AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA INFORMATYKI

PRACA DYPLOMOWA MAGISTERSKA

Prediction of resource usage of computational jobs

Przewidywanie zużycia zasobów przez zadania obliczeniowe

Autor:	*Mateusz Krzysztof Plinta*
Kierunek studiów:	*Informatyka*
Typ studiów:	*Stacjonarne*
Opiekun pracy:	*dr hab. inż Bartosz Baliś*

Kraków, 2020

Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. – Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

……

Contents

Chapter 1

Introduction

In the last half-century we witnessed an exponential growth in terms of computational power. Moore's law states that the number of transistors on integrated circuits doubles approximately every two years, which is an exponential growth. This observation is commonly known and over the years it has proved to be quite accurate [36]. Increase in the number of transistors on integrated circuits translates to an improvement in machine computational power, whose surge over the last 30 years can be seen in Fig. 1.1.

With the increase of computer power capabilities, expectations also have risen. Computing resources commonly have started to be transferred outside the local facilities, to huge, corporate data centers that offer on-demand availability of computer system resources, such as computing power and data storage. Highly configurable, accessible via APIs, and rented in the pay-per-use model, this type of service has commonly become known as *cloud computing* [27]. There is an apparent tendency in the industry to redirect computations to the cloud as it provides more flexibility, scalability, data backups and built-in security. Cost optimization is the primary reason for 47% of enterprises' cloud migration [1]. According to [2], 90% of companies are "in the cloud". Another research states that cloud data centers will process 94% of computational workloads in 2021 [3].

In recent years, the scientific computing community has also shown great interest in cloud computing capabilities. Although the pay-as-you-go model is not new for academic researchers, since for decades they used shared compute facilities and are fairly accustomed to per CPU-hour billing scheme, it proves beneficial for users who do not require 24/7 availability. Moreover, cloud solutions have another compelling feature, which

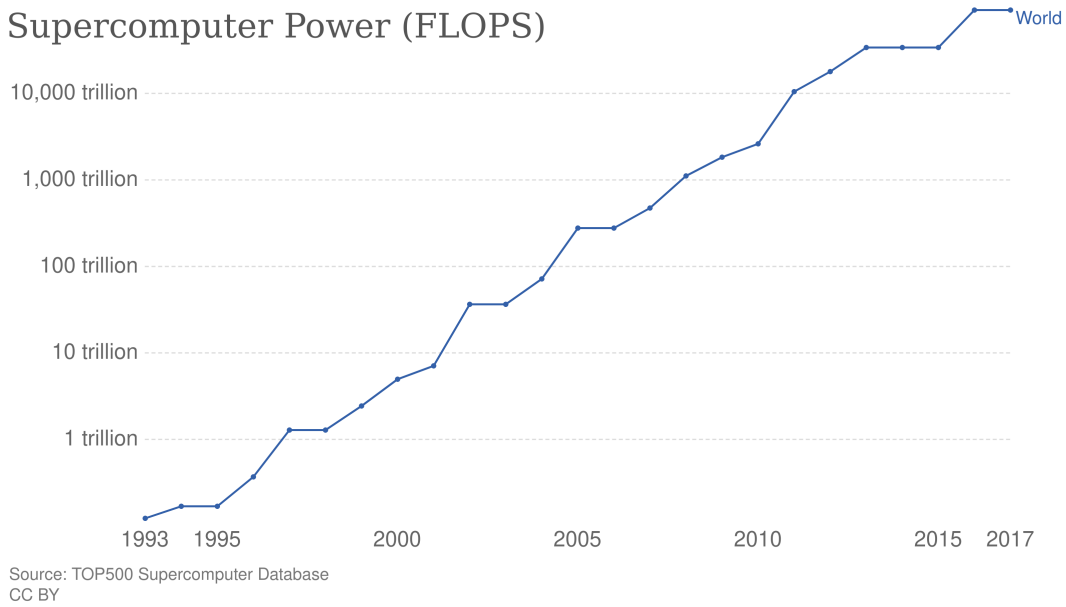

Figure 1.1: Exponential growth of supercomputing power as recorded by the TOP500 list [4]. Power was measured as the number of floating-point operations carried out per second (FLOPS) by the largest supercomputer in any given year. The vertical axis is logarithmic and the increase corresponds to the exponential growth of the number of transistors every two years.

