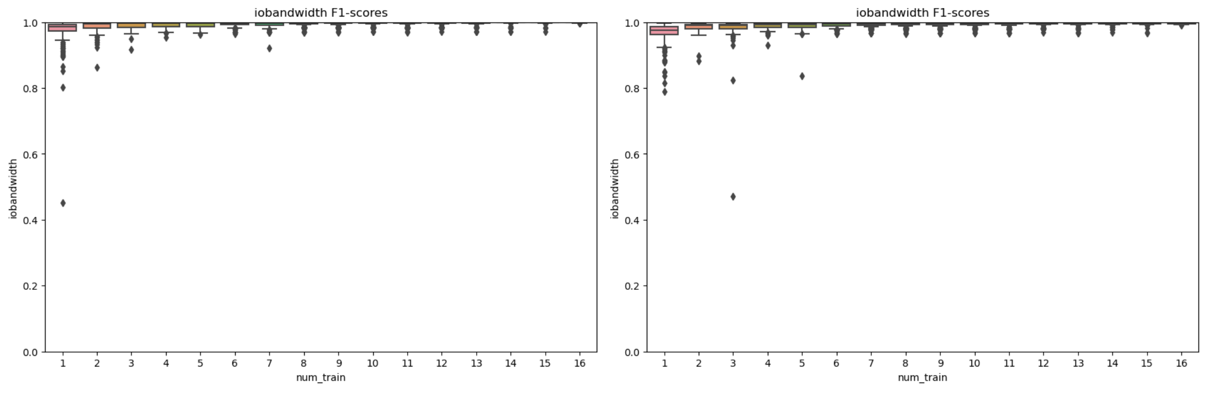


a) both balanced



b) both unbalanced

Figure 5.20: iometadata scores as we increase the number of training nodes.



a) both balanced b) both unbalanced

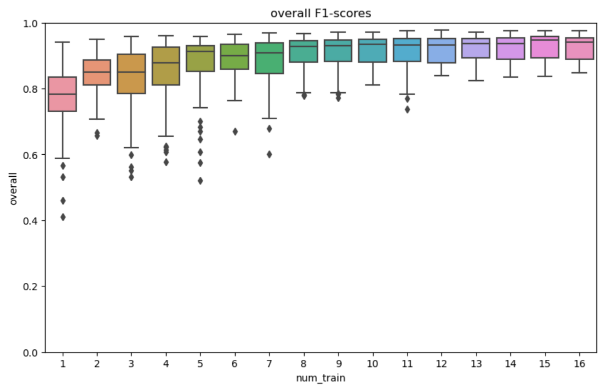
Figure 5.21: iobandwidth scores as we increase the number of training nodes.

As can be seen from Figure 5.13 the overall score stabilizes when using 5 or more training nodes. As for the other figures, the ones that stand out are Figure 5.14 and Figure 5.15 about *healthy* and *memeater*. In these cases class balancing can make a difference. More precisely, *healthy* improves when there's no class balancing, while *memeater* gets worse performances.

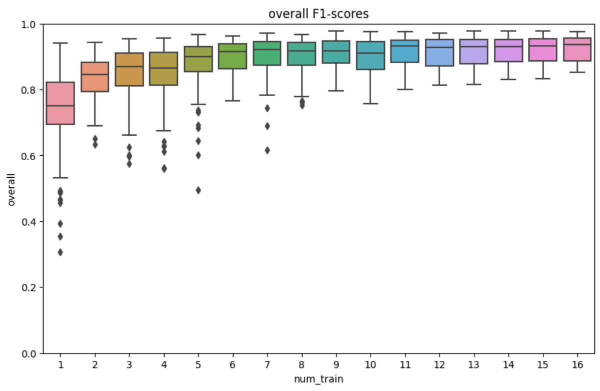
Since in the real world both testing data and training data may be not balanced and we don't want to hide the effects of imbalance from the results of our research, in the experiments where we choose the actual training nodes class balancing will be disabled both on the training and testing set.

Subsampling

As described in Section 4.2.3 and Section 5.1, for these experiments we executed 10 runs each time and the seeds are not fixed but based on the time the instruction is executed, in order to have full randomization and have results that are statistically valid. Shuffling, normalization and balancing (on both training and testing set) are all disabled. In the following figures part a) will be about subsampling disabled and part b) about subsampling enabled.

