ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN

COMPUTER SCIENCE AND ENGINEERING

Ciclo 37

Settore Concorsuale: 01/B1 - INFORMATICA

Settore Scientifico Disciplinare: INF/01 - INFORMATICA

JOB-LEVEL ONLINE PREDICTIVE MODELLING FOR SUSTAINABLE HPC SYSTEMS

Presentata da: Francesco Antici

Coordinatore Dottorato

Ilaria Bartolini

Supervisore

Zeynep Kiziltan

Esame finale anno 2025

Borsa di dottorato del Programma Operativo Nazionale Ricerca e Innovazione 2014-2020 (CCI 2014IT16M2OP005), risorse FSE REACT-EU, Azione IV.4 “Dottorati e contratti di ricerca su tematiche dell’innovazione” e Azione IV.5 “Dottorati su tematiche Green.” Il codice CUP della borsa è J35F21003070006.

Abstract

High-Performance Computing (HPC) systems are pivotal in addressing complex computational challenges across various scientific and industrial domains. However, their significant energy consumption and environmental impact pose critical sustainability challenges. One possible solution to tackle these challenges is the development of job level predictive models, with the aim of optimizing system throughput while minimizing environmental impact. While promising, these practices are currently not employed in HPC systems due to several important limitations which make them impractical in production environments. The research makes several contributions in addressing such limitations, and making job-level predictive modelling a practical solution towards more sustainable and efficient HPC environments. We first create two comprehensive datasets, namely PM100 and F-DATA, to overcome the scarcity of publicly available, fine-grained job-level data. These datasets facilitate in-depth analysis of job execution characteristics, such as power consumption and resource allocation, and serve as fundamental tools for job-level predictive modelling. Then, online ML-based predictive algorithms are developed to predict key job execution characteristics, including failure, power consumption, and memory/compute-bound nature. These models operate online, leveraging only submission-time features to infer the prediction into job scheduling and resource allocation decision-making. We employ our predictive models into frameworks, e.g. MCBound and UoPC, suitable for deployment in production environment. Such frameworks enable both system-level optimizations and end-user awareness, fostering improved end-user experience, performance and sustainability. This work emphasizes the importance of job-level predictive modelling for sustainable HPC workload management. By addressing the limitations of such practices, our research contributes to the broader mission of sustainable computing, setting the stage for more environmentally conscious HPC systems.

Contents

Chapter 1

Introduction

1.1 Research Context

High-Performance Computing (HPC) refers to the practice of aggregating computational power to solve complex computational problems (such as big data processing, simulations, or real-time analytics) at high speed. HPC systems, such as supercomputers, aggregate processing power, storage, and memory to achieve performance levels far beyond those of conventional computers.

HPC systems typically involve clusters of thousands of interconnected nodes, each containing multiple CPUs or GPUs, to perform tasks collaboratively. To enable efficient communication, the nodes are connected through high-speed interconnects like InfiniBand or Ethernet. HPC systems can scale up by adding more computational nodes, enabling them to handle increasing workloads as computational demands grow. To fully exploit this interconnected computational power and accelerate computations, HPC systems leverage parallel processing, i.e. they divide large computational tasks into smaller sub-tasks, distributed across multiple processors, and executed simultaneously.

HPC systems are not just feats of engineering, but also fundamental enablers of scientific advancements and discoveries. By providing the computational power required to simulate complex physical phenomena, analyze massive datasets, or optimize intricate systems, HPC has revolutionized research in fields such as climate modeling, genomics, astrophysics, drug discovery, and artificial intelligence

(AI). For example, HPC have been instrumental in training large-scale AI models, understanding global climate patterns, or employing massive-scale genome sequencing to accelerate medical research and personalized medicine. Without the capabilities of HPC, many of these breakthroughs would have been impossible or would have required decades to achieve.

1.2 HPC Sustainability Problems

Modern HPC systems are engineering marvels; however, such powerful machines come with significant sustainability challenges. These machines consume enormous amounts of power to operate their processors, memory, storage, and cooling infrastructure. Considerations on the cost of power generation, power delivery, and chiller/cooler infrastructures put 30MW as an upper limit for the power consumption of HPC systems [1]. Despite modern systems being projected to respect this limit, their energy requirements (comparable to those of a small town) result in substantial carbon footprints, making environmental sustainability a critical concern for the HPC community.

To this end, two main factors significantly affecting the system's energy consumption are job scheduling and end-user resource utilization. Addressing these aspects is crucial for reducing the environmental impact of HPC systems while ensuring they continue to drive innovation and scientific progress.

1.2.1 Uninformed Job Scheduling

Job scheduling is normally performed in an uninformed fashion, which means that the scheduling strategy does not take into account energy consumption and job execution characteristics to make the job scheduling decision. This results in having scheduling strategies which make the system reach concerning peaks of power consumptions [2, 3], as they have no understating of the relation between job executions and system power status. For instance, having systems operating over the 30MW limits or creating sudden and huge variation in the power consumption of the systems which can potentially damage the system components.

Moreover, they do not make scheduling decisions suited to the job execution

characteristics [4]. For example, they may schedule energy-intensive jobs to less energy-efficient nodes, increasing power consumption unnecessarily. Uninformed scheduling strategies fail to adapt to changing workloads or system states. They may continue running jobs inefficiently on certain nodes even when better alternatives exist, such as shifting workloads to low-power nodes during periods of low utilization. Furthermore, schedulers often allow users to request more resources than needed, leaving the over-allocated resources powered and consuming energy unnecessarily [5].

1.2.2 End-User Inefficient Resource Utilization

Jobs have different behaviors and requirements, and in order to execute them optimally, they need to be coupled with a correct configuration. Users are often unaware of their jobs' characteristics (e.g. resources required, power consumption, duration, failures) and pick a wrong or suboptimal configuration for the job execution. This includes overprovisioning or underprovisioning the resources requested, wrong type of setup (e.g. node frequency, power cap, wall-time) or types of resources requested [5]. This has a negative impact at job level, as with misconfiguration come errors and increased runtime, but it also has concerning consequences for the whole system, with allocated resources being underused or wasted [6, 7]. In fact, the resources require significant energy and cooling resources. Idle or underused components waste power, driving up operational expenses without delivering proportional computational output. In parallel, such energy consumption results in an unnecessarily increased environmental footprint for the system operations. Moreover, inefficient resource utilization limits the number of jobs that can be processed over time, reducing the system's overall productivity, which can be measured as suboptimal system throughput [6, 8]. In addition, the average time jobs wait in the queue increases as the system fails to match resource availability to user demands effectively, frustrating users, and delaying results.

1.3 Job-level Predictive Modelling

In order to obtain more effective HPC systems, it is thus fundamental to address problems related to their sustainability. As mentioned in the previous section, two main factors which are worsening HPC systems' sustainability and efficiency are unaware job scheduling and end-user inefficient resource utilization. By improving these aspects, it is possible to improve system throughput, while reducing its energy consumption and environmental impact.

1.3.1 Applications and Impact

A practical solution to address both uninformed job scheduling and end-user inefficient resource utilization is predicting job execution characteristics (e.g. job duration, failure, power/energy consumption, resources needed, performance metrics).

First, the predicted characteristics can be used to devise informed job scheduling strategies. The information on power or energy consumption of the jobs can be leveraged to perform power-constraining techniques (e.g. power-capping, delaying power-hungry jobs) or energy-aware scheduling (such as executing energy-intensive jobs on energy-efficient nodes or carbon-aware scheduling). In addition, the prediction of actual resources needed, and performance metrics can be used to pick the best execution setup for the job, aiming to increase its performance. All the cited solutions are instrumental to reduce the energy consumption of the system while ensuring optimal throughput.

Moreover, the predictive models can also be aimed at helping the end-users. By providing information on characteristics and actual resources needed by the job, the users can be helped in picking the correct job configuration, and thus optimizing the job performance, while minimizing resource wastage and energy consumption. Furthermore, prediction of energetic impact and cost (power/energy consumption and carbon footprint) of job executions can increase users' awareness on such topics, promoting downsizing (e.g. less requested resources or delaying the execution to a less carbon intensive period) of jobs in favor of a more sustainable (in terms of energetic footprint) execution on the system.

1.3.2 Use of Machine Learning

In recent years, Machine Learning (ML) has emerged as a cornerstone technology for predictive modeling in the context of HPC systems. ML techniques are particularly well-suited for job-level predictive modeling, as they excel at identifying patterns and relationships in complex, high-dimensional data. This capability has made ML the go-to approach for addressing challenges related to resource prediction, scheduling optimization, and energy efficiency in HPC environments.

ML has been applied to predict key job-level parameters such as runtime [9, 10] or power consumption [11, 12]. These techniques stand out as the optimal approach for job-level predictive modeling due to the following reasons:

- Data-Driven Insights: HPC systems generate vast amounts of operational and job-related data. ML algorithms are designed to analyze and learn from such large-scale data, uncovering intricate patterns that traditional methods struggle to detect.

- Flexibility and Adaptability: Unlike rule-based or heuristic approaches, ML models can be trained on diverse datasets and adapted to various HPC environments and workloads. This adaptability makes ML particularly valuable in heterogeneous and evolving systems.

- Prediction Accuracy: ML models can achieve high accuracy in predictions, especially when trained on well-preprocessed datasets. Accurate predictions are critical for optimizing scheduling decisions and reducing resource wastage.

- Scalability: ML techniques are inherently scalable, allowing them to handle the increasing size and complexity of modern HPC systems. Advanced models, such as deep learning, are particularly effective at scaling to high-dimensional datasets.

- Automation and Continuous Learning: ML models can automate the process of performance analysis and scheduling by continuously learning from new data, ensuring that predictions remain relevant even as workloads and system characteristics change.

1.3.3 Limitations of The State Of The Art

While job-level predictive modeling has seen significant advancements, there are important limitations which make them impractical in production environments. These gaps affect the accuracy, adaptability, and usability of predictive models, limiting their ability to drive sustainable and efficient HPC operations.

Lack of Comprehensive Job-Level Datasets One of the primary challenges in job predictive modeling is the absence of publicly available datasets containing detailed job-level characteristics. In past work [13], it has been shown that extraction of job data from federated grid architectures is non-trivial, yet feasible. Such data is insightful on the characteristics of the jobs executed on the system, and it is fundamental to train and validate predictive models. While some datasets [14, 15] include job runtime and basic resource utilization metrics, they often lack crucial features such as per job energy and power consumption and performance metrics (e.g. memory bandwidth, CPU efficiency or # of flops). The lack of such datasets hinders the development and validation of predictive models, which are instrumental to optimize both performance and energy efficiency.

Lack of Submission-time Models Job predictive modelling becomes particularly important when the predictions can be leveraged to make informed decisions on the job scheduling and resource allocations. This is possible only if the model can generate a prediction before the job is executed, i.e. at job submission-time. The only information available at job-submission time is the job submission-time features (e.g. *resources requested and user information*), and the data of the jobs completed by then. Past work on job level predictive modelling has mainly relied on data not limited to submission-time features [16, 17], making the approach not suitable for real-case scenarios.

Static and Offline models Many ML approaches [11, 12, 18] are trained only once offline using historical data, and do not simulate the real online scenario where job data are live and streaming in time. This setting presents two main problems. First, by not simulating the online environment, the models do not take into account the timeline of job submission and execution. This is fundamental

to ensure that the data in the training set always come before the one in the test set. Otherwise, the model is not evaluated in a realistic scenario, resulting in unreliable prediction performance. Second, the models are not updated over time to adapt to the workload changes or evolving system configurations, which is key to obtain accurate prediction. This can potentially lead to suboptimal prediction performance.

Moreover, such models are not designed to be deployed online in a production environment, making them useful for simulation and testing, but not for deployment.

No Predictive Models for Performance Characteristics A major gap in the state-of-the-art is the lack of models predicting job performance characteristics, such as Memory/Compute-Bound nature, memory bandwidth or #flops. Failure in understanding such job characteristics may lead to suboptimal performance, wasted resources and increased energy consumption. Conversely, as shown in [6, 8], knowing these characteristics before job execution allows to improve system throughput, while reducing significantly system energy consumption. The absence of these models restricts the ability of HPC systems to implement sophisticated scheduling strategies tailored to job-specific performance profiles.

Lack of Tools for End-User Awareness Current research and tools in job predictive modeling focus primarily on system-level optimizations, overlooking the role of end-users. No tools have been developed to help users understand the characteristics of their jobs, such as resource requirements or performance profiles. Providing such information could enable users to make more informed resource selection decisions during job submission, which could improve overall system efficiency and reduce unnecessary energy consumption [5]. For instance, a user could be informed about the actual resources needed for their job, allowing them to avoid overprovisioning or underprovisioning. This would not only enhance the user's experience but also contribute to a more sustainable use of HPC resources. Additionally, end-user tools could raise awareness about the environmental impact of their jobs, encouraging users to adopt more energy-efficient practices. This is particularly important in the context of HPC systems, where energy consump-

tion and carbon emissions are significant concerns. Raising user awareness on the energy impact of their jobs is fundamental to enforce energy-based pricing schemes [19, 20], so as to promote energy-efficient job configurations.

1.4 Research Contributions and Results

In light of the previous considerations, we can conclude that accurate job level predictive modelling is instrumental to reduce the energetic impact of HPC systems, while improving their throughput. However, current approaches lack some fundamental traits which would make them practical for a real system. Therefore, this research aims at filling the gaps in the state-of-the-art for job level predictive modelling, and contribute to the creation of workload management pipelines for more efficient and sustainable HPC systems. Table 1.1 summarizes the limitations of the state-of-the-art for job-level predictive modeling. For each of these limitations, we report the specific contributions of this work in overcoming them, and we highlight how this research differs from past work.

In the early stages of our research, we focused on finding voluminous datasets containing job execution data; this step is fundamental to develop the prediction algorithms and test their accuracy. Since the publicly available resources are scarce, and often do not present fine-grained job execution characteristics (e.g. power consumption, energy consumption, performance metrics), we studied methodologies to extract and create novel fine-grained job datasets from production HPC systems. The methodologies were applied to the data of two production HPC systems, namely Supercomputer Fugaku and Marconi100. We were able to then create two large scale datasets, which we eventually released to augment the public availability of job data, and empower the scientific community with important tools to foster research in job-level predictive modelling.

Then, we worked on job level predictive modelling, trying to address all the limitations of the state-of-the-art. To this end, we develop prediction algorithms which leverage only submit-time information to perform a prediction before the job execution. Our solutions are all *online*, meaning that they are designed to work in a real system where data are streaming in time. We test the prediction performance by keeping into account the actual timeline of the job data, so as

Aspect	State-Of-The-Art	Our Contributions
Job-Level Datasets	Limited datasets with coarse-grained job characteristics. Lacking important features for prediction, such as per-job power/energy consumption and performance metrics [14, 15].	Creation and release of fine-grained datasets extracted from production systems (PM100 and F-DATA) with detailed job execution characteristics, including power/energy consumption and performance metrics.
Submission-time Models	Models trained on post-execution data, unsuitable for real scenarios [16, 17].	Creation of submission-time prediction algorithms, which rely only on submit-time features.
Online Models	Models are static and offline, i.e., they are trained only once on historical data, witout being updated over time [11, 12, 18].	Development of Online prediction algorithms which continuously update the models to adapt to workload changes and evolving system configurations.
Performance Characteristics Prediction	Lack of models predicting job performance characteristics (e.g., memory/compute-bound nature).	Development of MCBound, a predictive framework for job performance characteristics (i.e., the memory/-compute bound nature of the job).
End-User Tools	No tools for end-user awareness.	Creation of UoPC, a tool to help users forecast their jobs' power consumption, aiming to promote energy-efficient practices.
Deployment Readiness	Models designed for simulation and testing, not for production environments.	Development of frameworks (MCBound and UoPC) which are suitable for deployment in real production systems.

Table 1.1: Limitations of the state-of-the-art in job-level predictive modelling and our contributions to address them.

to obtain reliable results. Our algorithms perform continuous model updating,

to adapt to the change of workload of the system and obtain better prediction performance w.r.t. to a static approach, as we will show in the next chapters. Furthermore, we designed predictive algorithms for job performance characteristics, namely the memory/compute-bound nature of the job, and predictive tools for the end-users. Finally, we note that we developed frameworks to easily deploy our predictive algorithms to a real system, providing the first operational tools to perform job level predictive modelling systematically in a production system.

In the next chapters, we first present the background to our research in Chapter 2. In Chapter 3 and 4 we present the two datasets we released, namely PM100 and F-DATA. Then, we present our studies on the prediction of different job execution characteristics, namely the job failure (Chapter 5), job power consumption (Chapter 6) and job memory/compute-bound nature (Chapter 7). In Chapter 8 we present our work on the creation of an end-user tool for job predictive modelling, and we finally conclude in Chapter 9.

Chapter 2

Background

2.1 HPC systems

2.1.1 History of HPC

HPC has evolved dramatically over the past few decades, transitioning from early mainframes and vector processors to today's powerful supercomputers capable of solving complex scientific, industrial, and societal problems.

Early days The roots of HPC can be traced back to the development of early electronic computers in the mid-20th century. Machines like the ENIAC (Electronic Numerical Integrator and Computer) and UNIVAC were among the first to perform large-scale computations. Although these systems were primitive by modern standards, they were groundbreaking for their time, enabling tasks such as ballistic trajectory calculations and census data analysis. The concept of parallel computing emerged during this era as a way to improve computational speed. By dividing tasks across multiple processors, early computer scientists realized they could achieve faster results. This principle became a cornerstone of HPC, laying the groundwork for future advancements.

The Advent of Supercomputing: 1960s and 1970s The 1960s marked the birth of supercomputing as a distinct field. Seymour Cray, often referred to as the "father of supercomputing," played a pivotal role in this era. Working at

Control Data Corporation (CDC), Cray designed the CDC 6600, considered the first true supercomputer. Released in 1964, the CDC 6600 achieved unprecedented processing speeds by introducing pipelining and parallel processing techniques, innovations that would become fundamental to HPC systems.

Cray's work continued into the 1970s with the development of the Cray-1, a system renowned for its distinctive cylindrical design and use of vector processing. Vector processors optimized mathematical operations on large datasets, making the Cray-1 ideal for scientific simulations and modeling. This period also saw the rise of government and academic interest in HPC, with systems being used for tasks like nuclear simulations, weather forecasting, and aerospace research.

Scaling Up: The 1980s and 1990s The 1980s and 1990s were transformative decades for HPC, driven by advancements in hardware, software, and networking. Parallel computing became more prominent, with the introduction of massively parallel processing (MPP) architectures. MPP systems, such as the Thinking Machines Corporation's CM-5, employed thousands of interconnected processors to achieve high speeds. These systems marked a shift from the vector processing paradigm to a focus on scalability.

The development of distributed computing further enhanced HPC capabilities. By connecting multiple systems via high-speed networks, distributed computing allowed researchers to leverage collective computational power. This era also saw the emergence of standardized programming models like MPI (Message Passing Interface) and PVM (Parallel Virtual Machine), which facilitated parallel programming and resource management.

Government initiatives, such as the U.S. Department of Energy's investment in supercomputing centers, spurred further advancements. Machines like IBM's Deep Blue demonstrated the power of HPC in specialized tasks, famously defeating world chess champion Garry Kasparov in 1997, a milestone in artificial intelligence and computational chess.

The Petascale Era: 2000s The early 21st century marked the beginning of the petascale era, where supercomputers achieved processing speeds measured in petaflops (quadrillions of floating-point operations per second). This milestone

was first reached by IBM's Roadrunner in 2008, a hybrid system that combined traditional processors with specialized accelerators.

This period saw an explosion in the demand for HPC, driven by the growth of data-intensive fields like genomics, climate modeling, and financial analytics. Advances in hardware, such as multicore processors and GPUs (Graphics Processing Units), enabled greater computational efficiency and parallelism. GPUs, initially developed for gaming and graphics, became a key component of HPC systems, offering immense processing power for tasks like machine learning and scientific simulations.

The rise of open-source software, such as Linux, also revolutionized HPC. Linux became the operating system of choice for supercomputers, providing a flexible and cost-effective platform for large-scale computation.

The Exascale Pursuit: 2010s to Present The 2010s and beyond have been defined by the pursuit of exascale computing, where systems achieve speeds exceeding one exaflop (10^{18} floating-point operations per second). This leap in performance has been made possible by innovations in architecture, interconnect technology, and energy efficiency.

Supercomputers like Fugaku in Japan, Frontier in the U.S., and LUMI in Europe have pushed the boundaries of HPC. Fugaku, launched in 2020, integrates specialized ARM processors to achieve world-leading performance while emphasizing energy efficiency. These systems are being used to tackle global challenges, from simulating the spread of COVID-19 to modeling climate change impacts.

2.1.2 HPC in modern society

In today's society, HPC systems are at the forefront of modern technological advancements, impacting a wide range of industries and societal functions. These systems have become critical in scientific research, as they allow for execution of large scale and computationally intensive workload, outside the capabilities of normal computing architectures.

In the scientific community, HPC enables breakthroughs in areas like genomics, material science, and particle physics, supporting researchers in modeling complex

phenomena that would be impossible or impractical to study in physical laboratories. In healthcare, HPC aids in drug discovery, disease research, and personalized medicine, allowing researchers to simulate biological processes and analyze extensive genetic data. Furthermore, in environmental science, HPC is essential for climate modeling and predicting natural disasters, helping societies prepare and mitigate the effects of climate change.

Supercomputers like Frontier and Fugaku exemplify how large-scale architectures are being leveraged for scientific discovery. These machines can perform more than 10^{15} operations per second, making them indispensable for solving complex equations, simulating natural phenomena, and analyzing vast datasets. In several cases, such architectures allow to solve urgent problems, which otherwise would require months. For instance, Fugaku was instrumental in accelerating COVID related research. It ran simulations to study how respiratory droplets spread in various environments, aiding in the development of effective social distancing and ventilation guidelines. Additionally, Fugaku was used to analyze potential drug candidates by simulating how various compounds interact with the virus at the molecular level, significantly speeding up the drug discovery process. Its unparalleled processing power allowed researchers to model complex scenarios in days that would have otherwise taken months, demonstrating the critical role of supercomputing in addressing global health crises.

As modern society's demand for computational power grows exponentially, HPC is not just a tool for specialized fields, but a foundational technology fundamental for modern innovation.

2.1.3 Using HPC systems: Jobs and Schedulers

Accessing and efficiently utilizing HPC systems requires specialized workflows, tools, and protocols. HPC systems are designed to manage the computational needs of thousands of users simultaneously, ensuring optimal resource utilization and minimizing conflicts. Most users access HPC systems remotely via secure shell (SSH) connections. Upon gaining access, users typically work within a shared environment, usually a Linux-based operating system, where they can upload data, manage files, and configure their computational tasks.

Jobs Users execute their applications on an HPC system under the form of *jobs*, which are instances of computational tasks executed on a set of system resources that are allocated for a finite amount of time. In fact, users do not directly interact with computational resources like processors or memory, instead they submit *jobs* to the system. To submit a *job*, the user prepares a *job script*, which is a configuration file that specifies the details of the *job* execution, such as the amount of hardware requested, the expected runtime of the job, the logging files and most importantly the commands, libraries and modules needed to execute the computational task. In Listing 2.1, we provide an example of a *job script* which requests 8 computational nodes for 2 hours of time. The *job* executes the `python` script `simulation.py` and saves the output of a computation in the `output.log` file. Depending on the *job scheduler* installed on the system (e.g. SLURM, PBS, etc), the *job script* must contain specific keywords (e.g. #SBATCH) to specify requirements for the *job*. In the case of Listing 2.1, the *job script* is written for the SLURM[1] *job scheduler*.

Listing 2.1: Example of a Job Script for the SLURM Job Scheduler

```
1
2  #!/bin/bash
3  #SBATCH --job-name=example\_job
4  #SBATCH --output=output.log
5  #SBATCH --nodes=8
6  #SBATCH --time=02:00:00
7  #SBATCH --partition=compute
8
9  module load python
10 python simulation.py
```

Once the *job script* is ready, the user submits it to the system through the *job scheduler*, and the *job* is then queued for execution.

Job Scheduling The *job scheduler* is a critical component of any HPC system. Such component is a software tool responsible for allocating resources and managing the queue of submitted jobs. The *job scheduler* dynamically allocates resources to maximize system utilization, while ensuring that high-priority jobs are executed promptly. Such software is highly scalable and high performance, since

[1] `https://slurm.schedmd.com`

they are required to handle scheduling of thousands of jobs per seconds, submitted by hundreds of different users.

The *job scheduler* executes *jobs* in the queue based on several factors, which are the priority policy, the resource availability and dependency. The priority policy is usually determined at system level, and may consider factors like job size, user group, or fairness policies. The *jobs* need to wait for the availability of the resources requested, which might result in having smaller jobs be executed sooner than larger ones. As the *Job Scheduler* also aims at maximizing the resource utilization and minimizing the *job* waiting time in the queue, such software may consolidate smaller jobs into available nodes or pause lower-priority jobs to make way for urgent tasks. Finally, some *jobs* may depend on the completion of others and hence are scheduled accordingly.

Once a job is submitted, most *job schedulers* allow the users to monitor their *job* and obtain information on its status (e.g. running, pending or failed) and execution metrics (e.g. power consumption, resource utilization and duration). Such information is generally stored in detailed logs and reports, generated after the *job* completes. Users can then analyze such logs to identify bottlenecks, optimize their code, or debug errors. Furthermore, these data can be fed into AI and data-driven models to predict *job* execution characteristics (e.g. power consumption, failure, duration, etc) and patterns.

2.2 Systems studied

During our research, we had the opportunity to work with two production HPC systems, namely Marconi100 and Supercomputer Fugaku, from which we extracted job-level data.

2.2.1 Supercomputer Fugaku

Supercomputer Fugaku, developed by RIKEN and Fujitsu in Japan, is one of the most powerful and advanced HPC systems ever created. Named after Mount Fuji, this system has set global benchmarks in supercomputing, excelling in both performance and versatility. Fugaku is one of the most effective supercomputers

in the world, tackling problems in: medical research, climate science, disaster management and AI research.

The development of Fugaku began as the successor to the K computer, another groundbreaking Japanese supercomputer that operated from 2011 to 2019. The project to design and build Fugaku started in 2014, and it was officially completed in March 2020. Despite becoming fully operational in 2021, it was heavily used during the outbreak of the COVID-19 pandemic to simulate droplet dispersion and identifying potential drug candidates. These contributions underscored its practical value in addressing global crises.

Table 2.1 summarizes Fugaku architecture and performance. Fugaku is built on Fujitsu's A64FX processor, the world's first CPU based on the Arm architecture designed specifically for HPC, differently from traditional x86 processors. The A64FX is equipped with 48 computational cores and is optimized for high memory bandwidth and energy efficiency. Thanks to more than 150K interconnected nodes, the system theoretical peak performance is around 537 petaflops[2]. The system is designed with energy efficiency in mind. It uses liquid cooling technology to maintain optimal operating temperatures, reducing the energy required for cooling while maximizing performance.

System characteristic	Description
Architecture	Armv8.2-A SVE 512 bit
OS	Red Hat Enterprise Linux 8
#Nodes	158.976
#Cores	48 + 2 assistant cores (per node)
Memory	HBM2, 32 GiB (per node)
Peak Performance	$\approx$ 537 Pflop/s (FP64)
Internal Network	Tofu D Interconnect (28 Gbps)

Table 2.1: Fugaku system architecture.

In the years since its deployment, Fugaku has consistently ranked among the world's most powerful supercomputers, claiming the number one spot in multiple categories (e.g. Top500, Graph500 and HPCG). Nowadays, after more than 4 years since its debut, it still occupies the $6th$ position in the latest (November

[2] 10^{15} floating-point operations per second

System characteristic	Description
#Nodes	980
#Processors (per node)	2x16 cores IBM POWER9 AC922, 3.1 GHz
#Accelerators (per node)	4 x NVIDIA Volta V100 GPUs, 16 GB
#CPU cores (per node)	32
Amount of RAM (per node)	256 GB
Peak performance	32 PFlop/s

Table 2.2: Marconi100 system characteristics.

2024) Top500 list [3] (a list ranking the world's most powerful supercomputers).

Fugaku relies on a proprietary operations management software[4], built as an extension of PBS [21] which features workload management operations like job manager, job scheduler, and functions that enable the recording and storage of job data.

2.2.2 Marconi100

Marconi100 is an ex-production supercomputer installed at CINECA,[5] Italy's leading HPC center. During its operational phase, which spanned from March 2020 to June 2023, it ranked among the world's fastest supercomputers, holding the 9th position in the June 2020 Top500 list. This achievement underscored its significance as a computational resource, particularly in academic research.

The system could deliver approximately 32 petaflops of peak performance, and it was based on IBM Power9 processors and NVIDIA Volta GPUs. Marconi100 was composed of 980 nodes, each featuring two 16-core processors and four GPUs, interconnected through Mellanox Infiniband EDR DragonFly+. This configuration facilitated efficient and high-speed computation for a wide range of applications in physics, climate modeling, biology, and materials science. It also offered 8 PB of storage and 256 GB of RAM per node to handle data-intensive tasks efficiently. The system architecture is summarized in Table 2.2.

[3] `https://top500.org/lists/top500/2024/11/`

[4] `https://www.fujitsu.com/global/about/resources/publications/technicalreview/2020-03/article10.html#cap-03`

[5] https://www.cineca.it/it

Marconi100 is endowed with Examon [22], a software which allows the monitoring of the system. Examon collects data concerning management, workload, facility, and infrastructure of the Marconi100 supercomputer, including hundreds of metrics measured on each computing node and hundreds of others gathered from sensors monitoring the system components.

2.3 Predictive Modelling Tools

In this section, we provide the necessary background to all the predictive modelling tools used in our research. First, we explain what is AI and why we use it, and then we present the ML and AI tools we used for job-level predictive modelling.

2.3.1 Artificial Intelligence

Recent years witnessed unprecedented leaps forward in scientific discovery and technology development. Such advancement has been largely driven by the integration of Artificial Intelligence (AI) and data-driven techniques (e.g. numerical simulations, Machine Learning (ML) and Deep Learning (DL)). AI refers to the simulation of human intelligence in machines designed to perform tasks that typically require human cognitive abilities. These tasks include learning from data, reasoning, problem-solving, understanding natural language, and perceiving or interpreting sensory input. AI systems range from narrow AI, specialized in specific tasks like language translation or image recognition, to the concept of artificial general intelligence (AGI), which would exhibit human-like cognitive versatility. Such solutions have become fundamental catalysts for scientific progress, enabling breakthroughs across numerous fields by automating complex tasks, analyzing massive datasets, and generating actionable insights.

In healthcare and genomics, AI has transformed drug discovery and personalized medicine. ML algorithms have drastically reduced the time needed to identify potential drug candidates, as seen during the rapid development of COVID-19 vaccines. Similarly, tools like DeepMind's AlphaFold have revolutionized biology by accurately predicting protein structures, a challenge that stumped scientists for decades. This advancement is opening new avenues in understanding diseases

and developing targeted treatments. In climate science, AI is helping researchers model and predict the impacts of global warming with unprecedented accuracy. By analyzing satellite imagery, climate data, and environmental patterns, AI systems are improving predictions of extreme weather events, aiding disaster preparedness, and enabling better resource management. In astrophysics, AI is being used to sift through terabytes of data from telescopes, detecting exoplanets, identifying black holes, and even revealing new insights about the formation of galaxies. Meanwhile, in material science, ML is accelerating the discovery of new materials, including those with properties suitable for renewable energy and quantum computing.

Despite being often considered as a modern breakthrough, AI is an idea rooted in the mid-20th century, when pioneers like Alan Turing and John McCarthy envisioned machines capable of mimicking human intelligence. Early AI research developed concepts like symbolic reasoning and rudimentary ML algorithms, but the limited computational power of the time rendered these theories largely impractical. Only in recent decades, with the advent of advanced computational centers, high-performance GPUs, and cloud computing, has AI truly come into its own. These innovations have made it possible to process massive datasets, train complex models, and perform calculations at speeds unimaginable to early researchers, transforming AI from a theoretical pursuit into a transformative force across industries.

Modern state-of-the-art AI models show incredible predictive performance, due to their capability of analyzing enormous quantity of data to discover patterns and learn correlations which are outside the reach of other predictive techniques (e.g. heuristics, exponential smoothing, imperative algorithms). For this reason, we employ different ML and AI models for our job-level predictive modelling studies.

2.3.2 AdaBoost

The Adaptive Boosting (AdaBoost) algorithm, introduced in [23], is one of the earliest and most influential boosting methods, designed to improve the accuracy of weak classifiers by iteratively combining them into a strong predictive model. AdaBoost is widely recognized for its simplicity, effectiveness, and versatility in both classification and regression tasks, making it a fundamental technique in the

machine learning toolkit.

The central idea of boosting is to combine the predictions of multiple weak learners (typically Decision Trees (DTs)), to create a single strong learner. Unlike bagging methods such as Random Forest, where models are trained independently, AdaBoost sequentially trains weak learners and adapts their weights based on their performance. This adaptive process allows the model to focus on difficult examples, progressively reducing errors. AdaBoost assigns higher weights to misclassified samples, forcing subsequent models to focus on these challenging cases. Each weak learner contributes to the final model based on its accuracy, with better-performing learners given higher weights. AdaBoost requires no parameter tuning for individual weak learners and can work with various base models, although DTs are commonly used.

At initialization, equal weights are assigned to all the training samples. All the weak learners are trained on the weighted dataset, and the error rate on all the samples is collected. The weights of misclassified samples is increased, while the ones of the correctly classified is decreased. This is done to put more focus on the samples which are harder to predict on the following iterations of the algorithm. Each learner is assigned with a weight too, based on its accuracy. Learners with lower error rates receive higher weights, indicating greater influence on the final prediction. Finally, the AdaBoost performs a prediction by combining the predictions of all weak learners using a weighted majority vote (for classification) or a weighted average (for regression).

AdaBoost is generally sensitive to noisy data and outliers since misclassified samples receive higher weights. Despite this, its interpretability and effectiveness have made it a foundational technique in ensemble learning.

2.3.3 XGBoost

XGBoost, short for eXtreme Gradient Boosting, is a powerful and widely used ML algorithm designed for supervised learning tasks such as regression and classification. Presented in [24], XGBoost is a scalable and efficient implementation of gradient boosting, which gained popularity due to its impressive performance on ML benchmarks.

The foundation of XGBoost lies in gradient boosting, similarly to the AdaBoost, i.e. combining weak learners (usually decision trees (DTs)) into a strong predictive model. Gradient boosting optimizes a loss function by sequentially adding DTs that correct the errors of previous models. Unlike traditional boosting methods, XGBoost incorporates several advanced techniques to enhance its efficiency and performance, such as:

- Regularization: XGBoost includes L1 and L2 regularization to prevent overfitting and improve generalization.
- Tree Pruning: The algorithm uses a maximum depth parameter and employs a post-pruning strategy to avoid over-complex DTs.
- Handling Missing Data: XGBoost automatically handles missing values in the dataset, estimating their impact during training.
- Parallel Processing: By leveraging parallelism, XGBoost speeds up tree construction and evaluation, making it highly efficient for large datasets.
- Customizable Loss Function: Users can define custom loss functions to tailor the algorithm to specific use cases.

The algorithm starts by initializing predictions for the target variable with a baseline value (often the mean for regression or a constant for classification). Then, at each iteration, a new DT is fitted to the gradient of the loss function with respect to the current predictions. This gradient represents the errors the model needs to correct. This new tree is added to the model, and its predictions are scaled by a learning rate (a hyperparameter) to control the contribution of each tree.

XGBoost minimizes a combined objective function comprising the loss function (e.g., mean squared error for regression) and a regularization term that penalizes model complexity. The process repeats for a specified number of iterations or until the model achieves a desired level of accuracy. After all iterations, the predictions of the ensemble model (the sum of all trees' outputs) are used to make final predictions.

2.3.4 Logistic Regression

Logistic Regression (LR) [25] is a widely used ML method for binary classification tasks (it can be extended to multi-class classification with variants like multinomial LR). Despite its name, LR is a classification algorithm, not a regression technique. It models the probability of a sample belonging to a specific class, making it ideal for problems where the outcome is categorical, such as spam detection, medical diagnosis, and customer churn prediction.

LR leverages the logistic function (also called the sigmoid function), which maps real-valued inputs to probabilities between 0 and 1. It assumes a linear relationship between the input features and the log-odds of the target class. This simplicity makes logistic regression interpretable and computationally efficient, even with large datasets.

The algorithm initializes a set of weights (parameters) for the input features, which will be optimized during training via optimization algorithms (e.g. gradient descent). LR models the log-odds (logarithm of the odds) of the target class as a linear combination of the input features, as shown in Equation 2.1; in the equation β_0 is the intercept, and $\beta_1, \beta_2, \ldots, \beta_n$ are the weights. The log-odds are converted into probabilities using the sigmoid function, which maps the probability in the range [0, 1]. LR optimizes the weights by minimizing the log loss (cross-entropy loss), which measures the difference between predicted probabilities and actual labels. After several iterations of the optimization algorithm, the trained LR predicts the class of a sample by applying a threshold (e.g., 0.5) to the predicted probability. If $P(y = 1|x) \geq 0.5$, classify the sample as 1; otherwise, classify it as 0.

$$\text{log-odds} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n \tag{2.1}$$

LR is easy to implement and provides interpretable coefficients, showing the impact of each feature. It works well with small to medium-sized datasets and does not require extensive computational resources. However, logistic regression has limitations, such as its reliance on the linearity assumption and reduced performance with highly complex or non-linear data. Despite this, its effectiveness, interpretability, and probabilistic nature make logistic regression a cornerstone of

machine learning and a baseline for many classification tasks.

2.3.5 Random Forest

The Random Forest (RF) algorithm is a widely used learning method for classification, regression, and other tasks. RF is an ensemble method, which leverages several DTs to enhance predictive accuracy, reduce overfitting, and improve robustness in handling diverse data types. Its lightweight nature, coupled with the ability to deliver reliable predictions, has made it a standard in various domains, from medical diagnostics and predictive modeling, to finance and environmental science.

The concept of DT, which form the foundation of RF, dates back to the 1960s, while RF itself was introduced in 2001 [26]. The algorithm incorporates two key innovations, such as bootstrap sampling (i.e. creating multiple subsets of data for tree training) and feature randomization (i.e. selecting a random subset of features at each split to reduce correlation among trees). RF outperforms single DTs and even other ensemble methods in many scenarios, especially when dealing with noisy or missing data [26].

RF employs n instances of DTs (usually ~ 100). The training phase consists of training each DT independently on a different subset of the data. In addition, each DT is trained on a different feature set of the data, selected randomly. Then, RF infers on new data, by considering the predictions of all the trained DTs. For classification tasks, it aggregates DTs predictions using majority voting, while for regression tasks, it calculates the average predicted value.

2.3.6 K-nearest neighbors

The K-Nearest Neighbors (KNN) algorithm [27] is a non-parametric learning method used for classification and regression tasks. It is based on a vector distance approach, which makes it highly effective in applications involving pattern recognition, recommendation systems, and anomaly detection.

Such method was first introduced in the 1950s as a statistical method for pattern recognition. The algorithm gained popularity due to its simplicity and

effectiveness, becoming a foundational technique in data science and artificial intelligence. Over time, KNN has evolved with various adaptations to optimize its efficiency, particularly for large datasets.

KNN predicts outcomes based on the proximity of data points in feature space. It operates on the assumption that similar data points share similar characteristics. The algorithm is non-parametric, as it does not make any prior assumptions about the data distribution. In fact, KNN employs a lazy learner method, i.e. it does not build a model during training; instead, it memorizes the dataset and performs calculations during inference.

The algorithm infers on new data as follows:

1. Chooses the Value of k, where k is the number of nearest neighbors to consider when making a prediction. A low k value (e.g., 1) can make the model sensitive to noise, while a high k can smooth predictions but risk losing detail.