differentiates them from existing supercomputer facilities – they abstract the underlying compute components, ranging from hardware infrastructures through operating systems to even software packages. This enables the scientists and developers to gain full control over their clusters, allowing for installation of custom pieces of software at the system level – a feature hardly met when dealing with supercomputer centers. For providers offering software as a service paradigm, the scientist is deprived of necessity of installing, maintaining and optimizing the application, as she or he only needs to be compliant to specific API [34].

Many works, including [19, 34], dedicated to assessing capabilities of cloud computing solutions for scientific purposes deemed them as feasible and advantageous, especially for projects requiring moderate levels of computational resources and seeking cheaper solutions compared to supercomputer centers, grids or commodity clusters.

Scientific workflows [37] are an example of a leading paradigm for scientific computing which strongly leverages cloud computing. Workflows are widely used to orchestrate complex computational applications. They enable users to express multi-step computa-

tional pipelines easily, in such a way that the jobs are efficiently and reliably executed on distributed infrastructures [20]. Scientific workflows are used in a variety of different domains of science, and are utilized for image processing, simulation, data analysis etc. Existing works, including [20] and [7], aiming to evaluate the role of cloud computation in workflow execution, state that the advantages are quite substantial. They derive from the service-oriented architecture and virtualization of clouds and include elasticity, provenance, reproducibility, lease-based provisioning, support for legacy environments and applications and also the illusion of infinite resources.

1.1 Background

As mentioned previously, arguably the majority of scientific workflows are executed in the cloud. Efficient workflow execution depends primarily on proper task scheduling and resource allocation [10, 17, 35]. A crucial issue in workflow execution management algorithms is accurate prediction of task execution times (to correctly schedule tasks into time slots), and resource usage (to correctly allocate tasks to available machines).

Older methods for cost-effective managing of workflows typically revolved around analytical modeling techniques, which required significant effort to develop [14, 40], or statistical techniques, which usually have not been very accurate [39, 41]. Currently there is a tendency to combine machine learning and data mining techniques in order to create a best performing, self-adaptive prediction and scheduling mechanism. Usually, in a prediction process, taken into account are only CPU utilization, run-time, I/O of a process and memory usage. Several works also opt for combining of ”online” techniques – actively working on the prediction process during execution of a task, and ”offline” methods – traditionally estimating resources using post-processing and history data [13, 31].

1.2 Problem statement

The main research problem tackled in this Thesis concerns employing machine learning approach to predict time and resource consumption of computational jobs of scientific workflows. We assume that, based on the collected history of workflow executions, including information on processing job types and their input parameters, as well as ma-

chine configuration parameters, it is feasible to train applicable prediction models, yielding accurate estimates. In particular, we will investigate several classification and regression methods in hope of categorising job runs by the intensity of a particular resource as well as predicting run-time of tasks. It should be noted that we consider only individual jobs of workflows. Scheduling of entire workflows is out of scope of this Thesis.

1.2.1 Objectives and methodology

The main objectives of this Thesis and methods to achieve them are as follows:

- To collect execution data of computational task runs (jobs) in scientific workflows and the statistics of their resource utilization, appropriate for leveraging machine learning methods and training prediction models. The data is collected based on individual experiments conducted on various cloud platform providers, namely Amazon Web Services (AWS) and Google Cloud Platform (GCP). The captured parameters of a job include: the command and parameters used to run the job, names and sizes of input and output files, and machine configuration, e.g. its CPU and available memory.
- To develop and evaluate prediction models, based on the gathered data, using a variety of techniques, such as binary classification using logistic regression as well as regression analysis using decision trees, random forest, *k*-nearest neighbours, and deep neural networks.

1.3 Thesis organization

This Thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 describes details about data collection procedure. Chapter 4 presents the prediction methodology using machine learning approach. Next, chapter 5 contains evaluation of the gathered results, then chapter 6 shows other insights into our methodology and suggests future research. Finally, chapter 7 concludes the work.