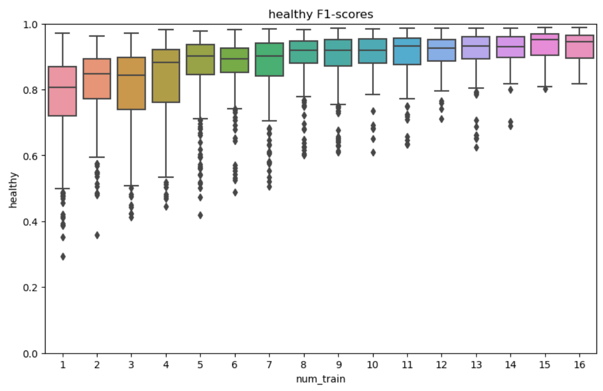


a)without subsampling

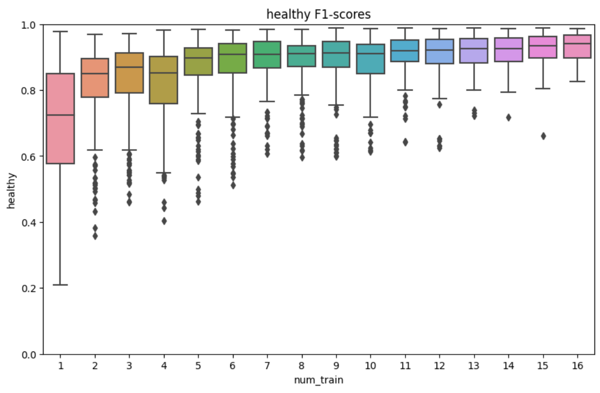


b)with subsampling

Figure 5.22: overall scores as we increase the number of training nodes

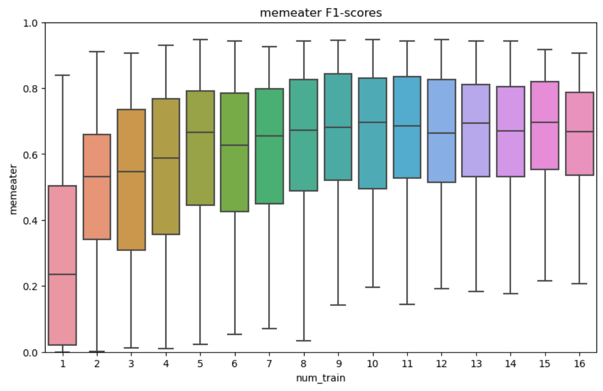


a)without subsampling

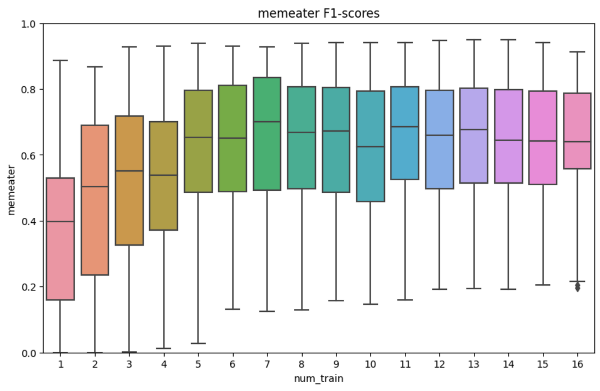


b)with subsampling

Figure 5.23: healthy scores as we increase the number of training nodes

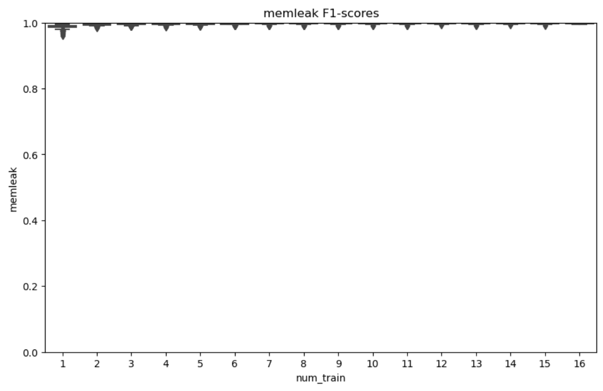


a)without subsampling

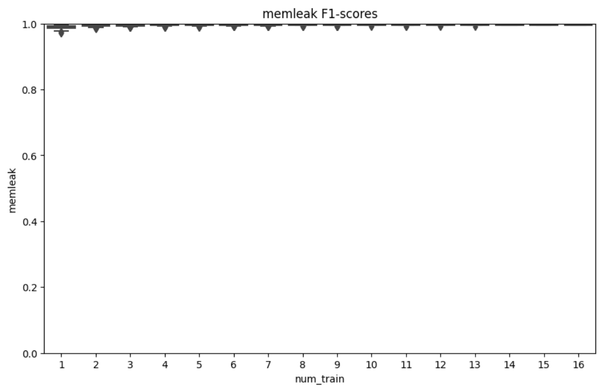


b)with subsampling

Figure 5.24: memeater scores as we increase the number of training nodes

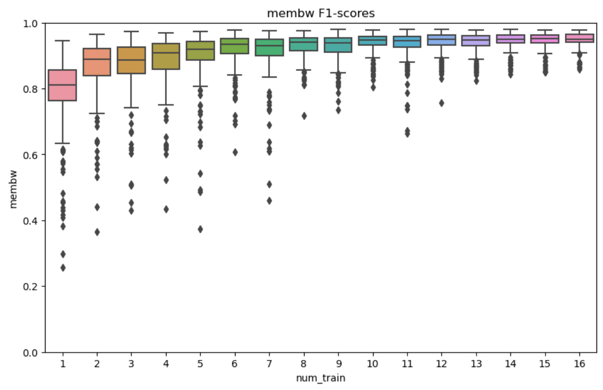


a)without subsampling

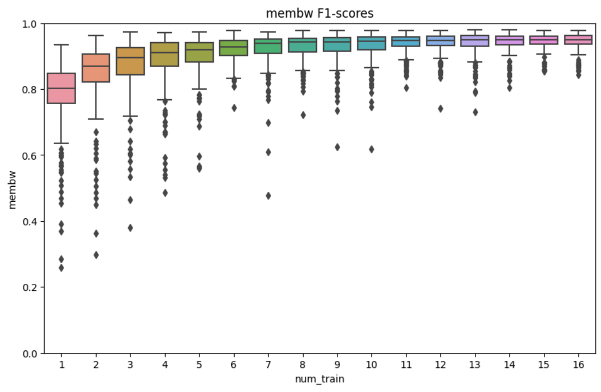


b)with subsampling

Figure 5.25: memleak scores as we increase the number of training nodes

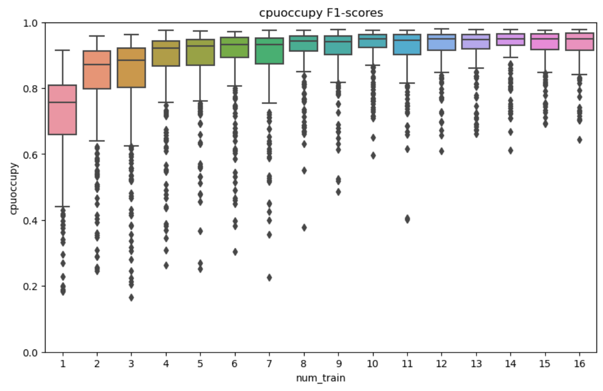


a)without subsampling

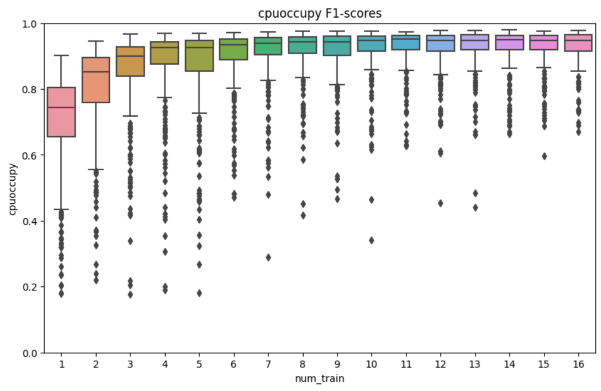


b)with subsampling

Figure 5.26: membw scores as we increase the number of training nodes

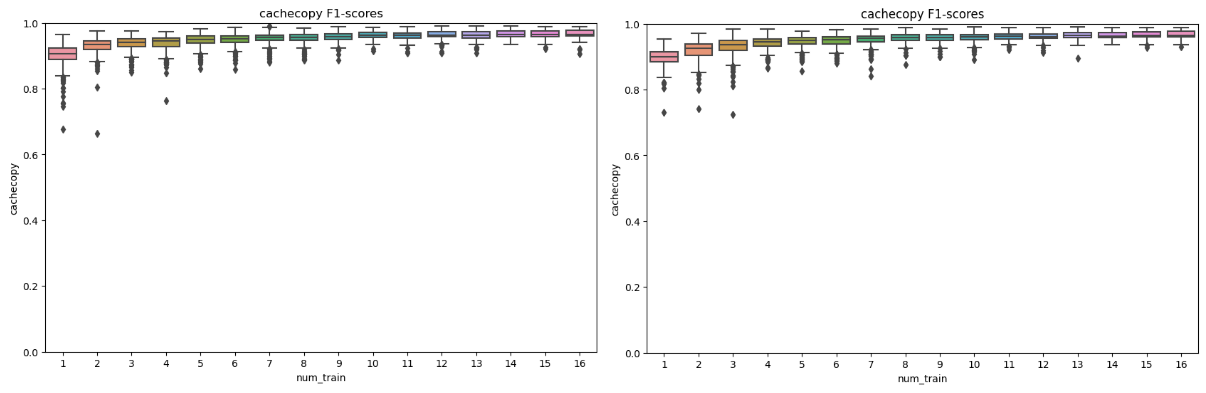


a)without subsampling



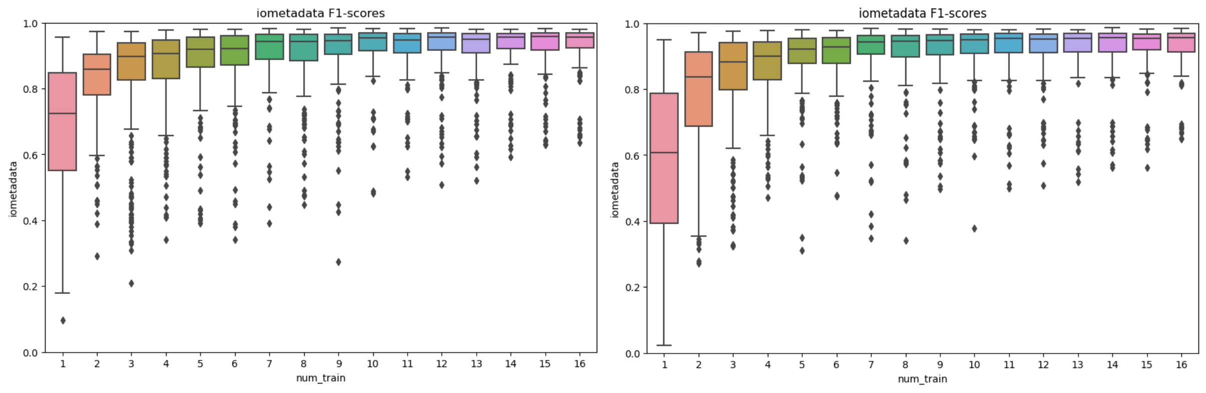
b)with subsampling

Figure 5.27: cpuoccupy scores as we increase the number of training nodes



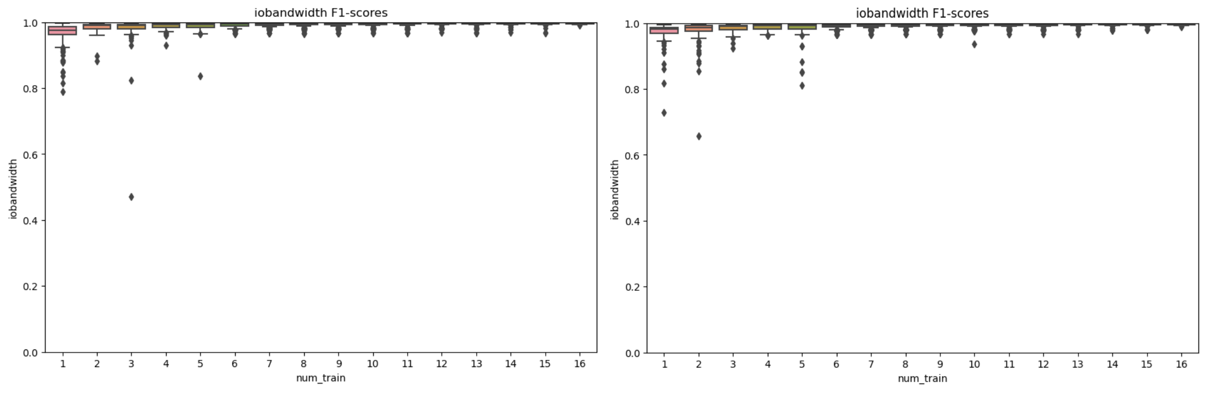
a)without subsampling b)with subsampling

Figure 5.28: cachecopy scores as we increase the number of training nodes



a)without subsampling b)with subsampling

Figure 5.29: iometadata scores as we increase the number of training nodes



a)without subsampling b)with subsampling

Figure 5.30: iobandwidth scores as we increase the number of training nodes

From Figure 5.22 we can see that subsampling doesn't impact negatively the classification performances (a thing that can be observed even for other faults). However with subsampling enabled the execution of the scripts doesn't go much faster (for example, when using 5 nodes as training set and testing on all the other nodes, there's roughly a 3 minutes difference). Subsampling is much more important in order to save up memory space, especially when using an high number of features. For these reasons, in the experiments where we chose the actual training nodes, we didn't use subsampling.

General conclusions

As can be seen from the discussion and figures above, we can conclude that most of the time the F1-scores stabilize when using 5 or more training nodes. Moreover in the experiments where we choose the actual training nodes we will not use normalization, shuffling, class balancing and subsampling for the reasons already described in the sections dedicated to each one of them.

To summarize here these reasons, normalization when enabled gives us worse F1-scores. Shuffling doesn't impact much the classification performances and some times the median of the F1-scores are higher when shuffling is disabled. Class balancing will be disabled because we don't want to hide the effects of imbalanced classes in our research. Finally subsampling doesn't affect much the classification performances and it gives very low speedup, while being much more important for saving up memory space when using a very high number of training nodes; for this reason it will be disabled, since in choosing the actual training nodes we will not use more than 5 training nodes.