2. Calculates Distance: The algorithm computes the distance (such as Euclidean, Manhattan, Minkowski) between the new data point and all other data points in the training set.

3. Identifies Nearest Neighbors: The k closest data points are selected based on the computed distances.

4. Makes Predictions: For classification the class label with the highest frequency among the k neighbors is assigned to the new data point (majority voting). Conversely, for regression, the average value of the target variable among the k neighbors is used as the prediction.

The non-parametric nature of the KNN removes the need for heavy model training and big availability. However, KNN suffers the curse of dimensionality, as with growing dataset sizes, the computational cost of calculating distances increases, posing challenges for large-scale applications.

2.3.7 SBert

Sentence-BERT (SBERT) [28] is an advanced variation of the BERT (Bidirectional Encoder Representations from Transformers) model [29], designed specifically for producing high-quality sentence embeddings. Developed by Nils Reimers and Iryna Gurevych in 2019, SBERT extends BERT's capabilities to better handle tasks requiring semantic understanding of sentence pairs, such as semantic search, question answering, and text clustering. SBERT is heavily used in tasks like semantic search, question answering and text clustering.

The original BERT model, introduced by Google in 2018, revolutionized natural language processing (NLP) by enabling deep contextualized representations. It employed a transformer architecture with a bidirectional attention mechanism to pretrain on large-scale text corpora using masked language modeling and next sentence prediction. While highly effective, BERT required computationally expensive pairwise input comparisons for sentence-level tasks. SBERT optimizes BERT's transformer architecture for efficient computation of sentence similarities, addressing BERT's computational limitations.

SBERT is created by fine-tuning a Siamese network architecture employing BERT models on sentence similarity tasks. Such a solution allows to derive fixed-length vector representations for sentences, which enable semantic text similarity tasks to be performed not only faster, but also more accurately, as shown in [28].

Chapter 3

PM100: A Job Power Consumption Dataset of the Marconi100 System

Job power consumption refers to the amount of electrical power consumed by the job while executing its computational tasks on the system resources. Hence, the total power consumption of a job is computed by aggregating the power consumption of all the resources allocated to the job during its execution. Predicting the power consumption of a job before its execution would allow to forecast the whole system's power consumption. Past work proved the feasibility of predicting job power consumption by leveraging on workload manager information [12, 15, 30, 31]. The prediction can then be exploited by the workload manager to perform techniques like power capping [2, 32].

All the referenced work addressed the prediction task by using data-driven techniques exploiting a structured dataset. Due to the steady development of such techniques, the availability of quality data extracted from a production HPC system has thus become a leading priority. Such resources are, however, often limited due to the inherent complexity of collecting structured data for job power characterization in a production system.

Power consumption measurement relies on, among others, hardware sensors and different software interface. To give an idea, most of the modern systems have

in-band software interface (available within the operating system of the compute node) for the power measurement of compute elements, whereas for node-level and component-level measurement, they rely on out-of-band interface (observable in the management network) and smart power switches which monitor cluster power consumption (observable in the facility management network). Job information instead requires monitoring the workload manager, which is possible from the login and master node. Accessing simultaneously to all these data resources requires different privilege levels and monitoring software validation procedures. While this is relatively easy in a test environment, it is not the case in a production machine. On the other hand, job characteristics of a production machine are more relevant for predictive studies as they reflect the behaviour of a multitude of real user. Test clusters tend to have synthetic and small-scale workload submitted from a smaller set of users with larger idle time, which limits generality.

In order to fill the lack of resources for job power prediction, we propose an approach to extract job power consumption data from workload manager data and node power metrics logs, which can easily be obtained through system plugins. Moreover, we created a large job dataset named PM100, with fine-grained job power consumption information. The PM100 dataset is derived from the M100 workload [33], a holistic dataset extracted from a large-scale production HPC, using the approach that we propose and is accessible through Zenodo [34].

To the best of our knowledge, the only publicly available dataset for job power consumption is presented in [15]. The dataset contains 80K jobs using CPU cores and is extracted from two production HPC clusters, with 560 and 728 nodes, respectively. Our work differs from [15] in a number of ways. First, our dataset is based on the first holistic dataset M100 of a more powerful tier-0 production supercomputer (Marconi100) and contains many more jobs (80k vs 230k). Second, as the nodes of Marconi100 are equipped with GPUs, the collected power consumption data refers to two types of jobs (those using only cores and also GPUs). Third, the data presented in [15] lack job information present in PM100, such as the job exit state, making the dataset not suitable for the job failure analysis and prediction tasks presented in Section 3.3. Finally, our approach can be applied to any public workload data with node power metrics logs so as to create new job power consumption datasets. With the results of our work, we strive to empower

the HPC community with tools to drive research in optimizing system performance and power consumption.

3.1 M100 Dataset

The M100 workload data [33] is collected during two and a half years of operation of Marcon100. It is the first holistic dataset of a tier-0 supercomputer, and it is the largest (49.9 TB in size before compression) publicly available. It contains data ranging from the computing nodes' internal information such as core load, temperature, power consumption, to the system-wide information, including the liquid cooling infrastructure, the air-conditioning system, the power supply units, workload manager statistics, and job-related information. For our purposes, we are interested in the job data and node power metrics.

Job data The job data is collected in the job table plugin. We focus on the data that describes the jobs present in the workload by features related to their submit-time, run-time and end-time. The first category contains the information available when a job is submitted, such as submission time, number of requested resources, user information and system state. The second category comprises the information about the job launch, such as waiting time, execution start time, and the actual number of allocated resources. At job termination, the end-time features are collected, e.g., ending time, duration and outcome of the execution. The full list of job features is available at the dataset repository.[1] The job termination features do not contain job power consumption, so we need to extract this information from the power metrics logs of the nodes present in M100.

For the purposes of our work, we consider only a part of the dataset[2] and use only the data collected between May 2020 and October 2020. The reason is that this is the only period where the dataset contains information on the requested resources, which is useful to give a more in-depth description of each job and could be exploited for prediction tasks. The considered period contains around 1 million

[1] `https://gitlab.com/ecs-lab/exadata/-/blob/main/documentation/plugins/job_table.md`

[2] `https://doi.org/10.5281/zenodo.7588815`

jobs.

Node power metrics The power consumption data of the system components is contained in the IPMI plugin, which collects several metrics on cluster nodes, such as *ambient temperature, node temperature, fan speed, node power, CPU power, memory power*. The full list of metrics present in the IPMI data, and their relative sampling time, is reported in the original documentation of the dataset.[3] The values of all the metrics are collected every 20 seconds on all the system nodes. The final data is saved in a table, divided by node and collection time.

For our purposes, we consider only the metrics $ps0\_input\_power$ and $ps1\_input\_power$, which contain the power consumption values recorded at the input of the two power supplies of the nodes. Thus, the power consumption of a node n at time t_i can be obtained by summing $ps0\_input\_power_{n,t_i}$ and $ps1\_input\_power_{n,t_i}$.

3.2 PM100 Dataset Creation

In this section, we first discuss the information that needs to be included in the data to apply our methodology. Then, we describe how we selected the jobs in M100 to include in the dataset PM100; this step is crucial to guarantee data soundness. We then explain how to extract job power consumption data starting from the power metrics logs of the nodes present in M100. At the end, we discuss the job features present in PM100.

Data requirements The methodology we propose in this work can be applied to any workload manager data and node power metrics logs, providing the following information.

The workload manager data need to contain, for each job j, its start time, end time, and the nodes $nodes_j$ allocated to the job j. This information will be used to filter out the exclusive jobs and to extract the job j power consumption p_j.

The node power metrics logs must keep record of the power consumption values of the single nodes of the systems, at different timestamps t_i. Such values are used to compute the job j power consumption p_j, at the different time t_i.

[3] `https://gitlab.com/ecs-lab/exadata/-/blob/main/documentation/plugins/ipmi.md`

In this work, we consider the node n power consumption at time t_i as the sum of $ps0\_input\_power_{n,t_i}$ and $ps1\_input\_power_{n,t_i}$. However, if the $ps0\_input\_power$ and $ps1\_input\_power$ values are missing, the node power consumption can be defined differently, in accordance with the data present in the node power logs.

Job filtering In Marconi100, multiple jobs can run on the same node at the same time. Therefore, node power consumption depends on the power consumption of the execution of multiple jobs on the node's resources. We are not aware of a methodology to evaluate accurately the contribution of each job execution to the node power consumption. Thus, we consider only the jobs that run alone on all the allocated nodes throughout their execution, to provide accurate power consumption information.

In order to filter out the exclusive jobs, we implement a pipeline defined as follows. First, we create a hash-table for each node n, where the keys are the timestamps t_i of a fixed period of time Δ, sampled every θ seconds. The value related to the key t_i is a list containing the IDs of the jobs running on n during that particular timestamp t_i. For each job j, we round the start time (ceiling rounding) and the end time (floor rounding) to the closest t_i (referred to as $start\_time_j$ and end_time_j). We then add the job ID to the lists of all the t_i that fall between $start\_time_j$ and end_time_j, for all the nodes $n \in nodes_j$ allocated to j. Finally, from the resulting tables, we filter out all the jobs j that are the only members of the lists hashed by t_i falling between $start\_time_j$ and end_time_j for all $n \in nodes_j$.

Despite considering all the t_i of all the jobs is a costly operation, it can be very useful for future work. For instance, it can be used to study the power consumption of concurrent jobs and how their power consumption intertwine during their execution. Moreover, one can augment the data by considering the power profiles of the concurrent jobs just in the timespans where they ran alone.

Job power consumption extraction In order to extract job power consumption, we need to perform a data post-processing pipeline to correlate each job to the power consumption caused by its execution. We define the power consumption of an exclusive job j as a list p_j, where each element is the job power consumption computed at t_i, for all the t_i intersecting the job execution. The job power con-

Input : $job\ j, ps0\_input\_power, ps1\_input\_power$
Output : p_j
$p_j = []$
$t_i = start\_time_j$
$\theta = 20$
while $t_i \leq end\_time_j$ **do**
  $p\_nodes_{j,t_i} = 0$
  for $n \in nodes_j$ **do**
    $p\_nodes_{j,t_i} = p\_nodes_{j,t_i} + ps0\_input\_power_{n,t_i} + ps1\_input\_power_{n,t_i}$
  end
  $p_j = p_j + [p\_nodes_{j,t_i}]$
  $t_i = t_i + \theta$
end

Algorithm 1: Job power consumption extraction

sumption at time t_i is obtained as the sum of the power consumption of the nodes $n \in nodes_j$ allocated to the job j (p_nodes_{j,t_i}). As mentioned in Section 3.1, the power consumption of a single node n at time t_i is the sum of $ps0\_input\_power_{n,t_i}$ and $ps1\_input\_power_{n,t_i}$. Thus, p_nodes_{j,t_i} can be computed as shown in Equation 3.1. Combining the definitions, we can create the job power consumption values list as shown in Equation 3.2.

$$p\_nodes_{j,t_i} = \sum_{n \in nodes_j} ps0\_input\_power_{n,t_i} + ps1\_input\_power_{n,t_i} \tag{3.1}$$

$$p_j = [p\_nodes_{j,start\_time_j}, \ldots, p\_nodes_{j,end\_time_j}] \tag{3.2}$$

Algorithm 1 lists the steps performed to extract the power consumption of each exclusive job filtered in the previous step. For each job j that has $end\_time_j - start\_time_j > 0$, we initialize p_j as an empty list. The length of p_j will be $end\_time_j - start\_time_j/\theta$, where θ is the sampling time of the power values in the power data (20 seconds in our case), namely the distance in time between two consecutive power measurements in the data. We iterate over all the t_i intersecting the execution of j, (i.e. $start\_time_j \leq t_i \leq end\_time_j$), and we compute the p_nodes_{j,t_i}. At the end of each iteration, p_nodes_{j,t_i} is added to p_j and t_i is updated. The algorithm returns p_j containing the power consumption values of the job j during its execution.

The code to perform the full pipeline is available on the GitHub repository[4] of the dataset. We run the scripts on a machine endowed with 32 cores and 256 GB of RAM, and the runtime of the whole process was around 8 weeks, with only the job power consumption extraction requiring more than 7 weeks. The lengthy computation is mainly due to the size of the initial dataset, which contains around 1 million jobs. The filtering process removed all the jobs that do not run exclusively on the nodes and with which we encountered problems due to missing values in the power data. The final dataset ($\sim$ 100 MB) contains 231,238 jobs and is accessible through Zenodo [34].

Job features Each job in PM100 is represented with a set of features, most of which are imported from the original dataset M100. After inspecting the original feature values, we notice that some information is duplicated or not meaningful enough. For instance, multiple fields are related to the names of the allocated nodes, so we aggregate them into a single feature called *nodes*. We also observe that some features contain the same information in different formats, e.g., *time_limit_str* and *time_limit*, thus we keep only one of such features. We further remove the features that have the same values across all the jobs, such as *power_flags*.

After this initial pre-processing, we modify some features to make the underlying information more explicit and easy to access. In particular, we engineer the features related to the resources. In the original data, each job record contains two fields listing the amount of requested and allocated resources, namely *tres_req_str* and *tres_alloc_str*. This data is structured as a comma-divided list of features (#nodes, #cores, #GPUs, and amount of RAM) and their values, e.g. *"nodes=1,cpus=32,gpu=1,mem=256G"*. We unpack the list of features and their values to individual features f_k, where $f \in \{num\_nodes, num\_cores, num\_gpus, mem\}$ and $k \in \{req, alloc\}$. Finally, we add the *power_consumption* feature, which is the list p_j of job power consumption values recorded during its execution, as computed in the previous paragraph.

Overall, each job has 32 features, which are documented in the GitHub repository of the dataset.

[4] `https://github.com/francescoantici/PM100-data`

3.3 Dataset Overview

In this section, we offer an overview of the PM100 dataset. First, we conduct an Exploratory Data Analysis (EDA) to understand the dataset and its underlying patterns, such as distributions, correlations, and relationships between certain features. As the description of the original M100 dataset [22] did not provide any such analysis, we first start with the jobs and then continue with their power consumption. This process provides insights into the nature of the data, which could be useful for prediction purposes and choosing the appropriate modeling techniques. We conclude the section with some examples of prediction tasks that can be performed with the dataset.

3.3.1 Job analysis

With job analysis, we investigate the classical workload characteristics, such as job submission date, exit state, duration, and allocated resources.

Submission date In Figure 7.2, we plot the distribution of the number of jobs (y-axis) submitted per day (x-axis), and its Kernel Density Estimate (KDE). The KDE (blue solid line) provides a probability density function estimation of the job submission per day, and it is plotted to spot patterns or seasonality more easily. Although we may expect a uniform distribution of job submission throughout the days, the figure shows that neither seasonality nor uniformity is present in the data. This is not unexpected. The PM100 dataset does not include all the jobs submitted to the system. Also, in a real HPC environment, jobs maybe submitted non-uniformly due to several reasons, like scheduled maintenance, holidays, and node failures and outages. Moreover, occasionally, it may be necessary to dedicate (a part of) the cluster to certain computations, as it happened in our case with COVID-19-related tasks.[5]

[5]Quoting an e-mail from the HPC system support to the users: "Due to a COVID-related urgent computing activity, most of M100 nodes will not be available for the standard production from Friday, November 6, 4 pm, to Monday, November 9, 4 pm. This will cause a significant increase in the waiting times and the impossibility to run jobs with more than 60 nodes".

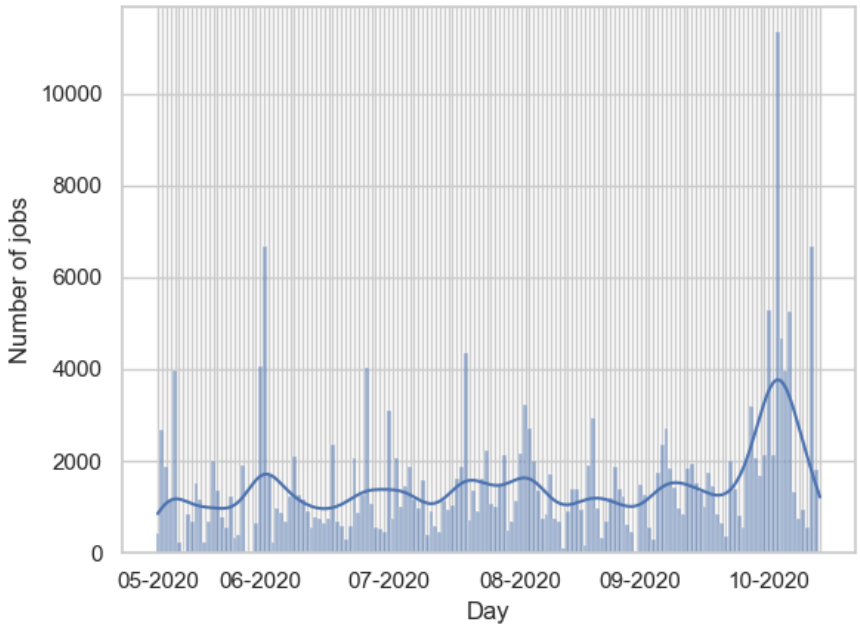

Figure 3.1: The distribution of the jobs throughout the days.

Exit state The exit state of a job represents its execution outcome. Since the jobs in our dataset are executed on a production machine, we expect to have a workload composed of mainly successfully completed jobs. This is because production machines are not used for tests, but only to execute stable jobs. We present in Figure 3.2 the distribution of the possible outcomes. As expected, the vast majority of the jobs (77%) are successfully *completed*. The second most frequent category is *failed*, with the 14%. These are the jobs that fail due to generic errors encountered during their execution, such as bugs in the code or errors in the job script. Some jobs are *cancelled* by their user during their execution due to reasons like discovery of bugs or errors in the code by checking intermediate results or of misconfigurations in batch scripts (e.g. run-time duration configuration is not enough). They represent 5% of the jobs in the dataset. With similar percentage (4%), there is the *timeout* category, referring to the jobs that exceeded the time limit set by the user or the system. Less than 1% of jobs run *out of memory* (*oom*) due to misconfiguration and underestimation of memory requirements. Similarly,

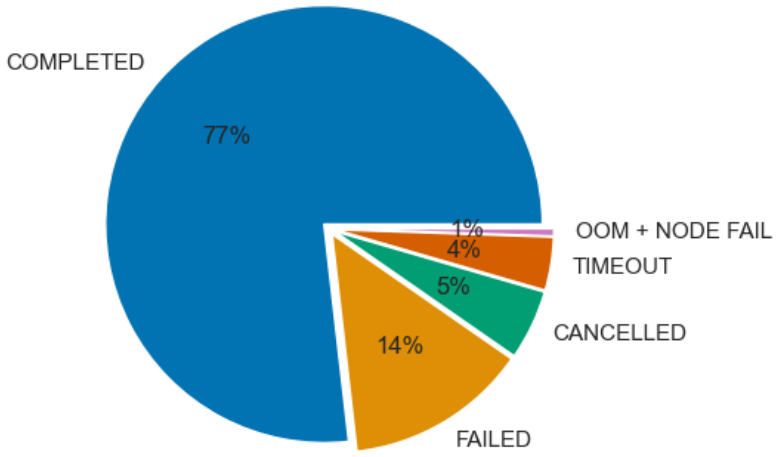

Figure 3.2: Distribution of job exit state.

less than 1% of the jobs fail due to *node fail*. This phenomenon is very rare in production machines, indeed, *oom* and *node fail* jobs combined account for around 1%. Even though the dataset does not reflect the whole system load, the expected unbalancing towards the successfully completed jobs is still present and yet there exist a significant amount of jobs that did not successfully complete ($\sim$ 50K).

Duration In Figure 3.3, we present the job duration distribution. In an HPC system, job duration may range from a few seconds to several days, depending on the allocated resources, the operations performed, and the system requirements (some systems may allow long executions while others not). The figure reveals that the majority of jobs has a duration of less than 100 minutes. Very few jobs run for more than a day, meaning that the jobs in our dataset are mainly short to medium length. This can be a characteristic of the users (not submitting jobs that require excessive computation) or due to a system setting (lengthy executions are not allowed). In the figure we also investigate how job duration is related to job

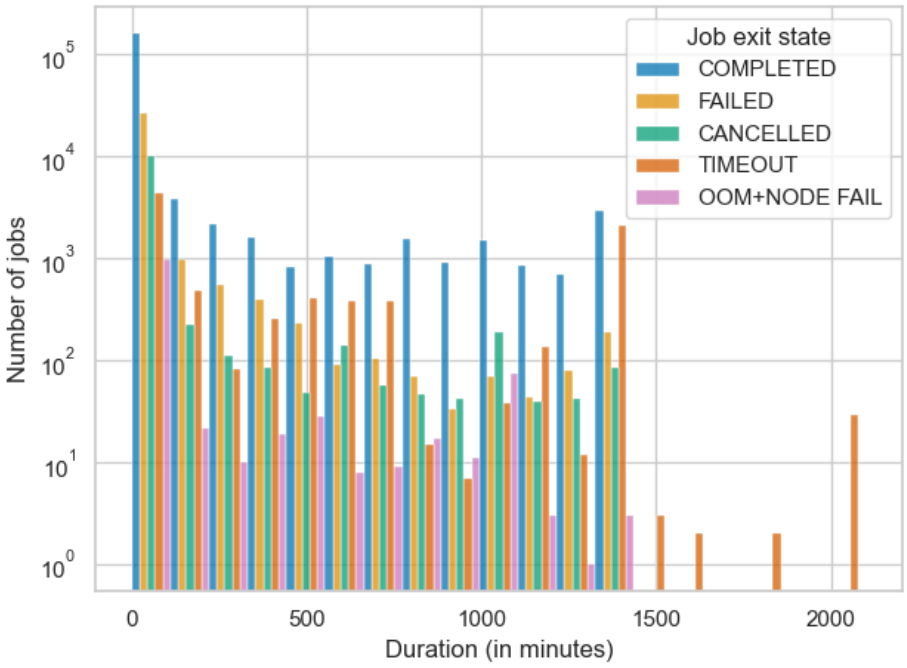

Figure 3.3: Distribution of job duration, divided by exit state.

exit state. We observe all possible outcomes with jobs running up to 1300 minutes. We observe further that the jobs that run for more than 1300 minutes all timeout. This is consistent with the information provided in the official documentation of Marconi100[6], stating that the system limits the job duration to 24 hours (1440 minutes) except some particular cases.

Allocated resources The dataset is dominated by the jobs using both cores and GPUs (91%). In Figures 3.4, 3.5, 3.6, and 3.7, we show the distribution of the amount of allocated nodes, cores, GPUs and RAM, respectively. From Figure 3.4, we can conclude that the majority of the jobs uses just one node for their execution. No job uses a significant portion ($\geq$ 20%) of the system nodes (980 in total), meaning that the jobs in our data mainly represent small-scale executions. This is also reflected in Figures 3.5, 3.6, and 3.7. Indeed, we notice an evident spike in these figures in correspondence to the amount of cores (128), GPUs (4),

[6]https://wiki.u-gov.it/confluence/pages/viewpage.action?pageId=336727645

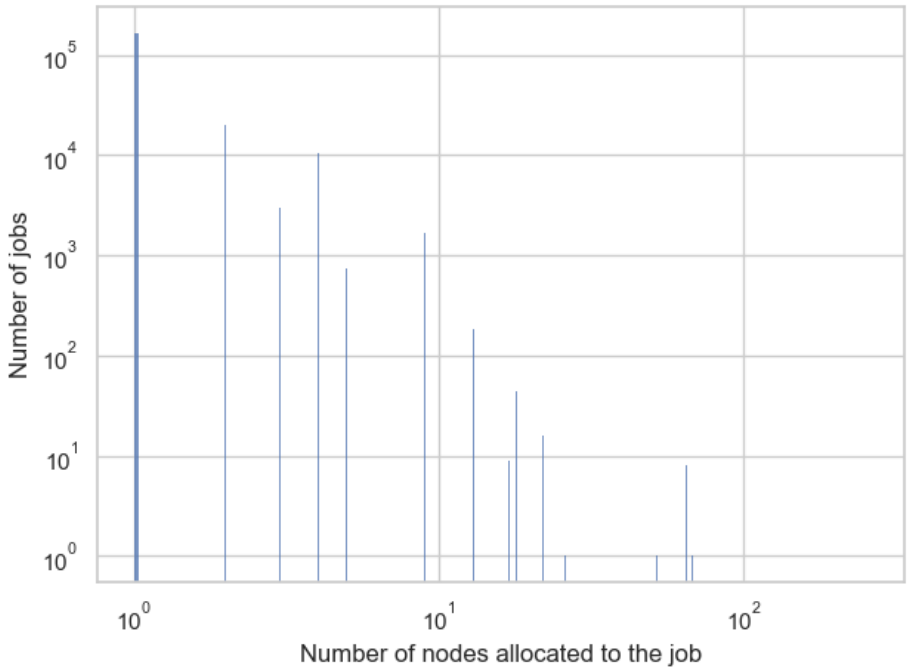

Figure 3.4: Distribution of the number of allocated nodes.

and RAM of a node (256 GB).

3.3.2 Job power consumption analysis

With job power consumption analysis, we investigate whether jobs differ in their power consumption values and trends, as well as the influence of GPU usage.

Power consumption As discussed in Section 3.2, *power_consumption* is the feature containing the list of power consumption values of the job throughout its execution. We start our analysis by plotting in Figure 3.8 the *power_consumption* values of 7 randomly chosen jobs whose resource usage is reported in Table 3.1. We observe that the jobs with similar features (single-node jobs 1 and 2) tend to behave similarly in their power consumption. As the number of allocated nodes, cores and GPUs increase, so does the power consumption values. We also observe that the consumption trend of the jobs vary. For instance, while jobs 6 and 7 fluctuate between higher and lower values, the others show a smoother behaviour.

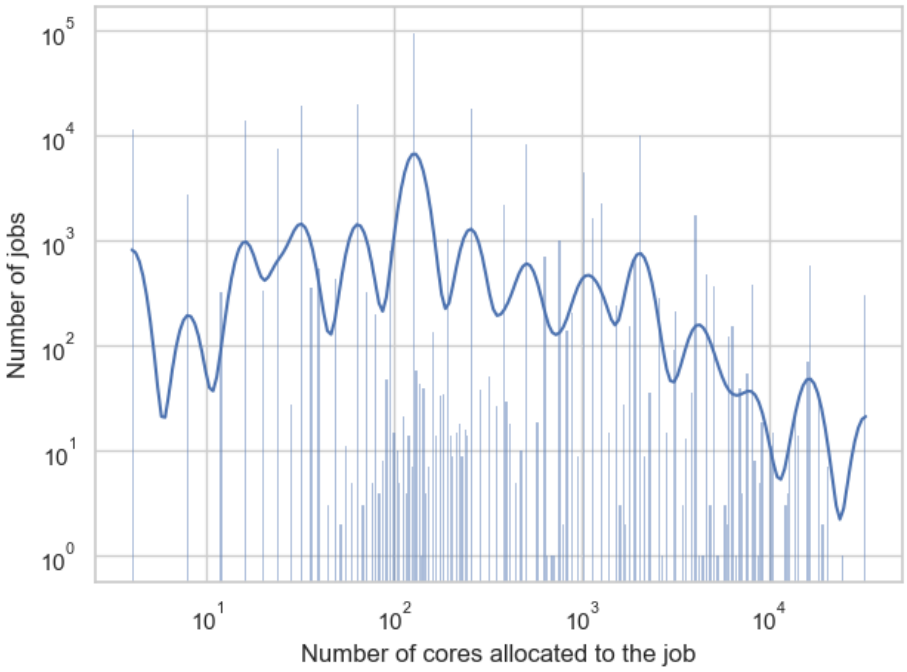

Figure 3.5: Distribution of the number of allocated cores.

We can conclude that our dataset contains a diverse set of jobs in terms of power usage.

GPU influence on power consumption Since the jobs are extracted from a heterogeneous machine, we want to explore how the GPU usage impacts power consumption. In this analysis, we consider the individual values of the jobs' instantaneous power consumption at each time point rather than their time distribution. Thus, we merge all the values of all the power consumption lists into a single set, independently of the job. Then, we use this data to plot Figures 3.9 and 3.10. In Figure 3.9, we show the distribution of the set of power consumption values by distinguishing between the jobs using only cores (9%) from those using also GPUs (91%). The figure shows that jobs using only cores tend to reach lower power values compared to the ones using also GPUs. This behavior is normal in heterogeneous systems, since GPUs have higher functional unit density and are usually characterized by a larger Thermal Design Power (TDP) than cores. In

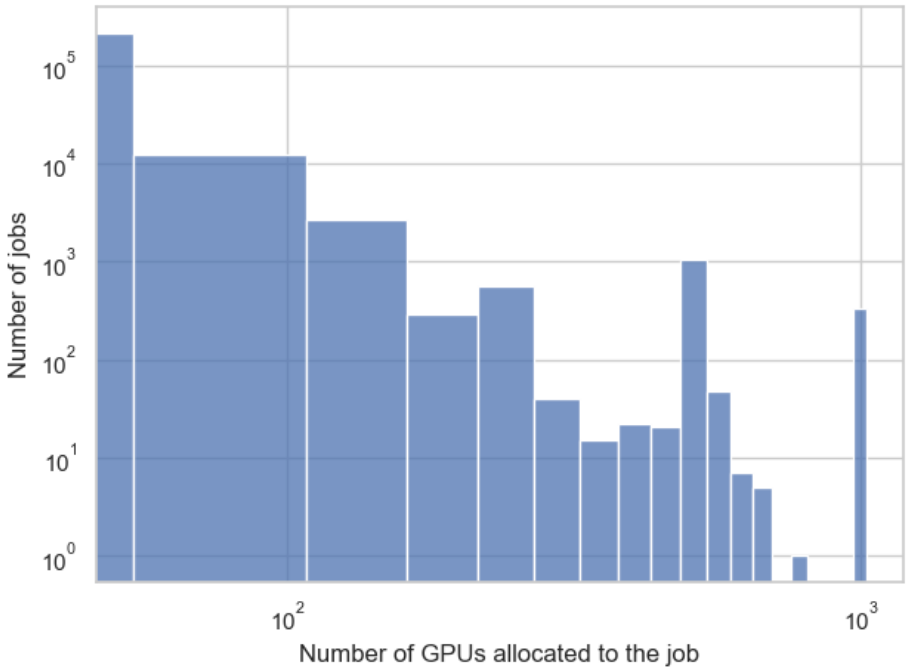

Figure 3.6: Distribution of the number of allocated GPUs.

Job #	#Nodes	#Cores	#GPUs	RAM
1	1	40	4	74
2	1	128	4	237
3	4	64	16	512
4	8	1024	32	1900
5	12	1536	48	2695
6	16	2048	64	3800
7	20	1280	80	4804

Table 3.1: Amount of allocated resources of jobs in Figure 3.8.

Figure 3.10, we focus on the single-node jobs and plot the distribution of the instantaneous power consumption values, again by separating the jobs using only cores from those using also GPUs. We see that independently of the number of allocated nodes, the range of power consumption values of jobs using also GPUs are higher than those using only cores, validating the findings of the analysis of the previous plot.

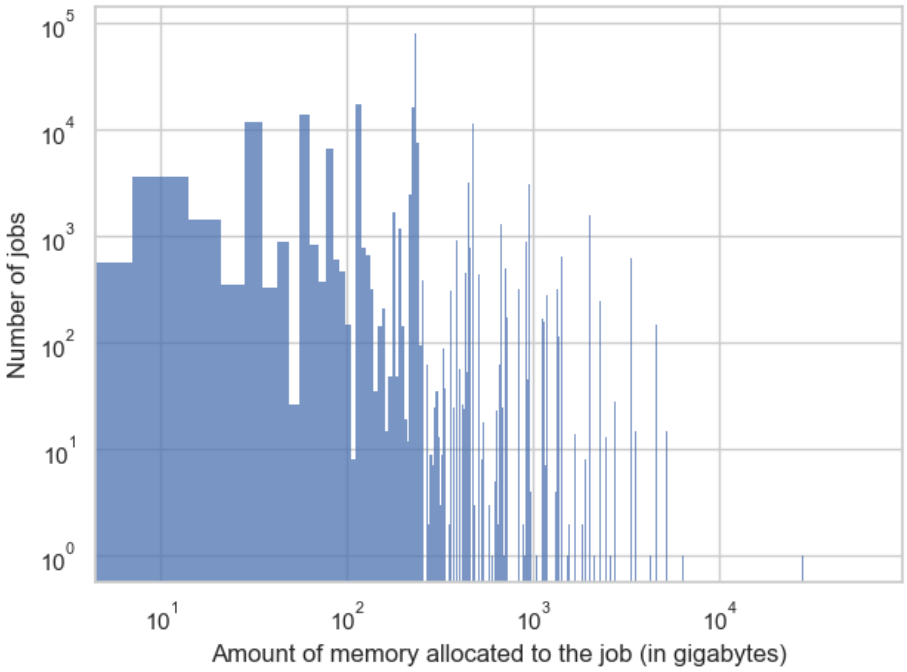

Figure 3.7: Distribution of the amount of allocated RAM.

3.3.3 Prediction Tasks

Our dataset enables performing various prediction tasks in HPC systems such as job duration prediction, job failure prediction, and job power consumption prediction.

Job duration prediction This concerns forecasting the execution duration of a job before its allocation in the system.

This information can be useful to develop dedicated workload management strategies, as shown in [35], aiming at improving system performance and achieving high quality of service (QoS) levels. Past work, such as [36, 10], explored the task by relying on machine learning techniques, while others like [35] tackled the problem by employing a data-driven heuristic algorithm in the scope of an online job dispatching problem. Such approaches can easily be performed on our dataset by exploiting the *run_time* feature of the jobs as target.

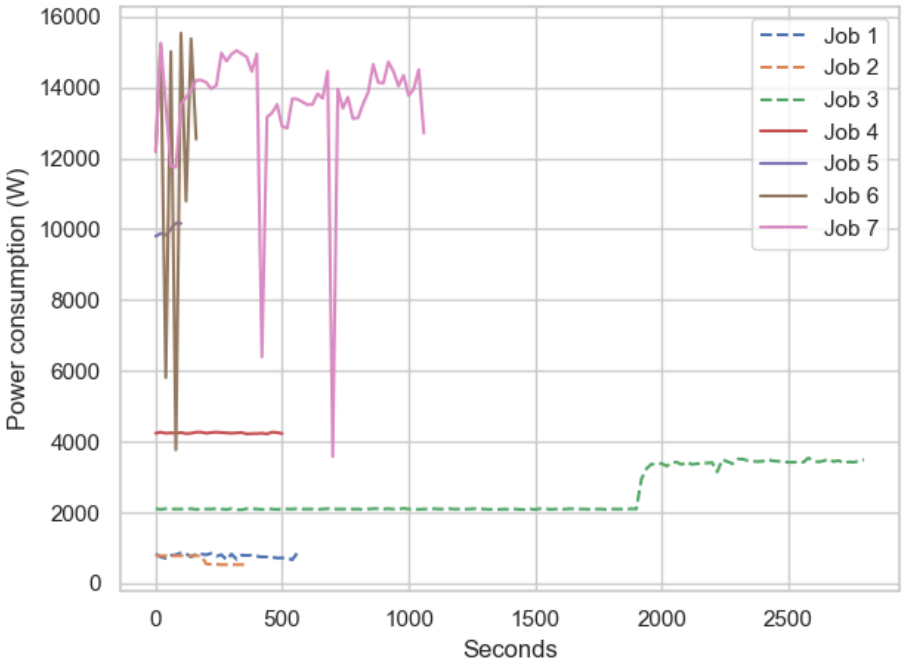

Figure 3.8: Job power consumption during execution.

Job failure prediction This is one of the hottest topics in the area of workload prediction, and it concerns forecasting possible failures during the execution of a job before its allocation in the system. Failing jobs unnecessarily occupy resources which could delay other jobs, adversely affecting the system performance, QoS and power consumption. Similar to the job duration prediction, forecasting failures a priori would allow to adopt ad-hoc workload management strategies, as shown in [37]. Several past work addressed the job failure prediction task. For instance, [9, 18, 37] relied on data-driven techniques aimed at predicting job failure by analysing workload features. As discussed in Section 3.3.1, our dataset contains the exit state of each job, which can be found in the *job_state* feature and used as a target in a classification task.

Job power consumption prediction This is about predicting the power consumption caused by the execution of a job on the system. It can assist the development of power-aware workload management techniques to optimize the system

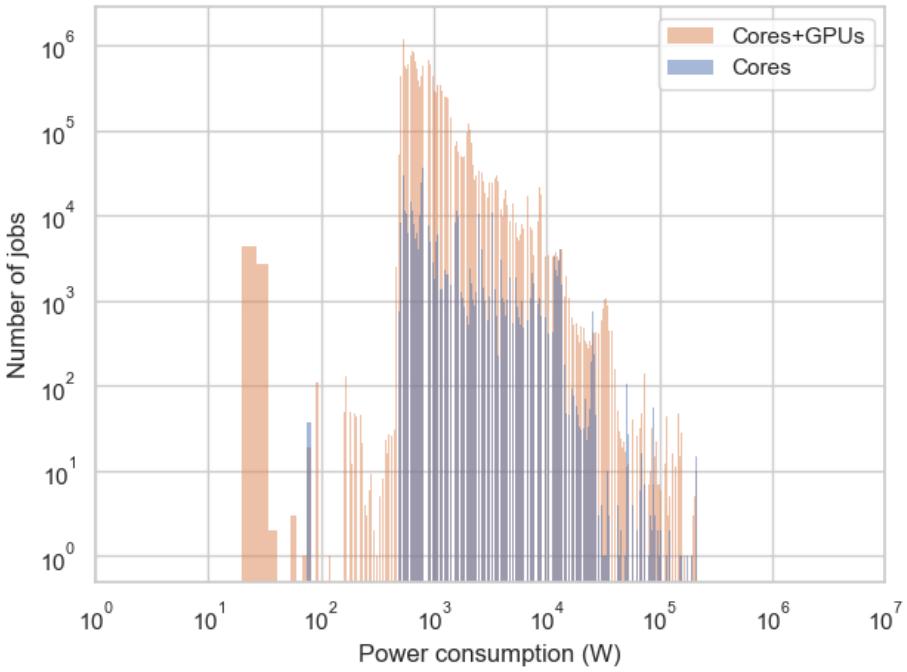

Figure 3.9: Distribution of job power consumption.

performance and power consumption, as shown in [2]. The prediction can be performed in different ways; for instance, by predicting the power consumption values of a job throughout its execution time, as done in [31]. Alternatively, it can be performed by predicting the average, or the maximum power consumption value, as done in [12, 15, 30]. The data in PM100 can be used for both purposes. As discussed in Section 3.2, each job has the *power_consumption* feature, which is a list of power consumption values computed at each timestamp intersecting the job execution. This feature can be used as it is for the first purpose, and its average and maximum values can easily be computed for the second.

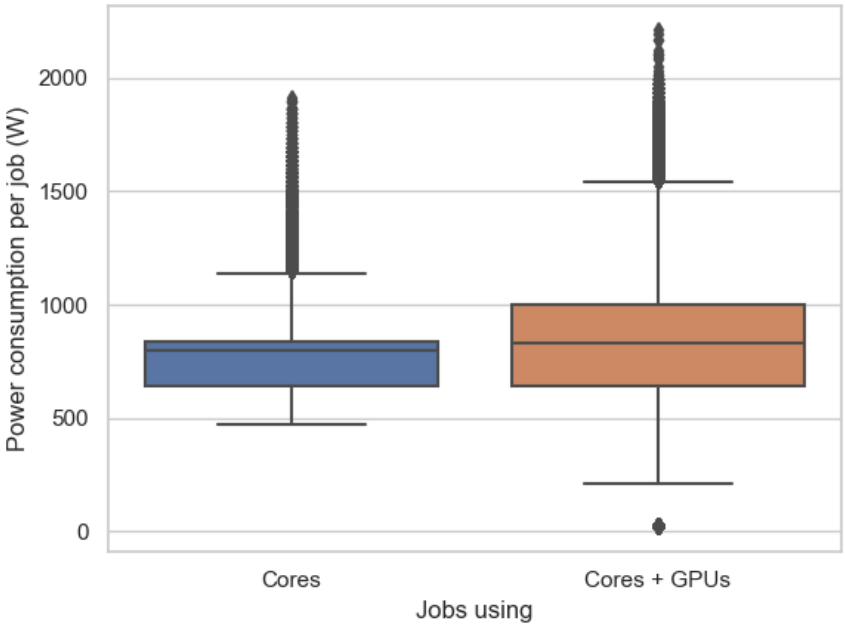

Figure 3.10: Power consumption of single-node jobs.