Chapter 2

Related work

The prediction of execution time and resources usage is an important topic in tasks scheduling on large computational infrastructures. Thus, a fair number of studies have focused on workflow deployments on such infrastructures. Some researchers focus on predictions of entire workflows [29] [15], while others investigate the modular character of workflows and the possibility of predicting individual workflow activities [22]. However, in those studies the input of the processing task is not as thoroughly examined as in this work. At most, generic attributes such as file names or problem sizes are considered.

Another approach is presented in the work of Kecskemeti *et al.* [21]. In their article, they propose a technique for predicting generic background workload by means of simulations, which are supposed to provide additional knowledge of the underlying private cloud systems, enabling the support for activities like cloud orchestration and workflow enactment. Their technique compares long-running scientific workflows with their cloud simulations equivalent, and based on this they predict workloads of the future jobs. As a workflow they use a biochemical application on both real and simulated cloud infrastructures. Our study however does not account for generic background workload but only for resource utilization of jobs executed only by the regarded user.

Different solution was presented by Caron *et al.* [11], where they used workload prediction based on identifying similar past occurrences of the current short-term workload history for efficient resource scaling. Their algorithm also predicts the system usage by extrapolating beyond the identified patterns. However our work focuses on prediction using machine learning methods.

Yet another proposal by Mao *et al.* [26] argues that a single prediction algorithm is not sufficient to estimate workloads in complex cloud computing environments. As such, that devised a self-adaptive prediction algorithm combining linear regression and neural networks to predict workloads. They also evaluated their approach on public cloud server solutions, and the accuracy of their solution is considerably better when compared to purely neural network or linear regression-based approaches. Our study, however, evaluates different machine learning methods.

Tudor *et al.* [28] investigate the possibility of predicting execution time of the workflow based on features of the input that can be extracted inexpensively, given a sufficiently long history of observed past executions. The method involves exploring a space of estimation functions (regression models), each operating on a possibly different combination of features extracted from the input, to find the ones that maximize prediction accuracy. They demonstrate on a specific case study involving the Weka implementation of the C4.5 machine learning algorithm. For C4.5, a variety of regression models can be trained to reach acceptable asymptotic prediction errors of about 20%, substantially better than the baseline predictor, consisting of the average of past observed execution times. While promising, their method remains a knowledge-intensive task.

In the work of Silva *et al.* [13], the authors show different approach to resource prediction, in which they use data collected from 5 scientific workflows, equipped with information regarding CPU utilization, input/output processes, execution time and memory usage of a task. They devised a method capable of automatically characterizing workflow task requirements based on the profiles of said workflows. The main properties of their solution include: accurate prediction of task's execution time, peak memory usage and disk space usage based on the size of the tasks' input data. The algorithm checks whether the parameters of a dataset are correlated or not. Upon lack of any correlation, the data is divided into smaller subsets with the use of clustering. The authors also propose an online prediction process, in which job executions are actively monitored and the most recent data collected is taken into account when calculating the estimates. The results indicate superiority in case of accurate estimates, of online method compared to the more traditional, offline one.

Thanh-Phuong *et al.* [31] model workflow execution times as functions that depend on workflow inputs as well as on cloud features. Cloud features describe properties of the virtual machine type in which the task is executed. Models are built using regression

methods based on historical executions of that workflow in the cloud. These methods include: linear regression, regression trees, ensemble methods such as bagging using regression and random forest. Their team tested the solution on three different cloud providers: AWS (EC2), GCP (GCE) and RackSpace Cloud. Evaluated workflows include Montage, POV-Ray, Wien2k and Blender. The ASKALON workflow management system (*WMS*) [33] is used to manage workflow system. The scheduler executes workflow tasks on each virtual core of a virtual machine. The system collects resource utilization information using Linux system calls to obtain CPU, memory usage, I/O operations by instrumentation wrapper injected in every task. The workflow management system measures file transfer which is interpreted as bandwidth calculated from file transfer time. The authors also use clustering techniques to subdivide training data into subgroups based on input, and calculate prediction model separately for the each cluster. The results show that the proposed approach clearly outperforms existing prediction approaches, which are primarily based on pre-runtime parameters. The authors also conclude that their solution achieves better accuracy when using random forest than with other algorithms. Also, authors observed that two types of tasks are harder to predict than others: task with short execution times and bandwidth dependent tasks.