5.5 The actual training nodes

In this section we will present the results regarding the clustering vs multiple run random experiments. We will also be able to compare these results against the node-specific models. Unless otherwise stated, we will refer to the F1-scores and introduce the other evaluation metrics only when needed to prove a point. Moreover we will train the multiple run random models and the clustering-based models on up to 5 training nodes, because in Section 5.4 we saw that the results stabilize after that threshold.

In the box plots that will be shown, a white circle with black edge represents the average. Moreover for each fault, there will be three boxes: the left blue one shows the results of the clustering experiments across all testing nodes, the middle orange one shows the results of the multiple run random experiments (across all the testing nodes and all the runs), the right green one represents the results of node-specific models (again, across all the testing nodes), whose experimental setup has already been discussed in Section 5.1. A box plot is a way to depict the variation of data through their quartiles.

5.5.1 One single node

Clustering

When selecting a candidate for a single cluster, the algorithm gave us N21 as result, as shown in Table 5.1. Figure 5.31 can also help the reader see how the nodes are distributed in the 3D space.

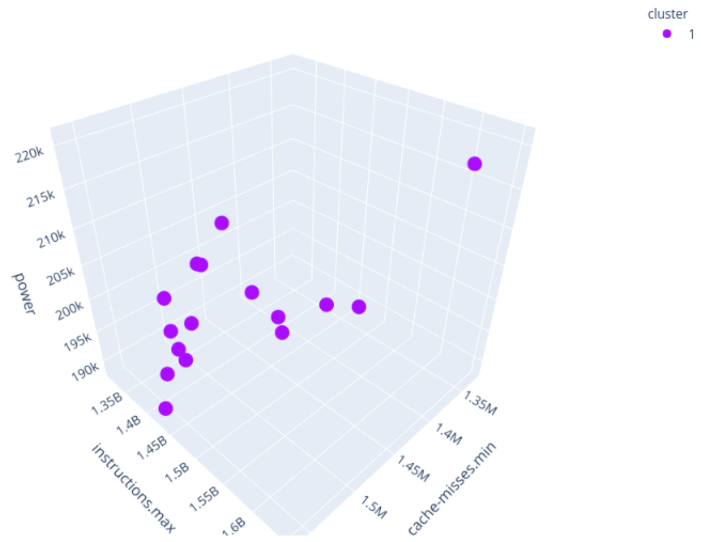


Figure 5.31: Scatter plot with clusters

Cluster	Node	Distance
1	N21	0.4008287862566875

Table 5.1: Nodes representative of each cluster

Results

For this particular experiment, we also wanted to compare the overall results between clustering and all the random runs (from “r1” to “r10”) in Figure 5.33; we also wanted to see in a scatter plot how the nodes have been selected, as can be seen in Figure 5.34. In Figure 5.32 there are the F1-scores for each fault and the reader can compare clustering vs random vs node-specific models.

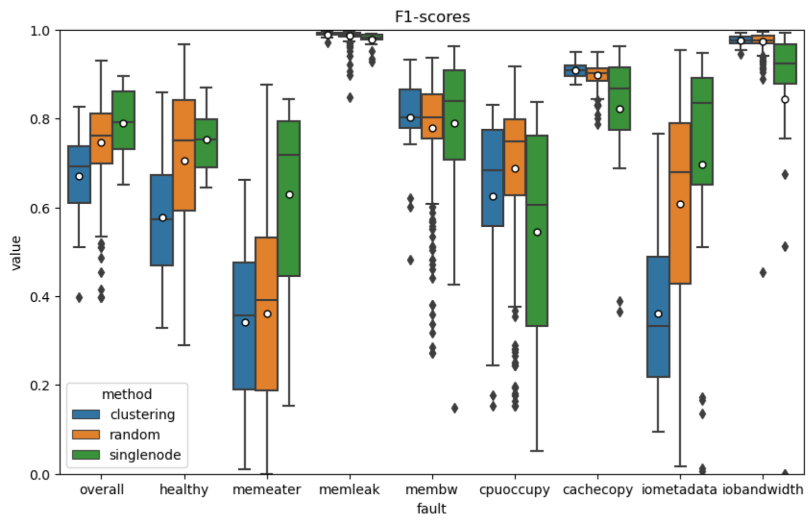


Figure 5.32: F1-scores clustering vs multi-run random vs node-specific models

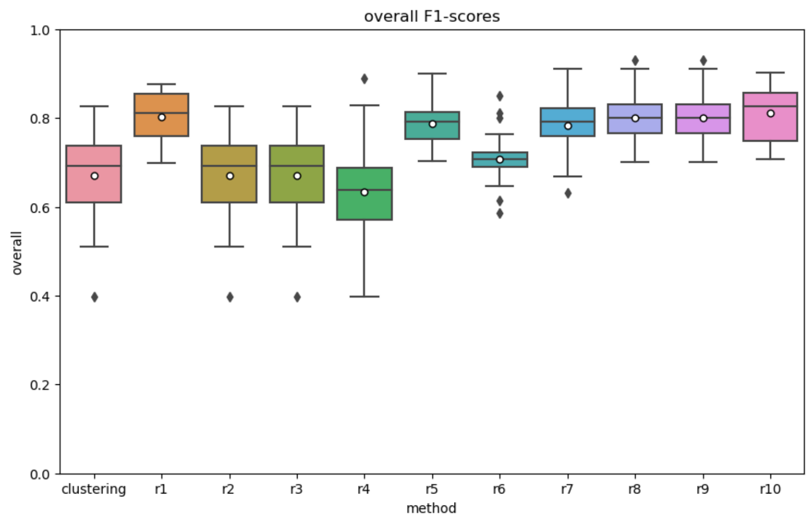


Figure 5.33: F1-scores clustering vs 10 run random

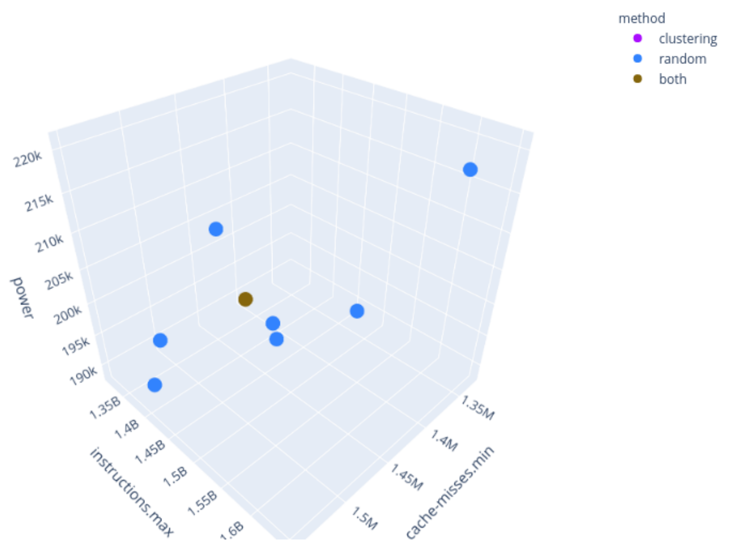


Figure 5.34: clustering vs random, training nodes scatter plot

From the figures above we can deduce that when using only 1 training node, random selection clearly outperforms clustering. Moreover when using only 1 training node, node-specific models outperform multi-node models.

5.5.2 Two nodes

Clustering

When choosing candidates for 2 clusters, the GMM algorithm gave us nodes N15 and N10, as can be seen in Table 5.2. The specific clusters can be observed in Figure 5.35.

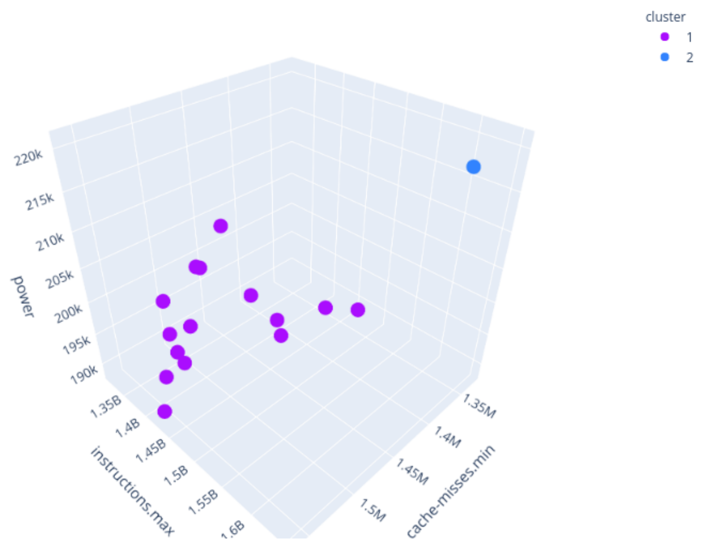


Figure 5.35: Scatter plot with clusters

Cluster	**Node**	**Distance**
1	N15	0.5551379436529936
2	N10	0.003576257131923944

Table 5.2: Nodes representative of each cluster

Results

For this experiment we will show the F1-scores in Figure 5.36, the sensitivity in Figure 5.37 and the specificity in Figure 5.38.