Chapter 4

F-DATA: A Fugaku Dataset for Holistic Job-centric Predictive Modelling

Due to technical difficulties in the data collection discussed in Chapter 3, public resources do not contain some important job execution characteristics, such as *power consumption* and *performance metrics* (e.g. *#flops*, *memory/compute-bound class* and *memory bandwidth*). Information like *user name* and *job name* reveal insights on users' job execution patterns, and hence are crucial for developing accurate predictive models. On the other hand, *power consumption* and *performance metrics* can help design workload management strategies to improve system throughput and energy efficiency, as shown in [6, 8, 38]. While for the *power consumption* we released PM100 (Chapter 3), no public resource contains *performance metrics*, making their prediction currently infeasible. Moreover, PM100 has some limitations. first, it only contains a subset of all the jobs executed on the system, namely the ones which ran exclusively on the resources; second, it contains the data of only 5 months, for a total of ∼200K job data.

To address the aforementioned problems, we present F-DATA, a workload dataset for job-centric predictive modelling in HPC systems. It contains the data of around 24 million job executions on Fugaku[1] Supercomputer, over more than

[1] https://www.fujitsu.com/global/about/innovation/fugaku/

three years of system usage, and is publicly available in Zenodo.[2] The sensitive job data appears both in anonymized and irreversibly encoded versions. The encoding is based on an NLP model and, as confirmed by our experimental study, it retains sensitive but useful job information for prediction purposes without violating privacy. F-DATA is the first public dataset containing i) the data of one of the most powerful supercomputers, ii) the job feature *performance metrics* along with several others (e.g. *power consumption*, *duration* and *exit state*), which are useful for a multitude of job-centric predictive modelling, and iii) an encoding of the sensitive data. By releasing F-DATA publicly, we empower HPC researchers and practitioners with a resource to foster the development of ML-based predictive models aimed at guaranteeing the sustainable development of HPC systems.

4.1 Dataset Creation

Fugaku relies on a proprietary operations management software,[3] which enables the recording and storage of job data in an instance of a PostgreSQL [39] database. We query the database via its interface and retrieve the data of the jobs executed on the system in each month between March 2021 and April 2024. F-DATA is thus composed of 38 smaller datasets (for a total of 28 GiB of data), named as YY-MM, each containing the data of the jobs executed during the month (MM) of the year (YY). We consider the data as of March 2021, when the system became available for public usage.[4]

Original job features Each job data extracted from the database contains features concerning the job submission, execution and completion. To the first category belongs the information available at job submission time, such as the job *user information* (e.g. *username* and *user id*), the *submission time* (i.e. when the user submits the job to the system) and the *requested resources* by the user (e.g. *# of cores, amount of memory, # of nodes*). When the job starts running, the execution features can be collected, such as the *start time* (i.e. when the job

[2] `https://doi.org/10.5281/zenodo.11467483`

[3] `https://www.fujitsu.com/global/about/resources/publications/technicalreview/2020-03/article10.html#cap-03`

[4] `https://www.fujitsu.com/global/about/innovation/fugaku/`

starts) and the *resources allocated* (i.e. actual amount of resources allocated to the job). At job completion, it is possible to access execution outcome characteristics, such as the *duration*, the *exit code*, the *power consumption* and the *performance counters*. The *exit code* is an integer value in the range [0-255] representing whether the job execution was successful or not. The *power consumption* is the power consumption of the resources allocated to the job during its execution, and it can be collected from different hardware components, like a node, CPU and RAM. The *performance counters* store the amount of hardware-related operations (e.g. # of memory read/write requests and # of floating point operations) performed by the job, which allow to gain insights on the job resource utilization.

We extend the job data by deriving new job features from the original features, and by encoding the sensitive data, which we explain next. The full list of 44 job features can be found in Zenodo.[2]

Derived features For each job, we derive the *exit state* and a series of *performance metrics* features, starting from the *exit code* and *performance counters* features. The *exit state* of a job is a label that describes the outcome of the execution, which can be successful or not. This feature is directly related to the job *exit code* which is 0 if a computation ends without any error, or an integer number in the range [1-255] in case of errors. Hence, we label the *exit state* of a job as *completed* if the *exit code* is 0, and as *failed* otherwise.

The *performance metrics* provide high-level information on the job resource utilization. Such information is fundamental to characterize a job execution, aiming to improve job and system level throughput and energy efficiency [6, 8, 40]. The job *performance metrics* we compute are *#flops*, *mbwidth*, *opint* and *pclass*. The $\#flops_j$ is the number of floating point operations per second performed by the job j and is computed as in Equation 4.1. In the equation, $perf2_j$ is the fixed amount of operations, while $perf3_j$ is the number of operations per 128-bit SVE (Scalable Vector Extension), which is multiplied by 4 since the A64FX of Fugaku is 512-bit SVE.The memory bandwidth $mbwidth_j$ is the amount of memory bytes moved per second during execution. In Equation 4.2, $perf4_j$ and $perf5_j$ are summed in order to obtain the total number of requests to the memory, as they represent the amount of memory read and write requests, respectively. Then, they

are multiplied by the size of the memory requests, (256 bytes of cache line size), to obtain the total amount of memory bytes moved. The cores of Fugaku nodes are grouped by 12 in Core Memory Groups (CMGs). Since the $perf4_j$ and $perf5_j$ values are generated by summing all the values collected by each core for the whole CMG, these values need to be divided by 12 to eliminate redundant information. Since both *#flops* and *mbwidth* are computed per second, we divide the values by the job duration ($duration_j$). The operational intensity *opint*, which is the amount of floating point operation per byte of the job execution, is computed as the ratio between *#flops* and *mbdwidth*.

$$\#flops_j = \frac{perf2_j + (perf3_j * 4)}{duration_j} \tag{4.1}$$

$$mbwidth_j = \frac{(perf4_j + perf5_j) * 256}{duration_j * 12} \tag{4.2}$$

Finally, we generate the performance class label *pclass*, which can be either *memory-bound* or *compute-bound*. They refer to the jobs whose performance is bound by the memory access rate or by the system's arithmetical performance, respectively. This feature can be generated as shown in [41], by computing the *ridge point* of the system, i.e. the ratio between the system peak attainable performance (maximum number of floating point operations per second) and memory bandwidth. We label all the jobs with *opint* greater than *ridge point* as *compute-bound*, and the others as *memory-bound*.

Anonymization of the sensitive data Publication of job data is possible upon effective protection of the sensitive data of the users and the system [20, 42]. Anonymization [43] is one of the most used techniques to protect sensitive data, and it consists of altering data in a way that prevents the original information to be identified. In F-DATA, the values requiring anonymization are the *user name*, *job name*, *job id* and *job environment*. Those values could indeed reveal the user identity, thus violating personal privacy, and disclose confidential details about the research or work being conducted, which could violate internal privacy policies on intellectual property or non-disclosure-agreements. For analysis purposes, such features are kept in the dataset, but they are transformed as follows. For each

feature, we take the list of all the values, without duplicates. As the data are originally ordered chronologically, the values list will be ordered by the time of first appearance in the dataset. The list index i is then used to generate the anonymization for a value of a feature f, as f_i (e.g. the first *user name* in the dataset becomes $username\_0$).

Encoding of the sensitive data As argued in [42], using anonymized data may compromise the effectiveness of prediction models. This is because the anonymization process may remove important information about the data, which is fundamental for the prediction task. In particular, the anonymized values of the sensitive data do not provide any information on the job nature, which is fundamental for job-centric predictive modelling. For instance, the *user name* and *job name* are fundamental to understand the user behavior and job execution patterns, while the *job environment* provides information on the software and hardware used to run the job. In the context of job-centric ML-based predictive modelling, we discovered that encoding job data with an NLP model improves the prediction performance of ML models, with respect to using the data in the standard integer format (as our experimental results in Chapter 3,6,7 and 8 will show). Thus, we encode the deanonymized version of the sensitive data with an NLP model and add it to the dataset as the *sensitive data encoding* feature.

We rely on the NLP model SBert [28], a state-of-the-art sentence embedding model. In the next chapters, we will show how we use SBert in our experiments for job level predictive modelling. We implement SBert leveraging the *sentence transformers*[5] library, with the pre-trained model *all-MiniLM-L6-v2*[6], since it has the best trade-off between prediction quality and speed [28].

The *sensitive data encoding* feature for a job data is generated by merging the deanonymized *user name*, *job name* and *job environment* features into a comma-separated string, and then encoding it with SBert. To this end, we do not consider the *job id* for the encoding, as it is an integer number and its original format does not provide any further information on the job nature with respect to the anonymized one. It is not possible to recreate the original values from the *sensitive*

[5] `https://www.sbert.net`

[6] `https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2`

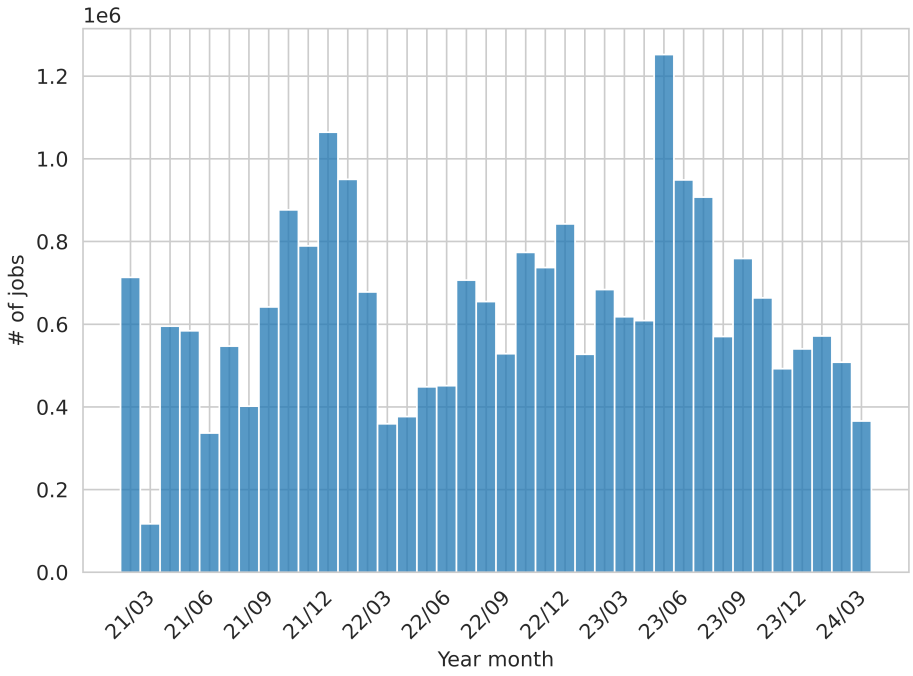

Figure 4.1: Distribution of jobs' submission by month

data encoding [44]. We thus safely include it in the dataset, aiming to foster the development of effective predictive models, without violating privacy concerns.

4.2 Dataset Overview

In this section, we provide an overview of F-DATA. First we analyze the job data, then we discuss the possible prediction tasks and the limitation of the dataset.

Job analysis We inspect the distribution of several job features. More specifically, in Figure 4.1, we show the distribution of the amount of job executions divided by month. We observe no clear patterns or seasonality. Except for April 2021, the amount of data is steadily over 300K jobs a month, with peaks in June 2023 (more than 1.2 million of data) and in January 2022 (around 1 million of data).

Jobs in HPC systems are executed on a set of nodes. In Figure 4.2, we show

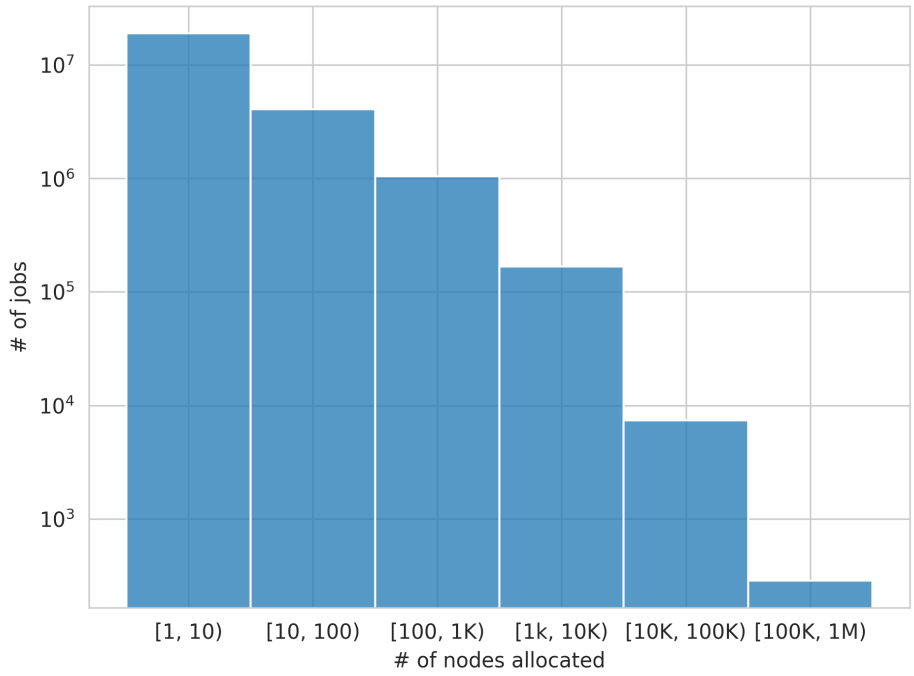

Figure 4.2: Distribution of jobs' *# of nodes allocated*

the amount of nodes allocated per job. We observe that the majority of the jobs (around 19 million) uses up to 10 nodes, meaning that the Fugaku workload is mainly composed of jobs using limited resources. However, the dataset contains also around 10K jobs executed on a significant portion of the system, using more than 10K nodes.

Depending on the complexity of the application, HPC jobs can run from a few seconds to several days. In Figure 4.3, we show that the dataset covers all values in the range of edge values, with a predominance of the short jobs (around 15 million jobs ran for less than an hour). We observe that jobs running for many days are rare (some hundreds), while thousands of jobs run for around one day.

Being a production system, the jobs submitted to Fugaku are expected to be mainly successful, as users are accounted for their job executions and failed jobs result in additional cost for the computational resources. This can be observed in the distribution of the job *exit state*, in Figure 4.4. The figure shows that the great majority of the jobs (more than 21 million) are completed. We can conclude

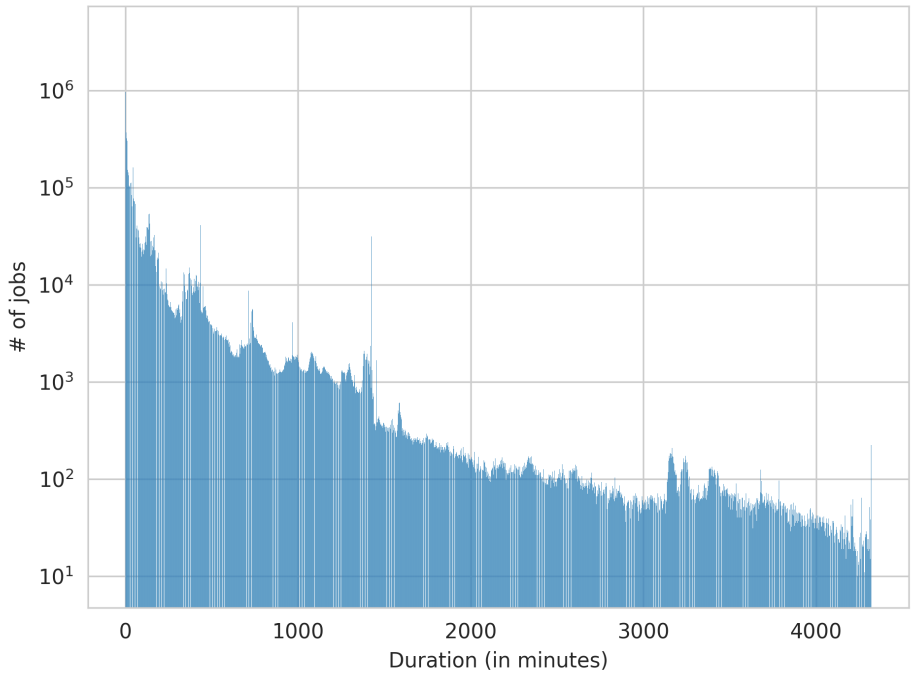

Figure 4.3: Distribution of jobs' *duration* in minutes

that the dataset is composed of mainly successful jobs, which are fundamental for the correct analysis of job behavior, as no failure alters or stops the execution. Yet, the dataset provides a significant amount of failed jobs (around 2.5 million), which can be used to study the job behavior and investigate the reasons for failed executions.

Figure 4.5 shows the distribution of the amount of job executions in each month, divided by their *pclass* values (i.e. *memory-bound* or *compute-bound*). We observe a high variability, due to the fact that the system workload and usage changes continuously. This is witnessed by the fact that the *pclass* distribution throughout the months is neither balanced nor stable; hence, the characteristics of the jobs running on the system are very different. While most of the jobs are *memory-bound* (almost 2/3), in some months (e.g. 21/07 and 23/01) the *compute-bound* jobs are the majority.

For job power prediction tasks, the job power consumption is usually normalized per node [11, 12]. In Figure 4.6, we show the minimum (*minpcon*), average

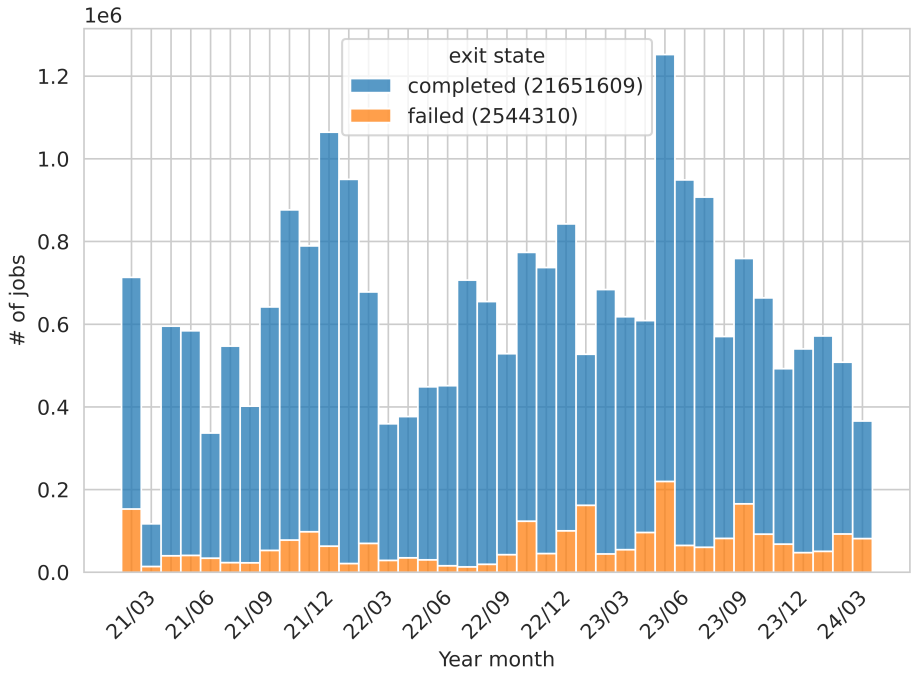

Figure 4.4: Distribution of jobs' *exit state* by month

(*avgpcon*) and maximum (*maxpcon*) job power consumption, normalized on the number of nodes allocated to the job. We observe that the *maxpcon* is shifted to the right (higher values of power consumption) with respect to the other two; the same holds for *avgpcon* with respect to *minpcon*. This is expected, the maximum power consumption is always greater than the average, and the minimum is always the lowest. Again, the dataset covers a wide range of power consumption values, from some watts to more than 200.

Prediction tasks The information included in the dataset allows for a multitude of job-centric prediction tasks. For instance, it allows for the prediction of job and system level power consumption, as shown in [12, 30]. By providing *minpcon*, *avgpcon* and *maxpcon*, jobs can be characterized in terms of full power consumption profile. Along power, it is also possible to explore the energy consumption prediction task, as each job data contains the *econ* feature giving the energy consumption in Watt-Hours. Such values can be used to estimate the carbon footprint

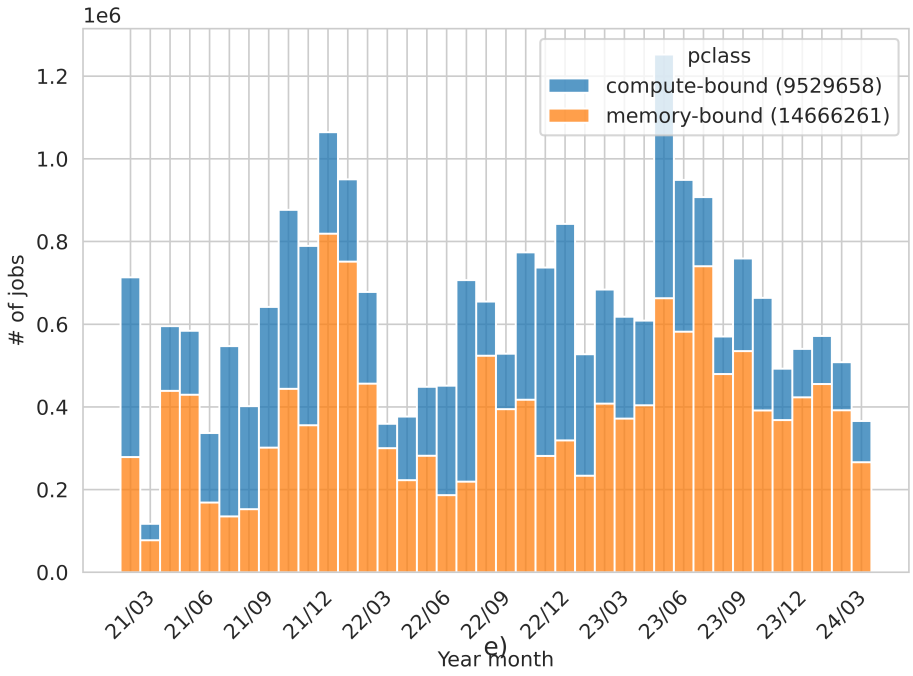

Figure 4.5: Distribution of jobs' *pclass* by month

of a job execution, which is fundamental to guide the sustainable development of HPC systems [45]. Moreover, all the *performance metrics* can be used as a target for a prediction model. Features like the *mbwidth*, *#flops* and *pclass* can be predicted to develop both co-scheduling techniques and specific hardware-software co-design techniques to improve system throughput and energy consumption, as shown in [6, 8] and [46, 47, 40], respectively. Furthermore, in Chapter 5, we will show how to use the *exit state* for job failure prediction tasks. Such an information might be useful to develop failure-aware scheduling strategies to minimize the system resource wastage [37]. Finally, it is possible also to predict values related to the time of the job execution, such as *duration* [48] or *ending time* [49]. These predictions can be used to develop better scheduling strategies accounting for job duration [35].

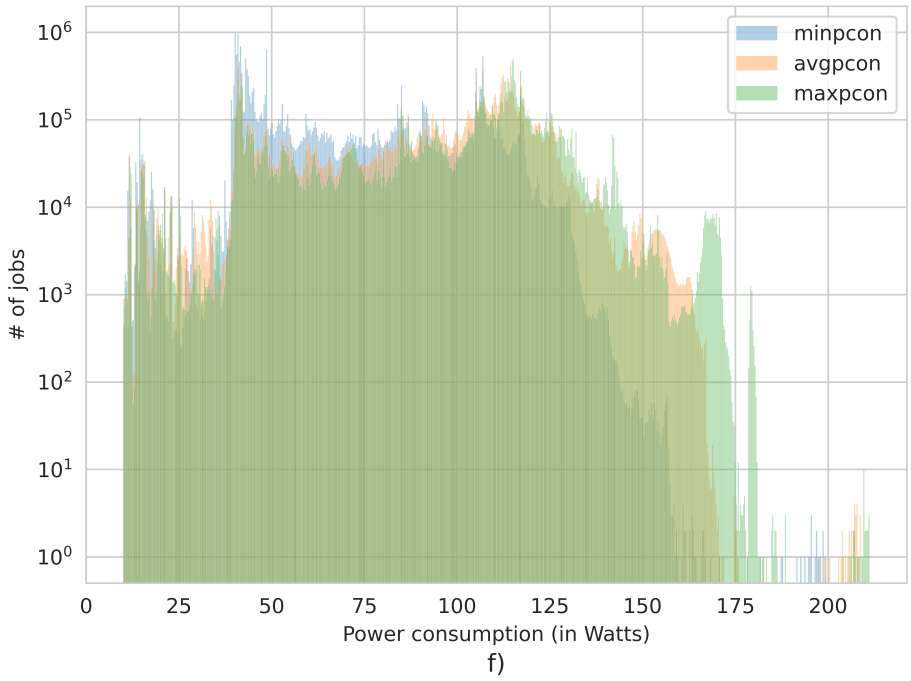

Figure 4.6: Distribution of jobs' *minpcon*, *avgpcon* and *maxpcon* per node

4.3 Experimental Study

We conduct an experimental study to i) showcase the value of the dataset in predictive modeling, and ii) demonstrate that the *sensitive data encoding* improves prediction performance with respect to the anonymized values. In the following sections, we first describe the experimental setup and then present our results.

4.3.1 Experimental Setup

We focus on the prediction of job *exit state*, *pclass*, *avgpcon*, and *duration* values, because they are often addressed in past work, as discussed in Section 4.2. Since the prediction target values are integers (*avgpcon* and *duration*) and labels (*exit state* and *pclass*), we face two regression and two classification tasks.

The experiments are run on a machine equipped with two AMD EPYC 7302 CPUs, 64 cores and 512 GB RAM, running Python 3.11.5 on Linux Fedora 37. The code necessary to repeat the experiments is available at F-DATA GitHub

repository.[7]

ML-based predictive models We need ML-based predictive models that can infer on new unseen data after being trained on historical data, and that are suitable for both regression and classification tasks. We opt for three widely used models, namely XGBoost (XG) [24], Random Forest (RF) [26], and k-nearest neighbors (KNN) [27].

In our experiments, we use the model implementations available in the Python *scikit-learn* library, [8] and instantiate them with the default settings provided by the library.

Evaluation metrics To evaluate the prediction performance of the models, we adopt the Mean Absolute Error (MAE) and accuracy, which are simple and widely used metrics for regression and classification tasks, respectively. The MAE is computed as the mean of all the absolute error on all the predictions, with respect to the ground truth values, and it is representative of the numerical error of the predictive model. The accuracy is a value between 0 and 1, computed as the ratio between the amount of correctly classified values and the size of the whole test set. To ease readability, we multiply it by 100 and express it in percentage form.

Job data preparation The ML models require the input job data to be encoded in a numerical format, i.e. a list of integers or floating points values. In our prediction tasks, we represent each job with its *job name*, *user name* and *job environment*. While these features are the most informative about job characteristics, they are also sensitive data and appear in the dataset as anonymized and as *sensitive data encoding* using Sbert. To adopt an NLP encoding approach (which we will use also for our experiments in the next chapters), we use directy *sensitive data encoding* (*sb_sensitive*), as well as *SBert* encoding of the comma-separated anonymized data (*sb_anon*)(e.g. "*jobname_0,username_0,jobenvironment_0*"). For the integer encoding (*int*), we assign an integer to all the sensitive feature values, as commonly done in ML tasks We note that this encoding is the same for the

[7] `https://github.com/francescoantici/F-DATA/`
[8] `https://scikit-learn.org/stable/`

original and anonymized values, since each unique value is mapped to an integer with the same order, regardless of the original format.

We train and test all the models with the *sb_sensitive*, *sb_anon*, and *int* encodings, for all the prediction tasks. This is done to see whether an NLP-model is able to extract more meaningful information about the job from the original deanonymized data and compare it to a standard (*int*) encoding.

Model training and testing To perform the prediction tasks, it is necessary to define a training set and a testing set. In an HPC context, the data of testing set need to always come after in time with respect to those of the training set, as otherwise the experiments would not be realistic, as discussed in Chapter 1.3.3. We consider as the training set the first 26 months of data, namely the jobs executed between 21/03 and 23/05, and we test on the data of the jobs executed between 23/06 and 24/04. The training set is composed of around 19 million job data, while the testing set has the remaining 6 million. We note that this is not an optimal setting for job-centric prediction, as the models are more accurate when they are updated frequently over time with recent data, as we will show in the following chapters. The setting is however sufficient to showcase the utility of the dataset and its features.

4.3.2 Experimental Results

We present our results in Figure 4.7, 4.8, 4.9 and 4.10. We observe in Figure 4.7, that for the *pclass* prediction, *sb_sensitive* obtains the best results with all the three models. In particular, it outperforms *sb_anon*, increasing the accuracy from a minimum of 4% (RF) to a maximum of 8% (KNN). The *int* and the *sb_anon* encodings obtain similar results with XG and RF, while *int* performs the worst with KNN. Concerning the *exit state* prediction (Figure 4.8), the *sb_sensitive* and *sb_anon* encodings obtain the same results with XG and RF, however, *sb_sensitive* is better with KNN. Here, *int* outperforms the other two with XG and KNN, while it is the worst with RF. The usefulness of the *sb_sensitive* is particularly evident in the *duration* and *avgpcon* prediction tasks, shown in Figure 4.9 and 4.10, respectively. In both cases, we observe a clear improvement in the prediction

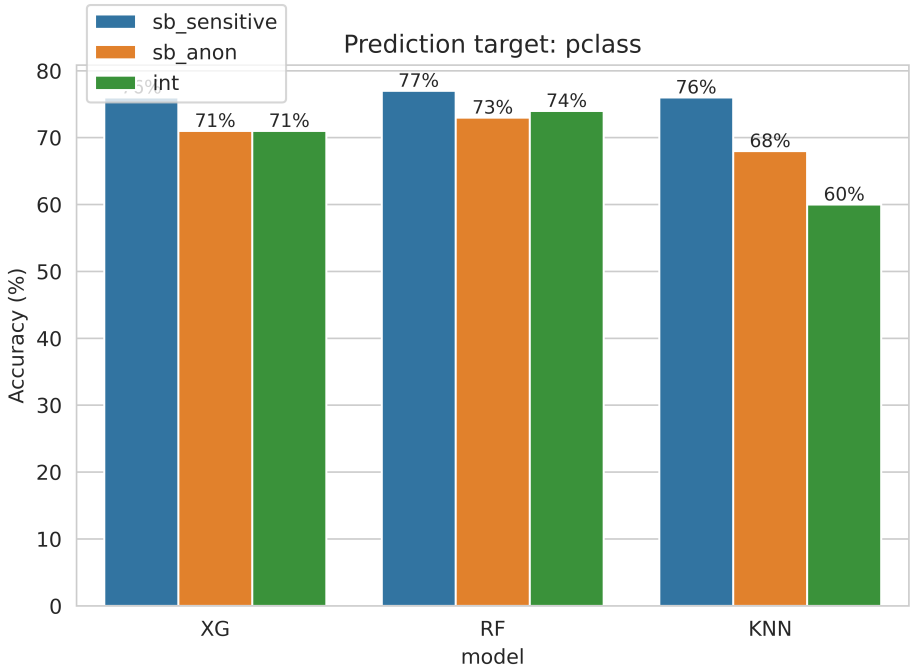

Figure 4.7: Model performance on the *pclass* prediction task. The higher the better.

performance with respect to the other encodings. The only exception is KNN in *avgpcon*, where all the encodings behave equally.

We conclude with these results that, i) the NLP encoding of the job data usually leads to better prediction than the standard *int* encoding, and ii) the *sensitive data encoding* retains more information about the jobs with respect to anonymized values and further improves the prediction performance, without violating data privacy.

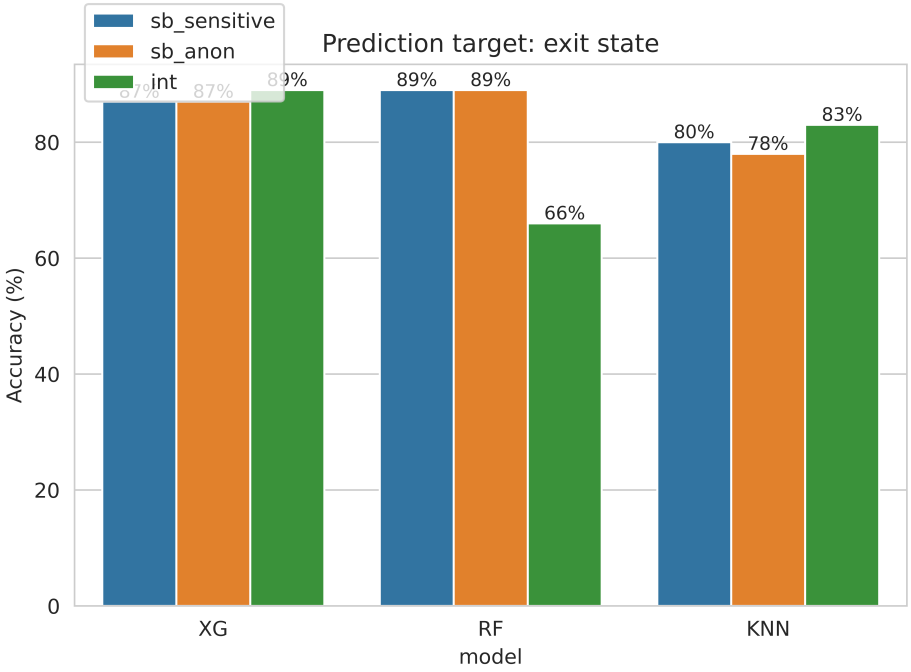

Figure 4.8: Model performance on the *exit state* prediction task. The higher the better.

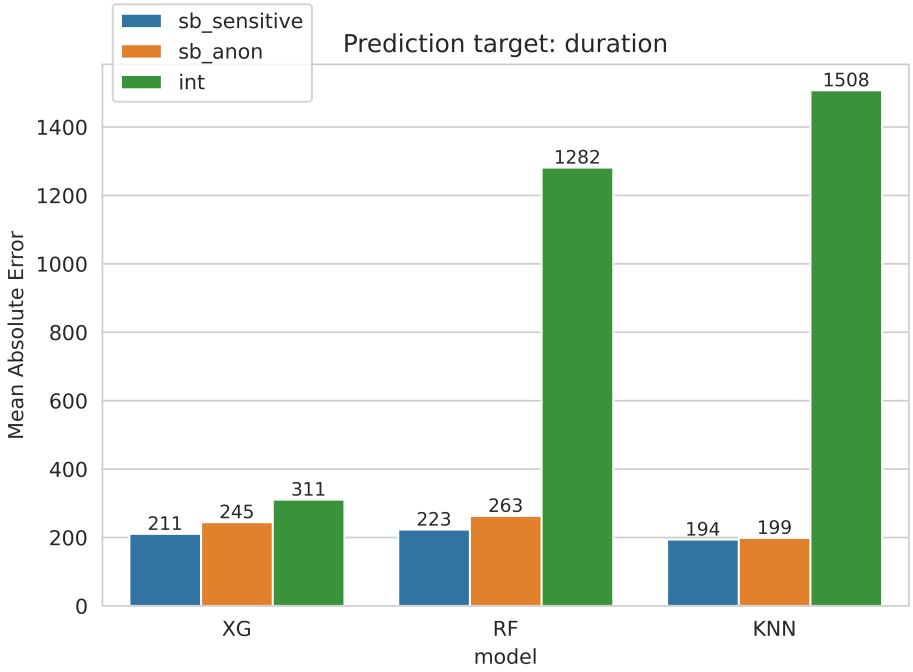

Figure 4.9: Model performance on the *duration* prediction task. The lower the better.

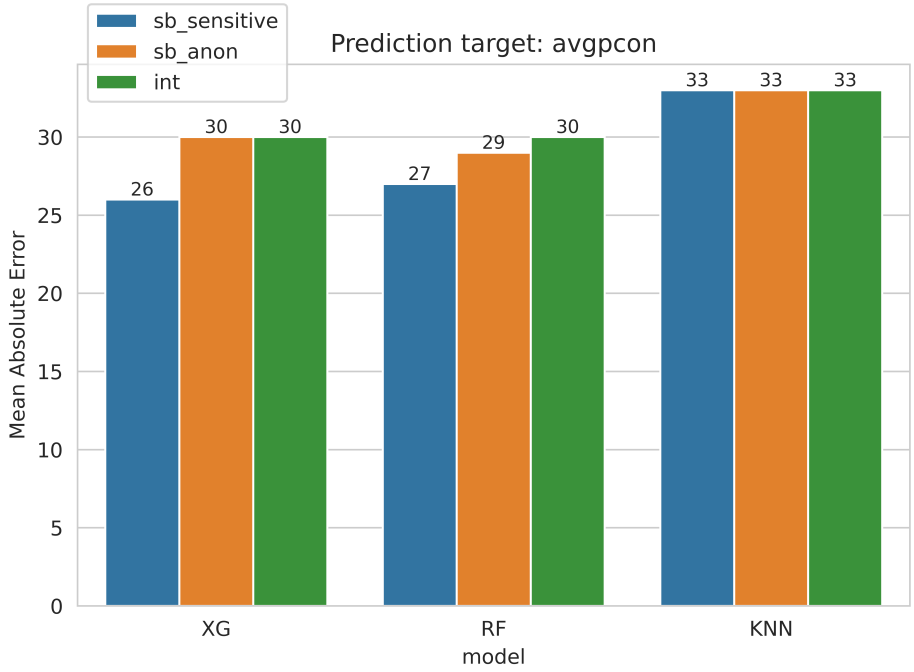

Figure 4.10: Model performance on the *avgpcon* prediction task. The lower the better.

Chapter 5

Job Failure

A critical problem of HPC systems is job failure. Whenever a job running on the system fails, it results in a waste of resources, time and power. The allocation of resources for a job is also responsible for delaying other jobs execution, reducing the efficiency of the system. One strategy to prevent this undesired behavior is to highlight the jobs that are most likely to fail, prior to their execution on the system. We distinguish between failures due to external factors, such as problems with the computing nodes, networking issues, workload manager downtime (*exogenous* failures) [50], and those due to internal reasons, such as wrongly configured submission scripts and software bugs (*endogenous* failures)[51]. We here focus on the latter category. Forecasting failures due to internal factors a priori would allow to adopt ad-hoc workload management strategies. For instance, we could enforce system power saving policies by exploiting job power prediction models, such as the one presented in [2]. Quantifying the power required by a job would allow us to estimate the system power consumption and reduce it by rescheduling jobs which are likely to fail to time spans when the system power consumption is lower. Furthermore, such prediction could be provided to the end-users to warn them on the risk of executing such, and hopefully prevent fallacious job executions.

We develop an ML-based classification approach to predict endogenous job failures. Our approach is applicable to data that can be collected from a production machine and leverages only the information available at job submission time (hence does not require any instrumentation of the users' code nor any change to standard

workload submission workflow). This information might have different formats, and text is among them. To extract more meaningful job information from such textual data, we employ NLP tools and improve the classification performance of the ML models. To the best of our knowledge, this is the first work that exploits an NLP method to represent jobs during classification. Contrary to the majority of the past studies which work on random splits of historical data, the proposed methodology can be deployed in an *online* context where jobs are continuously submitted by users to a real production system. We demonstrate the validity of our approach on a dataset extracted from M100, presented in Section 3.1.