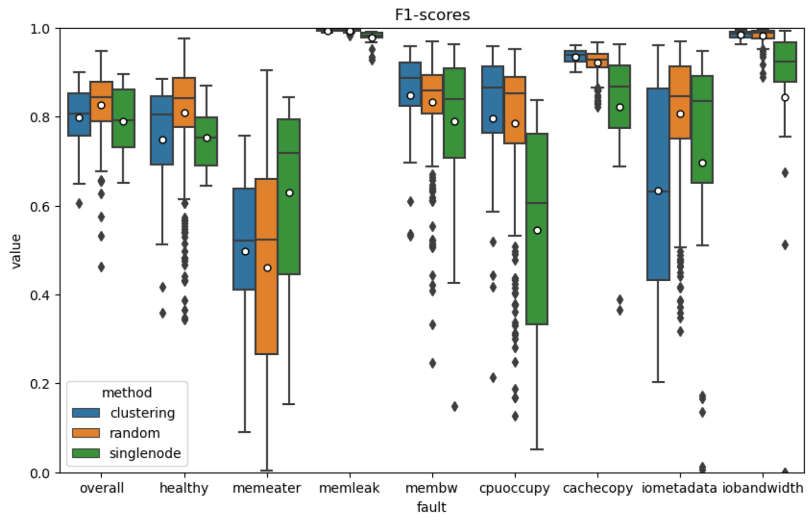


Figure 5.36: F1-scores clustering vs multi-run random vs node-specific models

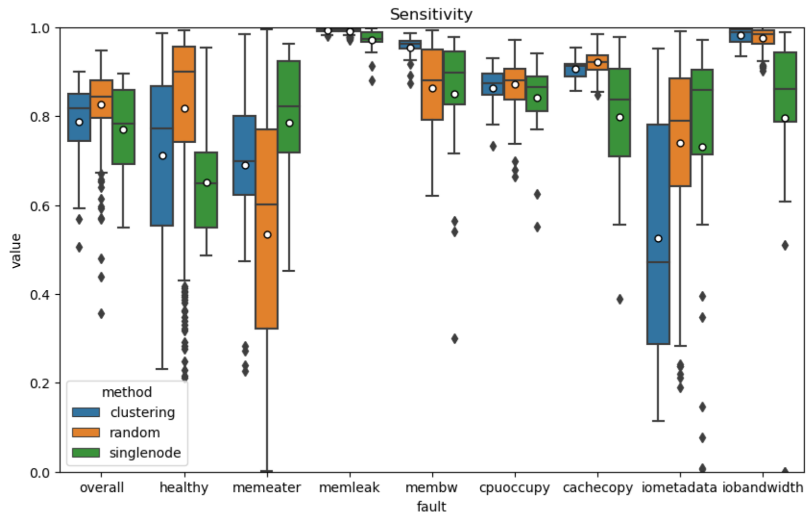


Figure 5.37: Sensitivity clustering vs multi-run random vs node-specific models

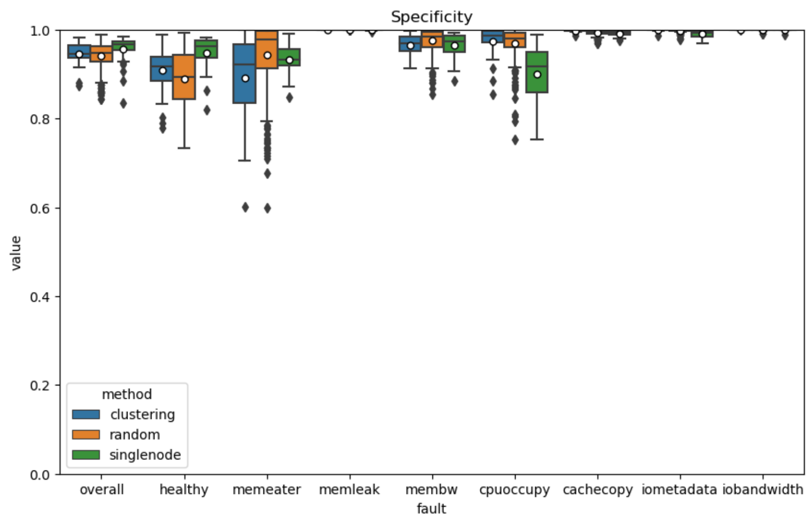


Figure 5.38: Specificity clustering vs multi-run random vs node-specific models

From Figure 5.36 we can see that with 2 training nodes, random selection still outperforms the clustering approach, although by a little margin. However we can also start to see that a single general multi-node model has an overall score that surpasses the one obtained with node-specific models. This is true even for many other faults.

Instead from Figure 5.37 and Figure 5.38 we can see that the random approach has lower sensitivity for *memeater* (and it also is highly variable since the box is wider) and higher and less variable specificity; this means that for *memeater* with the random approach we can be sure about the instances that are identified as not belonging to *memeater*, but less sure when the instances are classified as belonging to this fault. This is true also for the clustering approach, which however tends to remain more stable. As for *iometadata*, the clustering-based approach shows problems similar to the ones shown by *memeater* in the random approach.

5.5.3 Three nodes

Clustering

When searching for 3 candidate nodes via clustering, we obtained nodes N23, N10 and N19, as shown in Table 5.3. Instead Figure 5.39 shows how the clusters are distributed in the 3D space.

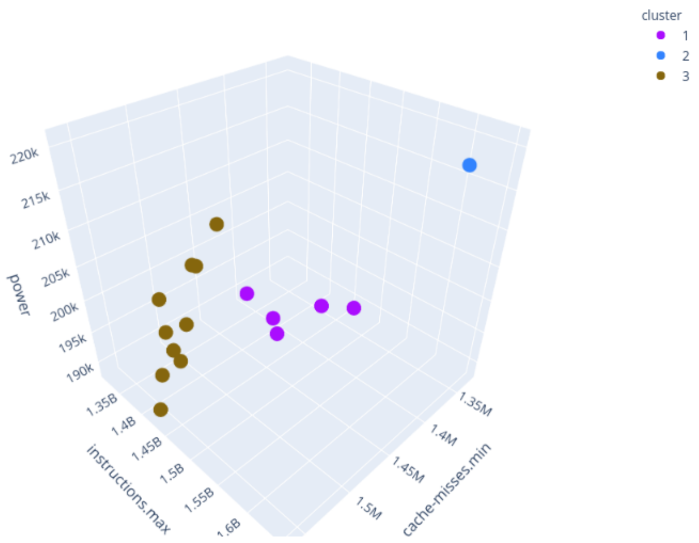


Figure 5.39: Scatter plot with clusters

Cluster	**Node**	**Distance**
1	N23	0.949795176961559
2	N10	0.003576257131923944
3	N19	0.5945542546369079

Table 5.3: Nodes representative of each cluster

Results

In Figure 5.40 we can see the F1-scores of the experiment.

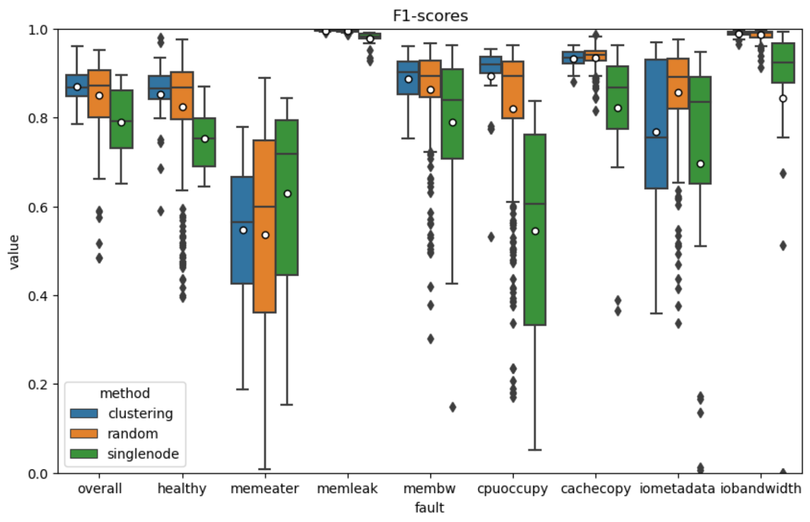


Figure 5.40: F1-scores clustering vs multi-run random vs node-specific models

From the Figure above we can see that generally the average of the clustering approach is higher than the one of the random approach; moreover both clustering and random approaches outperform the node-specific models. Finally the clustering approach seems to become more stable and reliable than the random approach, except for the *memeater* and *iometadata* faults.

5.5.4 Four nodes

Clustering

When selecting 4 candidates nodes via clustering we obtained the first half of node N20, the first half of node N10, the first half of node N9 and the first half of node N19, as shown in Table 5.4. Figure 5.41 shows instead how the clusters are distributed in the 3D space.

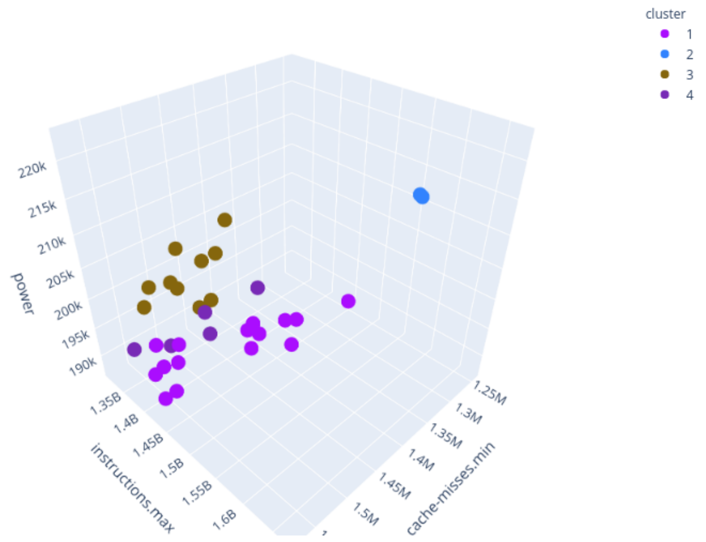


Figure 5.41: Scatter plot with clusters

Cluster	Node	Distance
1	N20_1	0.5037602723821104
2	N10_1	1.0249200129411562
3	N9_1	0.5873675358414548
4	N19_1	0.9497396060421003

Table 5.4: Nodes representative of each cluster

Results

Here we will show the F1-scores, sensitivity, specificity, false positive rate and false negative rate of the experiments.