5.1 Related Work

In this paper, we restrict the related work to the study of failures in large-scale systems at job/application level. In [52], the authors analysed workload traces in a grid, showing the correlations between failure characteristics and performance metrics. Works like [53, 54] tackled application failure prediction in cloud computing by using recurrent neural networks on resource usage data and performance logs, extracted from Google cluster workload traces. Also in [55] the authors relied on the resource usage data of a job to predict its failure, but in the scope of an HPC center.

These approaches do not take into account the human factors (error in the code, the submission, etc.), which are responsible for many job failures [18]. Therefore, the trend is shifting towards the use of data collected from a workload manager to predict failure using job features, as done in [18, 37, 9]. In [9], the authors use a decision tree algorithm to predict job failure on two HPC workloads. In [37], they survey several ML techniques to perform the same task on a Google cluster workload trace and other two HPC workloads. A similar approach is reported in [18] on another workload; in addition, they use NLP techniques to assign similar names to similar jobs executed by the same user. All this past work, which are most related to ours, evaluate their approach on random splits of data, which is not realistic because testing could be done on data which is chronologically placed in between the training data traces. Our work differs in two ways: (i) we propose to use NLP techniques to represent jobs for classification via all the job information

available at job submission time, (ii) our approach can be deployed in a more realistic *online* context and is thus evaluated on a streaming data, by continuously retraining the classification model on recent (past) data, and testing it on (future) data which has not been seen.

5.2 Methodology

In this section, we describe our methodology to job failure prediction. The workflow can be divided into two phases: (i) data preparation and (ii) job failure prediction.

5.2.1 Data preparation

To train and test our classifiers, we consider a part of the M100 dataset[1] and use only the data collected between May 2020 and October 2020. The reason is that this is the only period where the dataset contains information on the requested resources and the job EC, which we need for our prediction task. We collect the job data in a data frame and then prepare it for model training and inference.

Feature selection In order to describe the characteristics of a job in a classification task, we need to associate it with certain features. We focus only on job submit-time features, as we want to compute a prediction before job allocation. The features available in the dataset are listed in Table 6.1 along with their description. Jobs submitted by the same user and close in time tend to be similar because in a production HPC, users often submit jobs in batches referring to similar experiments and jobs in the same batch tend to have similar names and command. Thus, we believe that all these features are useful for our purposes. We note that user name and similar private data are omitted in the public dataset. However, CINECA granted us access under a non-disclosure agreement.

Job exit state labels For the training data, we need to assign a label to each job, indicating whether it has failed or not. In M100, one feature related to the

[1] `https://doi.org/10.5281/zenodo.7588815`

Name	Description	Type
Name	Job name assigned by the user	String
Command	Command executed to submit the job	String
Account	Account to be charged for job execution	String
User id	ID of the user submitting the job	Integer
Dependency	Jobs to wait for completion before execution	String
Group id	Group of the user submitting the job	Integer
Requested nodes	Specific nodes requested	List[String]
Num tasks per socket	Number of tasks to invoke on each socket	Integer
Partition	Name of the assigned partition	String
Time limit	Maximum allowed run time in minutes or infinite	Integer
Qos	requested quality of service	String
Num cpu	Number of rquested CPUs	Integer
Num nodes	Number of requested nodes	Integer
Num gpus	Number of requested GPUs	Integer
Submit time	Time of job submission	Timestamp

Table 5.1: Job features description.

execution outcome is the job Exit State (ES) label, which is assigned to each job by Slurm as an interpretation of the job's Exit Code (EC). This code is formed by a pair of numbers; we consider only the first one, which refers to a system response that reports success, failure, or the reason of an unexpected result from job launch. An EC value of 0 means successful completion, while any EC $\neq 0$ represents an error encountered during execution. Table 5.2 describes the ES labels assigned to the jobs in our dataset, along with their distribution. As seen in the table, the dataset is highly unbalanced. This is not surprising, because in a real production machine the failures should be minimized to guarantee correct functioning of the system. Nevertheless, the percentage of the jobs not successfully completed is more than 20% (more than 1 out of 6 million jobs), representing an important threat to the system performance.

According to the Slurm official documentation, the labels assigned by the scheduler may not be coherent with the actual EC, due to lack of proper synchronization between the signal emitted by the job exit and the data collected in the database. We therefore inspect the data and identify any possible discrepancy, e.g., a job with an ES label *completed* and an EC $\neq 0$. Our analysis reveals that more than

Name	Description	%
Completed	Job completed execution without errors	79%
Failed	Job terminated for an unknown reason	10%
Cancelled	Job did not start execution due to an error in submission	8%
Timeout	Job terminated due to reaching the time limit	2%
Out of memory	Job terminated due to more memory access than allocated	0.6%
Preempted	A higher-priority job delayed the job execution	0.1%
Node fail	Job terminated due to a failure in an allocated node	0.01%

Table 5.2: Job ES labels and their distribution in the M100 dataset.

70K jobs labelled differently than *completed* have an EC value of 0. This is confirmed by the difference between the percentage of the completed jobs (83%) and the jobs having an EC of 0 (89%). As a consequence, we discard the original labels and create new labels based on the job EC.

Despite the discrepancy between the original ES labels and EC, the highly unbalanced nature of the entire dataset (see Section 3.1) is observed also in the subset data we use in this study. In particular, while the percentage of jobs with EC = 1 is 9%, the percentage with EC > 1 is 2%. We therefore group all types of failures under the same category; discriminating among different fail modes is outside the scope of this work. Moreover, we are interested in failure caused by the workload itself, so we remove from the dataset all the jobs originally labelled as *cancelled* (failure due to user) and *node fail* (failure due to hardware). Eventually, we re-label the remaining data according to the following policy: for every job, we assign an ES label of *completed* if its EC is 0, *failed* otherwise. The final dataset after the relabelling is composed of 924,252 (89%) completed and 113,027 (11%) failed jobs. The distribution of the labels, throughout the months, is reported in Figure 5.1. We can observe that imbalance between the two classes of jobs appears in all the months, while the ratio between them changes considerably, showing that the workload is highly variable across time.

5.2.2 Online Predictive Algorithm

Feature encoding In order to compute a prediction for a job, we need to represent it suitably to feed into the classification models presented in Chapter 2.3. We

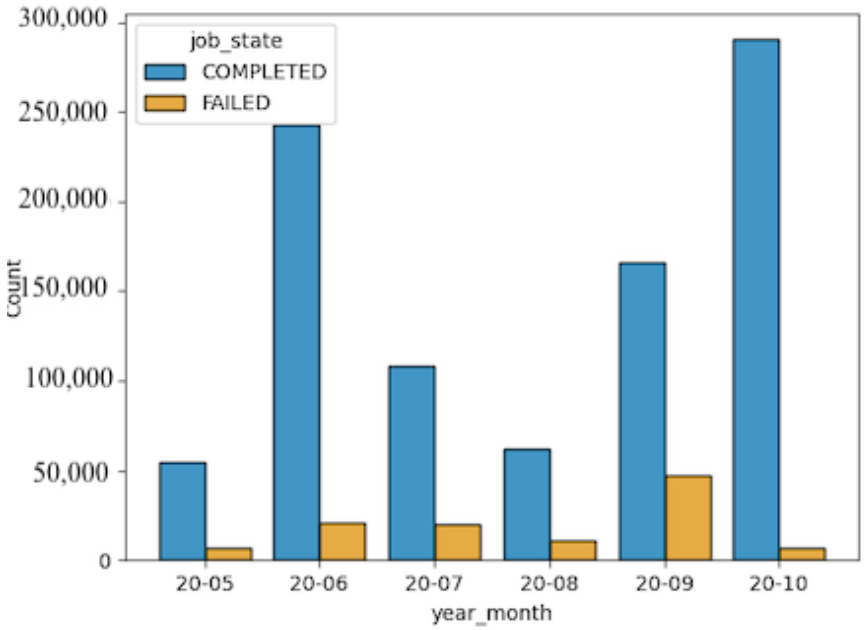

Figure 5.1: Job ES label distribution throughout the months in the final dataset.

achieve that by relying on job feature values, and we propose two different ways to encode them. In the first (INT), we assign an integer to the values which are not numerical, i.e. *name, command, account, dependency, requested nodes, partition, qos, submit time*, while setting all the missing values in the other fields (*num tasks per socket, time limit*) to a default value of 0. In the second encoding (SB), we first concatenate all the feature values into a comma divided string, e.g. *job1, run_job1.sh, [1, 10], 2020-10-01 15:30:00, account_1, partition_1, 0, normal, 4, 100, 2, etc.* Then we encode the string with SBERT, obtaining a 384-dimensional floating-point array.

We believe that with SBERT we can extract more fine-grained insights about job features expressed in natural language (e.g. *name, command, account*). This is because SBERT is designed to result in similar encodings with sequences with semantically similar contents. As we discussed in Section 5.2.1, jobs with similar names and command could belong to the same submission batch running similar operations. Therefore, features like *submit_time, name, account, command* could

reveal important patterns on the nature of the job and its workload. This is hard to recognize with the INT encoding, since similar natural language values will be mapped to different integer values, while they would have similar representation in SB, due to semantic similarity.

Classifier training and testing In our prediction task, it would not be realistic to do inference on a job by learning from the data of the future jobs submitted at a later time. We thus create the training and test sets by considering the timeline of the job data, keeping in the training set the data that comes before in chronological order the data of the test set.

We identify two settings in which a classifier can be trained and tested on a dataset. The first is the *offline* setting, where we consider the job data as a whole, train the model once on one portion of it, and test it using the data of the other portion in chronological order. To do this, we sort the jobs based on their submission time, split them into two, use the first split preceding in time as the training set, and the other as the test set.

The second setting, which we refer to as *online*, is more suitable to our context. We treat the job data as live and streaming in time, retrain the model periodically on a fixed size of recent data, and test it on future data that comes later (but near) in time. As we discussed in Section 5.2.1, the workload of an HPC system can be very similar in a short period, while may vary in the long term. As our experimental results confirm, a model trained once on data which slowly gets further in time to the test data could classify poorly compared to a model which is retrained continuously on data closer in time to the test data.

In the *online* setting, we use the time information provided by the *submit_time, start_time and end_time* features in order to simulate job submission and execution on a machine, and add the *day* feature as the submission date by extracting it from *submit_time*. We consider as the first training set all the jobs that were submitted in the first α days and not finished after the date of the first test set. Starting from the submission time of the first job not present in the first training set, we divide the data in batches in chronological order, where each batch contains the jobs submitted in the next β days. We then iterate over each batch, considering it as a new test set. At every iteration, the training set is updated with the data of

the last α days and the supervised models are retrained. With the unsupervised models, no actual re-training takes place, however the training set is extended for each new job in the test set with the jobs that finished before the submission time of the new job (with negligible overhead).

5.3 Experimental Study

In this section, we report our experimental study and discuss our results.

5.3.1 Experimental setting

All the experiments are conducted on a node of a small cluster equipped with two Marvell TX2 CPUs with 32 cores and 256 GB of RAM. No accelerator, such as GPU, is used in the experiments.

The classification algorithms are implemented with *scikit-learn* Python library. The sequence encoder model is provided by the *sentence transformers* library[2], while the weights for SBERT are pulled from huggingface.[3] We use the pre-trained model *all-MiniLM-L6-v2*[4], since it is the best trade-off between prediction performance and speed [28]. All the models are instantiated with the default setting provided by the library.

We set the hyperparameters as follows after an initial empirical evaluation. We use MWD of order $p = 2$ and set $k = 5$ in the KNN algorithm. As discussed in Section 5.2, the testing period strictly follows the training period. For the offline setting, we take the first 70% of the data as the training set and the remaining 30% as the test set. For the online, we fix the training interval α to 30 days, based on the trade-off between prediction performance and training/inference time. The time-span of data in each test set is $\beta = 1$ day. The implementation is available in a GitHub repository.[5]

The results are reported in Tables 5.3 and 5.4, where we distinguish between the job feature encodings (INT and SB), the supervised algorithms (DT, LR, RF), and

[2] `https://www.sbert.net`
[3] `https://huggingface.co`
[4] `https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2`
[5] `https://github.com/francescoantici/job-failure-predictor/`

the distance metrics of the KNN algorithm (CD and MWD). Each classification algorithm is evaluated using the two feature encodings and are compared with two simple baselines, namely majority and random. Both baselines ignore the input feature values. The majority returns the most frequent label observed in the training data, while the random generates predictions uniformly from the list of unique labels, so each class has equal probability. The results reported in Table 5.4 are averaged over 5 months between June 2020 and October 2020.

5.3.2 Results

We evaluate our models with metrics typically used for classification tasks, namely f1, precision and recall. Table 5.3 reports the results of the offline setting. The model that gives the best results overall is INT+RF. It achieves a f1 score of 71% and is very good at classifying the completed jobs, as the f1 score computed over such jobs is 98%. The prediction of the failures is somewhat harder, with a f1 score of 43%.

Overall, we observe that the supervised techniques perform better, but all the models struggle with the classification of the failed jobs, as most of them (with the exception of INT+DT) have lower recall than the random baseline in the failed class. Conversely, the classification of completed jobs is much easier, with the precision being $\geq 96\%$; this is probably due to the imbalance in the dataset (completed jobs are more abundant). This is compounded with the proportion between the completed and failed jobs varying significantly across different periods, as shown in Figure 5.1. Thus, with the offline setting, the model has a high risk of overfitting on the completed job examples (being more numerous) and of spectacularly underperforming when tested on jobs that fail.

This behaviour can be mitigated by retraining the models to adapt them to the workload and the class distribution shift over time. Indeed, Table 5.4 shows the results of the online setting, with notable improvements in the classification of the failed jobs. The SB encoding coupled with the clustering classifier using the Minkowski distance (SB+MWD) yields the best results overall, suggesting that properly extracting meaningful job information from textual data is beneficial. In terms of the f1 score, SB+MWD reaches 70%, outperforming all the supervised

models, which arrive to a maximum of 64% with SB+RF and INT+RF.

The classification of the completed jobs is good for all the models and their f1 scores are always above the 80%; the clustering methods have the highest precision (87%), while SB+RF has better recall (91% with respect to 83%). There is some minor drop in performance in the completed class compared to the offline setting (less overfitting), but the results are still solid. In the failed class, the clustering methods (SB+CD, SB+MWD) obtain a f1 score of 54% outperforming all the supervised algorithms. We observe a significant improvement with respect to the offline setting. Indeed, the best f1 score obtained over failed jobs in the offline setting (INT+RF) is increased by 20% by the best model in the online setting (SB+MWD and SB+CD); clearly, retraining the models helps to classify job failures.

As can be observed in both tables, the use of the SB encoding has a marginal impact with the supervised models, while the training time increases significantly in the online context (e.g., the training time of INT+RF is 25 seconds, while SB+RF requires 922 seconds). The increase in training time is not surprising, as the extraction of the text features through NLP involves the usage of a computationally hungry DN. We note, however, that the inference time remains very small and this is the operation that needs to be performed in real time without affecting the machine's normal workload (the retraining can be scheduled in less busy periods). On the other hand, in the case of the unsupervised models, SB improves the performance by 1-2% in almost every metric while no training time is incurred and the inference time always remains under a second. As we discussed in Section 5.2, with these models retraining is simply extending the training set (with negligible overhead) and classifying a new job requires a simple inference step (i.e., the new job is compared with those in the training set, projected in the feature space).

Model	T $F1_m$	T $Prec_m$	T Rec_m	C F1	C Prec	C Rec	F F1	F Prec	F Rec
Supervised									
INT+DT	0.30	0.50	0.48	0.55	0.96	0.38	0.06	0.03	**0.57**
INT+LR	0.54	0.62	0.53	**0.98**	0.97	0.99	0.10	0.26	0.06
INT+RF	**0.71**	**0.72**	**0.69**	**0.98**	**0.98**	0.98	**0.43**	**0.47**	0.39
SB+DT	0.38	0.50	0.50	0.70	0.97	0.55	0.06	0.03	0.45
SB+LR	0.66	0.70	0.63	**0.98**	**0.98**	0.99	0.34	0.43	0.28
SB+RF	0.55	0.54	0.61	0.95	0.97	0.92	0.16	0.11	0.30
Unsupervised									
INT+CD	0.52	0.52	0.58	0.92	0.97	0.87	0.11	0.07	0.28
INT+MWD	0.39	0.50	0.50	0.72	0.97	0.58	0.06	0.03	0.42
SB+CD	0.42	0.50	0.52	0.76	0.97	0.63	0.07	0.04	0.42
SB+MWD	0.42	0.50	0.52	0.76	0.97	0.63	0.07	0.04	0.42
Majority	0.49	0.50	0.48	**0.98**	0.97	**1.00**	0.00	0.00	0.00
Random	0.36	0.50	0.50	0.66	0.97	0.50	0.06	0.03	0.49

Table 5.3: Results in the offline setting, for both classes (T), completed class (C) and failed class (F) using precision (Prec), f1 and recall (Rec). In (T), we consider the macro averaged metrics ($F1_m$, $Prec_m$, Rec_m). The model name is composed of the feature encoding and the classification algorithm/distance metric. Best results are highlighted in bold.

Model	T $F1_m$	T $Prec_m$	T Rec_m	C F1	C Prec	C Rec	F F1	F Prec	F Rec	Time
Supervised										
INT+DT	0.60	0.64	0.63	0.80	0.84	0.79	0.41	0.44	0.46	1.27 + 0.005
INT+LR	0.46	0.53	0.51	0.85	0.79	0.95	0.06	0.26	0.06	78 + 0.3
INT+RF	0.64	0.69	0.64	0.84	0.84	0.87	0.43	0.54	0.41	25 + 0.12
SB+DT	0.61	0.62	0.63	0.80	0.84	0.78	0.41	0.39	0.47	455 + 0.09
SB+LR	0.60	0.66	0.60	0.85	0.82	0.89	0.34	0.50	0.30	84 + 0.4
SB+RF	0.64	0.70	0.63	0.86	0.83	0.91	0.41	**0.57**	0.35	922 + 0.4
Unsupervised										
INT+CD	0.68	0.69	0.69	0.84	0.86	0.82	0.52	0.52	0.56	**N.A. + 0.3**
INT+MWD	0.68	0.69	0.69	0.84	**0.87**	0.83	0.52	0.51	0.55	**N.A. + 0.3**
SB+CD	0.69	0.70	0.71	0.84	**0.87**	0.83	**0.54**	0.54	**0.59**	N.A. + 0.7
SB+MWD	**0.70**	**0.71**	**0.71**	0.85	**0.87**	0.83	**0.54**	0.54	**0.59**	N.A + 0.7
Majority	0.44	0.40	0.50	**0.87**	0.79	**1.00**	0.00	0.00	0.00	N.A.
Random	0.44	0.50	0.50	0.61	0.79	0.5	0.28	0.21	0.5	N.A.

Table 5.4: Results in the online setting, presented similarly to Table 5.3. The time (in sec) is the avg. training time per day and the avg. inference time per job (including the SB encoding time where applicable – "N.A." indicates the cases where SB is not applicable).

Chapter 6

Job Power Consumption

Efficient management of power resources in HPC systems is essential for several reasons. First, power consumption directly impacts the operational costs of HPC facilities, making it imperative for organizations to optimize it to minimize the costs. Furthermore, minimizing the power consumption is vital for environmental sustainability, to reduce the carbon footprint of the systems. Therefore, developing strategies aimed at minimizing the power consumption of the system, while guaranteeing its optimal performance, is of paramount importance.

This challenge can be addressed at workload level. Predicting power consumption of HPC jobs, prior to their execution on the system, allows to compute the system power consumption in advance, enabling better workload management decisions. Prior work [2, 3, 56, 57] leverages job power prediction models to infer power awareness in workload management strategies, such as power capping and workload scheduling. Hence, the need for accurate prediction models for job power consumption gained prominence in the HPC community.

Building on top of the online prediction algorithm presented in Chapter 5, we create a pipeline for the prediction of job power consumption. The task is defined as a regression problem, since we target the estimation of a mapping between the job workload manager data and its power consumption value. We present an online job power prediction algorithm which is able to efficiently predict two different targets, namely the maximum and average job power consumption. Such an algorithm is tested on a batch of job data extracted from F-DATA (Chapter

4), confirming that our online, NLP-based approach outperforms classic methods.

6.1 Related Work

In the past, several works have explored techniques to estimate power consumption prediction of jobs executed on large-scale systems.

Prior work, such as [16, 17], explored the use of workload manager information to perform power-related prediction on the execution of the job. In their work, differently from our approach, the job data is not limited to submission time, making the online prediction non-feasible.

In [12, 11], the authors propose to predict job average power consumption per node by using an RF model trained on historical data, without relying on an online algorithm.

In [30, 58] the authors propose simple non-ML algorithm which are updated over time. In particular, they propose to predict average job power consumption per node based on exponential smoothing of similar past jobs power consumption. This approach, though, relies only on categorical features of the jobs (*user id*, *group id*, *# tasks per node*, *submission time*), which are very general and less informative on the nature of the job. In fact, these approaches are shown to be outperformed by an RF model trained only once on all the data (as in [12]). The results obtained in this work show that an *online* RF model performs better than an *offline* one, confirming that our *online* solution is more accurate than the ones in [30, 58].

Differently from all those works, we propose to use a prediction algorithm which is both online and relies on NLP techniques to extract more meaningful insights from the job data. We apply the methodology proposed in Chapter 5, but to a new task and new dataset. Differently from the failure prediction, where the task is a binary classification problem, we apply the methodology to a regression task, i.e. the job power consumption prediction. Moreover, we validate the approach using different models and on different data with respect to Chapter 5.

Furthermore, differently from all the cited works on power prediction, we perform the prediction of average and maximum job power consumption values. Both values can be exploited for different power-aware scheduling strategies, such as

power capping.

For instance, in [57] the authors propose a constraint-programming based dispatching strategy exploiting job power consumption to perform power-capping. In [3] the maximum power consumption caused by the execution of the job to the system is used to make informed decisions about the scheduling of the jobs, while works like [56, 2] rely on the average job power consumption to perform the same task. Moreover, the prediction of the maximum job power consumption could be used to address the power supply demand from the supercomputing center to the electricity company, as shown in [59], aiming to estimate in advance the power load required by the system.

6.2 Methodology

In this section, we describe how we modify the online prediction methodology presented in Chapter 5 to apply it a new task, namely the job power consumption prediction of the jobs in the Fugaku dataset.

6.2.1 Data Preparation

To evaluate our prediction algorithm, we consider the subset of F-DATA containing the data collected between January and March 2022 ($\sim$ 1.5 million of jobs). Starting from the original job data presented in Chapter 4, we perform some data engineering steps to isolate the information we need to perform the prediction.

Feature selection As discussed also in Chapter 5, we can only rely on job submit-time features, since our goal is to predict the power consumption before job execution. After analyzing the full set of job features available in the dataset, we filter the submit-time ones. The final list of submit-time features of the job are listed in Table 6.1, along with their description.

Since users tend to submit jobs in batches containing similar experiments, the jobs submitted in the same batch are prone to have similar names, characteristics and perform similar operations. Given that the power consumption of a job depends on the computational operations it performs, jobs performing the same

Name	Description	Type
Job type	Category of the job (Batch, Step, etc.)	String
User	User name	String
Group	Name of the group the user belongs to	String
User id	ID of the user submitting the job	Integer
Group id	Group of the user submitting the job	Integer
Frequency	Requested frequency of the processor	Int
Job name	Name of the job	String
Host name	Name of the host node	String
Priority	The priority assigned to the job	Int
#Cores Requested	The number of cores requested	Int
#Nodes Requested	The number of nodes requested	Int
Arrival time	Time of job submission	Timestamp
Memory size limit	The limit to the memory size allocated	Int
Time limit	Maximum allowed run time in minutes	Integer
Environment	The set of environment variable	String

Table 6.1: Job features description.

or similar operations will have similar power consumption. Therefore, features like the user name, job name and environment variables, might be the key to identify similar jobs, and consequently, perform accurate job power consumption prediction.

In an initial experimental phase, we evaluate models' prediction performance on a smaller sample of the data, using different subsets and combinations of the features presented in Table 6.1. We observed that the use of particular subsets of features to represent a job is beneficial for both prediction performance and computation time, since the number of features to encode is smaller. The subset of features which yields the best predictive performance is composed of the following features, *user name, job name, # cores requested, # nodes requested* and *environment*. Therefore, we decide to use such features to represent each job in our dataset.

Job power consumption The prediction tasks require the definition of a prediction target for the training phase of the models. In this work, we focus on the maximum and average job power consumption. This information is present in each job j data, namely in the $maxpcon_j$ and $avgpcon_j$ features. The original

power consumption values range from few to millions of Watts, depending on the resources allocated to the job. This makes the prediction task very hard and the possible relative prediction error very high. In light of that, we decide to perform some data pre-processing to make the target more suitable for the regression task, as outlined hereafter.

We analyse the power traces of the nodes allocated to the jobs which run on multiple nodes. The analysis reveals that, during the job j execution, there is a small difference between the power consumption values of the different nodes allocated to the job ($nodes\_allocated_j$). Thus, for each $n \in nodes\_allocated_j$, its power consumption is well approximated by the average of the power consumption of all the nodes n in $nodes\_allocated_j$. Such assumption is particularly valid for HPC systems like Fugaku system, where the nodes have similar architectures which result in similar power consumption values. For different environments and systems (such as federated learning on different architectures), this assumption might not hold. However, using the average power consumption per node is a widely used proxy to evaluate the node-level power consumption of a job (regardless of the environment), especially in the context of job-level power prediction [12, 30], and workload scheduling [3, 2]. This allows us to predict each job power consumption as if it was running on a single node, ultimately making the prediction task less error-prone. Hence, the final job power consumption values used in the prediction tasks are defined as the average of $maxpcon_j$ (p_max) and $avgpcon_j$ (p_avg) per node, as shown in Equations 6.1 and 6.2.

$$p\_max_j = \frac{maxpcon_j}{\#nodes\_allocated_j} \tag{6.1}$$

$$p\_avg_j = \frac{avgpcon_j}{\#nodes\_allocated_j} \tag{6.2}$$

In order to gain more insights on the data for the prediction phase, we plot the distribution of p_max_j and p_avg_j in Figure 6.1. We do that also to show that the two values provide different information, thus needing two different prediction tasks. The values range from a minimum of 24 W (for p_max_j) and 20W (for p_avg_j), to a maximum of 200 W for both power values. The average values for p_max_j and p_avg_j are, 97 W and 90 W. Nevertheless, the majority of the values

are around 110 W for p_avg_j, while, for p_max_j, the values are shifted towards the 120W. We can observe that in both cases, there is a predominance of jobs in two power consumption bands, with the first being less than 50W per node, and the second being in the range 110-130 W. The values in between the two power bands are more uniformly distributed, while the jobs consuming very high amount of power ($\geq$ 130W) are in a very limited number with respect to the others.

This analysis shows that p_max_j and p_avg_j are indeed different, and predicting them defining two different tasks is necessary to obtain reliable results.

6.2.2 Job Power Consumption Prediction

The methodology, in terms of feature encoding and model training and testing, is the same one presented in Chapter 5. However, the task addressed is a regression and not a classification, hence the models employed need to be adapted to predict an integer value.

The target of the predictions are the p_max_j and p_avg_j value of a job, and we test our algorithm on both the tasks. From here on, we refer to the prediction of the p_max_j and p_avg_j as the *maximum* and *average* task.

6.2.3 Experimental Study

In this section, we present the experimental setting and the results of the tests conducted for the study.

6.2.4 Experimental Setting

We run our experiments on a machine equipped with two AMD EPYC 7302 CPUs with 64 cores and 512 GB of RAM.

The RF and AD algorithms are implemented with the *scikit-learn*[1] Python library, while the XG implementation is retrieved from the *xgboost*[2] library. The sequence encoder model is provided by the *sentence transformers* library[3], while

[1] `https://scikit-learn.org/stable/`
[2] `https://xgboost.readthedocs.io/en/stable/index.html`
[3] `https://www.sbert.net`

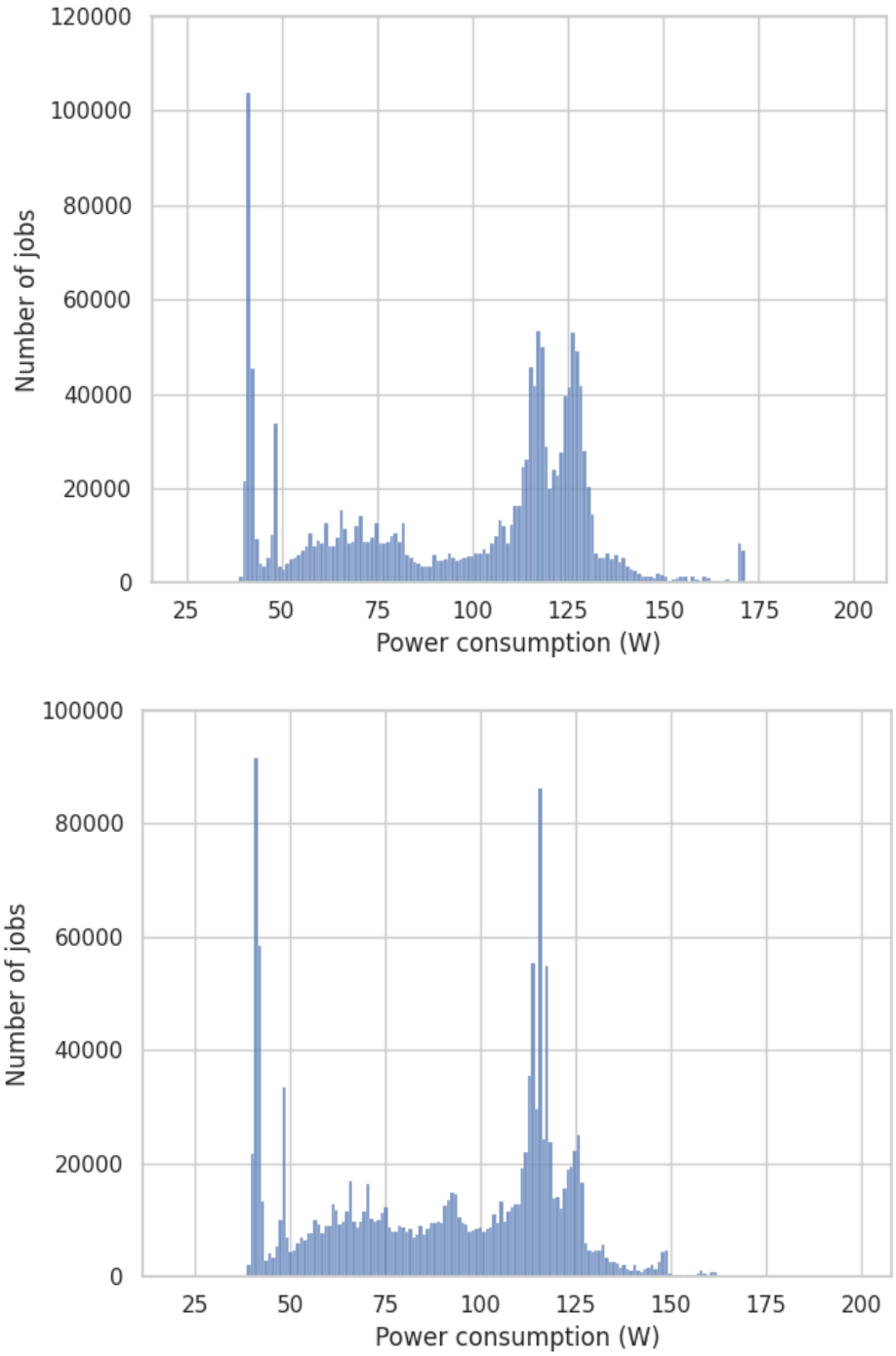

Figure 6.1: Distribution of p_max_j (above), and p_avg_j (below).

the weights for SBERT are pulled from huggingface.[4] We use the pre-trained model *all-MiniLM-L6-v2*[5], since it is the best trade-off between prediction performance and speed [28]. All the models are instantiated with the default setting provided by the libraries. The implementation and the details regarding the Python version and its packages are available in a GitHub repository.[6]

Job power prediction We evaluate the models for the prediction of job power consumption on several metrics typically used in regression tasks, namely Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE) and Coefficient of Determination (R2).

For the *offline* setting, the parameters are set as in Chapter 5, thus we take the first 70% of the data as the training set and the remaining 30% as the test set. Concerning the *online*, we experiment with three different values for the α, namely 15, 30 and 60. We observe that, for the Fugaku data, the best prediction performance is obtained with $\alpha = 60$. Therefore, differently from our previous work, we fix the training interval α to 60 (30 in the past work) days, while we keep $\omega = 1$ day. The training/testing step of the models is performed only if the data splits defined by α and ω contain more than 1 element each, otherwise, we move on to the following splits. We are able to compute 16 iteration over the data in the *online* settings, corresponding to the job data submitted between the 16^{th} and the 31^{st} of March 2022. Therefore, the results reported in Table 6.2.5 and 6.3 are the average of the results of the 16 tests.

For the evaluation phase, we distinguish between the job feature encodings (INT and SB) and the supervised algorithms (AD, XG, RF). Each regression algorithm is evaluated using the two feature encodings, and are compared with two simple baselines predicting constant values, namely c_max and c_avg. These baselines always predict the same value, ignoring the input feature values (thus we don't need to distinguish between INT and SB encoding since the input features are not considered for the prediction). The c_max always predicts the maximum

[4] `https://huggingface.co`
[5] `https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2`
[6] `https://github.com/francescoantici/job-pcon-predictor`

value among the power consumption of the jobs in the training set, while the c_avg always predicts the average one.

System power prediction Furthermore, after testing the models at job level, we want to evaluate the prediction performances at system level. We do that for two main reasons, (i) to see if our models are capable of reconstructing the system power state accurately, thus providing a tool which is able to predict the power consumption of a whole system by considering only the job submitted, and (ii) because the prediction error on a single job is either an overestimate or an underestimate of the actual power consumption, so we believe that such errors might cancel out each other at system level, given the large amount of jobs running concurrently.

We estimate the real system global power consumption for each hour ($psys_h$) of each day of the *online* testing phase. We start by computing the power consumption of the system for a single minute, by summing the power consumption of all the jobs $j \in J_{h,m}$ running concurrently on the system between the minutes m and $m+1$ of the hour h. The hourly power consumption of the system is then computed by taking the average of its power consumption for each minute of the hour, as shown in Equation 6.3 and 6.4.

$$psys\_avg_h = \frac{\sum_{m=1}^{60} \sum_{j \in J_{h,m}} p\_avg_j}{60} \tag{6.3}$$

$$psys\_max_h = \frac{\sum_{m=1}^{60} \sum_{j \in J_{h,m}} p\_max_j}{60} \tag{6.4}$$

The same methodology is used to compute the predicted system power, by replacing the true job power consumption values with the predicted ones.

6.2.5 Results

After computing all the experiments, we analyse the results obtained. First, we present the power prediction performance of the models in the *offline* and *online* setting. Finally, we show how well our models perform at system level.

Model	MAE (W)	MAPE (W)	MSE (W)	RMSE (W)	R2
INT+AD	33.43	0.53	1372.86	37.05	0.05
INT+XG	35.00	0.55	1643.22	40.05	-0.14
INT+RF	39.64	0.59	2029.66	45.05	-0.41
SB+AD	33.41	0.52	1337.30	36.56	0.07
SB+XG	**22.70**	**0.37**	**844.84**	**29.06**	**0.41**
SB+RF	28.15	0.50	1368.74	37.00	0.05
con_max	110.42	1.80	13632.96	116.76	-8.46
con_avg	33.93	0.59	1564.92	39.55	-0.09

Offline job power prediction Table 6.2 reports the results of the *offline* experiments. These show that the SB+XG model is the best-performing model for both the *maximum* and *average* task, obtaining a RMSE of 29.6 and 25.8, respectively. It is noticeable how the use of the SB encoding improves the performances of all the models. For instance, on the RF model, the R2 score goes from -0.15 (INT+RF) to 0.23 (SB+RF), confirming that a meaningful representation of textual features improves the prediction performance. The two baselines employed for the prediction tasks obtain poor results in both the *average* and *maximum* tasks. While the const_max baseline obtains the worst results for the prediction task, the const_avg baseline performs similarly to models trained with the INT encoding. These results highlight the difficulty of the prediction task and the importance of leveraging on a trained model to perform the prediction.

Online job power prediction In Table 6.3 we show the prediction performance of the models in the *online* setting. Coherently with the *offline* setting, the use of the SB encoding improves all the models performances significantly. For instance, the RMSE score of the INT+RF model is almost twice the one of SB+RF for both the *maximum* and *average* task. Moreover, the R2 score of all the models increases significantly with the SB encoding, going from values lower than zero to greater than 0.50 (INT+XG and SB+XG). In this setting, the models obtaining better results are the SB+XG and SB+RF. In the *maximum* task, both the models obtain the same MAPE of 27%, but the SB+XG reaches slightly better results in terms of MAE (18.70 vs 18.88), MSE (631.63 vs 655.32), RMSE (25.13 vs 25.59) and R2 (0.57 vs 0.55). Conversely, on the *average* task, the trend is reversed. Indeed, the

Model	MAE (W)	MAPE (W)	MSE (W)	RMSE (W)	R2
INT+AD	30.30	0.52	1161.94	34.1	0.0
INT+XG	25.16	0.38	886.54	29.77	0.24
INT+RF	31.76	0.44	1346.77	36.70	-0.15
SB+AD	29.72	0.51	1112.15	33.35	0.05
SB+XG	**19.90**	**0.32**	**666.05**	**25.80**	**0.43**
SB+RF	23.47	0.42	895.98	29.93	0.23
con_max	116.58	1.95	14758.07	121.48	-11.65
con_avg	31.18	0.55	1288.17	35.90	-0.10

Table 6.2: Results in the offline setting for the *maximum* (above), and *average* (below) power consumption task. For MAE, MAPE, MSE and RMSE metrics the lower, the better, while for R2 score the higher, the better. Best results are highlighted in bold.

SB+RF model obtain better results in terms of MAE (16.80 vs 17.14) and MSE (557.65 vs 557.81), while the MAPE (0.26), RMSE (23.61) and R2 (0.53) scores are equivalent.

The comparison of the results of the *online* and *offline* setting outlines that retraining the models is beneficial for prediction performance. The most noticeable enhancement is obtained in terms of the R2 score. Indeed, the score increases by 0.16 (*maximum* task) and 0.10 (*average* task) points. Based on the definition of the R2 metric, this represents a significant improvement in the model capability of capturing characteristics and variation of the target values.

We plot Figures 6.3 and 6.4 to visualize how much the models are able to capture the variability of the target values. We create the figures performing the following steps. First, we group all the job data belonging to all the test splits together in a list; then we sort them in ascending order by their actual power consumption values. We further split the sorted job data in 30[7] batches, with each batch identifying a range of true power consumption values. Finally, for each batch, we show the true (solid blue line) and the predicted (whisker) power consumption values of the jobs belonging to it. This allows us to understand how the model performs on the different ranges of job power consumption values.

[7]The number of splits is decided after an empirical evaluation to guarantee readability of the plots.

Model	MAE (W)	MAPE (W)	MSE (W)	RMSE (W)	R2
INT+AD	33.37	0.52	1346.27	36.69	0.08
INT+XG	37.75	0.41	2207.05	46.97	-0.50
INT+RF	43.01	0.57	2464.04	49.63	-0.68
SB+AD	28.74	0.44	1008.74	31.76	0.31
SB+XG	**18.70**	**0.27**	**631.63**	**25.13**	**0.57**
SB+RF	18.88	**0.27**	655.32	25.59	0.55
c_max	110.36	1.81	13647.47	116.82	-8.29
c_avg	34.21	0.59	1573.37	39.66	-0.07

We plot the results of the RF and XG models, both with the INT and SB encodings, to check that the improvement in the results presented in Table 6.3 actually corresponds to a better reconstruction of the target values. As expected, the distribution of the ranges is better approximated by the models with the SB encoding, in both the *maximum* and *average* tasks. The models with the INT encoding (Figure 6.3) tend to always predict the same values regardless of the target, while the SB encoding (Figure 6.4) seems to make the models more flexible and adaptable to the different distributions. Even though the prediction performance of the models is improved with the SB encoding, there is still evidence of power values that the models struggle in predicting[8]. For instance, the extreme values of the ranges, both for the *maximum* and *average* task. For the case of the jobs consuming low power, the absolute prediction error is quite small, given the low power values. Concerning the jobs consuming very high levels of power, we can observe by the power distribution plots in Figure 6.1 that such values are the least numerous in the dataset. Therefore, since their occurrence is very rare, it is very hard for the models to learn patterns to predict their values.

System power prediction Figure 6.5 presents the system-level evaluation of the best models for the *maximum* and *average* task in the online setting, namely SB+XG and SB+RF. The period of time represented in the plots is the testing period of the *online* setting, which concerns the jobs submitted between the 16^{th} and the 31^{st} of March 2022 (Section 6.2.4). We plot the figures to show how

[8]In Figure 6.4 there is an outlier in the predictions in correspondence to the true power values between 115 and 120 Watts. The reasons for this are still under investigation.

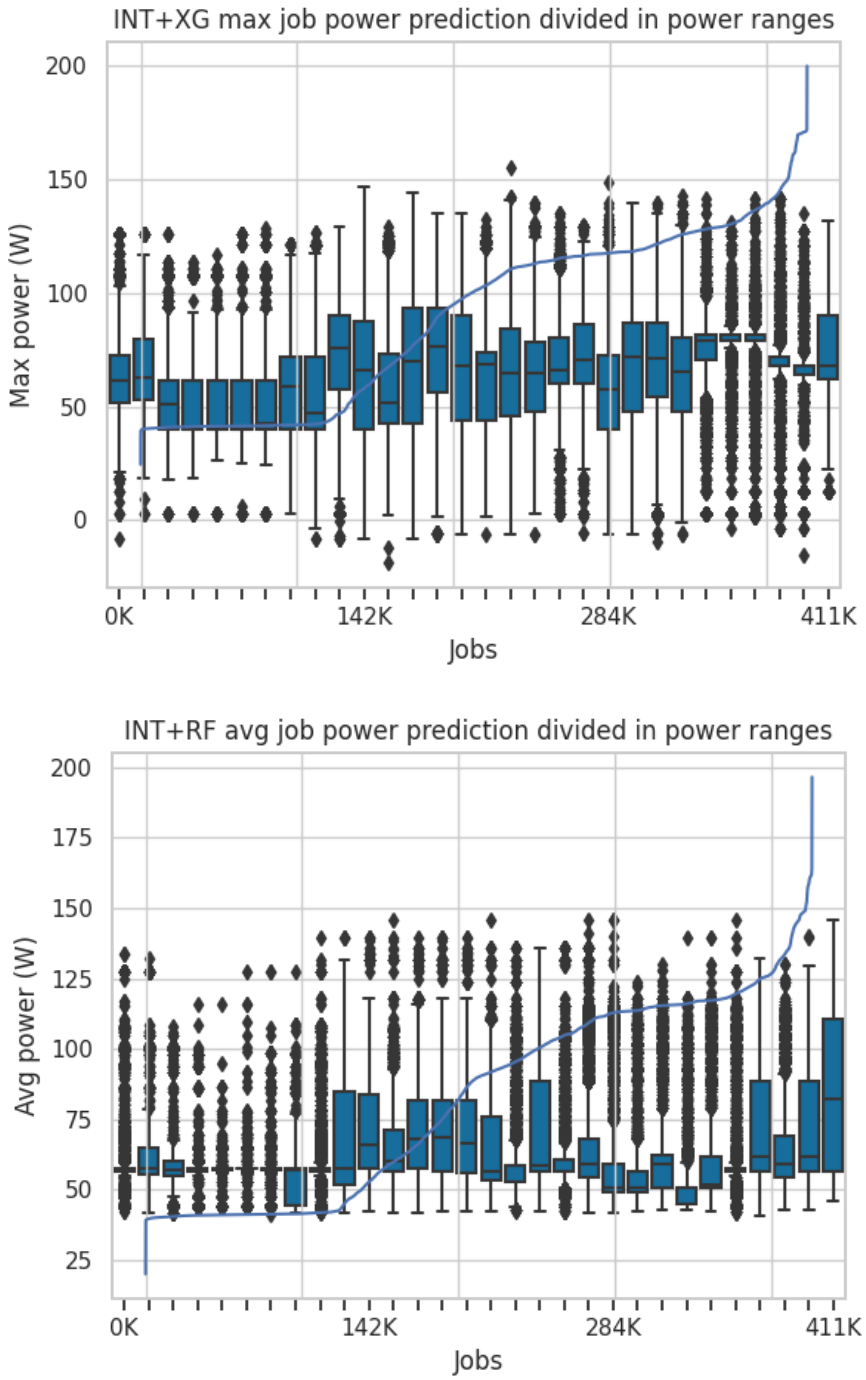

Figure 6.2: Job true and the predicted power for the *maximum* target with the INT+XG model (above), and for the *average* target with the INT+RF model (below). The x-axis represents the number of jobs in the test set.

Model	MAE (W)	MAPE (W)	MSE (W)	RMSE (W)	R2
INT+AD	30.29	0.51	1129.74	33.61	0.05
INT+XG	33.15	0.42	1519.56	38.98	-0.28
INT+RF	33.48	0.39	1667.66	40.83	-0.41
SB+AD	27.37	0.45	925.28	30.41	0.22
SB+XG	17.14	**0.26**	557.81	**23.61**	**0.53**
SB+RF	**16.80**	**0.26**	**557.65**	**23.61**	**0.53**
c_max	116.50	1.96	14756.10	121.47	-11.47
c_avg	31.22	0.55	1281.47	35.79	-0.08

Table 6.3: Results in the online setting for the *maximum* (above), and *average* (below) power consumption task. For MAE, MAPE, MSE and RMSE metrics the lower, the better, while for R2 score the higher, the better. Best results are highlighted in bold.

well the total true power consumption of the jobs (blue line) is approximated by the predictions of our models (orange line). The results confirm the hypothesis formulated in 6.2.4, i.e. that the prediction performance is improved at system level (MAPE of around 5% and a R2 ¿ 0.97 for both tasks) with respect to the job level. This can be caused by several factors. For instance, the predicted job power consumption prediction might be either an overestimate, or an underestimate of the actual values. These can balance each other out when summing the power consumption of the concurrent jobs, obtaining a better approximation of the overall power consumption.

In this scenario, it is important to specify that the reconstruction is an estimate of the systems' power consumption, which does not match with the actual power consumption of Fugaku. This is because we don't have access to other power consumption sources of the system (e.g. idle node power) and there are some missing information in our data which prevents us to reconstruct the exact load of the system through time. Hence, instead of seeing the Fugaku-typical constant load of about 19MW[9], we are seeing an increasing curve. Nevertheless, this plot shows that for the available data, our prediction model nearly matches the actual load of the system in Spring of 2022.

Finally, in Figure 6.6 we report the average training time for all the models

[9]https://status.fugaku.r-ccs.riken.jp/

employed in the experiments with the INT and SB encoding. In each heatmap we show the average, maximum and minimum training time of the model throughout the 16 days of the testing of the *online* setting. We do that to investigate if the training time has a very high variability through time, which would represent a non-predictable behaviour of our algorithm. The figures show that our models' training time does not present a significant dissimilarity between the minimum, average and maximum value. The difference is justified by the variability of the number of job data in the training splits, which range from a minimum of 1038542 jobs to a maximum of 1481037 (mean value of 1261219).

For the case of the INT encoding, the training time is almost negligible for the RF and XG models, while it is around 2 minutes for the AD.[10] As introduced in Section 6.2, the SB encoding maps the input to a 384 dimensional vector, with respect to the 5 dimensional of the INT representation. This step introduces a significant complexity to the model, since the feature space of the SB encoding is almost 80 times bigger than the INT one. RF relies on single features values correlation with the target, so its computation time is heavily influenced by the dimensionality of the input data (training time more than 60 times bigger in the case of SB encoding). The AD model is the one obtaining worse results in terms of training time for the INT encoding (90 seconds), while it performs better than RF with the SB encoding($\sim$ 7000 seconds less in every case). The XG model is the most robust in terms of training time, since despite the high-dimensionality of the input with the SB encoding, it is able to perform the training in around 3 minutes for the worst case scenario, making it easily deployable to a real system.

[10]Differently from the other two models, the scikit-learn API for the AdaBoostRegressor algorithm currently does not support the distribution of the training operations on multiple processors

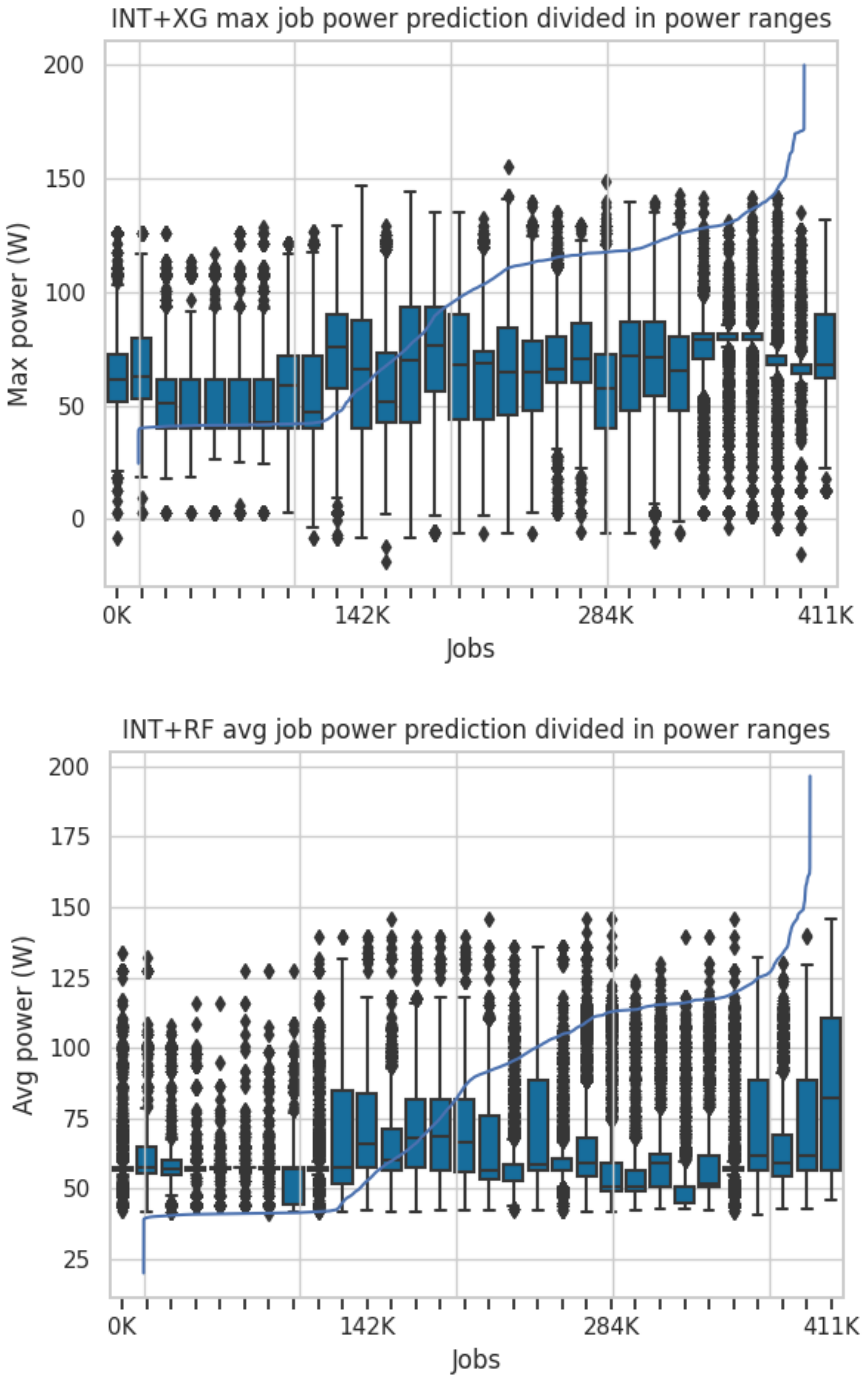

Figure 6.3: Job true and the predicted power for the *maximum* target with the INT+XG model (above), and *average* target with the INT+RF model (below). The x-axis represents the number of jobs in the test set.

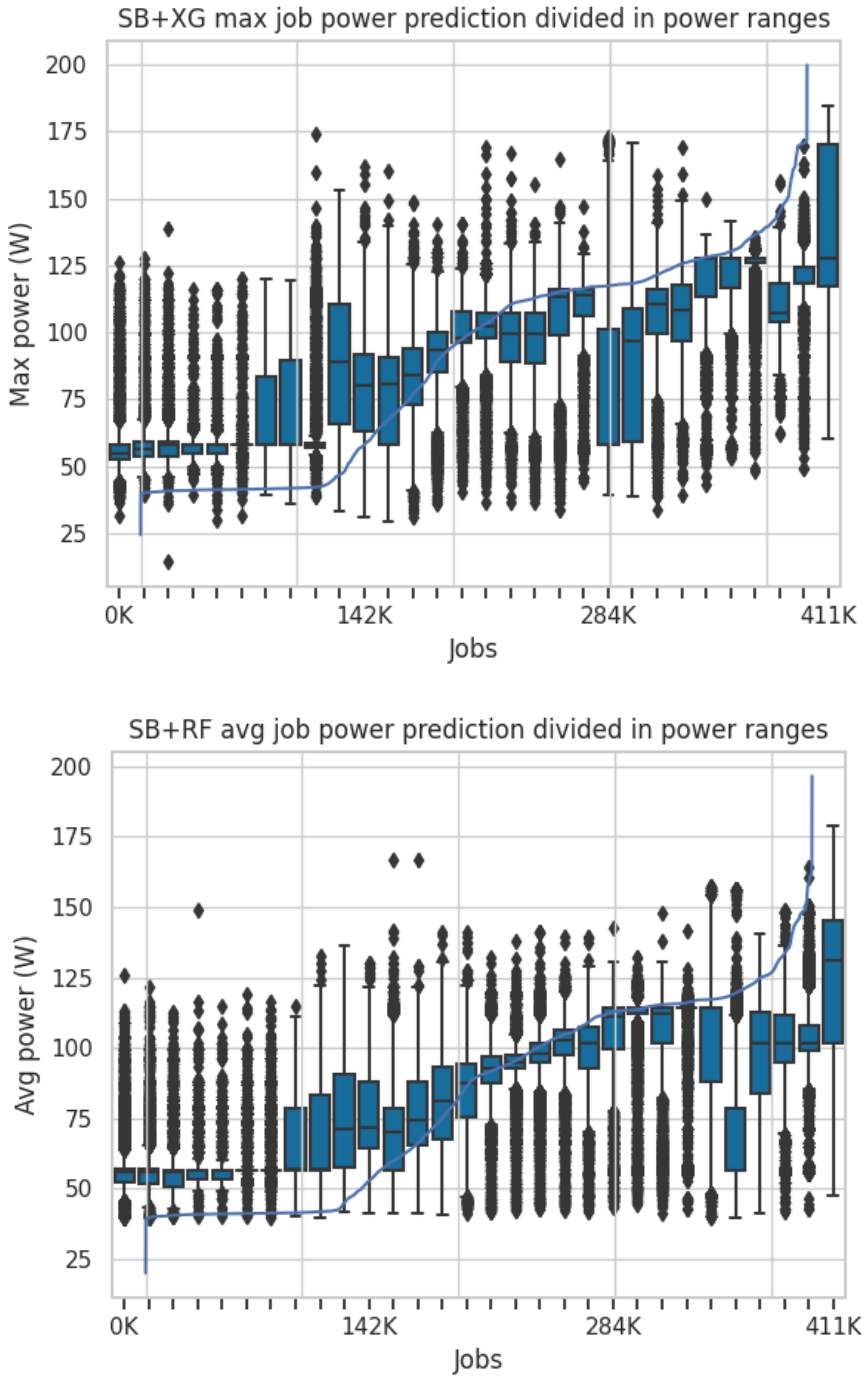

Figure 6.4: Job true and the predicted power for the *maximum* target with the SB+XG model (above), and for the *average* target with the SB+RF model (below). The x-axis represents the number of jobs in the test set.

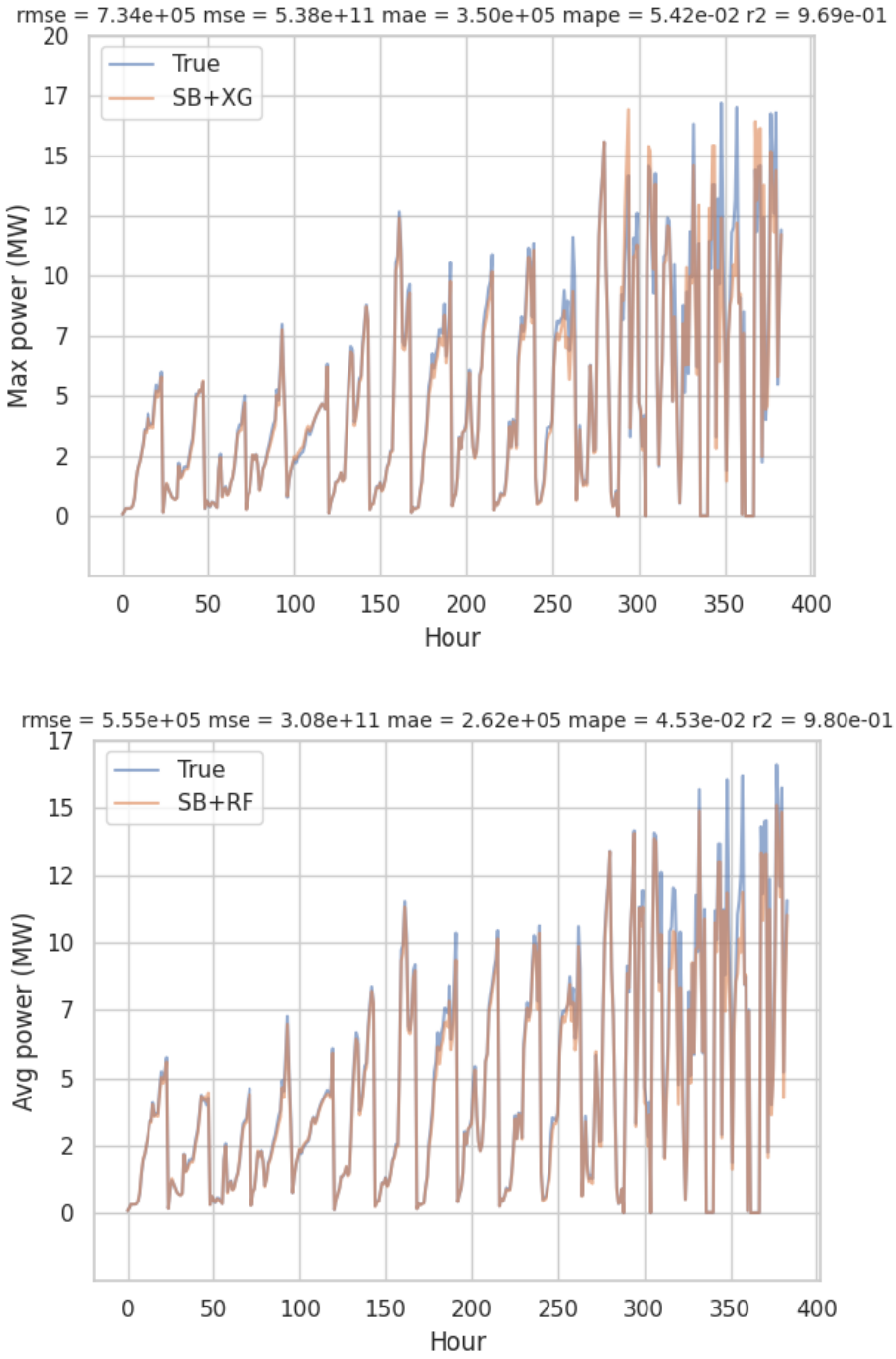

Figure 6.5: System true and predicted power for the *maximum* task with the SB+XG model (above), and for the *average* task with the SB+RF model (below).

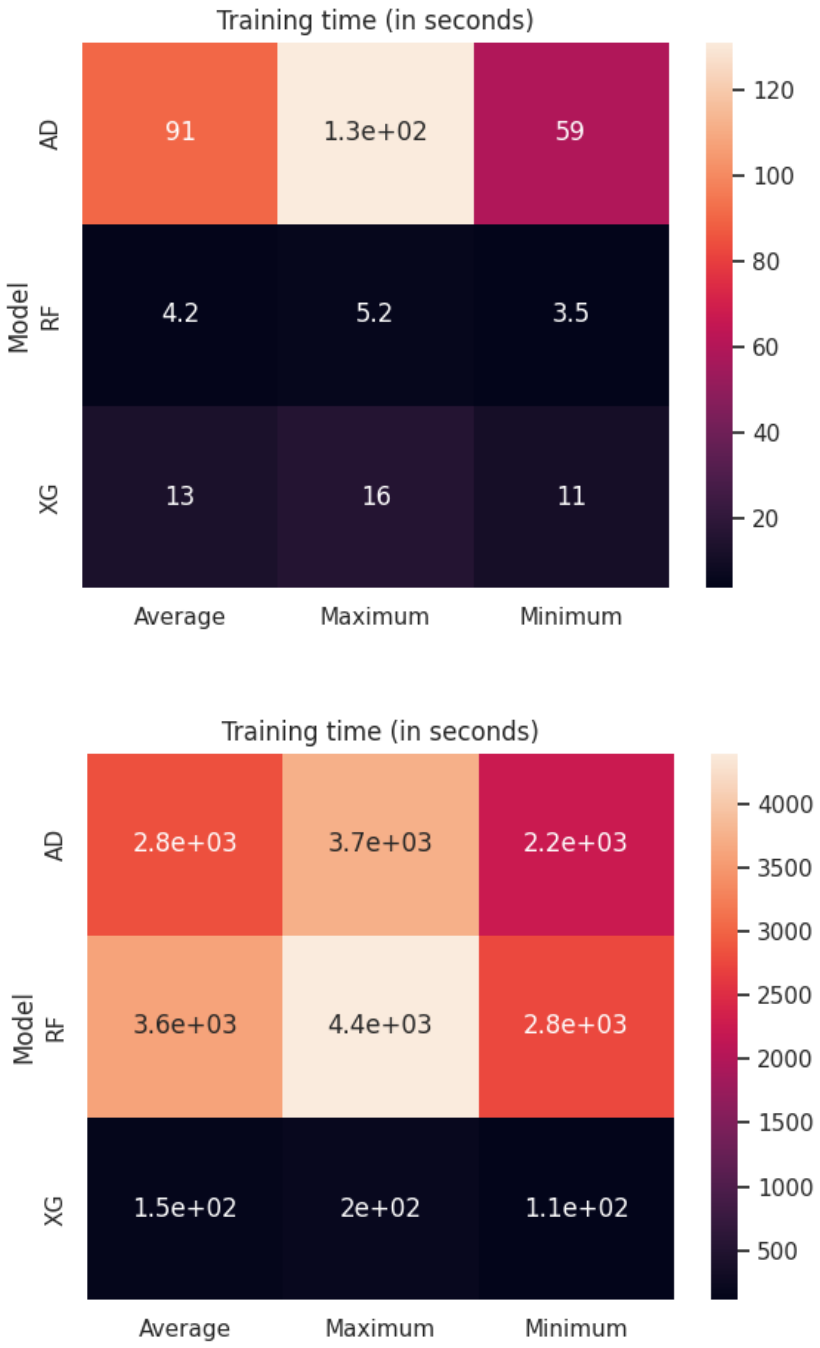

Figure 6.6: Models' training time with the INT (above) and SB (below) encoding.

Chapter 7

Job Memory/Compute-Bound Nature

HPC jobs, like any application, can be classified based on the intensity of their system resource usage as *memory-bound* and *compute-bound* [6, 8]. The first category refers to the jobs whose attainable performance are bound by their memory access rate, often measured as the utilization level of the available system memory bandwidth. In contrast, the *compute-bound* refers to the ones whose performance is bound by the system's arithmetical performance, often measured as the rate of double-precision floating-point operations computed per second. As jobs are often not engineered to simultaneously saturate the different system resource types, failure in identifying their category prior to their execution is likely to cause inefficiency in resource usage, system throughput and energy consumption [5, 6, 60].

Conversely, knowing if a job is *compute-bound* or *memory-bound* upon submission allows making informed decisions about its scheduling and execution. For instance, it can be used to design specific hardware-software co-design techniques [40, 46, 47, 61, 62], or a job co-scheduling strategy that allocates the same node to jobs with different characteristics [6, 8]. Both techniques have been proven effective in enhancing system throughput, while significantly reducing system energy consumption. Therefore, classifying jobs as *memory-bound* and *compute-bound*, prior to their execution, has the potential to improve the system energy efficiency and throughput, without the need of any intervention by the user, as shown in

[40, 6, 8, 46, 47, 61, 62].

To develop reliable classification models to predict the *memory/compute-bound* nature of a job before its execution, a large amount of labelled job data is needed. However, to the best of our knowledge, such a public dataset for a production system does not exist. Without prior knowledge on jobs' computational operations and memory usage, they can only be characterized by analyzing the performance metrics collected during the execution. This requires a systematic characterization technique leveraging the data collected during job execution. Job data analysis based on this characterization could also provide insights into the system usage; for instance, whether the users submit jobs optimized to fully saturate the different system resources and if specific actions can be enacted to improve system throughput.

Despite recognizing its importance, no past work has proposed a solution to systematically and seamlessly characterize and classify *memory/compute-bound* jobs in an HPC system before job execution, nor has proven the feasibility of such an approach. During our research, we tackled the aforementioned challenges by developing *MCBound*, the first online data-driven framework to classify HPC jobs before job execution as *memory-bound* and *compute-bound*, without user intervention. We propose a systematic characterization technique to generate a reference dataset from historical data for our initial classification model training. Using the proposed characterization technique, we analyze the data from 2.2 million job runs on the Supercomputer Fugaku to obtain insights into their *memory/compute-bound* characteristics. Moreover, we employ *MCBound* to classify the jobs executed on Fugaku during February 2024, obtaining an F1-macro average score [63] of at least 0.89 as prediction quality.

The *MCBound* framework is online in the sense that it works in real-time on live streaming data and periodically updates the classification model in the background. The job characterization is performed systematically, leveraging job performance metrics, system's specifics, and the *Roofline* model [41] technique. The classification is achieved through a prediction algorithm relying on NLP and supervised ML models, which is trained on historical and properly characterized job data, and is able to classify unseen jobs upon submission prior to their execution. The algorithm is periodically retrained over time on recent data. Our

framework can be configured ad-hoc to meet the needs and characteristics of the system on which it is deployed.

Towards implementing *MCBound* in a production HPC system, we extract from F-DATA 2.2 million jobs executed on Fugaku between December 2023 and March 2024. Our job analysis using our characterization technique reveals that the great majority of Fugaku jobs are *memory-bound* and users execute large numbers of *compute-bound* jobs with the system's default execution mode (i.e. 2.0 GHz), instead of the boost mode (i.e. 2.2 GHz), which may result in longer execution time, node-hours wastage and increased energy consumption.

We implement *MCBound* for Fugaku and evaluate the online prediction algorithm with over 700,000 jobs executed during February 2024. We study the impact of choice of recent data for periodic retraining and retraining frequency on prediction accuracy and runtime overhead of training and inference. We show that our approach is effective for the classification task with an F1-macro average score of at least 0.89 as prediction quality and it incurs low runtime overhead on the system.

To the best of our knowledge, this is the first approach to systematically and seamlessly characterizes and classify *memory/compute-bound* jobs in HPC systems before job execution, without requiring any intervention by the user.

7.1 Related Work

The use of the *Roofline* model to evaluate computational bottlenecks and characterize *memory-bound* and *compute-bound* applications is a standard in the field, and it has been done in several past work [62, 64, 65, 66]. In [64, 65], the authors rely on a technique based on the *Roofline* model for an in-depth analysis of application bottleneck in the cache memory. Whereas, in [66, 62] it is used to characterize *memory-bound* and *compute-bound* applications and evaluate the impact of optimization techniques on their execution. All the cited work characterized a few well-known kernels or benchmarks via visual analysis of the resulting *Roofline* model of the computations, while we here do it systematically on millions of real jobs, for which we have no prior knowledge on the operations performed.

In *MCBound*, we target predicting a new job characteristic prior to job execu-

tion, as runtime prediction [36, 17] may incur overhead on the system operations and necessitate modification to the regular workload submission workflow, as often thousands of jobs are submitted every second.

This work proves that online NLP-augmented RF and KNN models are effective also in predicting the *memory/compute-bound* job class, other than failure (Chapter 5), and power (Chapter 6). Moreover, we study the impact of choice of recent data for periodic retraining and retraining frequency on prediction accuracy and runtime overhead of training and inference. Finally, we integrate the algorithm as a component in a deployable framework.

7.2 MCBound Framework

In this section, we first describe the *MCBound* framework at a high-level, then detail its main components, and finally explain how it is deployed on the target system.

The framework is designed to be deployed in a real system where jobs are submitted and executed continuously, and various information regarding job submission, execution and completion (referred to as job data) is streaming in time. For this reason, *MCBound* employs an online prediction algorithm which shares the characteristics of the one presented in Chapter 5. First, it relies on job submission information only, as classification of a job before its execution can be done by leveraging only such data, and the historical data of the jobs that are already completed by that time. Second, to adapt to changes in the workload and guarantee accurate classification, the model is periodically updated over time by using the recently executed job data for training. The framework thus executes in two different modes: (i) periodic model retraining on recent job data and (ii) model inference on a newly submitted job. For this, the framework requires an operational data analytics framework that collects job data and stores them in a jobs data storage.

The framework is depicted in Figure 7.1, where the rectangular blocks represent the main components and the blue containers show how they are employed in two Continuous Integration/Continuous Delivery (CI/CD) workflows. The components are:

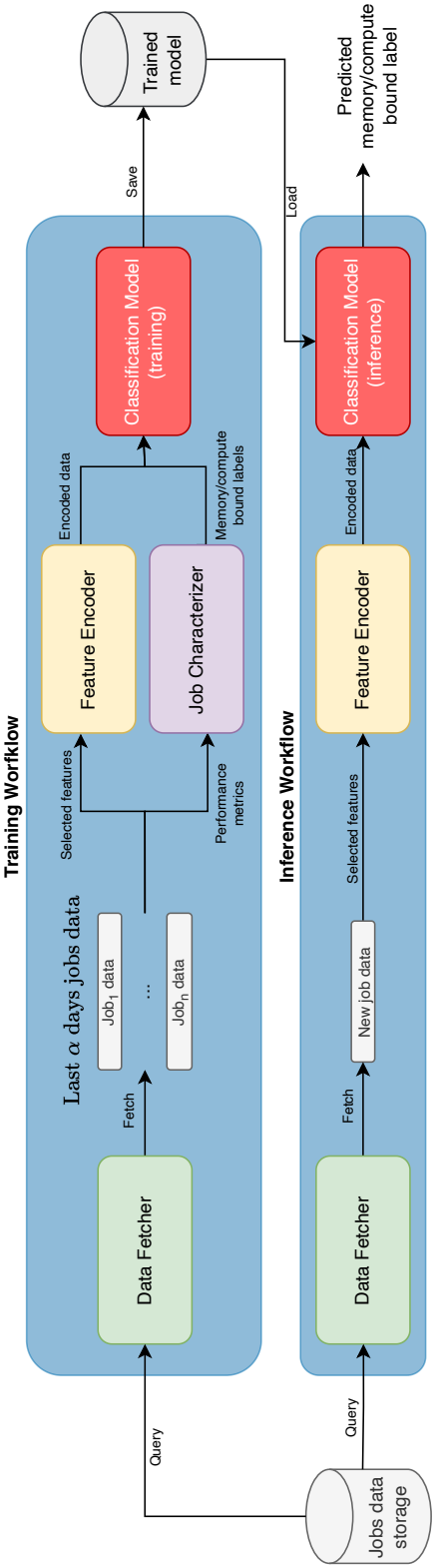

Figure 7.1: High-level scheme of the components and the workflows of *MCBound*.

- The *Data Fetcher*, which retrieves the job data by querying the jobs data storage.
- The *Feature Encoder*, which takes as input a series of raw job data, and returns the encoded data to be fed into the *Classification Model*.
- The *Job Characterizer*, which takes the raw job data as input and augments the job data with *memory/compute-bound* labels.
- The *Classification Model*, which uses the encoded data together with the *memory/compute-bound* labels for model training, and just the encoded data for model inference to classify each submitted job as *memory* or *compute-bound* prior to its execution.

When triggered:

- the *Training Workflow* fetches the data of the jobs executed in the last α days to generate an instance of a trained *Classification Model* once every β days;
- the *Inference Workflow* fetches the data of a new (unseen and not yet executed) job and generates a *memory/compute-bound* label for it.

MCBound targets the Fugaku supercomputer, however, the framework is designed to work universally for any system. It requires only that the jobs data storage is integrated in the system, containing job features referring to submission (such as *requested resources*, *user information*, *job name*), execution and completion (such as *duration* and *#nodes allocated*), and performance metrics (such as $\#flops$ and $\#moved\_memory\_bytes$). The architecture is designed to be modular and easy to customize for different systems, for instance by implementing different *Data Fetcher*, *Feature Encoder*, or *Classification Model*. All the framework components are software components implemented as Python classes, with a method for each functionality they provide.

7.2.1 Data Fetcher

The *Data Fetcher* is an interface to retrieve data from the jobs data storage, which contains information of executed and newly submitted jobs. At initialization time,

the class allows configuring the *Data Fetcher* object to interact with the specific data storage technology deployed in the target system (e.g., relational database, non-relational database, distributed file system). The class provides the *fetch* method, which takes as input either a *job_id*, or a *start_time* and *end_time*. With the first parameter the method fetches the data of a single job corresponding to the *job_id*, with the second instead the data of all the jobs executed between *start_time* and *end_time* are fetched. These parameters are used to generate an SQL query to the job's data storage. The query results in a list of job features and their values, which is then returned as the output of the *fetch* method.

7.2.2 Feature Encoder

This component represents job features in a format suitable to be fed into the *Classification Model*, i.e. an array of floating point values. The class is endowed with an *encode* method which takes as input the raw job data and outputs the encoded job data. Internally, the method filters out a subset of job features, selected empirically during an initial experimentation phase and accordingly to the chosen classification model. The corresponding feature values are then concatenated into a comma-separated string and encoded with Sentence-BERT (SBERT) [28]. The resulting representation of the feature string is a fixed-size 384-dimensional floating-point array, which constitutes the output of the *encode* method.

This method can be modified to select any subset of job features and to leverage any encoding technique (such as classical categorical mapping of feature values to integers, transformers or neural encoder/decoder models) able to map job features to a suitable format for the *Classification Model*.

7.2.3 Job Characterizer

The *Job Characterizer* component exploits the *Roofline* model [41] which represents the compute-memory ratio of a computation, and allows identifying if it is *compute-bound* or *memory-bound*. By using system specifics (i.e. peak performance and peak memory bandwidth), it computes the operational intensity *op* (mean operations per byte of memory traffic) of the ridge point op_r for a machine m, namely the minimum *op* needed to obtain the peak performance of m. This

value is then used to distinguish between *memory-bound* and *compute-bound* computations. In our case, computations are jobs, and machine m is a single node n of an HPC system.

The *Job Characterizer* class is initialized with the peak performance and the peak memory bandwidth of a single node n of an HPC system, and computes the operational intensity of the ridge point op_r. The *generate_labels* method of the class returns the *memory-bound* or *compute-bound* label of a job j given as input its feature values. These are the number of floating point operations ($\#flops_j$), the duration ($duration_j$), the number of nodes allocated ($\#nodes\_alloc_j$), and the amount of moved memory bytes ($\#moved\_memory\_bytes_j$). These features can be obtained by filtering the job execution statistics and performance metrics. Internally, the method computes the performance (p_j), the memory bandwidth (mb_j), and operational intensity (op_j) for j. As p_j is a measure of Flops per second, we divide the $flop_j$ by $duration_j$. Furthermore, since the *Roofline* model refers to a single node of the machine, the performance of j needs to be normalized on $\#nodes\_alloc_j$, obtaining for each job the per node average p_j, mb_j and op_j. Then, p_j, mb_j and op_j are computed as shown in Equations 7.1, 7.2 and 7.3. The *generate_labels* method returns *compute-bound* if op_j is greater than op_r computed at class initialization time, *memory-bound* otherwise.

$$p_j = \frac{\#flops_j}{duration_j * \#nodes\_alloc_j} \tag{7.1}$$

$$mb_j = \frac{\#moved\_memory\_bytes_j}{duration_j * \#nodes\_alloc_j} \tag{7.2}$$

$$op_j = \frac{p_j}{mb_j} \tag{7.3}$$

In this first version of *MCBound*, we focus only on the classes defined in the original *Roofline* paper [41]. However, by adding to the *Roofline* model the bandwidth of other hardware components (e.g. cache, interconnect and GPUs) it is possible to expand the *Job Characterizer* to create other labels for the job data, such as *interconnect-bound* and *GPU-bound*.

7.2.4 Classification Model

This component provides methods for the classification task via data-driven prediction algorithms. When an object of the class is created, the initialization method takes as input the name of the algorithm to employ. In our use case, we implement two instantiations, using supervised ML algorithms which are first trained on historical and properly characterized job data, before performing inference on unseen jobs. Such algorithms are the KNN and RF, presented in Chapter 2.3. The chosen algorithms spend complementary effort in their training and inference parts. While RF needs to dedicate a significant amount of time to training, the KNN does that for inference. Availability of algorithms with different learning nature allows choosing the best trade off between the quality of prediction and the runtime effort spent on it. We note that it is possible to implement any data-driven prediction algorithm for *Classification Model*, such as neural networks, other ML-based or even heuristic algorithms.

Once initialized, the *Classification Model* instance provides the *training* and *inference* methods. The *training* method takes as input two arrays containing respectively encoded job data and the corresponding *memory/compute-bound* labels. The input data is then used to train the *Classification Model* instance. The *inference* method can be called only after the *Classification Model* instance is trained. The method takes as input an array of encoded job data and generates a list of predicted *memory/compute-bound* labels for all the jobs.

7.2.5 MCBound Deployment

We implement *MCBound* as a *flask*[1] backend, providing APIs to perform the operations of the framework. Flask is endowed with a built-in development server, but it can be also easily deployed to any HTTP server. We also provide a Docker [67] configuration, to distribute the framework as a container and make it scalable through container orchestration techniques, such as Kubernetes [68].

The workflows shown in Figure 7.1 are implemented as Python scripts leveraging the framework APIs to perform the necessary steps. The trained model

[1] `https://flask.palletsprojects.com/`

instances are saved to the machine file system by using the skops.io library,[2] in order to handle and maintain different versions of the models.

We provide a `deploy` script for the first deployment of *MCBound*. The script first executes the *Training Workflow* script to generate the trained instance of the *Classification Model*, and then the *flask* application of *MCBound* is started. Finally, a cronjob [69] is scheduled to re-execute the *Training Workflow* script every β days.

The online prediction starts when a trained instance of the *Classification Model* is generated. Then, the trained instance is used by the *Inference Workflow* script to generate predictions for the all jobs submitted in the following β days. Within this period, the inference on a job can be triggered in two different ways depending on how and when the prediction is needed: at each new job submission, or by periodically querying the jobs data storage to retrieve the accumulated new job data. After β days, the cronjob for the *Training Workflow* script is re-triggered, an instance of the *Classification Model* is trained, and the framework is ultimately reloaded.