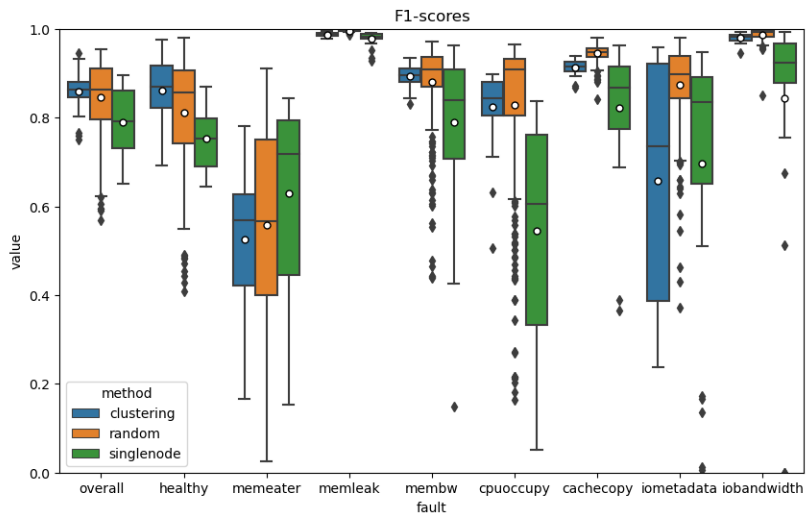


Figure 5.42: F1-scores clustering vs multi-run random vs node-specific models

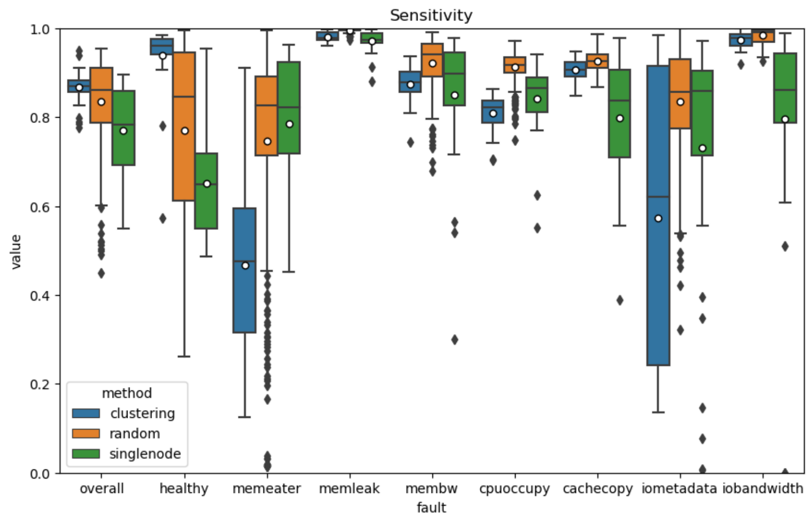


Figure 5.43: Sensitivity clustering vs multi-run random vs node-specific models

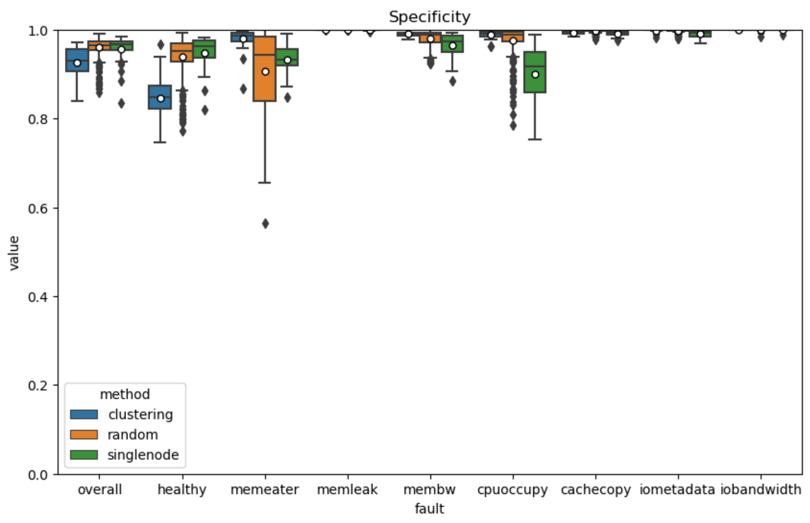


Figure 5.44: Specificity clustering vs multi-run random vs node-specific models

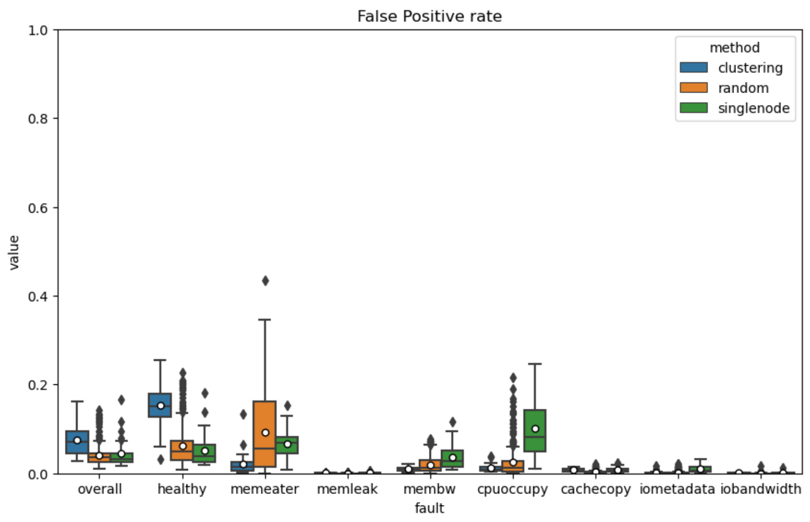


Figure 5.45: False Positive rate clustering vs multi-run random vs node-specific models

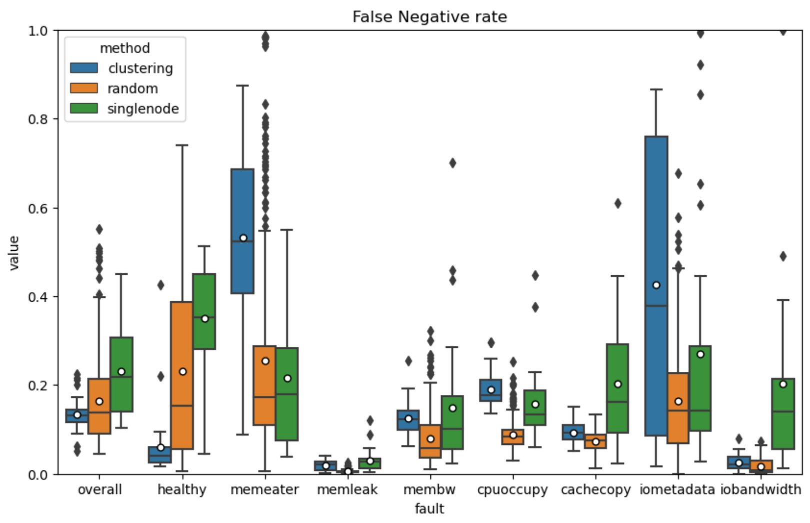


Figure 5.46: False Negative rate clustering vs multi-run random vs node-specific models

The conclusions we can draw from Figure 5.42 are similar to the one we explained in Section 5.5.3: clustering seems to become more stable and reliable than the random approach, but we still have problems with *memeater* and *iometadata*.

As for sensitivity and specificity, for the clustering results we can see a similar behaviour explained in Section 5.5.2: we can see that we have lower sensitivity for *memeater* (and it also is highly variable since the box is wider) and higher and less variable specificity; this means that for *memeater* with the clustering approach we can be sure about the instances that are identified as not belonging to *memeater*, but less sure when the instances are classified as belonging to this fault. We have similar problems with *iometadata*, whose sensitivity is highly variable, while the specificity is more stable and higher.

Finally false positive rates (Figure 5.45) are generally lower and more stable than false negative rates (Figure 5.46), which can be interpreted as low probability of false alarm but an high probability of miss rate.

5.5.5 Five nodes

Clustering

When searching for the best 5 training nodes via clustering we obtained the first half of N20, the first half of N10, the first half of N9, the first half of N19 and the second half of N10, as can be seen in Table 5.5. Figure 5.47 shows instead the distribution of the clusters in the 3D space.

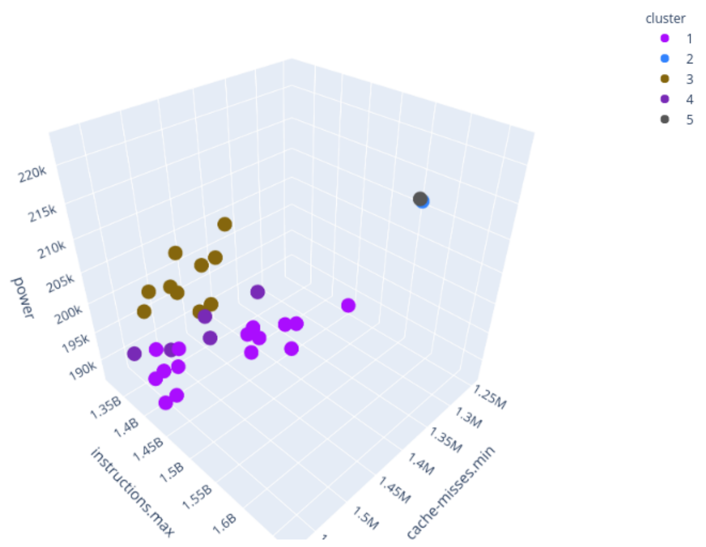


Figure 5.47: Scatter plot with clusters

Cluster	**Node**	**Distance**
1	N20_1	0.5037602723821104
2	N10_1	0.003576257131923944
3	N9_1	0.5873675358414548
4	N19_1	0.9497396060421003
5	N10_2	0.0038146707432278617

Table 5.5: Nodes representative of each cluster

Results

For this particular experiment, we also wanted to compare the overall results between clustering and all the random runs (from “r1” to “r10”) in Figure 5.49; we also wanted to see in a scatter plot how the nodes have been selected, as can be seen in Figure 5.50. The other figures instead show the F1-scores, sensitivity, specificity, false positive rate and false negative rate.

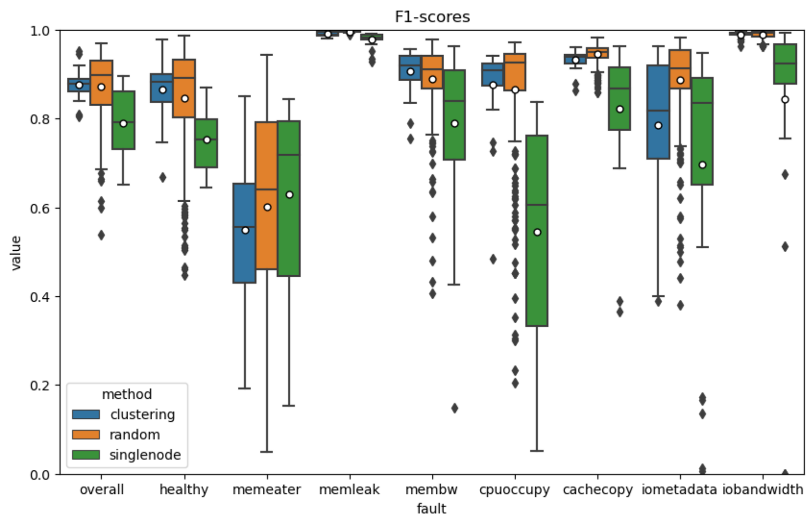


Figure 5.48: F1-scores clustering vs multi-run random vs node-specific models

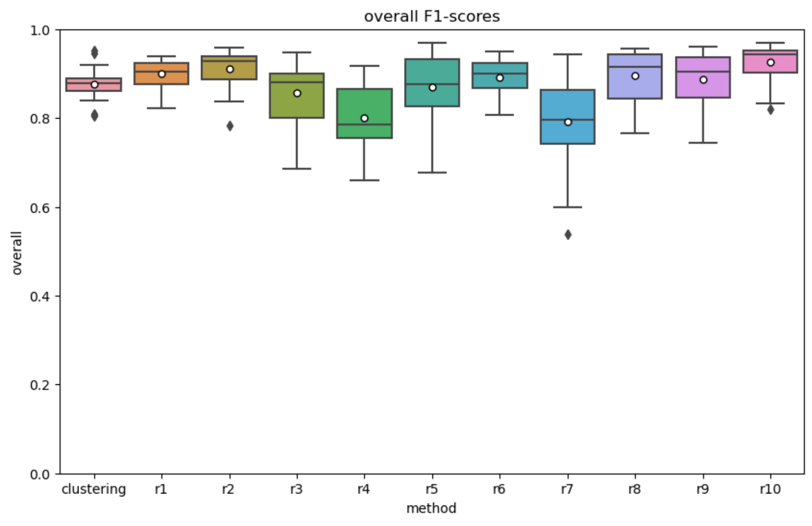


Figure 5.49: F1-scores clustering vs 10 run random

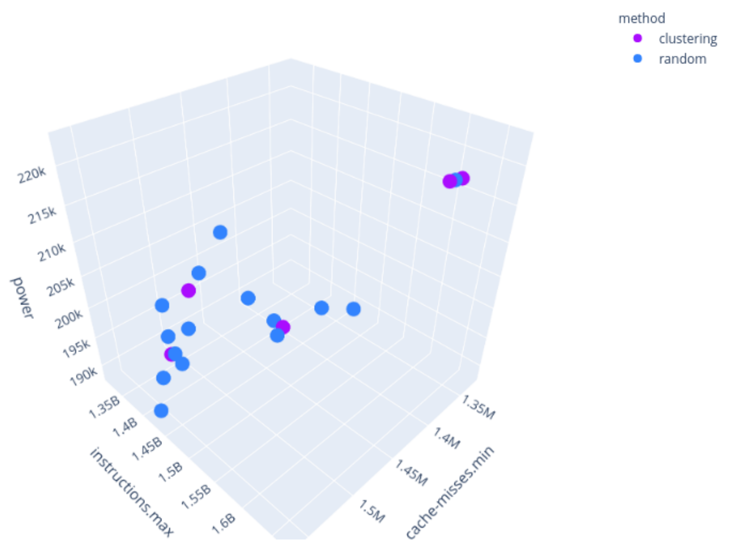


Figure 5.50: clustering vs random, training nodes scatter plot

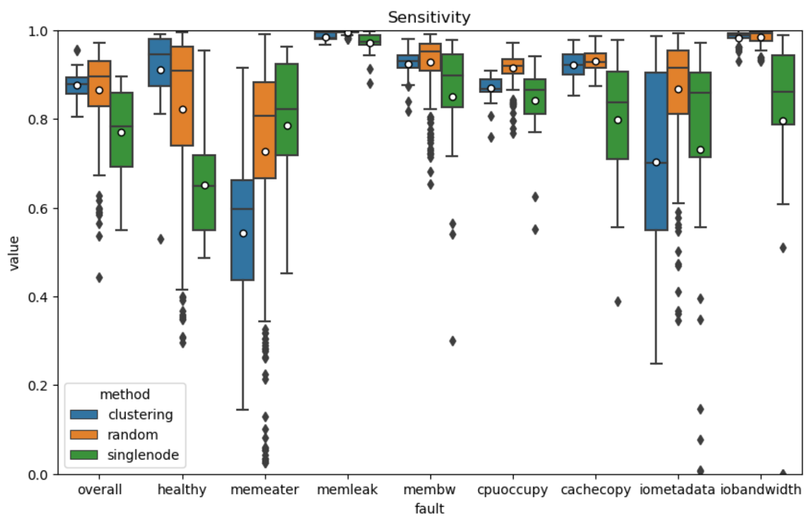


Figure 5.51: Sensitivity clustering vs multi-run random vs node-specific models

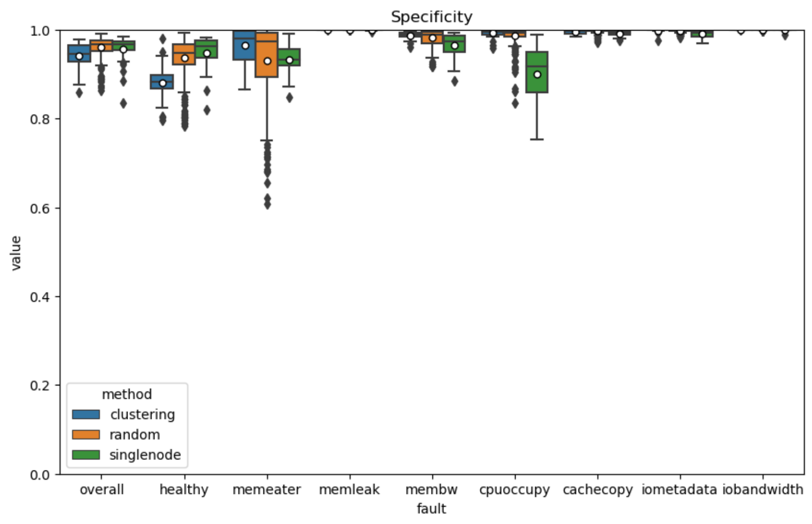


Figure 5.52: Specificity clustering vs multi-run random vs node-specific models

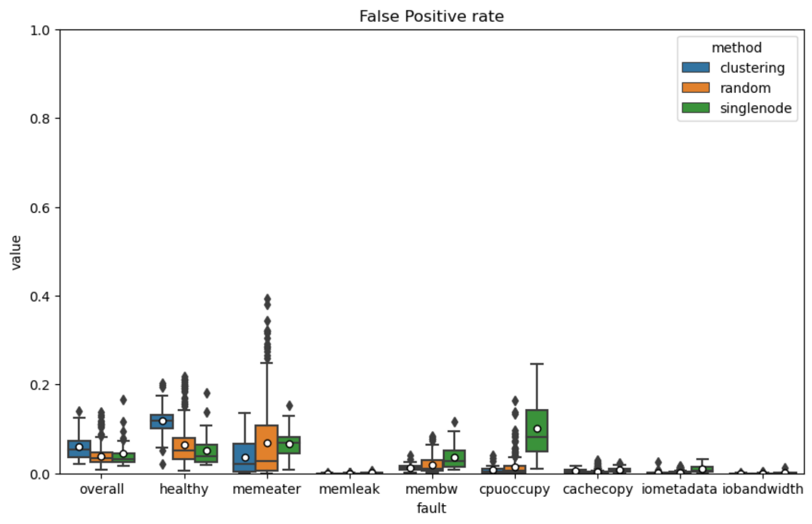


Figure 5.53: False Positive rate clustering vs multi-run random vs node-specific models