7.3 Memory/Compute-bound Characterization and Analysis of Fugaku Jobs

In this section, we apply our characterization approach to the job data obtained from the Fugaku system and analyze the outcome. The characterization is necessary to acquire the ground truth for the prediction algorithm evaluation in Section 8.4, while the analysis allows to obtain insights into the system usage; for instance, whether the users submit jobs optimized to fully saturate the different system resources and if specific actions can be enacted to improve system throughput.

7.3.1 Fugaku Job Traces

For our experiments, we use the F-DATA dataset. This data includes information about job submission (such as *submission time*, *requested resources*, *user*

[2] `https://skops.readthedocs.io/en/stable/index.html`

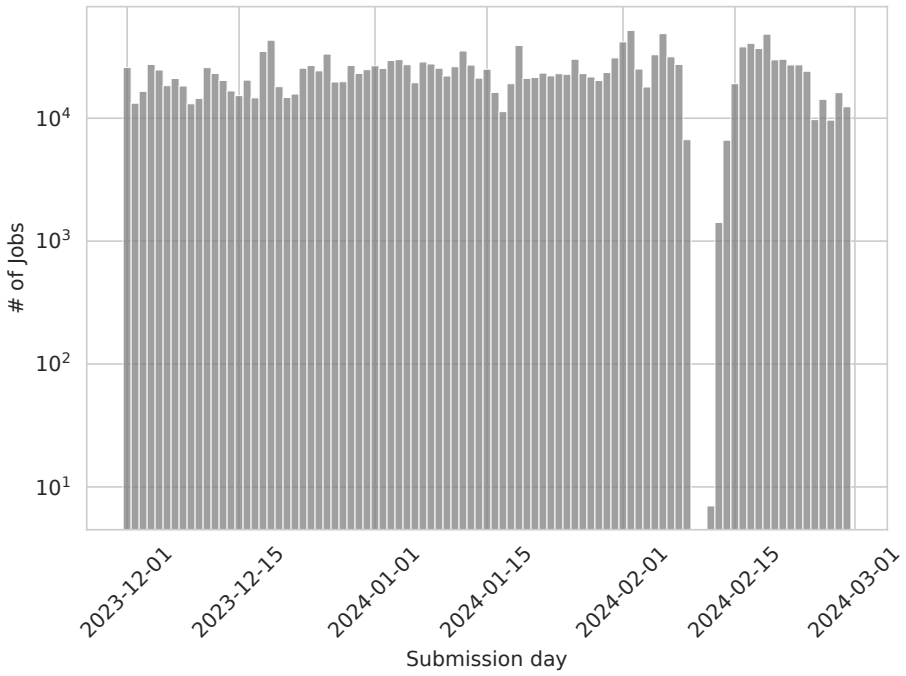

Figure 7.2: Job submission distribution over time.

information, and *system state*), as well as job execution and completion (such as *duration* and *performance counters*). The *performance counters* can be leveraged to compute performance metrics on the job execution, such as $\#flops$ and $\#moved\_memory\_bytes$.

Fugaku is used by hundreds of users, submitting thousands of jobs to the system every day. We extract from F-DATA the data of 2.2 million jobs submitted and executed between December 2023 and March 2024. Figure 7.2 shows the distribution of the jobs over the entire period. We observe that the job submission rate is uniform except for a few days in early February, when a scheduled maintenance caused the shutdown of the system.

7.3.2 Job Characterization Setup

Following the methodology presented in Section 7.2.3, we extract the peak performance and peak memory bandwidth of a Fugaku node from system's spec-

ifications.[3], which are 3380 GFlops/s in FP64 and 1024 GByte/s, respectively. The peak performance reported refers to FX1000 boost-mode configuration (i.e. frequency = 2.2 GHz for the A64FX CPUs), as we need to consider the best performance attainable by the machine. Based on these characteristics, the ridge point is at an op_r of $\approx$ 3.3 Flops/Byte, which is used for the job labelling.

As described in Section 7.2.3, for job j we compute p_j and mb_j (and consequently op_j), via $\#flops_j$ and $\#moved\_memory\_bytes_j$. These two values in the target system are computed starting from the performance counters ($perf2$, $perf3$, $perf4$, $perf5$). In Fugaku, $perf2$ and $perf3$ correspond to the FP_FIXED_OPS_SPEC and FP_SCALE_OPS_SPEC A64FX_PMU_Events, while $perf4$ and $perf5$ refer to BUS_READ_TOTAL_MEM and BUS_WRITE_TOTAL_MEM [70]. In Equation 7.4, $perf2$ is the fixed amount of operations, while $perf3$ is the number of operations per 128-bit SVE, and it is multiplied by 4 since the A64FX of Fugaku is 512-bit SVE. In Equation 7.5, $perf4$ and $perf5$ are summed in order to obtain the total number of requests to the memory, as they represent the amount of memory read and write requests, respectively. Then, they are multiplied by the size of the memory requests, (256 bytes of cache line size), to obtain the total $\#moved\_memory\_bytes_j$. Moreover, the cores of Fugaku nodes are grouped by 12 in Core Memory Groups (CMGs). The $perf4$ and $perf5$ values are generated by summing all the values collected by each core for the whole CMG. Therefore, these values need to be divided by 12 to eliminate redundant information.

$$\#flops_j = perf2_j + (perf3_j * 4) \tag{7.4}$$

$$\#moved\_memory\_bytes_j = \frac{(perf4_j + perf5_j) * 256}{12} \tag{7.5}$$

Once we compute p_j and mb_j (and op_j), we label the job j based on the comparison of op_r of the ridge point and op_j.

[3] https://www.fujitsu.com/global/about/innovation/fugaku/specifications/

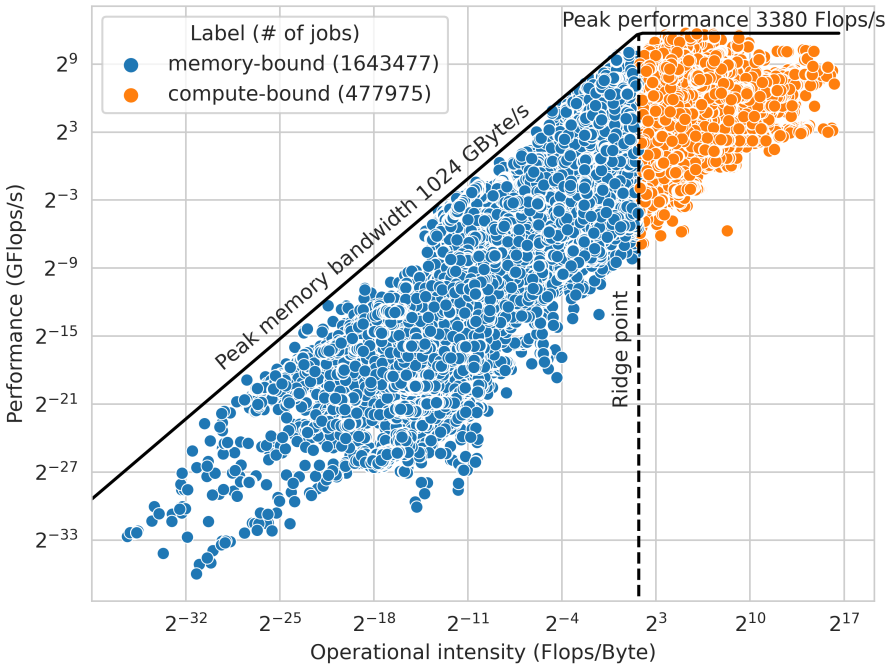

Figure 7.3: *Roofline* model of the job data.

7.3.3 Fugaku Job Analysis

Figure 7.3 shows the collective *Roofline* model, which reports in x-axis *op* measured as Flops/Byte, and in y-axis p in GFlops/s. We observe that the distribution of the operational intensity of the jobs submitted to the Fugaku system is significantly skewed toward values lower than the ridge point. Moreover, as reported in Table 7.1, the number of *memory-bound* jobs is around 3.5 times as the number of *compute-bound* jobs. Figure 7.4 reports the distribution of each job type over the entire period. We notice that the proportion between the *memory-bound* and *compute-bound* jobs is constant in time, suggesting that this difference is a characteristic of the studied Fugaku workload. This is interesting considering that the A64FX of the Fugaku system has been co-designed for *memory-bound* jobs [71], and thus, a more balanced job distribution would be expected.

We can also see that many jobs are far from the *Roofline*. This is particularly notable in the *memory-bound* area, where only a few clusters of jobs are close to the peak memory bandwidth line. The same can be observed in the *compute-*

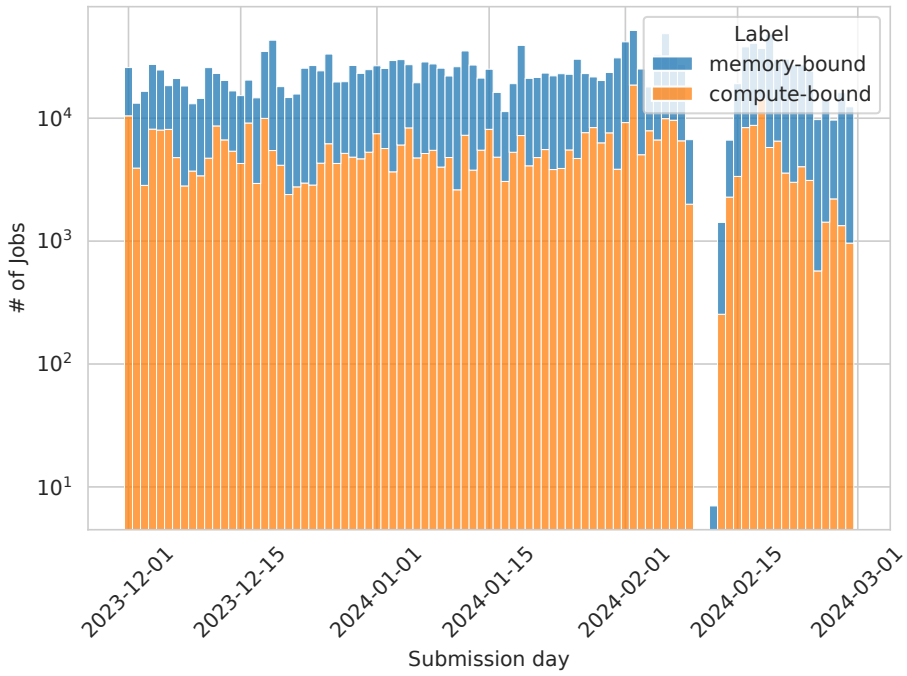

Figure 7.4: Distribution of job types over time.

bound area, where only some jobs with operational intensity around the ridge point touch the peak performance line. This means that while there are some well-engineered jobs saturating fully the resources, it is not the case for the majority of the jobs. Therefore, leveraging *MCBound* to classify *memory/compute-bound* jobs has the potential to guide job scheduling, for instance by enacting co-scheduling of *memory-bound* and *compute-bound* jobs on the same node, or by adjusting the amount of resource allocated to the job, and thus to reduce system resource wastage.

Figure 7.5 shows the distribution of jobs in the *Roofline* plane by highlighting the node frequency selected by the user at job submission time. In Table 7.1, we see that around 54% of the *memory-bound* jobs are executed in normal mode (frequency=2.0 GHz), while only around 30% of *compute-bound* jobs in boost mode (frequency=2.2 GHz). Moreover, Figure 7.5 shows that there is no observable correlation between the user-selected frequency at submission time and the position of the given job in the *Roofline*. These observations suggest that users do not nec-

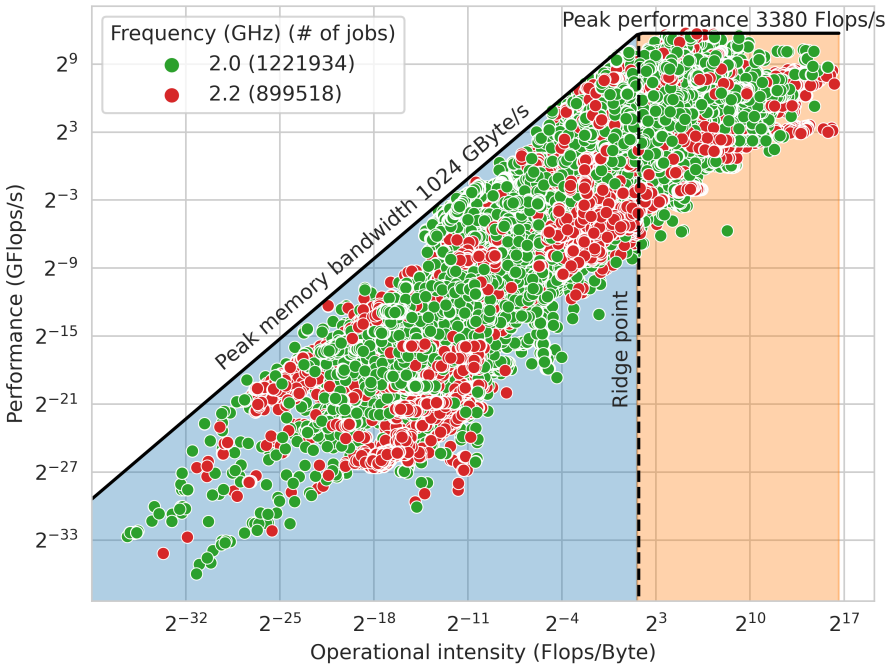

Figure 7.5: *Roofline* model of the job data, divided by frequency.

Table 7.1: Distribution of job types.

Frequency	*memory-bound*	*compute-bound*	Total
2.0 GHz (normal mode)	891,056	330,878	1,221,934
2.2 GHz (boost mode)	752,421	147,097	899,518
Total	1,643,477	477,975	2,121,452

essarily choose appropriate frequencies for their jobs. Indeed, *memory-bound* jobs do not benefit from running at higher frequencies, as their performance bottleneck is the memory bandwidth, while *compute-bound* jobs are likely to increase their performance at higher frequencies, possibly resulting in shorter execution time and energy savings. Therefore, another advantage of leveraging *MCBound* is the possibility to guide frequency selection, and thus the improvement of system energy efficiency and throughput.

7.4 Experimental Study

In this section, we present the implementation of the *Classification Model* of *MCBound* for Fugaku, experimentally evaluate the online prediction algorithm, and discuss the results.

7.4.1 *Classification Model* Implementation for Fugaku

We rely on the *scikit-learn*[4] library for the ML models and use their default implementation. The SBERT model is provided by the *sentence transformers* library[5], while the weights are pulled from Huggingface [6]. We use the pre-trained model *all-MiniLM-L6-v2*[7], since it has the best trade-off between prediction quality and speed [28]. The code we used for the implementation will be released in a public repository.

We conduct an initial empirical evaluation of the dataset to find the best set of features to represent the Fugaku jobs. The set should be representative enough for jobs to maximize the prediction accuracy while concise enough to minimize the runtime overhead in processing them. In our previous work on job power consumption of Fugaku jobs (Chapter 6), we found that the best set is composed of *user name, job name, #cores requested, #nodes requested* and *environment*. Our experiments confirm their value in our prediction task and that including the additional feature *frequency requested* improves the prediction performance. We therefore use *frequency requested* and those of Chapter 6 as augmented feature set.

As mentioned in Section 7.2, the *Inference Workflow* can be triggered periodically. For Fugaku, we do it once every β days, using the job data accumulated since the last trigger. We save the job characterizations and encodings of every trigger of the *Training Workflow* and *Inference Workflow*, in order to reuse them and avoid redundant computations during the future triggers of the *Training Workflow*.

The *Training/Inference Workflow*s are performed on a machine detached from Fugaku, accessible via HTTP calls. Thus, no overhead to the HPC computing

[4] `https://scikit-learn.org/stable/`
[5] `https://www.sbert.net`
[6] `https://huggingface.co`
[7] `https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2`

resources is incurred. As the data were already collected for logging purposes, the only additional storage required is for the saved trained model, which is negligible in today's HDD (around 1GiB)

7.4.2 Online Prediction Algorithm Evaluation

To evaluate the online prediction algorithm, we implement an `evaluate` Python script, which is executed once at the end of the testing period. This evaluation targets the assessment of the prediction quality as well as the incurred runtime overhead.

Evaluation setup We employ the two ML models, KNN and RF, described in Section 7.2.4. The models are trained on portions of the data of the jobs executed between December 1^{st}, 2023 and January 31^{st}, 2024 and tested on a subsequent time window composed of over 700,000 jobs executed between February 1^{st} and 29^{th}, 2024.

Prediction quality is measured using the F1-macro average score [63] - a widely used metric for classification problems - computed as the mean of the F1-score obtained on specific *memory-bound* and *compute-bound* classes. The F1-score on a single class is computed as the harmonic mean between the precision and recall on the target values. Hereafter, we will refer to the F1-macro average as F1. The F1 for a model is computed at the end of the testing period by our `evaluate` script, on all the predictions generated by all the *Inference Workflow* executions. The ground truth labels necessary for F1 have been acquired via Fugaku job data characterization, as described in Section 7.3.

The runtime overhead of the algorithm refers to the time spent in training and inference, which are computed as the average of all the *Training Workflow* and *Inference Workflow* runtimes. While job characterization time is negligible ($1 * 10^{-6}$ seconds per job), the encoding incurs a higher overhead ($2 * 10^{-3}$ seconds per job) which still is negligible. We note, however, that job encoding takes place during *Training Workflow* only once at the first deployment of *MCBound*. This is because there are no encodings of the historical job data at the beginning. As models are retrained during future triggers of the *Training Workflow*, the job

encodings can be retrieved from the previous *Inference Workflow* computations. Thus, we do not include the job characterization and encoding time in the training time, while we include the encoding time in the inference time.

Experimental setup We conduct three experiments. As described in Section 7.2, the online prediction algorithm retrains a model using the recently executed job data (the last α days' data), and continues to do so periodically (at every β days). In the first experiment, we use different combinations of α and β values, to find the best time window of the recent data for periodic retraining and retraining frequency. We iterate $\alpha \in \{15, 30, 45, 60\}$ and $\beta \in \{1, 2, 5, 10\}$. We avoid $\beta = 0$, i.e. retraining upon each new job submission, as it incurs excessive overhead, as well as exclude larger values of β so as not to delay model update for long.

We are not interested in using more than $\alpha = 60$ days of training data either, as otherwise the model would have to deal with a large amount of data during RF training and KNN inference, possibly increasing the runtime overhead. Moreover, the workload of an HPC system is variable and training based on "older" data is not beneficial for prediction. We demonstrate this in our second experiment, where the initial model training is done using the best α found in the first experiment, and then successively the model is retrained using the data of all the past days, without forgetting the data older than α days. We refer to this setting as α^+ time window.

To observe whether the amount of data used within a given α time window influences the prediction quality, in our third experiment we retrain the models using a θ subset of the last α days of data. To this end, we iterate θ *in* $\{10^2, 10^3, 10^4, 10^5\}$ after having analyzed the average training data size.

The experiments are run on the machine where the framework is currently deployed for testing purposes, which is equipped with two AMD EPYC 7302 CPUs, 64 cores and 512 GB RAM, running Python 3.11.5 on Linux Fedora 37. The code of the experiments will be released in a public repository.

7.4.3 Experimental Results

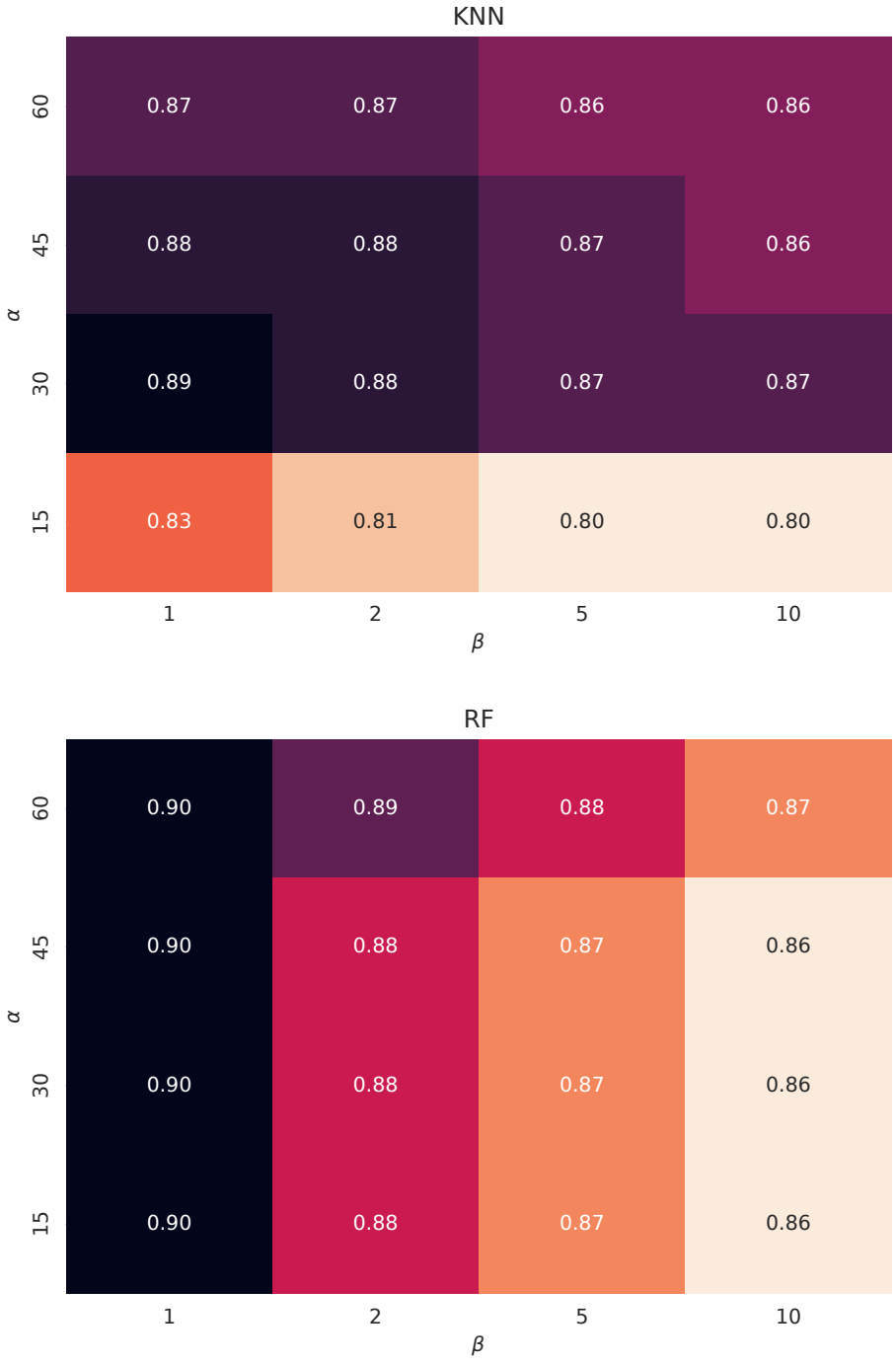

Figure 7.6: F1 of KNN (above) and RF (below) with different α and β values.

Experiments with α and β values Figure 7.6 shows F1 values over different combinations of α and β values. In both models, as β increases, F1 decreases, due to less frequent model training and knowledge update. We therefore consider the best retraining frequency as $\beta = 1$ (once a day). As α increases, the models behave differently. The parametric model RF tends to benefit from training using data spanning to a larger time window, probably because parameter tuning becomes more precise, but we observe no gains with $\alpha > 15$ when $\beta = 1$. Whereas, the non-parametric nature of KNN does not benefit from "older" data. In the specific case of $\beta = 1$, the best performance is attained with $\alpha = 30$ and then declines with greater values. We theorize that the workload has more similarities within 30 days. Given that KNN inference works by finding similar data, training on data older than 30 days infers past job behavior.

Figure 7.7 shows the average daily training time across various values of α. KNN training time is almost negligible, with a maximum duration of 0.32 seconds with $\alpha = 60$. In fact, KNN training consists of just building a model instance, which stores the training data for future inference, and no parameter is tuned. Conversely, RF requires an actual training phase with parameter tuning, and as α grows, so does the training time with the amount of data growing, up to almost 3 minutes. However, RF reaches the best prediction already with $\alpha = 15$, when the training time is lowest (around 26 seconds).

In Figure 7.8, we show the daily average inference time per job (including job encoding time) across various values of α. RF has a constant inference time, as inference is done through tuned parameters independently of α. Though this value is around $2 * 10^{-6}$, it is dominated by the average job encoding time ($2 * 10^{-3}$). KNN inference is about finding similarities among the entire training data, thus inference time would grow with larger values of α. However, the inference time is again dominated by the encoding time and is around $2.3 * 10^{-3}$, not changing much across different values of α. Still, the inference time per job of $\alpha = 30$ is negligible (milliseconds) w.r.t. job average waiting time for scheduling (time spent until the scheduling decision of a job, after its submission), which is around 3 minutes in the observed period. This means that neither of the models would incur overhead on the job submission workflow of the system. From Figures 7.6, 7.7 and 7.8, we can conclude that the best algorithm settings are $\alpha = 15$ (RF) and $\alpha = 30$ (KNN),

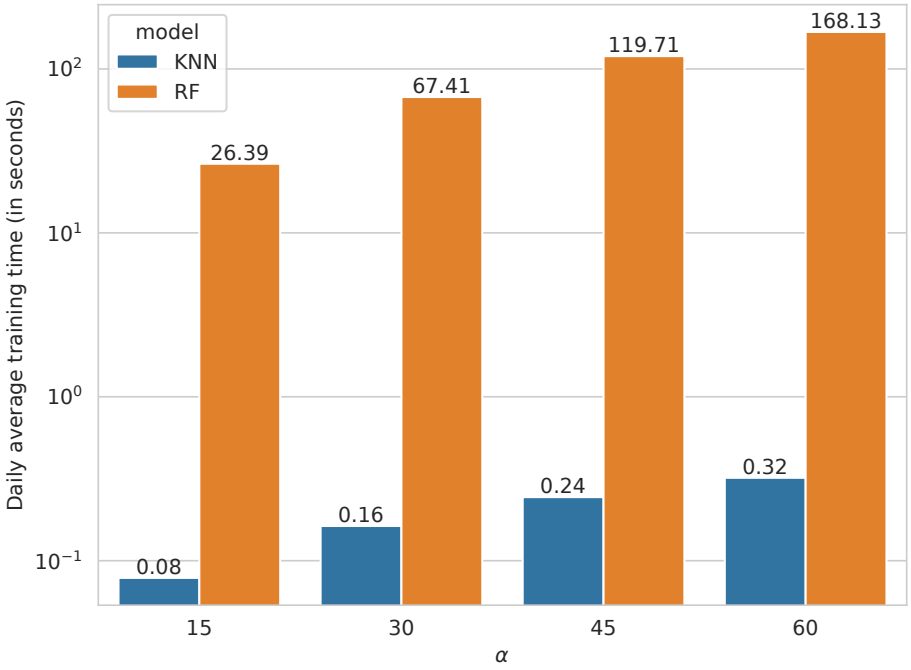

Figure 7.7: Average model training time variation when $\beta = 1$.

coupled with $\beta = 1$.

We further compare the RF and KNN to a simple baseline that maps a tuple of (*job name, # of cores requested*) to a *memory/compute-bound* label (which can be seen as a KNN with $k = 1$ on the features *job name, # of cores requested*). The baseline is updated over time using the same online algorithm, with $\alpha = 30$ and $\beta = 1$ (as the best KNN settings). While this solution is simpler, it is also less accurate than ours (F1-score: 0.83 vs 0.90), justifying the need for our approach.

Experiments with α^+ Starting with the best α and β value combinations as described in the previous experiment, we observe no improvement in prediction considering the α^+ time window during training. F1 of RF with α^+ is 0.90, which is the same as $\alpha = 15$. This is not surprising, as we saw in the previous experiments that increasing α does not change F1 when $\beta = 1$. F1 of KNN instead decreases to 0.86 with α^+ from 0.89 with $\alpha = 30$. This supports our hypothesis that jobs are most similar within 30-days.

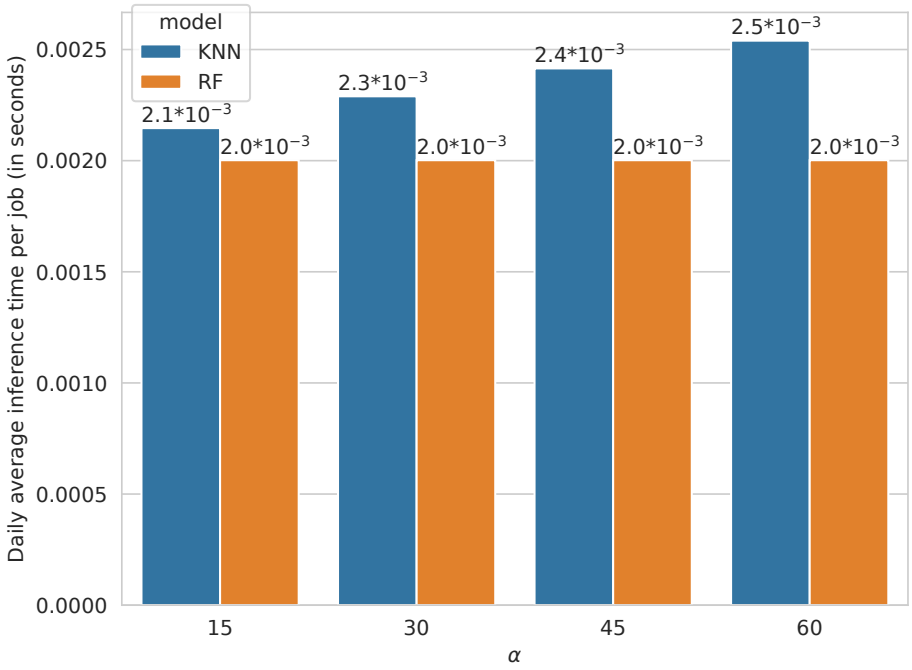

Figure 7.8: Average job inference time variation when $\beta = 1$.

Moreover, the growing time window jeopardizes the training time of RF and the inference time of KNN, as they are both dependent on the training data size. The average training time of RF increases from 26.39 seconds with $\alpha = 15$ to more than 200 seconds with α^+. Differently, the training time increase in KNN is marginal, going from 0.16 seconds with $\alpha = 30$ to 0.39 seconds with α^+. Conversely, while the average inference time per job of RF remains the same, the KNN time increases but slightly, going from $2.3 * 10^{-3}$ seconds per job with $\alpha = 30$ to around $2.5 * 10^{-3}$ with α^+.

This experiment confirms that a sliding time window, which filters the recent job data for retraining, is beneficial for the proposed online algorithm both for prediction accuracy and overhead on the system's operations. Therefore, in the following experiment, we fix α to its best values of 15 (RF) and 30 (KNN).

Experiments with θ For a given α retraining time window, we select a subset θ of data points either randomly, or by considering the jobs with the most recent

ending time. When sampling data randomly, we repeat model training with 5 different random seeds[8] and average the results of the 5 different trained models.

Figures 7.4.3 and 7.9 show F1 values of KNN and RF using latest and random data over different values of θ. We observe that having more data within a fixed time window improves the prediction accuracy for both sampling approaches, where the best result is obtained by using all the available data. Interestingly, random sampling is more effective consistently across all θ values. This can be attributed to the fact that Fugaku jobs are usually submitted in batches of identical jobs, and job data very near in time might lead to replicated data during training. A higher percentage of such replicated training data would result in a less general model, while sampling the data randomly smoothes this effect. Our hypothesis is supported by the fact that with smaller values of θ, i.e. from 10^2 to 10^4, the F1 difference between the two sampling approaches is significant (up to 0.26), while the gap reduces drastically (down to 0.02) with $\theta = 10^5$. In fact, the more jobs take part in training, the less batches of identical jobs would impact the model, as the percentage of replicated data would drop.

Discussion of results The best settings of the algorithm, in terms of retraining data time window α and retraining frequency β, are $\alpha = 15, \beta = 1$ days for RF and $\alpha = 30, \beta = 1$ days for KNN, using all the available training data. With these settings, we obtain accurate predictions (F1=0.90 for RF and F1=0.89 for KNN), at the expense of 26 seconds for RF and 0.16 for KNN daily average training time, and average inference time per job of $2.0{*}10^{-3}$ seconds with RF and $2.3{*}10^{-3}$ with KNN. As the average number of submitted jobs per day in the observed period is 25K, the daily overhead of model inference can be estimated as around 50 seconds for RF and 60 for KNN. The overall daily training and inference overhead is negligible w.r.t. to job average waiting time for scheduling, which is around 3 mins during our observation.

We conclude that regardless of the model used, the online prediction algorithm is suitable to be deployed in a production system, as it provides accurate predictions with negligible overhead on the job submission workflow of the system. We highlight that our approach can have a significant impact on the system power, en-

[8]The random seeds used for the experiments are 520, 90, 1905, 7, 22

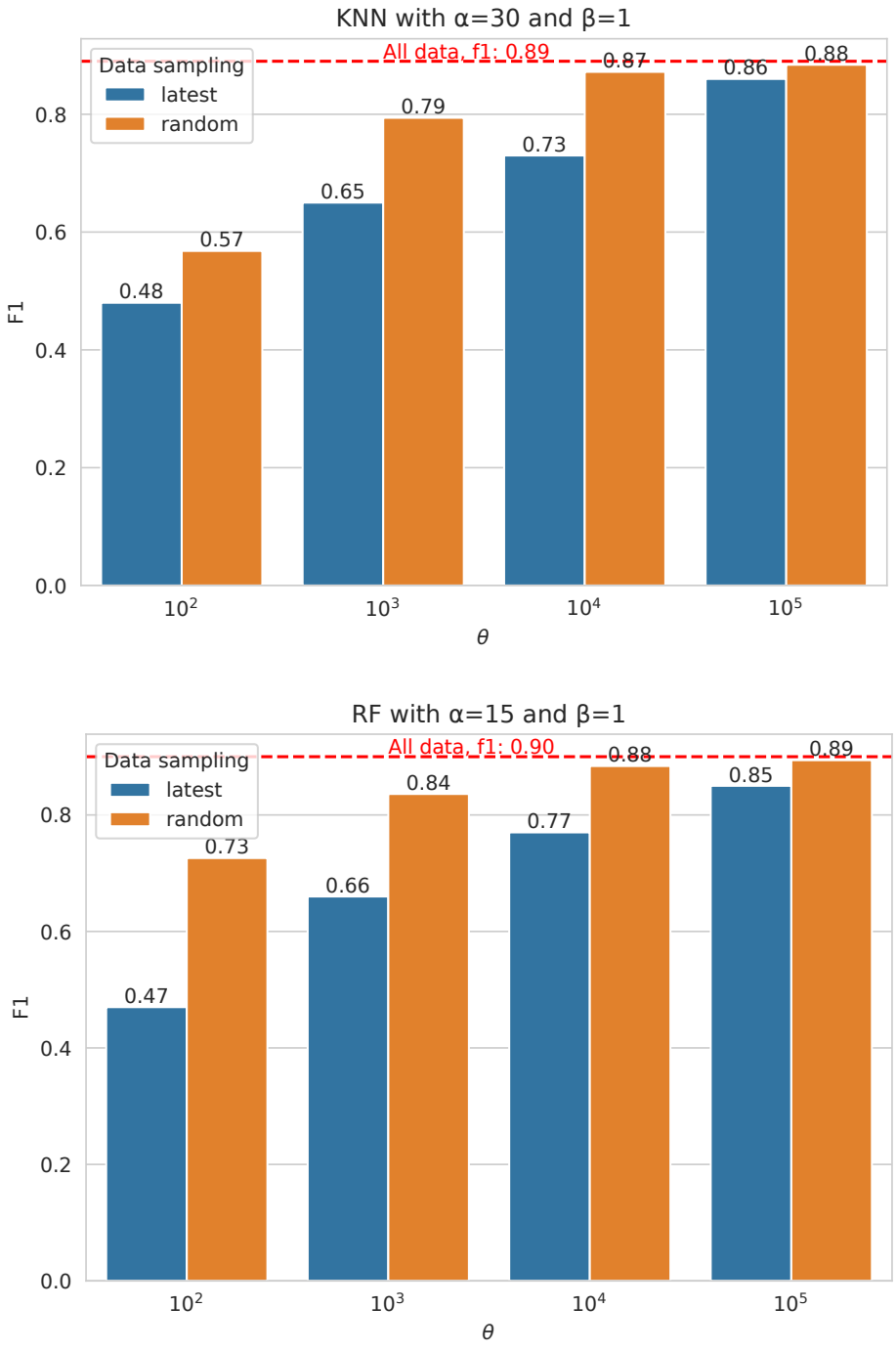

Figure 7.9: F1 of KNN (above), and RF (below) with different θ values.

ergy and performance. We estimate the impact based on previous work on job and frequency mode characterization [72]. The authors showed that using the boost mode on Fugaku for *compute-bound* jobs can reduce the job execution duration by 10% (with respect to normal mode), while using the normal mode for *memory-bound* jobs can reduce the job power consumption by 15% (with respect to boost mode). Our algorithm classifies 90% of the jobs correctly, hence we could perform semi-automatic frequency selection and obtain the following improvements. There are 750k *memory-bound* jobs executed in boost mode, with an average power consumption of 5000 W, and an average duration of 6000 seconds. By executing them in normal mode, we could have reduced the power consumption by around 680W per job, saving 450MW of power, and 14 GJoules of energy, at the system level. Moreover, there are 330k *compute-bound* jobs executed in normal mode, with an average duration of 13,500 seconds. By executing them in boost mode, we could have saved around 20 minutes of computation per job, and more than 1,700 hours of overall system computation.

We note that these kinds of improvements can be potentially obtained in any system where the nodes' frequency can be decided by the user, and thus our results are not limited to the Fugaku system.

Chapter 8

End-user Tool

The prediction algorithms presented so far are working at system-level, i.e. they employ just one model considering the data of all the jobs by all the users. While these approaches are valid, they present some limitations when employed in production environments. Such prediction algorithms rely on models which require training on voluminous historical execution data of jobs submitted by diverse users, as otherwise prediction effectiveness can be compromised [73]. In a real system, however, it is non-trivial to collect large amounts of data from multiple users. This can be due to privacy concerns [74] or difficulties in the data collection phase (caused by for instance different privilege levels requirements and monitoring software validation procedures) [75]. An observation is that in production environments, users tend to submit jobs performing similar operations (due to for instance running similar experiments) and consequently with similar power consumption [11]. Hence, job power consumption prediction using only the historical data of its own user is likely to improve accuracy, while eliminating the need for large amounts of data from various users. However, there can be significant variances in the job power consumption values of even a single user (due to for instance change of experiments), which renders the prediction task non-trivial, requiring a prediction model tailored to a user's behavior and job execution characteristics.

Another limitation of the existing approaches is that they are usually designed to be employed at the system level, without taking into account the end-users. From a user perspective, knowing the energy cost of their jobs is useful in systems

where power-based pricing strategies are in place [20]. It has been argued that to encourage the adoption of greener systems, the energy cost of jobs needs to be made explicit and accounted for in the pricing scheme [19]. Providing such information allows to raise end-users awareness on the environmental impact of their jobs, encouraging users to adopt more energy-efficient practices. For instance, if users are informed on the power consumption of their job, they could decide to submit it at a different time, or with a different configuration (e.g. amount of requested resources), to minimize the cost and the energetic impact of their workload.

To address the aforementioned limitations and challenges, we introduce a new online framework UoPC to predict the power consumption of jobs submitted to production HPC environments.[1] Our contributions can be summarized as follows:

- We design UoPC, which exploits ML-based predictive models. UoPC eliminates the need for model training and large amounts of data. It builds a separate and simpler model for each user by relying solely on the user's data, differing from the existing approaches.

- We provide an easy-to-use Python implementation of UoPC, which can be either used by the end-users as a stand-alone tool or integrated into the workload management system to enable power/energy-aware job scheduling strategies.

- We deploy UoPC for the Supercomputer Fugaku, a production HPC system hosted at the RIKEN Center for Computational Science, in Japan, and evaluate it on a very large dataset of job runs on Fugaku, obtaining a prediction error of only 10%, while incurring a small overhead on the standard workload submission workflow.

UoPC exploits KNN (Chapter 2.3) prediction algorithm, which is non-parametric and thus does not require model training. Again, the KNN is augmented with SBert to encode the textual features present in the job data. The framework streamlines the standard prediction process for a newly submitted job before its execution, requiring only the information available at job submission time and the

[1] `https://github.com/francescoantici/UoPC`

data of k (which can be as low as 5) previous job executions by the same user. This makes our approach more practical for production environments than the existing solutions. Moreover, the implementation requirements of UoPC transcend architectural boundaries, offering a solution applicable to different systems and workloads.

We tested the Fugaku deployment of UoPC on the job execution data of more than 1.3 million jobs submitted by more than 700 different users between February and May 2024. Our experimental results show that UoPC outperforms recent ML-based solution in predicting the *average* and *maximum* job power consumption while significantly reducing the overhead on the system operation and the amount of historical data needed. Our approach has also proven effective in predicting the system power consumption, obtaining an error of only 4%.

8.1 Related Work

The work related to job power consumption prediction is discussed in Chapter 6.1. All the cited works (including our solution in Chapter 6) train the model with a large amount of data which may not be always available, as discussed previously, especially per user. If we were to retrain the model, the approach of [11] (the only one building a model for each user) would be even more impractical in a real system, which are used by hundreds of users. Hence, training a model per user with a large amount of data would result in excessive overhead on the system operations.

In Chapter 6 and [12] there is an evaluation of the performance at the system level as well. Our approach is more accurate than that of [12], as they obtain an error around 9% against our 4%. Concerning our previous solution in Chapter 6, we obtain a similar error on the average system power consumption (around 4%), but UoPC is better for the maximum one (5% against our 4%).

All the cited approaches are designed to work at the system level, and they do not provide a tool for end-users. We instead enable end-users to estimate the power consumption of their jobs, while improving the prediction accuracy at the same time. Finally, they all build their supervised models by merging the data of all the users, while our approach keeps separate the data of different users, thus

preventing possible privacy concerns related to data sharing.

8.2 UoPC Framework

In this section, we describe UoPC's functioning. We first provide an overview of the framework, and then detail its components and implementation.

8.2.1 UoPC Overview

The UoPC framework is designed to predict job power consumption in a real HPC system, where jobs are submitted and executed continuously, and various information regarding job submission, execution and completion (referred to as job data) is streaming in time. In this context, the prediction for a job before its execution can be performed leveraging exclusively features available at job submission, together with the historical data of the jobs that are completed by that time.

The only implementation requirement for UoPC is the presence of a data collection infrastructure gathering the job execution data per user. Such an infrastructure should update the *user dataset* with the record of completed jobs, including all the features regarding job submission (such as *requested resources*, *user information*, *job name*), job execution and completion time (such as *duration*, *#nodes allocated*, and *power consumption*). The user should be able to interact with their dataset at any time. This is not a strict requirement, since modern systems are typically endowed with monitoring software that permits the collection of job data [76, 77, 78], including power consumption [79, 80]. Moreover, systems usually provide a user-friendly interface to this data [81]. For users who are not system admins (usually the great majority), such tools allow them to retrieve only their data, due to privacy and security concerns. The data we require are restricted to a single user, therefore both privacy concerns and technical difficulties in the data collection do not represent a problem.

Practically speaking, our framework works as an inference engine without having to train large statistical models. UoPC takes as input the data of a new job submission. The high-level functioning of its prediction algorithm for a new job is

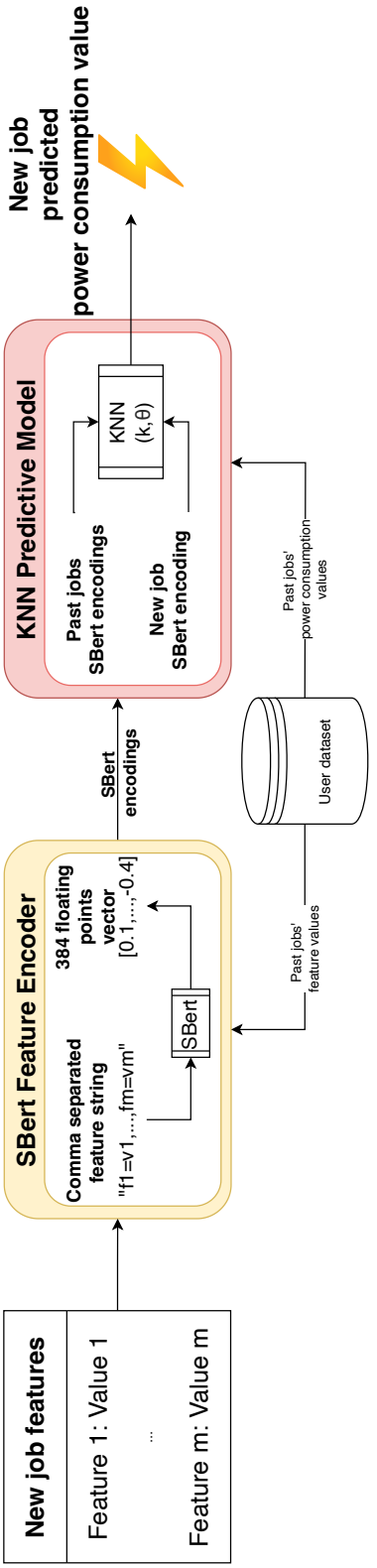

Figure 8.1: High-level functioning and components of UoPC.

presented in Figure 8.1. The main components of UoPC are:

- The SBert Feature Encoder, which takes as input a series of job feature values and returns the encoded data to be fed into the KNN Prediction Model.
- The *KNN Prediction Model*, which predicts the power consumption of a new job based on the encoded job data and the data of the past job executions of the same user.

UoPC is designed to work for any system where resources are allocated to job executions. We provide an easy-to-use Python implementation for UoPC, which can be used as a stand-alone tool by the end-user, or deployed in a workload management system. The components are software components implemented as Python classes, with a method for each functionality they provide.

8.2.2 SBert Feature Encoder

As job feature values are in a textual format, this component converts the job information into a standard numeric format suitable to be fed into the *KNN Prediction Model*. The conversion is performed by the *encode* method of the class, which takes as input a list of feature values describing the job at submission time (e.g. *"user_1","job_1","48","1","env_1,2000"*). Internally, the feature values are concatenated into a comma-separated string (e.g. *"user_1,job_1,48,1,env_1,2000"*) and encoded with an instance of a pre-trained SBert model.The final output of the *encode* method is a 384-dimensional floating-point vector (e.g. [-1.2,...,0.3]).

As shown in the previous chapters (Chapter 5, 6 and 7), an NLP encoding allows extracting more meaningful information from the job data, in the scope of predicting HPC job characteristics. We here identify additional advantages in terms of generality and data protection. Jobs submitted to different systems, or to the same system at different times, are likely to be described by different feature sets. SBert does not require a fixed feature set contrarily to typical ML models , thus can be used in a variety of settings. Additionally, we note that users tend to consider their job data sensitive and avoid its easy access and storage. Ideally, they prefer their data to be obfuscated and not available in a non-encoded format. SBert addresses this concern as it projects the information on a latent space.

Moreover, this step complicates the association between the original user-sensitive information and the encoded data used for training [44].

8.2.3 Predictive Algorithm

The prediction algorithm is based on the KNN model, and a specific model instance is built for each user. Predicting a job power consumption based on its user allows to maximize the accuracy of the KNN algorithm, while minimizing the amount of data needed, as explained before. Moreover, it also makes the framework suitable for systems where multiple user data may not be available (e.g. cloud or edge).

At initialization, the class requires the user dataset[2]. Then, it implements a *predict* method, which takes as input the SBert-encoded job submission data and generates a power consumption prediction for the job execution as follows:

1. The user dataset is queried to look for past job execution data.
   (a) If the dataset has at least k data points, the algorithm moves on to the next step. Otherwise, the prediction cannot be performed, and an error is returned; this is due to the constraint on the minimum number of data points required by KNN.
   (b) Conversely, if the dataset contains more than θ data points, we set a cap to the number of points that will be passed to the KNN module, to keep the inference time low (as applying KNN to a smaller set of points is computationally faster). To do that, we sort the data points by their completion time (w.r.t. the job arrival time) and keep the first θ points from the sorted list.
2. A KNN instance is built on the resulting past data, and it is used to generate power consumption prediction for the job execution.

The parameters k and θ are set by the UoPC user.

We focus on the job power consumption per node. Nevertheless, the component can easily be configured to predict power consumption at the CPU, GPU or

[2]We consider the user dataset to be accessible in a tabular data frame format, e.g. CSV, TSV, Parquet, JSON, etc.

memory level. Moreover, the energy consumption of a job can be computed as its average power consumption multiplied by its duration. Therefore, UoPC can also be used to estimate the job energy consumption per node as the predicted average power consumption multiplied by the duration predicted by the user.

UoPC can also be used to estimate the power consumption of the whole system at a given time t. The total power consumption of a job can be computed as the predicted job power consumption multiplied by the number of nodes allocated. By summing the total power consumption of all the jobs running concurrently at a given t, we obtain an estimate for the system. We note however that this estimation concerns only the job executions, as the total system power consumption depends also on other factors, such as cooling system or idle nodes power consumption.

8.2.4 UoPC Implementation

We provide an `install` script to install all the required dependencies at the time of the first deployment. Then, a `predict` script can be used to analyze job submission data and perform the prediction; the script takes as input either a series of comma-divided named parameters or a file from which to read the feature values. The frequency of the prediction depends on the use case. It can be called periodically (e.g. after a certain number of jobs are submitted) or when needed without any periodicity.

We provide a Docker [67] configuration, to distribute the framework as a container and make it scalable through container orchestration techniques, such as Kubernetes [68].

8.3 UoPC Deployment for Fugaku

In this section, we present the deployment of UoPC for the Supercomputer Fugaku to experimentally evaluate the prediction algorithm.

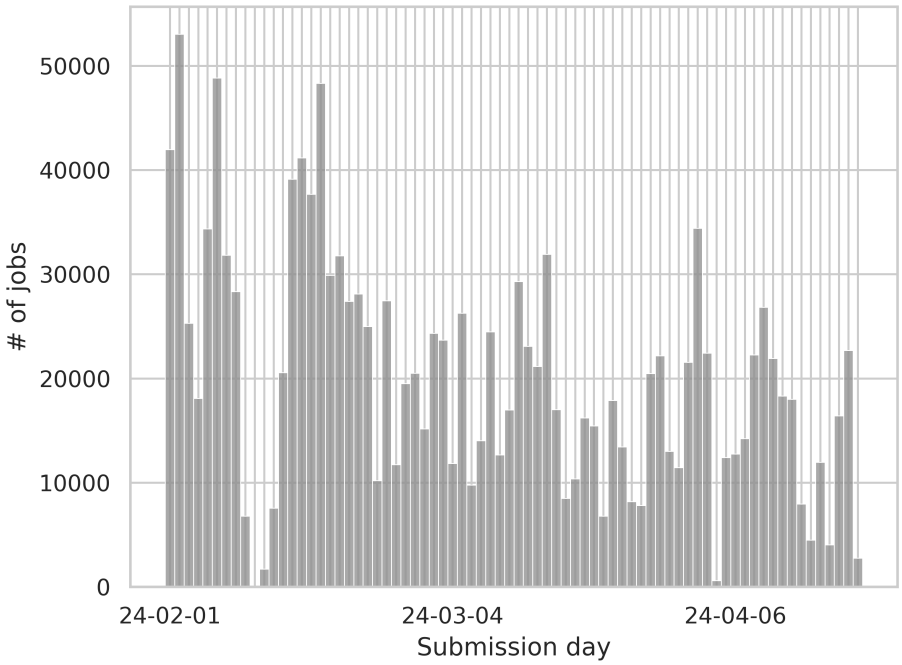

Figure 8.2: Distribution of jobs submitted to Fugaku between Dec.'23 and May'24.

8.3.1 Fugaku Dataset

To evaluate our prediction algorithm on real HPC jobs, we extract the data of almost 3 million jobs executed on Fugaku between December 2023 and May 2024, from F-DATA.

Figure 8.2 shows the distribution of the job submission in time. We notice that the number of submitted jobs is steadily higher than 10k per day, with a mean value of 20k jobs submitted per day. The only exceptions are in the first days of February and April, where scheduled system maintenance caused a system shutdown. In Figure 8.3, we show the distributions of the number of users, divided by their number of job data in the dataset. Out of more than 700 users, the great majority has less than 500 job traces. This justifies and favors our approach, since it can work well with a small amount of historical data.

As in Chapter 6, we consider the job power consumption recorded at the node level. We remind that Fugaku's job manager prevents node sharing among jobs, hence the power consumption value recorded on a node depends only on a single

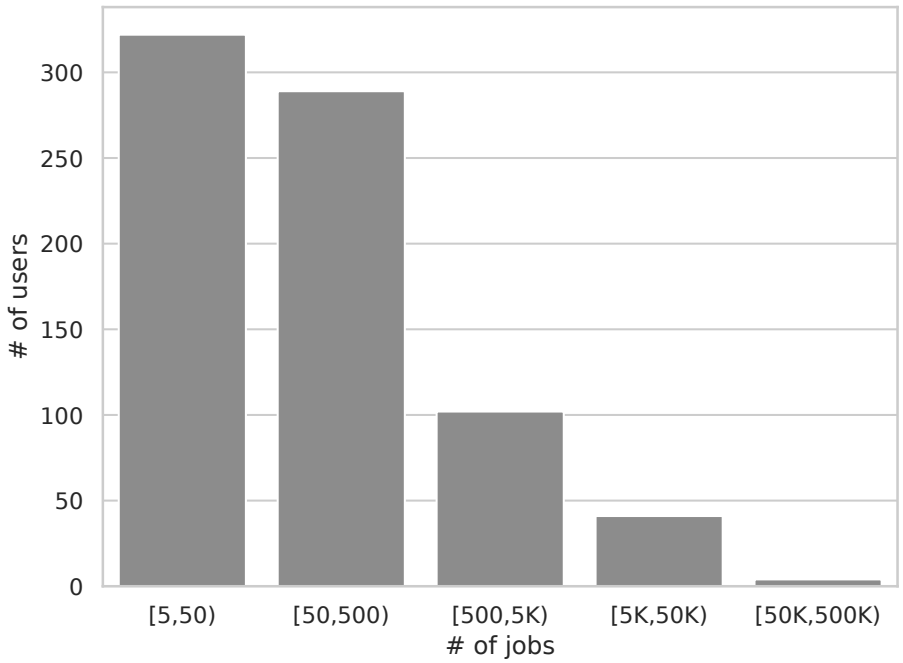

Figure 8.3: Distribution of users, divided by the number of jobs in the dataset.

job execution and there is no interference caused by other job executions (besides the typically shared resources like storage and network). The final value of the job power consumption is computed as the sum of the power consumption values recorded on all the nodes allocated to the job during its execution.

8.3.2 Data Preparation for Prediction

For our prediction task, we need to associate each job with a set of features representative of its characteristics. In general, the job power consumption depends on the computational operations performed and the amount of hardware used. Consequently, jobs performing similar operations and having a similar type and amount of hardware allocated, are likely to have similar power consumption. As argued in [82] and Chapter 6, features like the *job name* and *requested resources*, are key in identifying similar jobs, and consequently performing accurate job power consumption prediction. Since we work with the data of a single user, we exclude the *username* feature which is common to all the job data. The feature set we

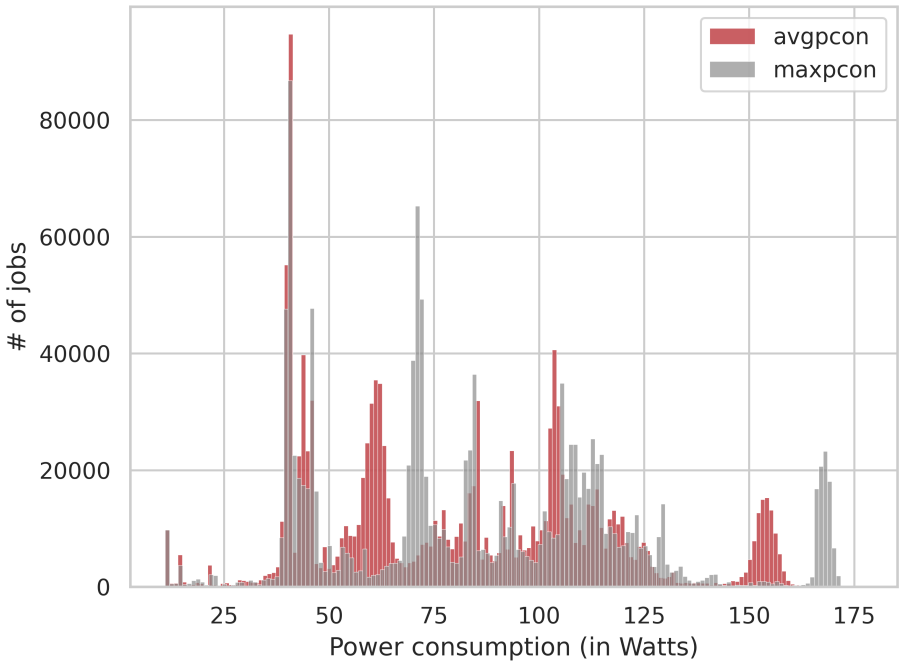

Figure 8.4: Distribution of average and maximum power consumption per node of jobs.

use to represent the job data is composed of *job name*, *# cores requested*, *# nodes requested*, *requested node frequency*, and *environment*. We chose this set based on our previous work on the task (Chapter 6), and upon empirical evaluation of which feature set yields the best prediction.

As for the prediction target, we focus on the average and maximum job power consumption, as in Chapter 6. Such values are normalized on the number of nodes, obtaining *avgpcon* and *maxpcon*. This allows us to predict the power consumption of each job as if it were running on a single node and makes the prediction task less error-prone. This strategy was adopted in past work [12, 15] and proven useful, especially in the context of workload scheduling [38, 83].

In Figure 8.4, we show the distribution of the *avgpcon* and *maxpcon* per job. We notice that the majority of the values lies in the range of 40W-110W, with a maximum value located around 40W. For values greater than 140W, the *avgpcon* is shifted to the left of *maxpcon*. This is explainable by the fact that for a single

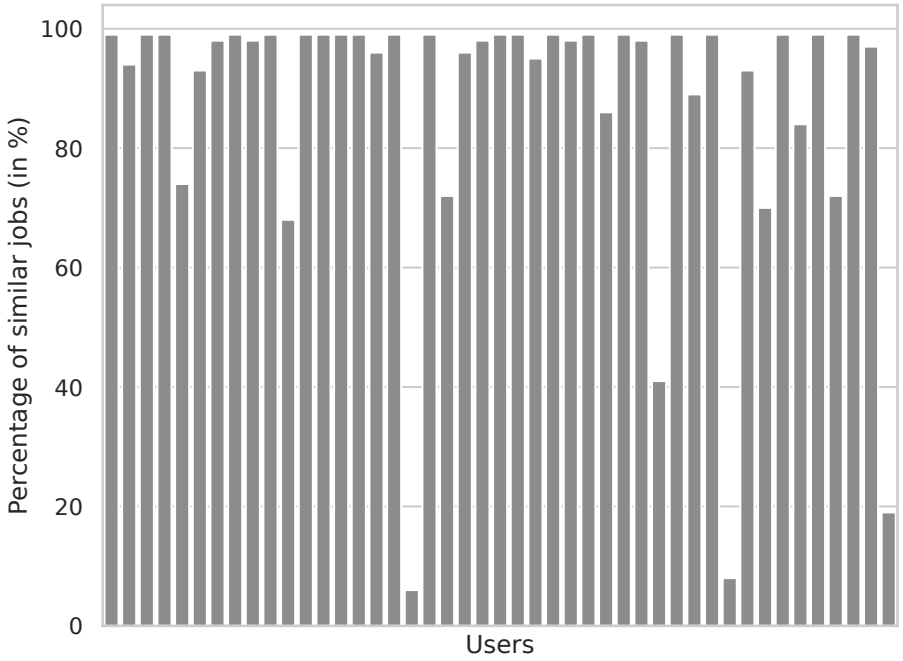

Figure 8.5: Percentage of similar jobs for the top 45 users.

job, the *maxpcon* is always greater or equal to the *avgpcon*. Thus, jobs may reach a high peak of power consumption (maximum), while maintaining a lower average power consumption throughout their execution.

Earlier in this chapter we argued that while users tend to submit similar jobs with similar power consumption values, a significant variance may appear across these values. Figures 8.5 and 8.6 reveal that this is the case for the Fugaku users and their jobs. In both figures, we show only the top 45 users (out of more than 700), i.e. the users with the largest amount of job data in the dataset. This is done to ease the readability of the plot, and also because such users submitted $\sim$ 85% of all the jobs executed on the system in the studied period, meaning that they are a relevant sample for analysis. Figure 8.5 shows the percentage of similar jobs per user. We say that two jobs of a user are similar if they have the same *job name*, *# cores requested*, *# nodes requested*, *requested node frequency*, *environment* (following the features decided earlier to represent similarity), and a difference of less than 5 Watts in the *avgpcon* and *maxpcon* values. We observe that the

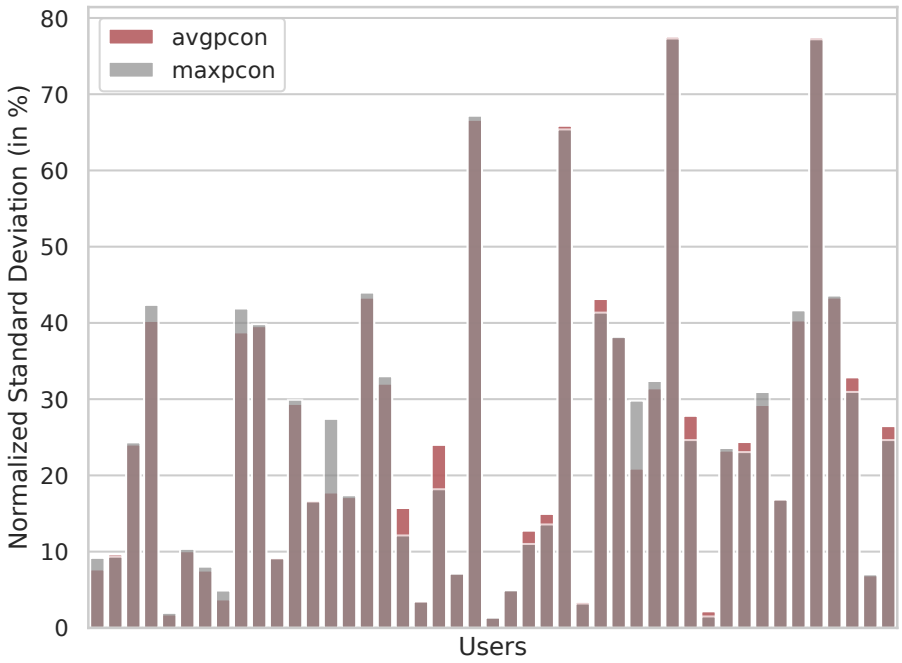

Figure 8.6: Normalized tandard deviation (on the average value per user) of the power consumption values for the top 45 users. The two bars of a user are placed one in front of the other.

majority of the users submits more than 80% of similar jobs, which supports the idea of user-based prediction.

Figure 8.6 instead shows the normalized (on the average value per user) standard deviation of the *avgpcon* and *maxpcon* values per user. We notice that for some of the users with many similar jobs, the job power consumption values vary a lot (up to more than 70% of the normalized standard deviation). Conversely, some users with few similar jobs have indeed low values of normalized standard deviation. We note that exogenous factors (e.g. room temperature) do not significantly affect power consumption of Fugaku jobs [84]. Hence, the normalized standard deviation shown in Figure 8.6 depends solely on endogenous factors, such as the resources allocated to the job or the operations it performs.

These observations confirm that the prediction task is indeed non-trivial and that a proper prediction model tailored to a specific user's behavior and job execution characteristics is necessary.

8.3.3 Online Prediction Algorithm Implementation

We rely on the *scikit-learn*[3] Python library for the ML models and use their default implementation. The SBert model is provided by the *sentence transformers* library,[4] while the weights are pulled from Huggingface.[5] We use the pre-trained model *all-MiniLM-L6-v2*[6] since it provides the best trade-off between prediction quality and speed [28].

To decide the k and θ parameters of the prediction algorithm, we experimented with different values of θ ($50, 100, 200, 500, 1000, 2000$, and 5000) and k ($5, 10, 20$, and 50). We observed that, for any value of θ, $k = 5$ always yields the best prediction performance. Conversely, the prediction accuracy stops increasing for $\theta > 500$, while the inference time grows up to several seconds. Hence, we set $k = 5$ and $\theta = 500$. UoPC is implemented and executed with Python 3.12, and the experiments are run on a machine equipped with two AMD EPYC 7302 CPUs with 64 cores and 512 GB of RAM.

8.4 Experimental Study

In this section, we experimentally evaluate the UoPC online prediction algorithm and discuss the results.

8.4.1 Online Prediction Algorithm Evaluation

Metrics To evaluate the prediction quality, we compute the Mean Absolute Percentage Error (MAPE), and the R^2 score, between the actual value (prediction target) and the predicted job power consumption value. The MAPE is the mean of all absolute percentage errors between the predicted and the actual values. Conversely, the R^2 score represents how much of the variation in the target values is predictable from the model. R^2 is not suitable to evaluate the numerical error, but it is meaningful to evaluate the adaptability and quality of the prediction

[3] `https://scikit-learn.org/stable/`
[4] `https://www.sbert.net`
[5] `https://huggingface.co`
[6] `https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2`

model. Generally speaking, an accurate regression model for a job should obtain a MAPE lower than 20% [85, 86], and an R^2 of at least 0.50 [11].

We also evaluate the prediction overhead to see how the algorithm impacts the system workload submission workflow. We define overhead as the computational time incurred by the prediction algorithm. The overhead of any ML-based prediction algorithm is composed of training and inference time. We define training time as the average time needed to train the model on the past data, and inference time as the time required to perform a prediction for a single job. When training is repeated periodically, it contributes to the overhead. A practical algorithm should incur a small overhead on the system operations, to be seamlessly integrated in the default workload submission workflow. A proxy to measure the overhead of our algorithm is to compare its time requirement to the average job waiting time (the time spent between the submission and the insertion in the job execution queue).

Baselines From the related work discussed in Chapter 6.1, we can conclude that RF is the the most used and effective model to predict job execution characteristics. The best results for power consumption prediction have been obtained with the online algorithm presented in Chapter 6. The algorithm involves a daily rebuilding of the model, on the data of the jobs executed in the last 60 days, from all the users. We employ both RF and KNN (the predictive model of our algorithm) in this algorithmic setting and refer to these baselines as A-RF and A-KNN, with "A-" meaning all users' data. To support that our approach would incur significant runtime overhead if the user-based models needed training, we also employ RF as predictive model in our algorithm, which we refer to as U-RF, with "U" meaning user-based. The algorithm of UoPC thus corresponds to U-KNN. Finally, we consider a simple baseline which predicts a job's power consumption by fitting a Linear Regression (LR) model (see Chapter 2.3) on the power consumption of the last k (i.e. 5) jobs executed by the same user.

Testing on historical data To test on historical data, we use the time information provided by the *submit_time, start_time* and *end_time* features to simulate the actual timeline of job submission and execution on a machine. We use this temporal information also to guarantee that the data used for model building always

comes before in chronological order the data of the test set. For a fair comparison, we evaluate both approaches on the same test set. Since the baseline requires 60 days of historical data also during the first training, the test set starts with the jobs submitted as of February $1st$.

For the UoPC algorithm, we iterate over all the job data and for every job j we perform prediction. To this end, we create the user-specific dataset by selecting the jobs belonging to the same user that were completed before the submission of j. This guarantees a realistic set-up where the prediction for a given job cannot be made with the data of future jobs.

Since both approaches require the SBert data encoding, we create them initially for all the jobs executed before February $1st$. Then, when we test a job j, we save its encoding to be retrieved for future training and prediction. This is done not to increase the overhead, as the encoding time for a single job is negligible ($2 * 10^{-3}$ seconds), while it can be significant for a set of training data. A similar time-saving mechanism can be enacted when UoPC is deployed in a real system.

We employ both approaches to predict *avgpcon* and *maxpcon* for every job, keeping them as two distinct learning tasks. Finally, we compute the evaluation metrics on all the predictions performed on the test set data.

System power consumption We evaluate the prediction quality also at the system level, as done in Chapter 6, to understand how accurately the prediction algorithm can reconstruct the system power consumption, and thus, whether it can be an effective tool to predict the whole system power consumption by analyzing only the submitted jobs. As discussed in Chapter 6, the prediction error at a single job level can be either an overestimation or an underestimation of the actual power consumption. Hence, such errors might cancel out each other at the system level, given the large number of jobs running concurrently.

We consider all the jobs for which we computed the predictions, and we group them based on the hour of the day when they were running. For all the hours of all the days $d \in D$, we compute the system power consumption as the sum of the actual power consumption of all the running jobs. Then, we compute the daily system power consumption $psys_d$, as the average of all the hourly ones. By replacing the actual job power consumption with the generated prediction,

multiplied by the number of nodes allocated to the jobs, we can also compute the predicted system power consumption $\overline{p}sys_d$.

Differently from Chapter 6, we evaluate the numerical error of the predictions by computing the *mean error* score in Equation 8.1, as defined in [12]. In addition, we also calculate the R^2 score.

$$mean\ error = \frac{1}{|D|} \sum_{d \in D} \frac{|psys_d - \overline{p}sys_d|}{psys_d} * 100 \tag{8.1}$$

8.4.2 Experimental Results

Job power consumption prediction In Table 8.1, we show the prediction performance of UoPC (U-KNN) and the baselines defined in Section 8.4.1. We exclude LR from the table as it obtains a very high MAPE ($> 100\%$). As expected, U-RF turned out to be computationally expensive and did not terminate within a week. We conclude that the method is not suited to a production environment. As A-RF outperforms A-KNN both in prediction performance and overhead, from now on we consider A-RF as the only baseline. While both UoPC and A-RF lead to accurate power consumption prediction, our algorithm outperforms the baseline for both the *avgpcon* and *maxpcon* prediction tasks. In the *avgpcon* task, our algorithm improves the MAPE score of the A-RF by 40%, going from 14% to 10%. It also improves the R^2, from 0.76 to 0.80. For the *maxpcon*, the MAPE of our approach is stable at 10%, while it slightly increases to 15% for the A-RF baseline. The R^2 of our approach is 0.01 lower in the *avgpcon*, however, it is steadily higher than the baseline value of 0.76. These results show that our approach not only obtains a very low prediction error in general terms but is also more accurate than A-RF.

In Figures 8.9-8.8, we compare UoPC against A-RF across single users. Figure 8.7 shows the distribution of the MAPE values of the approaches on all the users. We observe that the majority of the MAPE values obtained by UoPC is less than 10%, while A-RF obtains generally higher MAPE values. This means that overall, A-RF obtains a higher error w.r.t. UoPC. This is highlighted also in Figure 8.8, where we show the MAPE obtained by the two approaches on the top 45 users (as in Figure 8.6). Here we notice that except for a couple of users (where the A-RF

Approach	*avgpcon*				*maxpcon*			
	MAPE (%)	R^2	Avg Train Time (s)	Avg Inf Time (s)	MAPE (%)	R^2	Avg Train Time (s)	Avg Inf Time (s)
User based (U-)								
UoPC (U-KNN)	**10**	**0.80**	**0**	**0.08**	**10**	**0.79**	**0**	**0.08**
U-RF	-	-	-	-	-	-	-	-
All data (A-)								
A-KNN	16	0.60	0	9	17	0.58	0	9
A-RF	14	0.76	2080	0.13	15	0.76	2080	0.13

Table 8.1: Results for the *avgpcon* and *maxpcon* prediction tasks. For MAPE lower values indicate better results, while for R^2 higher values are preferred. For training and inference time, the lower the better. A "-" means that no result was obtained in a week of computation. The best results are highlighted in bold.

error is anyway similar), UoPC obtains a significantly lower error on the single users.

The high R^2 score obtained by our approach outlines its capability to accurately predict the variability in the power values. To observe this, we split the jobs based on their actual power consumption into power ranges of 20W. We then plot in Figure 8.9 the distribution of the ground truth and the predicted values in each range using both approaches. We observe that the UoPC distributions are more aligned with the ground truth w.r.t. to the A-RF ones. This is particularly noticeable in the ranges [100, 140) and [20, 40), which is also the most populated range, as shown in Figure 8.4. Both A-RF and UoPC struggle in the ranges [160, 180) for the *avgpcon* task, and [140,160) for the *maxpcon* prediction tasks. This can be explained by the low density of data in such areas, as shown in Figure 8.4. Nevertheless, UoPC still approximates the distribution better than A-RF.

Amount of data required In Figure 8.10, we show the average amount of job execution data per month used by the prediction algorithms. The amount used by the UoPC algorithm is computed by summing the dimensions of all the user datasets used during prediction. We observe that the amount needed by the A-RF is always over 1.1 million job traces, while UoPC requires around 800k data points. Even with fewer data, UoPC outperforms A-RF.

Overhead In Table 8.1, we also compare the overhead of the two approaches. Since the A-RF baseline requires daily retraining, we report the average daily training time. Our approach does not require a training phase, and the only overhead is the inference time. We observe that the training overhead of the A-RF is non-negligible, as it is more than half an hour per day. Thus, our approach saves time and resources, while resulting in a more accurate prediction.

The inference time of the two approaches is comparable and negligible compared to the average job waiting time in the system (up to 3 minutes). However, our algorithm still manages to predict in almost half of the A-RF time, 0.08 seconds against 0.13. Considering an average of 20k job submissions per day, if both approaches were to be deployed in a real system to analyze all the jobs systematically, the average daily inference time would be around 45 minutes for the A-RF

and around 25 for UoPC. We note that these numbers reflect the worst-case scenario where 20k jobs arrive in the system simultaneously and need to be processed right away.

System power consumption prediction Figure 8.11 compares the estimated system power consumption values with UoPC and A-RF to the actual values for both the *avgpcon* and *maxpcon* tasks. The drops in values are due to the system shutdown, as can be seen also in Figure 8.2. We observe that UoPC outperforms A-RF both in terms of *mean error* and R^2 in both tasks. It obtains an R^2 greater than 0.97 and an error smaller than 5% for both tasks, meaning that our approach is accurate enough to be used to predict the system-level power consumption. We note that UoPC's purpose is not to predict system-level power consumption; however, given its accuracy, such a prediction can be instrumental in guiding power-aware scheduling strategies in power-constrained systems.

8.4.3 Discussions

Our experimental results show that, with respect to the state-of-the-art ML methods using all users' data, UoPC obtains better predictions (lower MAPE), while incurring a smaller overhead and requiring less amount of data. An important consideration is that even if the improvement in the MAPE value is low, it can have a big impact when the prediction is used, for instance, to estimate the energy consumption of a job. In the Fugaku data, the average job duration is 10k seconds, while the average job power consumption is 5576W. If we consider job energy consumption as job power consumption multiplied by its duration, a prediction more accurate by just 1% would lead to a more precise job energy consumption estimation of around 156Wh. Considering this at the system scale, where on average 20k jobs are submitted per day, the improvement reaches around 3,000kWh. To put this number in perspective, the average US household energy consumption is 30kWh per day [87]. Hence, an improvement of 4% and 5%, respectively, in job power consumption prediction tasks, can be a highly significant achievement. For instance, when using a more accurate model in a scheduling algorithm, the scheduling decisions would allow for more energy savings and better performance

Approach	*avgpcon*						
	MAPE (%)	R^2	Avg Train Time (s)	Avg Inf Time (s)	MAPE (%)	R^2	
User based (U-)							
UoPC (U-KNN)	**18**	**0.11**	**0**	**0.07**	**22**	**0.13**	
U-RF	-	-	-	-	-	-	
All data (A-)							
A-KNN	22	-0.1	0	0.18	27	-0-03	
A-RF	20	0.07	400	0.12	24	**0.13**	

Table 8.2: Results for the *avgpcon* and *maxpcon* prediction tasks on PM100 data. For MAPE lower values indicate better results, while for R^2 higher values are preferred. For training and inference time, the lower the better. A "-" means that no result was obtained in a week of computation. The best results are highlighted in bold.

for each job execution.

We validated our approach also on PM100, presented in Chapter 3. In such data, the job power consumption data includes the GPU power contribution, and we tested UoPC on predicting the total power consumption, not just the CPU's power usage. Such values span from a few watts to more than 1kW, making the prediction task significantly more challenging. A summary of the obtained results is shown in Table 8.2, similarly to those presented in Table 8.1. Again, our algorithm improves A-RF (which is again better than A-KNN). In the *avgpcon* task, UoPC obtains a MAPE of 18% against a MAPE of 20% in A-RF, while not requiring model training and just using a fraction of the data. UoPC also obtains a better R^2 score (0.11 against 0.07 of A-RF). For the *maxpcon*, while the MAPE of our approach increases to 22%, it is still lower than 24% of A-RF, with the R^2 value being 0.13 in both cases. These results confirm that UoPC is a valid approach for other systems as well, including those with GPUs. UoPC does not require any structural changes to be adapted to other systems; only a preprocessing step is needed to account for potential differences in the job feature sets.

One limitation of our approach is that a prediction cannot be performed until the kth job execution of a user. However, k can be set to lower values to minimize this inconvenience. Alternatively, UoPC can be coupled with a simple linear

regression model or with A-RF (when it is possible to obtain all the user's data), until the user executes the kth job. Another limitation could be that our approach has been tested on jobs executed in exclusive mode. However, as described in Chapter 3, no current method can accurately determine jobs' node power consumption when jobs run concurrently, hence no public dataset offers such data. If future monitoring of resources enables precise job attribution, UoPC could be easily modified for this goal.

Finally, we note that UoPC can be seamlessly modified to predict other job features, such as *job failure, duration*, and *performance metrics*. These predictions can be used in conjunction with power consumption prediction to make informed decisions on job scheduling.