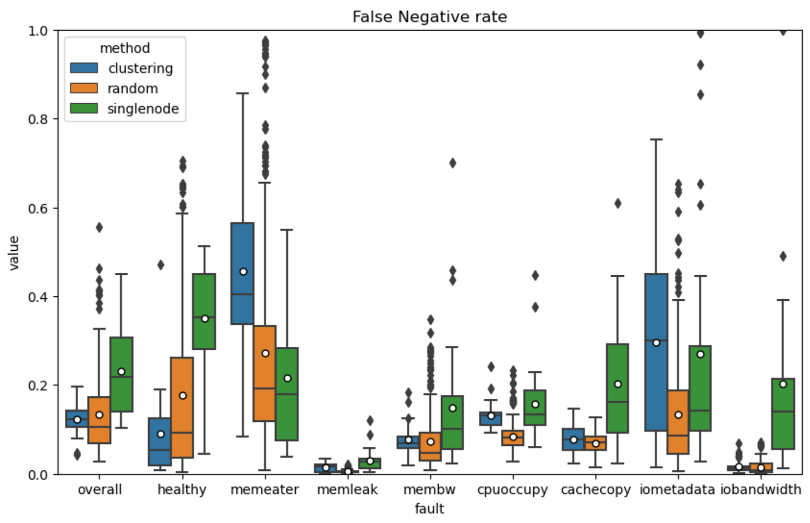


Figure 5.54: False Negative rate clustering vs multi-run random vs node-specific models

From Figure 5.49 we can clearly see that even if some lucky random choices give higher averages, the clustering approach has good results and is more stable and reliable. This is confirmed by the F1-scores shown in Figure 5.48.

Sensitivity and specificity confirm what we have said in Section 5.5.4 and 5.5.2. Figure 5.53 and 5.54 highlight again some problems in *memeater* and *iometadata*, where the false positive rate of clustering (that is the probability of false alarm) is lower than the false negative rate (that is the miss rate). This is reflected in the lower scores and high variety of them when we look at these faults in Figure 5.48.

5.5.6 Discussion of results

Single-node models outperform multi-node models only when using 1 training node. When using 2 training nodes, the multi-node approach (in this case the random one) starts to outperform node-specific models. This trend will become more clear even for the clustering-based approach when using an higher number of training nodes.

With a low number of training nodes (1 or 2), the random approach gives better results than the clustering-based approach. However when the number of training nodes is equal or greater than 3, the clustering-based approach gives averages comparable to the random approach but it is more stable and reliable.

We identified some problems with *memeater* and *iometadata* both for clustering and random. More precisely the sensitivity tends to be low and the specificity tends to be high. Moreover the false positive rates are generally lower and more stable than the false negative rates. These problems have been addressed at the end of Section 5.5.4 and Section 5.5.5.

Chapter 6

Conclusions and future work

6.1 Summary

During the work executed for this thesis, we extracted the data of the CooLMUC-3 HPC system using libDCDB and crafted the best *signatures* by exploiting the knowledge gained from previous research. We then conducted various experiments with a specific purpose: finding the best way to design a single general multi-node model and compare its performances against node-specific models.

First of all we started with some initial k-fold cross validation experiments, in order to see how the techniques used in our previous work [Cov20] could influence the classification results obtained with the new dataset used for this thesis. Then we executed a metaparameter exploration, in order to find the best metaparameters for the rest of the experiments. We conducted this exploration by using an incremental random approach and multiple runs, in order to have statistically significant results. This kind of experiments was also useful for determining that the results stabilize over a certain number of training nodes.

Having found out on how many nodes to train on, we had to answer another question: which specific nodes do we need to use for training? To answer the previous question we compared the incremental random approach with multiple runs against the incremental clustering-based approach. The idea of clustering was to train the model on nodes which are representative of a specific cluster, in order to have a model that generalizes better and works well on novel data. Clustering was executed using hand-picked metrics exploiting the use of expert knowledge (more specifically we used *cache-misses.min*, *instructions.max* and *power*) and filtering out the data which did not belong to *healthy* and *HPL*; the idea was to make the process of choosing nodes to train on independent of *fault injection*, which can be impractical, costly and not scalable when the number of nodes increases.

6.2 Future work

Even though our research shows promising results, further developments can be made, especially for solving the problems with *memeater* and *iometadata*.

First of all, some exploratory analysis could be done by removing *memeater*, *iometadata* and *iobandwidth* faults, in order to see if the overall results improve. However obviously this would not solve the problem, but it would be interesting to see how with our configuration the overall results change without these faults and if they improve.

Then, to actually try to solve the aforementioned problems, we could try to identify better metrics for clustering. Indeed the poor performances of the mentioned faults could be related to a low impact of these faults on the chosen metrics for clustering.

Finally, instead of using expert knowledge to pick specific metrics for clustering, one could try to apply a dimensionality reduction technique, such as PCA, to the whole vector of metrics and use the first k (for example 3) resulting metrics.

6.3 Conclusions

The initial exploratory experiments with k-fold cross validation on 4 random nodes gave interesting results, especially when shuffling was enabled, meaning that the strategy we used for designing the *signatures* during our previous internship work [Cov20] may be good, but class balancing and shuffling should be investigated further. For this reason we conducted a metaparameter exploration.

During the metaparameter exploration we found out that the classification results stabilize when using 5 or more nodes as training set. By interpreting the results of these experiments, we also decided to avoid the usage of shuffling, normalization, subsampling and class balancing in order to have better results and at the same time to be transparent about some problems encountered with some faults.

After executing the clustering vs random experiments, whose results have been shown in Section 5.5, we found out that the random approach gives better results when the number of training nodes is low, while the clustering-based approach gives overall average results comparable to the random approach, but it is more stable and reliable, when using an high number of training nodes. Moreover we also proved that training a multi-node model on a reasonable number of nodes (such as 3 or above) gives better results than node-specific models.

All of this means that we have contributed to the research field in the way described in Section 1.3. More precisely, we provided a methodology for finding the subset of training node necessary for multi-node models and for choosing which specific nodes to use as part of the training set. We also showed that multi-node models can outperform node-specific models. This is an important achievement, since multi-node models are more scalable.

We also identified for some configurations some problems regarding the performances of *memeater* and *iometadata* both for clustering and random. More precisely the sensitivity tends to be low and the specificity tends to be high. Moreover the false positive rates are generally lower and more stable than the false negative rates, which can mean that we have an low probability of false alarm but an high probability of miss rate for these faults. Future work to address these issues has been presented in Section 6.2.

Bibliography

[AAB+14] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker. The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165, 2014.

[AB94] Douglas G Altman and J Martin Bland. Diagnostic tests. 1: Sensitivity and specificity. *BMJ: British Medical Journal*, 308(6943):1552, 1994.

[ABC+05] Narasimha R Adiga, Matthias A Blumrich, Dong Chen, Paul Coteus, Alan Gara, Mark E Giampapa, Philip Heidelberger, Sarabjeet Singh, Burkhard D Steinmacher-Burow, Todd Takken, et al. Blue gene/l torus interconnection network. *IBM Journal of Research and Development*, 49(2.3):265–276, 2005.

[Alp20] Ethem Alpaydin. *Introduction to machine learning.* MIT press, 2020.

[AZA+19] Emre Ates, Yijia Zhang, Burak Aksar, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. Hpas: An hpc performance anomaly suite for reproducing performance variations. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[BBC+08] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[BBL+19] Andrea Borghesi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. A semisupervised autoencoder-based approach for anomaly detection in high performance computing systems. *Engineering Applications of Artificial Intelligence*, 85:634 – 644, 2019.

[BFSO84] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees.* CRC press, 1984.

[BGF$^+$10] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B. Woodard, and Hans Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 111–124, New York, NY, USA, 2010. Association for Computing Machinery.

[BL16] Elisabeth Baseman and Lissa. Interpretable anomaly detection for monitoring of high performance computing systems. 2016.

[Cov20] Vito Vincenzo Covella. Relazione finale di tirocinio. `https://dochub.com/covinc93heron/pqb0g5YRqyvJvZzRJ2nx67/report-pdf?dt=hcDiEKGQFQw9Lz-rxYQD`, 2020.

[DBMS79] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart. *LINPACK Users Guide*, 1979.

[DLR77] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[GC15] Ana Gainaru and Franck Cappello. *Errors and Faults*, pages 89–144. Springer International Publishing, Cham, 2015.

[HE17] Saurabh Hukerikar and Christian Engelmann. Resilience design patterns: A structured approach to resilience at extreme scale. *arXiv preprint arXiv:1708.07422*, 2017.

[Ho95] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[Ho98] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8):832–844, 1998.

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997.

[HW10] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers.* CRC Press, 2010.

[IK04] Tsuyoshi IDÉ and Hisashi KASHIMA. Eigenspace-based anomaly detection in computer systems. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, page 440–449, New York, NY, USA, 2004. Association for Computing Machinery.

[IPI$^+$15] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, et al. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[JWHT13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[Kni07] Will Knight. Ibm creates world's most powerful computer. *New scientist. com news service, June*, 2007.

[Mit99] Tom M. Mitchell. Machine learning and data mining. *Commun. ACM*, 42(11):30–36, November 1999.

[MMC13] RS Mitchell, JG Michalski, and TM Carbonell. *An artificial intelligence approach.* Springer, 2013.

[Net] Alessio Netti. *Development of Data-Driven Dispatching Heuristics for Heterogeneous HPC Systems.* PhD thesis.

[NKB$^+$20] Alessio Netti, Zeynep Kiziltan, Ozalp Babaoglu, Alina Sîrbu, Andrea Bartolini, and Andrea Borghesi. A machine learning approach to online fault classification in hpc systems. *Future Generation Computer Systems*, 110:1009 – 1022, 2020.

[NMA$^+$19] Alessio Netti, Micha Mueller, Axel Auweter, Carla Guillen, Michael Ott, Daniele Tafani, and Martin Schulz. From facility to application sensor data: Modular, continuous and holistic monitoring with DCDB. *CoRR*, abs/1906.07509, 2019.

[NTOS20] Alessio Netti, Daniele Tafani, Michael Ott, and Martin Schulz. Correlation-wise smoothing: Lightweight knowledge extraction for hpc monitoring data, 2020.

[ONTS20] Gence Ozer, Alessio Netti, Daniele Tafani, and Martin Schulz. Characterizing hpc performance variation with monitoring and unsupervised learning. In Heike Jagode, Hartwig Anzt, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, pages 280–292, Cham, 2020. Springer International Publishing.

[PCKI11] A. Pecchia, D. Cotroneo, Z. Kalbarczyk, and R. K. Iyer. Improving log-based field failure data analysis of multi-node computing systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 97–108, 2011.

[pdt20] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[PK20] Eva Patel and Dharmender Singh Kushwaha. Clustering cloud workloads: K-means vs gaussian mixture model. *Procedia Computer Science*, 171:158 – 167, 2020. Third International Conference on Computing and Network Communications (CoCoNet'19).

[Pow08] David Powers. Evaluation: From precision, recall and f-factor to roc, informedness, markedness & correlation. *Mach. Learn. Technol.*, 2, 01 2008.

[Pow20] David MW Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*, 2020.

[PRT$^+$18] Davy Preuveneers, Vera Rimmer, Ilias Tsingenopoulos, Jan Spooren, Wouter Joosen, and Elisabeth Ilie-Zudor. Chained anomaly detection models for federated learning: An intrusion detection case study. *Applied Sciences*, 8(12), 2018.

[PVG$^+$11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[Qui14] J Ross Quinlan. *C4. 5: programs for machine learning.* Elsevier, 2014.

[RN$^+$13] Stuart Russel, Peter Norvig, et al. *Artificial intelligence: a modern approach.* Pearson Education Limited London, 2013.

[SW98] Claude Shannon and Warren Weaver. *The mathematical theory of communication.* University of Illinois Press, Urbana, 1998.

[TAZ$^+$17] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. Diagnosing performance variations in hpc applications using machine learning. In Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes, editors, *High Performance Computing*, pages 355–373, Cham, 2017. Springer International Publishing.

[VdSD95] Aad J Van der Steen and Jack J Dongarra. *Overview of recent supercomputers*. Citeseer, 1995.

[Wa10] Margaret Wright and al. *The opportunities and challenges of exascale computing*. http://science.energy.gov/. U.S. Department of Energy, 2010.

[XJ96] Lei Xu and Michael I Jordan. On convergence properties of the em algorithm for gaussian mixtures. *Neural computation*, 8(1):129–151, 1996.

[YSM11] Guoshen Yu, Guillermo Sapiro, and Stéphane Mallat. Solving inverse problems with piecewise linear estimators: From gaussian mixture models to structured sparsity. *IEEE Transactions on Image Processing*, 21(5):2481–2499, 2011.

Appendix

Libraries and code

The code with the scripts used is available at the following GitHub repository: `https://github.com/DarthVi/hpcscripts`. The main directory and the "computeNodesCSV" directory contain the scripts used for the extraction of the data from Apache Cassandra and for preprocessing it in order to obtain a single CSV for each compute node, as described in Section 3.2. The "multinode" folder contains instead all the scripts, written in Python and Bash, used in this thesis for our multi-node fault detection research.

The scripts written in Python use the following libraries: Pandas [pdt20], tqdm, Scikit-Learn [PVG+11], Scipy, Imbalanced-Learn, Matplotlib, Seaborn, Jupyter, Streamlit, Plotly.

Code for data extraction and data processing

dcdbquery has been used to get all the sensors available in DCDB. More specifically the command is `dcdbconfig -h 127.0.0.1 sensor listpublic` and the output has been saved in a text file. Later this text file has been modified in order to have one sensor per line (lines in the form `<sensorname> : <sensorname>` become `<sensorname>`) and saved as `sensorlist_trimmed.txt`. Another text file has been created and contains the name of compute nodes we are interested in per line. This file, named `compute_nodes.txt`, is read by `createSensorListPerNode.py`, a Python file whose purpose is to select a subset from `sensorlist_trimmed.txt`. This subset contains only the sensors related to the compute nodes written inside `compute_nodes.txt` and it is saved as `sensorlist_data.txt`. The content that needs to be put inside `compute_nodes.txt` can be retrieved from the log file of the corresponding SLURM job. The code that relies on `compute_nodes.txt` requires that each node is written into this file one per line and without the character "/", so "/mpp3/r02/c05/s02" becomes "mpp3r02c05s02".

Once `sensorlist_data.txt` has been obtain from the procedure previously described, the Bash shell script `splitdata.sh` can be executed in order to obtain all the CSVs with sensor data in them. More specifically there will be one CSV for each sensor, all of them located in the root directory. Moreover this shell script needs 3 command line arguments to be run: the starting timestamp, the ending timestamp (both of them retrievable in the log file of the corresponding SLURM job) and the number of processes to spawn in order to get the data in parallel. The data in all the CSVs obtained can be merged into a single CSV per compute node by running `mergeStatPerNode.py`, which also reads from `compute_nodes.txt`. This Python script will also automatically align the data and check for monotonic columns, converting them to their delta equivalent. The resulting CSVs will be stored in the `computeNodesCSV` folder. `mergeStatPerNode.py` accepts 2 optional command line arguments: the interpolation method (`-i` or `--interpolationmethod`) and the order (`-o` or `--order`, in case the interpolation method is "polynomial" or "spline").

The creation of feature vectors is handled by `createFeatureVectors.py`, that needs to be run inside the folder where the previously defined CSVs are located. When running this Python file, 9 command line arguments can be specified:

`-w` : `--timewindow` the duration in seconds of the aggregating window, the default value is 60;

`-f` : `--features` the number of features to produce for each metric in the dataset, it can be 6 or 11, the default value is 6;

`-s` : `--stepsize` the stride/step size of the rolling window, the default value is 1;

`-p` : `--processes` the number of processes to spawn to speed up the program and execute some parts of it in parallel, the default value is 4;

`-c` : `--chunk` the number of CSV rows to read at each iteration, the default value is 10000; this is useful for processing large CSV files;

`-r` : `--corr` a string translated into a boolean value (possible values: 't', 'f', 'true', 'false'), that indicates wether to compute pairwise correlations among the features, the default value is False;

`-l` : `--horiz` a string translated into a boolean value (possible values: 't', 'f', 'true', 'false'), that indicates whether to compute horizontal aggregation of the CPU-specific metrics, the default value is False; if True for each CPU-specific metric, the script will compute the minumum, maximum, 25th and 75th percentile and the mean;

`-t` : `--label` a string that indicates which labelling strategy to use, it can be 'mode' or 'last'; the default value is 'last';

`-n` : `--normalize` a string translated into a boolean value (possible values: 't', 'f', 'true', 'false'), that indicates wether to perform min-max scaling as normalization technique before computing the feature vectors.

The resulting files will have names following the form `<nodename>_<numfeatures>f_<windowsize>s_<numstep>step.csv` (for example `N1_11f_60s_1step.csv`) and will be located in the same directory where the CSVs from which they are generated are located, namely `computeNodesCSV`.

Code for the experiments

The code for multi-node fault classification is located inside the "multinode" folder. We coded the initial preliminary experiments in the file `randomCrossValidation.py`, whose command line arguments can be retrieved appending `-h` after calling it in the console.

Feature selection is handled by `DT_featureSelection_rus.py` for the version that uses k user defined most important features while `RFECV_featureSelection_optimal_rus` uses the RFECV algorithm to find the minimal number of best features.

For the metaparameters exploration we heavily used the `inmemoryMultirunRandom MultinodeTrainAndTest_featureSelected_allmetrics_updateCSV.py` which loads upfront all the necessary data in memory in order to save time while running multiple random runs. Moreover this scripts appends the results of each run to the `classification _results.csv` for each evaluation metrics, in order to be able to use them later. Another file instead, does not append the results each time and immediately computes the box plots after executing the experiments: `inmemoryMultirunRandomMultinodeTrainAndTest _featureSelected_allmetrics.py`.

The single-node baseline has been obtained running `singlenodeTrainAndTestSplit _featureSelected_allmetrics.py`; the command line arguments needed can be retrieved by using the commmand `-h`.

The code for clustering is contained inside the "multinode/Clustering" folder and can be run using the command `streamlit run clusteringApp.py`. The web interface gives all the information needed for performing clustering and allows the user to choose where to save the results.

The Python script `selectedMultinodeTrainAndTest_featureSelected _allmetrics_withbaseline.py` is used to run the multi-node training and testing experiments with nodes selected via clustering. The command line argument needed can be retrieved via the `-h` (`--help`) command.

We also coded auxiliary scripts to handle the creation of box plots and other scatter plots: these are `create_clustering_random_boxplot_multi.py` and `scatterplot_clustering_vs_random.py`. These scripts can be customized by the user in case changes are needed.

Most of the scripts need the "FileFeatureReader" folder/module available in the root directory order to be able to read and parse the feature selected saved in a .txt file. The scripts in this module are built using the Builder design pattern.