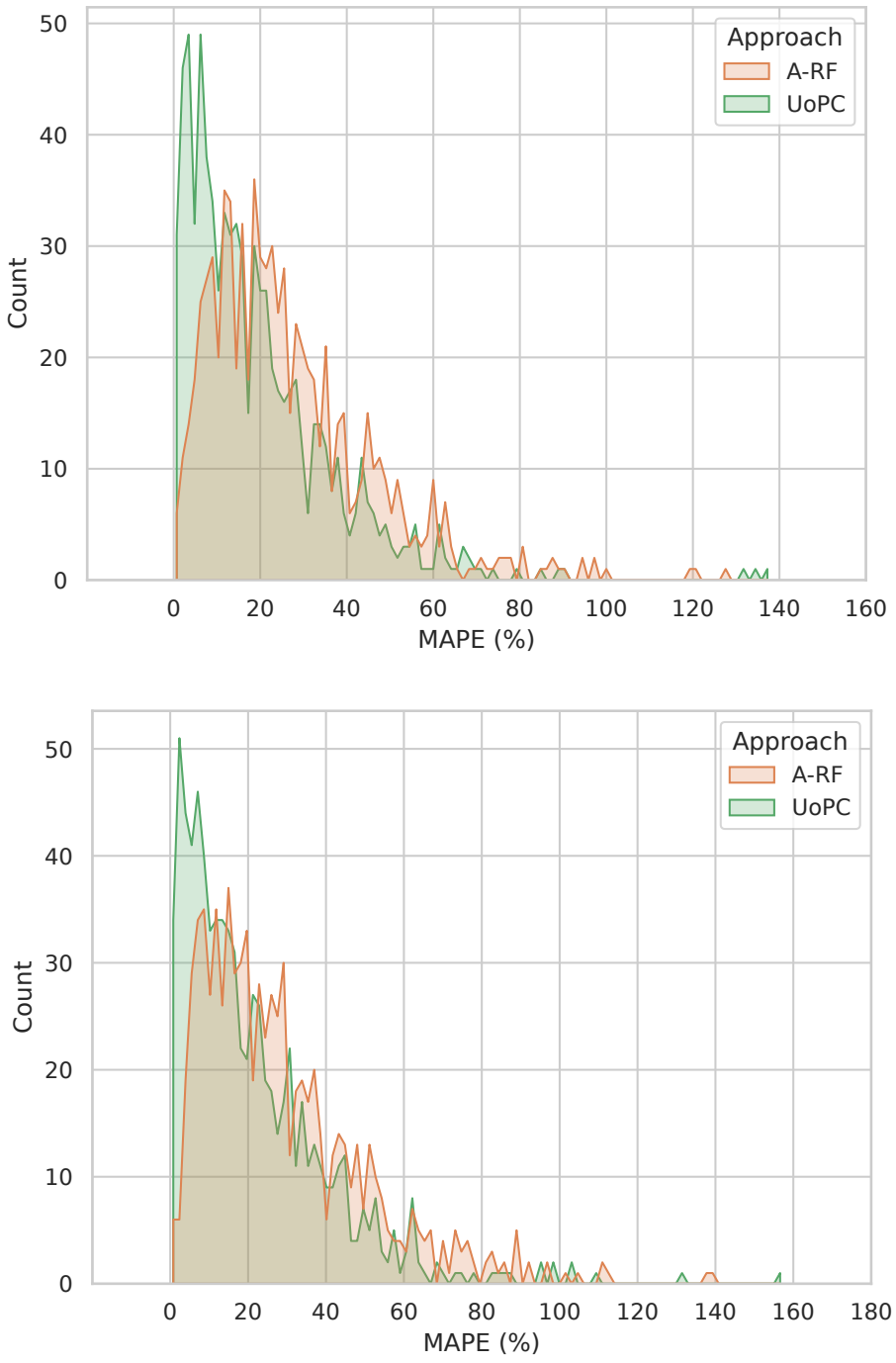

Figure 8.7: Distribution of the MAPE values per user for the *avgpcon* (above) and *maxpcon* (below) prediction.

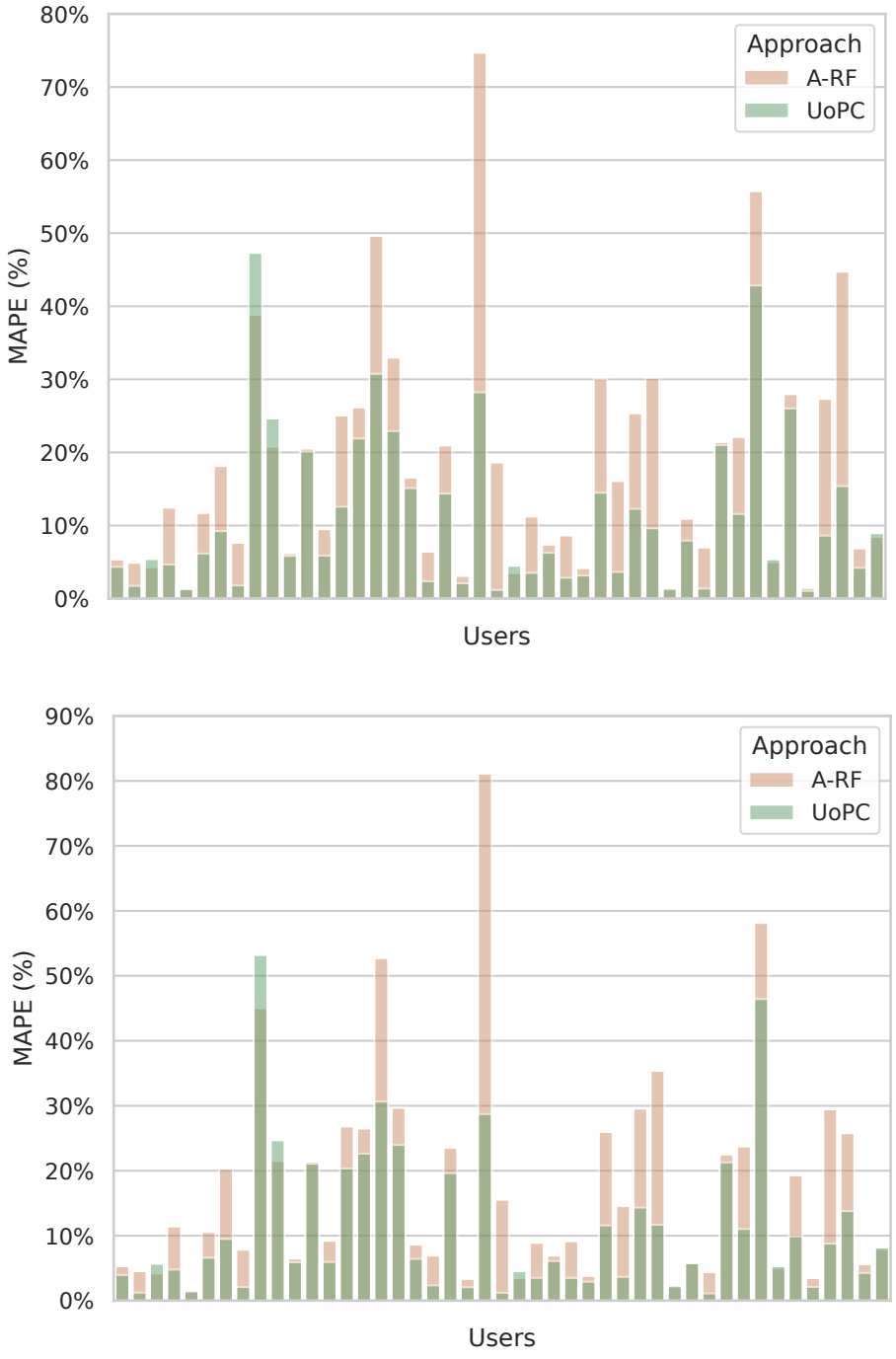

Figure 8.8: MAPE per user of the *avgpcon* (above) and *maxpcon* (below) prediction for the top 45 users. The bars are placed one in front of the other.

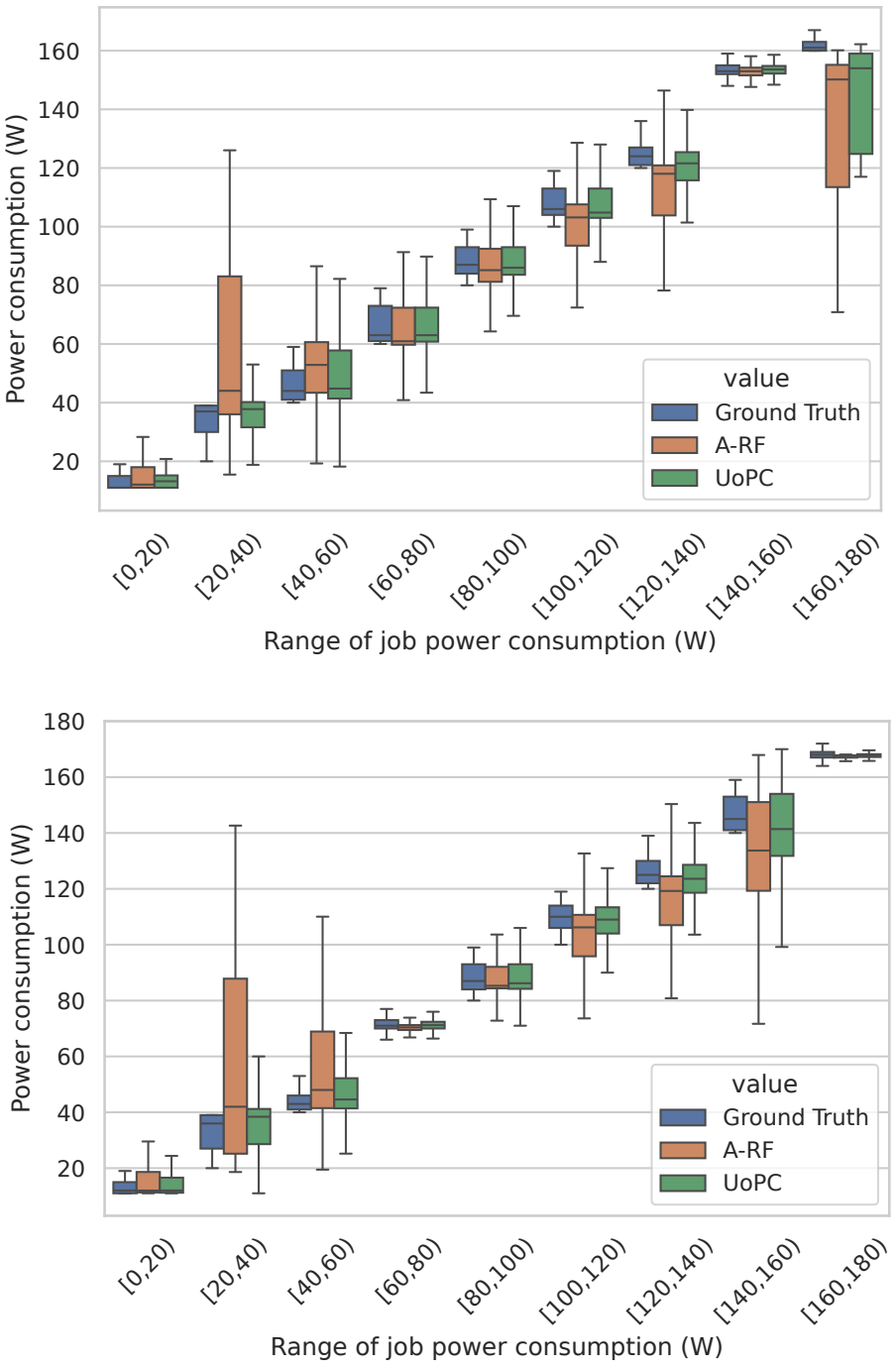

Figure 8.9: Distribution of actual and predicted *avgpcon* (above) and *maxpcon* (below) in ranges of 20W.

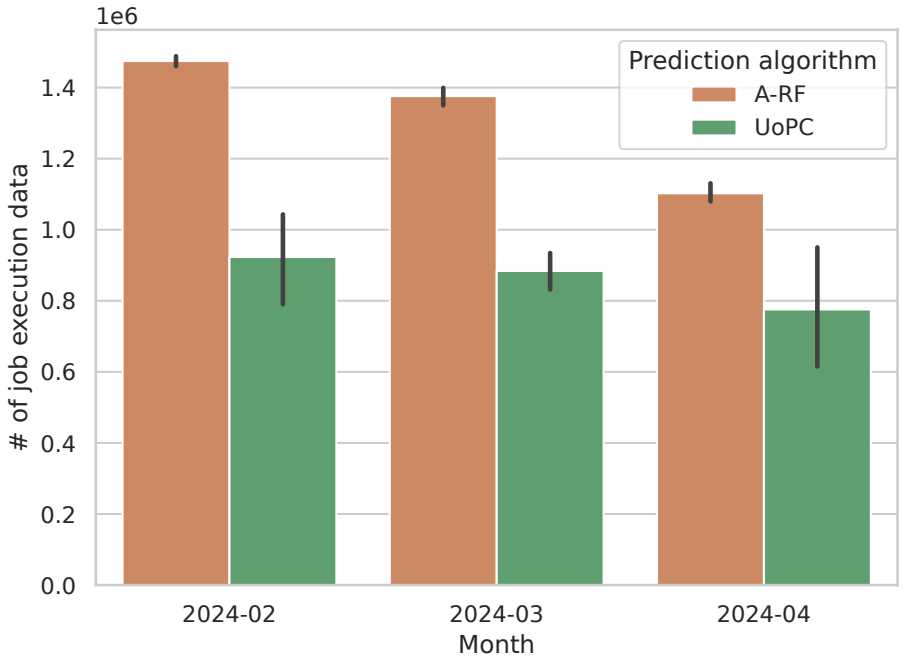

Figure 8.10: Distribution of the average amount of data per month used by the prediction models.

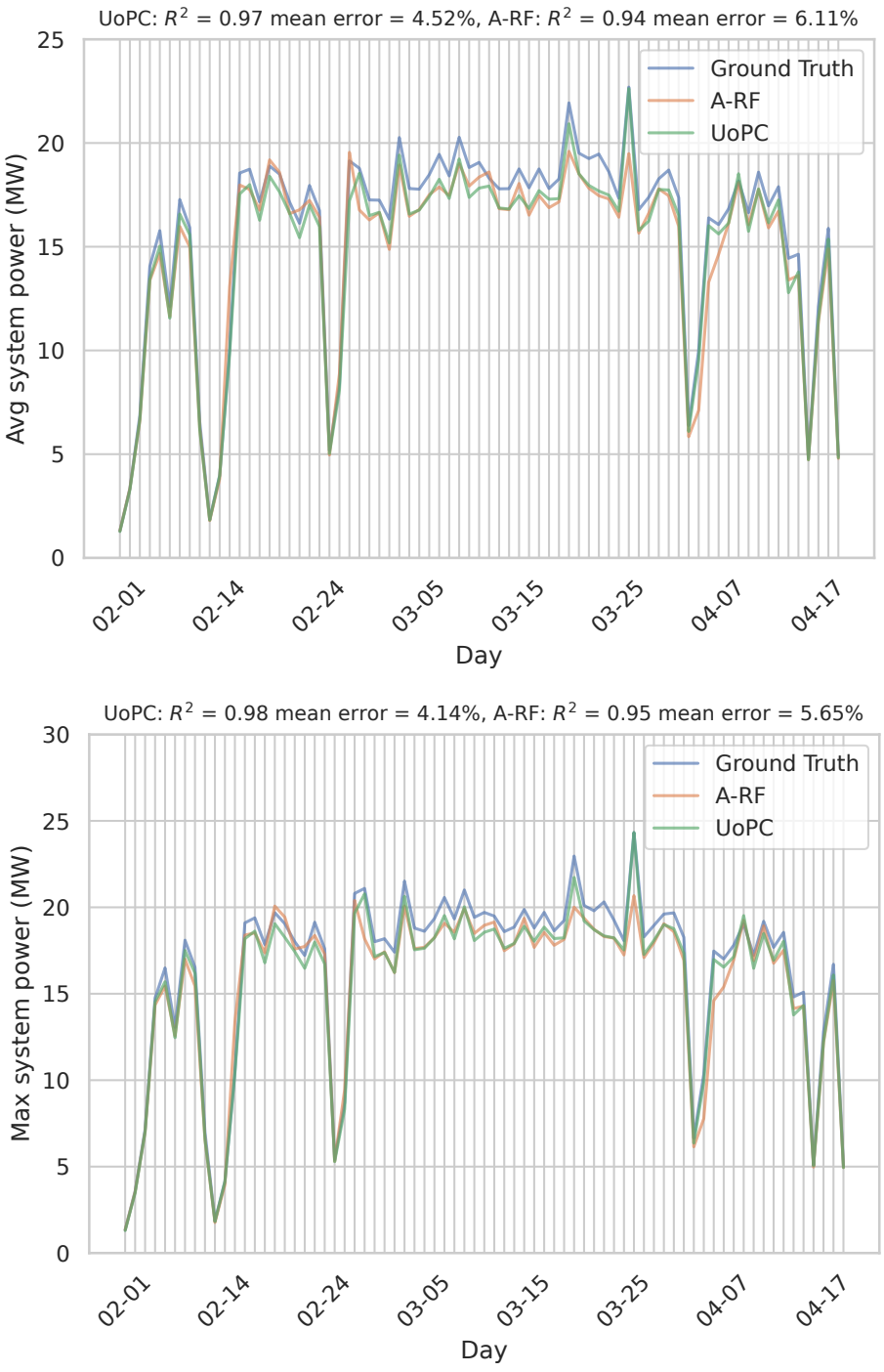

Figure 8.11: System power consumption prediction for the *avgpcon* (above) and *maxpcon* (below) tasks.

Chapter 9

Conclusion

In this work, we explored the development of job level predictive models, to enhance the effectiveness and sustainability of HPC systems. By addressing critical challenges in the state-of-the-art, this research has contributed with novel methodologies and tools to improve the environmental and operational efficiency of HPC environments. This chapter summarizes the primary contributions, highlights their significance, and outlines directions for future work.

9.1 Summary of Contributions

This research contributed in 5 main areas, which are in line with the limitations of the state-of-the-art in job-level predictive modelling, presented in Chapter 1.3.3. Such areas are the following.

Novel Comprehensive Job-Level Datasets During our research, we created and publicly released two extensive datasets from production systems, namely PM100 and F-DATA. Such datasets were created to address the lack of publicly available, large and fine-grained job-level data. They contain comprehensive information on job execution characteristics, including per job power consumption and performance metrics. F-DATA is the current largest job-level dataset, and the only one containing performance metrics. Moreover, is the first dataset containing the execution data of a former $1st$ most powerful supercomputer in the world, for

all its operational period. PM100, on the other hand, is the first dataset containing the actual job power consumption values (on different resources such as node, memory and CPU), sampled every 20 second during job execution. Their release has empowered the scientific community by fostering further research in job level predictive modeling for HPC systems.

Submission-time Models All the predictive models we developed rely only on submission-time information to perform a prediction on a job. This allows to make the models suitable to scenarios where the prediction can be leveraged to make informed decision on job scheduling or resource allocation.

Online Models Our algorithms are designed to work in an online context, where job data are live and streaming in time. This is fundamental to ensure that when the models are tested, this is done in a reliable way, and the results are actually meaningful for a real production environment. Moreover, we update our models over time to adapt to the change of workload in the system. This allows to obtain better prediction performance with respect to an offline scenario, as shown in Chapter 5 and 6, which was the standard in job level predictive modelling.

Furthermore, we developed operational framework, such as MCBound (Chapter 7.2) and UoPC (Chapter 8). Such frameworks can be easily configurable for each system's specifics, and rely on a software architecture which can be seamlessly deployed in a real production systems, as we showed for the case of Supercomputer Fugaku.

Predictive Models for Job Performance Characteristics This work is the first to address the prediction of job performance characteristics. In Chapter 7, we present a systematic methodology to characterize and predict the memory/compute-bound nature of HPC jobs. This kind of prediction is fundamental to enable decisions on the job execution based on the job performance, while allow to improve the system throughput while minimizing the energy consumption [6, 8].

End-User Tools Finally, in Chapter 8, we also present a user-level prediction framework. This framework is the first which targets also the use from the end-

users, thus becoming the first end-user tool. Such tools are crucial to i) improve end-user awareness in favor of a more environmentally sustainable usage of the HPC resources, and ii) encourage the adoption of energy-based pricing schemes [19].

9.2 Research Significance and Considerations

The findings of this dissertation underline the importance of job level predictive modelling in overcoming HPC sustainability challenges. By leveraging recent ML techniques and introducing novel datasets, this work bridges the gaps in existing methodologies and demonstrates how predictive insights can lead to substantial improvements in energy efficiency, system throughput, and user satisfaction.

Furthermore, this research emphasizes the need for user-awareness tools in HPC environments. By enabling end-users to make informed decisions regarding job configurations, the developed frameworks contribute to a collaborative effort between system designers and users toward achieving sustainable HPC practices.

Explainability The predictive models we developed are based on ML techniques (i.e., RF and KNN), which allow for several degrees of prediction interpretability. The RF can be used to identify the most important features for the prediction (in the case of RF), while the KNN reveals what data leads to perform a certain prediction (KNN). Such analysis are fundamental to validate how the models work in production environments, for accountability and trustworthyness purposes. Moreover, this provides insights on the job execution characteristics, and allows to study how the importance of the different features changes over time and across different systems. For instance, it could be observed how on different systems the power and energy are dependent on different causes and characteristics. This could be expanded by using adding the binaries, job scripts inspection with LLMs, or other configuration files to the input of the models, so as to have more fine-grained information on which specific components or workload characteristcs influence the prediction outcome.

Technical Integration with Job Schedulers Our predictive models are all suited for integration with job schedulers of production systems. In the previous chapters, we showed how are approaches always incur a negligible overhead on the system operations. This is fundamental to ensure that the scheduling strategies can be deployed in production systems, without incurring in performance penalties or alterations to the normal workload submission pipeline of the system. In pratical terms, the integration can be achieved through the development of a software pluging for the job scheduler, which interacts with the APIs of our frameworks to obtain the predictions. Such plugins can be developed in different programming languages, depending on the job scheduler used. Popular scheduling softwares, e.g. SLURM [88], FLUX [4] and PBS [21], support the integration of external plugins written in C/C++ and Python. Such plugins are usually event-based, meaning that they are triggered by specific events in the job scheduling process (e.g. job submission, job start, job end, etc). For our purposes, a plugin can be triggered when a job is submitted to the scheduler, and it can then interact with our prediction framework to feed the job data and obtain the predictions for that job. Finally, the predictions can be leveraged to make an informed scheduling decision on the job execution. For instance, the predicted *power consumption* can be used to allocate resources in a way that minimizes the overall power consumption of the system, or the predicted *memory/compute-bound* nature of the job can be used to allocate resources in a way that maximizes the performance of the job. We foresee the development of such a software, so as to integrate our predictive models with the job scheduling pipeline of production systems.

9.3 Future Directions

Provided the results of our research, we foresee several possible future directions. First, we would like to find other production system datasets, aiming to test our predictive models on other data and validate the results obtained so far. In this endeavor, we would like to obtain data extracted from computing environment (e.g. cloud, edge, fog), so as to expand our approaches beyond HPC systems only. Second, we would like to expand our predictive models to other job characteristics, such as duration or energy. In the context of the study presented in Chapter

7, we want to expand our classification to other classes, such as GPU-bound, I/O-bound or network-bound. This would allow to have even more fine-grained information for scheduling and resource allocation strategies. Finally, we want to integrate our predictions into informed job-scheduling strategies. More specifically, we are interested in carbon-aware scheduling. Our power prediction models can be adapted to the prediction of per-job carbon emission. Such information can be used to perform job scheduling targeting the minimization of system's carbon footprint.

9.4 Concluding Remarks

Sustainability considerations are fundamental in every aspect of modern society. When it comes to HPC system, these become not only necessary, but also timely. In our research, we aimed at improving the state of job level predictive modelling to make them suitable for production environment. With our contributions, we hope to have made steps forward towards i) the actual deployment in productions of such solutions, and ii) the employment of more environmental conscious HPC practices. We believe that it is paramount to raise awareness on the importance of this research's topic, to inspire further advancements in the field and contribute to the broader mission of sustainable computing.

Bibliography

[1] Oreste Villa, Daniel R Johnson, Mike Oconnor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, et al. Scaling the power wall: a path to exascale. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 830–841. IEEE, 2014.

[2] Andrea Borghesi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. Scheduling-based power capping in high performance computing systems. *Sustainable Computing: Informatics and Systems*, 19:1–13, 2018.

[3] Dineshkumar Rajagopal, Daniele Tafani, Yiannis Georgiou, David Glesser, and Michael Ott. A novel approach for job scheduling optimizations under power cap for arm and intel hpc systems. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 142–151. IEEE, 2017.

[4] Tapasya Patki, Dong Ahn, Daniel Milroy, Jae-Seung Yeom, Jim Garlick, Mark Grondona, Stephen Herbein, and Thomas Scogland. Fluxion: A scalable graph-based resource model for hpc scheduling challenges. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 2077–2088, 2023.

[5] Felippe Vieira Zacarias, Paul Carpenter, and Vinicius Petrucci. Memory demands in disaggregated hpc: How accurate do we need to be? In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 1–6. IEEE, 2021.

[6] Jens Breitbart, Josef Weidendorfer, and Carsten Trinitis. Case study on co-scheduling for hpc applications. In *2015 44th International Conference on Parallel Processing Workshops*, pages 277–285, 2015.

[7] Jason Hall, Arjun Lathi, David K Lowenthal, and Tapasya Patki. Evaluating the potential of coscheduling on high-performance computing systems. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 155–172. Springer, 2023.

[8] Jens Breitbart, Simon Pickartz, Stefan Lankes, Josef Weidendorfer, and Antonello Monti. Dynamic co-scheduling driven by main memory bandwidth utilization. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 400–409, 2017.

[9] Anupong Banjongkan, Watthana Pongsena, Nittaya Kerdprasop, and Kittisak Kerdprasop. A study of job failure prediction at job submit-state and job start-state in high-performance computing system: Using decision tree algorithms. *Journal of Advances in Information Technology*, 12(2), 2021.

[10] Nasim Ahmed, Andre LC Barczak, Mohammad A Rashid, and Teo Susnjak. Runtime prediction of big data jobs: performance comparison of machine learning algorithms and analytical models. *Journal of Big Data*, 9(1):67, 2022.

[11] Alina Sîrbu and Ozalp Babaoglu. Power consumption modeling and prediction in a hybrid cpu-gpu-mic supercomputer. In *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings 22*, pages 117–130. Springer, 2016.

[12] Andrea Borghesi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. Predictive modeling for job power consumption in hpc systems. In *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, pages 181–199. Springer, 2016.

[13] Georges Da Costa, Marios D Dikaiakos, and Salvatore Orlando. Nine months in the life of egee: a look from the south. In *2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 281–287. IEEE, 2007.

[14] Dmitry Duplyakin and Kevin Menear. Nrel eagle supercomputer jobs. 02 2023.

[15] Tirthak Patel, Adam Wagenhäuser, Christopher Eibel, Timo Hönig, Thomas Zeiser, and Devesh Tiwari. What does power consumption behavior of hpc jobs reveal?: Demystifying, quantifying, and predicting power consumption characteristics. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 799–809. IEEE, 2020.

[16] Bruce Bugbee, Caleb Phillips, Hilary Egan, Ryan Elmore, Kenny Gruchalla, and Avi Purkayastha. Prediction and characterization of application power use in a high-performance computing environment. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 10(3):155–165, 2017.

[17] Keiji Yamamoto, Yuichi Tsujita, and Atsuya Uno. Classifying jobs and predicting applications in hpc systems. In *High Performance Computing: 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018, Proceedings 33*, pages 81–99. Springer, 2018.

[18] Jie Li, Rui Wang, Ghazanfar Ali, Tommy Dang, Alan Sill, and Yong Chen. Workload failure prediction for data centers. *arXiv preprint arXiv:2301.05176*, 2023.

[19] Andrea Borghesi, Andrea Bartolini, Michela Milano, and Luca Benini. Pricing schemes for energy-efficient hpc systems: Design and exploration. *The International Journal of High Performance Computing Applications*, 33(4):716–734, 2019.

[20] Md Sabbir Hasan, Frederico Alvares de Oliveira, Thomas Ledoux, and Jean-Louis Pazat. Enabling green energy awareness in interactive cloud application. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 414–422. IEEE, 2016.

[21] Hanhua Feng, Vishal Misra, and Dan Rubenstein. Pbs: a unified priority-based scheduler. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 203–214, 2007.

[22] Andrea Borghesi, Carmine Di Santi, Martin Molan, Mohsen Seyedkazemi Ardebili, Alessio Mauri, Massimiliano Guarrasi, Daniela Galetti, Mirko Cestari, Francesco Barchi, Luca Benini, et al. M100 exadata: a data collection campaign on the cineca's marconi100 tier-0 supercomputer. *Scientific Data*, 10(1):288, 2023.

[23] Yoav Freund, Robert Schapire, and Naoki Abe. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612, 1999.

[24] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[25] Chao-Ying Joanne Peng, Kuk Lida Lee, and Gary M Ingersoll. An introduction to logistic regression analysis and reporting. *The journal of educational research*, 96(1):3–14, 2002.

[26] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[27] Evelyn Fix and Joseph Lawson Hodges. Discriminatory analysis. nonparametric discrimination: Consistency properties. *International Statistical Review/Revue Internationale de Statistique*, 57(3):238–247, 1989.

[28] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[29] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and et al. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 NAACL: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[30] Théo Saillant, Jean-Christophe Weill, and Mathilde Mougeot. Predicting job power consumption based on rjms submission data in hpc systems. In *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings 35*, pages 63–82. Springer, 2020.

[31] Shigeto Suzuki, Michiko Hiraoka, Takashi Shiraishi, Enxhi Kreshpa, Takuji Yamamoto, Hiroyuki Fukuda, Shuji Matsui, Masahide Fujisaki, and Atsuya Uno. Power prediction for sustainable hpc. *Journal of Information Processing*, 29:283–294, 2021.

[32] Sean Wallace, Xu Yang, Venkatram Vishwanath, William E Allcock, Susan Coghlan, Michael E Papka, and Zhiling Lan. A data driven scheduling approach for power management on hpc systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 656–666. IEEE, 2016.

[33] Andrea Borghesi, Carmine Di Santi, Martin Molan, Mohsen Seyedkazemi Ardebili, Alessio Mauri, Massimiliano Guarrasi, Daniela Galetti, Mirko Cestari, Francesco Barchi, Luca Benini, Francesco Beneventi, and Andrea Bartolini. M100 dataset, https://zenodo.org/records/7588815, January 2023.

[34] Francesco Antici, Mohsen Seyedkazemi Ardebili, Andrea Bartolini, and Zeynep Kiziltan. PM100: A Job Power Consumption Dataset of a Large-Scale HPC System, July 2023.

[35] Cristian Galleguillos, Zeynep Kiziltan, Alina Sîrbu, and Ozalp Babaoglu. Constraint programming-based job dispatching for modern hpc applications. In *Principles and Practice of Constraint Programming: 25th International Conference, CP 2019, Stamford, CT, USA, September 30–October 4, 2019, Proceedings 25*, pages 438–455. Springer, 2019.

[36] Qiqi Wang, Hongjie Zhang, Jing Li, Yu Shen, and Xiaohui Liu. Predicting job finish time based on parameter features and running logs in supercomputing system. *The Journal of Supercomputing*, 78(17):18551–18577, 2022.

[37] Mohammad S Jassas and Qusay H Mahmoud. Analysis of job failure and prediction model for cloud computing using machine learning. *Sensors*, 22(5):2035, 2022.

[38] Basit Qureshi. Profile-based power-aware workflow scheduling framework for energy-efficient data centers. *Future Generation Computer Systems*, 94:453–467, 2019.

[39] Michael Stonebraker and Lawrence A Rowe. The design of postgres. *ACM Sigmod Record*, 15(2):340–355, 1986.

[40] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound gpu applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 191–202. IEEE, 2014.

[41] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[42] Siavash Ghiasvand and Florina M Ciorba. Assessing data usefulness for failure analysis in anonymized system logs. In *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 164–171. IEEE, 2018.

[43] Suntherasvaran Murthy, Asmidar Abu Bakar, Fiza Abdul Rahim, and Ramona Ramli. A comparative study of data anonymization techniques. In *2019 IEEE 5th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing,(HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pages 306–309. IEEE, 2019.

[44] Zhaozhen Xu, Zhijin Guo, and Nello Cristianini. On compositionality in data embedding. In *International Symposium on Intelligent Data Analysis*, pages 484–496. Springer, 2023.

[45] Baolin Li, Rohan Basu Roy, Daniel Wang, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Toward sustainable hpc: Carbon footprint estimation

and environmental implications of hpc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2023.

[46] Kazi Asifuzzaman, Mohammad Alaul Haque Monil, Frank Liu, and Jeffrey S Vetter. Evaluating hpc kernels for processing in memory. In *Proceedings of the 2022 International Symposium on Memory Systems*, pages 1–6, 2022.

[47] Marcelo Orenes-Vera, Esin Tureci, David Wentzlaff, and Margaret Martonosi. Dalorex: A data-local program execution and architecture for memory-bound applications. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 718–730. IEEE, 2023.

[48] Kevin Menear, Ambarish Nag, Jordan Perr-Sauer, Monte Lunacek, Kristi Potter, and Dmitry Duplyakin. Mastering hpc runtime prediction: From observing patterns to a methodological approach. In *Practice and Experience in Advanced Research Computing*, pages 75–85. 2023.

[49] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. Predicting job completion times using system logs in supercomputing clusters. In *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 1–8. IEEE, 2013.

[50] Elvis Rojas, Esteban Meneses, Terry Jones, and Don Maxwell. Analyzing a five-year failure record of a leadership-class supercomputer. In *2019 31st SBAC-PAD*, pages 196–203. IEEE, 2019.

[51] Sheng Di, Hanqi Guo, Eric Pershey, Marc Snir, and Franck Cappello. Characterizing and understanding hpc job failures over the 2k-day life of ibm bluegene/q system. In *2019 49th Annual IEEE/IFIP DSN*, pages 473–484. IEEE, 2019.

[52] Hamid Fadishei, Hamid Saadatfar, and Hossein Deldari. Job failure prediction in grid environment based on workload characteristics. In *2009 14th CSICC*, pages 329–334. IEEE, 2009.

[53] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. Failure prediction of jobs in compute clouds: A google cluster case study. In *2014 IEEE ISSRE Workshops*, pages 341–346, 2014.

[54] Tariqul Islam and Dakshnamoorthy Manivannan. Predicting application failure in cloud: A machine learning approach. In *2017 IEEE ICCC*, pages 24–31. IEEE, 2017.

[55] Wucherl Yoo, Alex Sim, and Kesheng Wu. Machine learning based job status prediction in scientific clusters. In *2016 SAI*, pages 44–53, 2016.

[56] Deva Bodas, Justin Song, Murali Rajappa, and Andy Hoffman. Simple power-aware scheduler to limit power consumption by hpc system within a budget. In *2014 Energy Efficient Supercomputing Workshop*, pages 21–30. IEEE, 2014.

[57] Andrea Borghesi, Francesca Collina, Michele Lombardi, Michela Milano, and Luca Benini. Power capping in high performance computing systems. In *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*, pages 524–540. Springer, 2015.

[58] Danilo Carastan-Santos, Georges Da Costa, Millian Poquet, Patricia Stolf, and Denis Trystram. Light-weight prediction for improving energy consumption in hpc platforms. In Jesus Carretero, Sameer Shende, Javier Garcia-Blas, Ivona Brandic, Katzalin Olcoz, and Martin Schreiber, editors, *Euro-Par 2024: Parallel Processing*, pages 152–165, Cham, 2024. Springer Nature Switzerland.

[59] Hyunsoo Kim, Jiseok Jeong, and Changwan Kim. Daily peak-electricity-demand forecasting based on residual long short-term network. *Mathematics*, 10(23):4486, 2022.

[60] Eduardo R Rodrigues, Renato LF Cunha, Marco AS Netto, and Michael Spriggs. Helping hpc users specify job memory requirements via machine learning. In *2016 Third International Workshop on HPC User Support Tools (HUST)*, pages 6–13. IEEE, 2016.

[61] Daren Lee, Ivo Dinov, Bin Dong, Boris Gutman, Igor Yanovsky, and Arthur W Toga. Cuda optimization strategies for compute-and memory-bound neuroimaging algorithms. *Computer methods and programs in biomedicine*, 106(3):175–187, 2012.

[62] Ang Li, Weifeng Liu, Mads RB Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. Exploring and analyzing the real impact of modern on-package memory on hpc scientific kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[63] Marina Sokolova, Nathalie Japkowicz, and Stan Szpakowicz. Beyond accuracy, f-score and roc: A family of discriminant measures for performance evaluation. volume Vol. 4304, pages 1015–1021, 01 2006.

[64] Nan Ding and Samuel Williams. *An instruction roofline model for gpus*. IEEE, 2019.

[65] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2013.

[66] Diogo Marques, Helder Duarte, Aleksandar Ilic, Leonel Sousa, Roman Belenov, Philippe Thierry, and Zakhar A Matveev. Performance analysis with cache-aware roofline model in intel advisor. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 898–907. IEEE, 2017.

[67] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[68] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[69] Michael S Keller. Take command: cron: Job scheduler. *Linux Journal*, 1999(65es):15–es, 1999.

[70] Fujitsu Limited. A64fx pmu events, 2019.

[71] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, et al. Co-design for a64fx manycore processor and" fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[72] Yuetsu Kodama, Tetsuya Odajima, Eishi Arima, and Mitsuhisa Sato. Evaluation of power management control on the supercomputer fugaku. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 484–493. IEEE, 2020.

[73] Viraj Kulkarni, Manish Gawali, Amit Kharat, et al. Key technology considerations in developing and deploying machine learning models in clinical radiology practice. *JMIR Medical Informatics*, 9(9):e28776, 2021.

[74] Abdul Hameed, Alireza Khoshkbarforoushha, Rajiv Ranjan, Prem Prakash Jayaraman, Joanna Kolodziej, Pavan Balaji, Sherali Zeadally, Qutaibah Marwan Malluhi, Nikos Tziritas, Abhinav Vishnu, et al. A survey and taxonomy on energy efficient resource allocation techniques for cloud computing systems. *Computing*, 98:751–774, 2016.

[75] Lucas Baier, Fabian Jöhren, and Stefan Seebacher. Challenges in the deployment and operation of machine learning in practice. In *ECIS*, volume 1, 2019.

[76] Gideon Juve, Benjamin Tovar, Rafael Ferreira Da Silva, Dariusz Król, Douglas Thain, Ewa Deelman, William Allcock, and Miron Livny. Practical resource monitoring for robust high throughput computing. In *2015 IEEE International Conference on Cluster Computing*, pages 650–657. IEEE, 2015.

[77] Nitin Sukhija and Elizabeth Bautista. Towards a framework for monitoring and analyzing high performance computing environments using kubernetes

and prometheus. In *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation*, pages 257–262. IEEE, 2019.

[78] Jie Li, Ghazanfar Ali, Ngan Nguyen, Jon Hass, Alan Sill, Tommy Dang, and Yong Chen. Monster: an out-of-the-box monitoring tool for high performance computing systems. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 119–129. IEEE, 2020.

[79] Maxime Colmant, Pascal Felber, Romain Rouvoy, and Lionel Seinturier. Wattskit: Software-defined power monitoring of distributed systems. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 514–523. IEEE, 2017.

[80] Jonathan Muraña, Sergio Nesmachnow, Fermín Armenta, and Andrei Tchernykh. Characterization, modeling and scheduling of power consumption of scientific computing applications in multicores. *Cluster Computing*, 22:839–859, 2019.

[81] Masahiro Nakao, Hidetomo Kaneyama, Masaru Nagaku, Ikki Fujiwara, Atsuko Takefusa, Shinichi Miura, and Keiji Yamamoto. Introducing open ondemand to supercomputer fugaku. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 720–727, 2023.

[82] Adedolapo Okanlawon, Huichen Yang, Avishek Bose, William Hsu, Dan Andresen, and Mohammed Tanash. Feature selection for learning to predict outcomes of compute cluster jobs with application to decision support. In *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1231–1236. IEEE, 2020.

[83] Hamidreza Khaleghzadeh, Ravi Reddy Manumachu, and Alexey Lastovetsky. Efficient exact algorithms for continuous bi-objective performance-energy optimization of applications with linear energy and monotonically

increasing performance profiles on heterogeneous high performance computing platforms. *Concurrency and Computation: Practice and Experience*, 35(20):e7285, 2023.

[84] Yuichi Tsujita, Atsuya Uno, Ryuichi Sekizawa, Keiji Yamamoto, and Fumichika Sueyasu. Job classification through long-term log analysis towards power-aware hpc system operation. In *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 26–34. IEEE, 2021.

[85] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for {Large-Scale} advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[86] Kevin Assogba, Eduardo Lima, M Mustafa Rafique, and Minseok Kwon. Predictddl: Reusable workload performance prediction for distributed deep learning. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 13–24. IEEE, 2023.

[87] U.S. Energy Information Administration. How much electricity does an american home use?, 2024.

[88] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.