2.1 Summary

While majority of presented works are based on similar ideas, i.e. creating regression models for usage prediction of scientific workloads, our primary goal is to create a two-step prediction system. First step revolves around classifying intensity of resources usage of a job, whereas second step relies on using history data complemented by the classified resources intensity parameters information, in a regression analysis in order to predict execution time of a job. Regression is performed using a variety of different machine learning methods, and the finest ones are distinguished. While [31] also presents a two-step approach into prediction of runtime of a task, we aim to simplify the process by using logistic regression for classification purposes in the first stage of our system, instead of deriving resource utilization regression models. To the best of our knowledge, no previous research used described two-step prediction in this context.

Chapter 3

Data collection methodology

This chapter focuses on the process of collecting execution logs and parsing them into usable data. It briefly describes the technologies used during the data gathering, as well as the concept of scientific workflows and the specific workflows used for the purpose of this work. Finally, a detailed overview of the data collection process and the system architecture is presented.

3.1 Scientific workflows

Scientific workflows (WF) are a common tool amongst scientists, allowing them to express multi-step computational tasks in an easy and straightforward way [37]. Workflows typically can be defined as directed acyclic graphs (DAG) that describe the dependencies between the tasks and manage the data (and control) flow. Workflow size can vary from few tasks to a very large number (tens of thousands or more) of tasks grouped into multiple parallel stages separated by pre and postprocessing tasks. Scientific workflows are the types of workflows used in a scientific application, e.g. astronomical mosaics creator Montage [9], Vina – used for molecular docking [38] , Kinc – gene co-expression network analysis [32] or Soybean workflow (SoyKB) – genomic analysis [23].

Fig. 3.1 depicts an example of a basic workflow. First, data marked as *Input* is split into three parts. On each of those parts, proper computational task is run. Finally, each branch returns the results of the task, finalizing the workflow.

Here, we are mainly interested in the characteristics of individual workflow jobs,

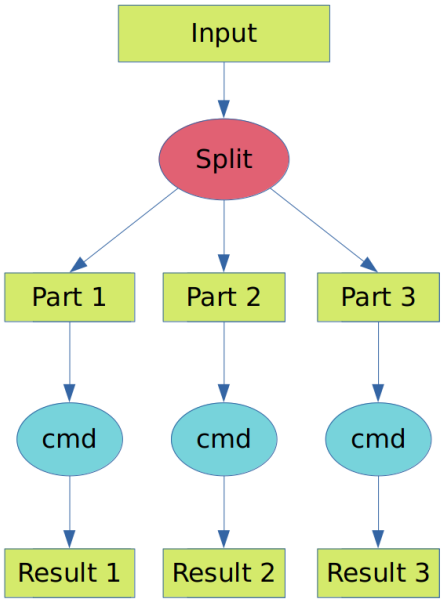

Figure 3.1: Exemple basic workflow. Data loaded from input is split into three parts which are processed in parallel by *cmd* tasks, leading to workflow results.

rather than workflows as a whole, however, the workflow that the job is part of is also part of this characteristic.

For the purposes of this paper, the following scientific workflows have been used.

Montage

Montage[1] is a toolkit allowing for creation of images utilizing the Flexible Image Transport System (FITS). The results produce custom space mosaics. The Montage distribution is founded by National Science Foundation and was previously founded by the National Aeronautics and Space Administration's (NASA) Earth Science Technology Office. For the purposes of this study, two different versions of Montage workflow have been utilized, to which, from now on, we will refer to as Montage[2] and Montage2[3]. Fig. 3.2 and 3.3, respectively, show the structure of Montage and Montage2 workflow graphs.

[1] `http://montage.ipac.caltech.edu/news.html`
[2] `https://github.com/hyperflow-wms/montage-workflow`
[3] `https://github.com/hyperflow-wms/montage2-workflow`

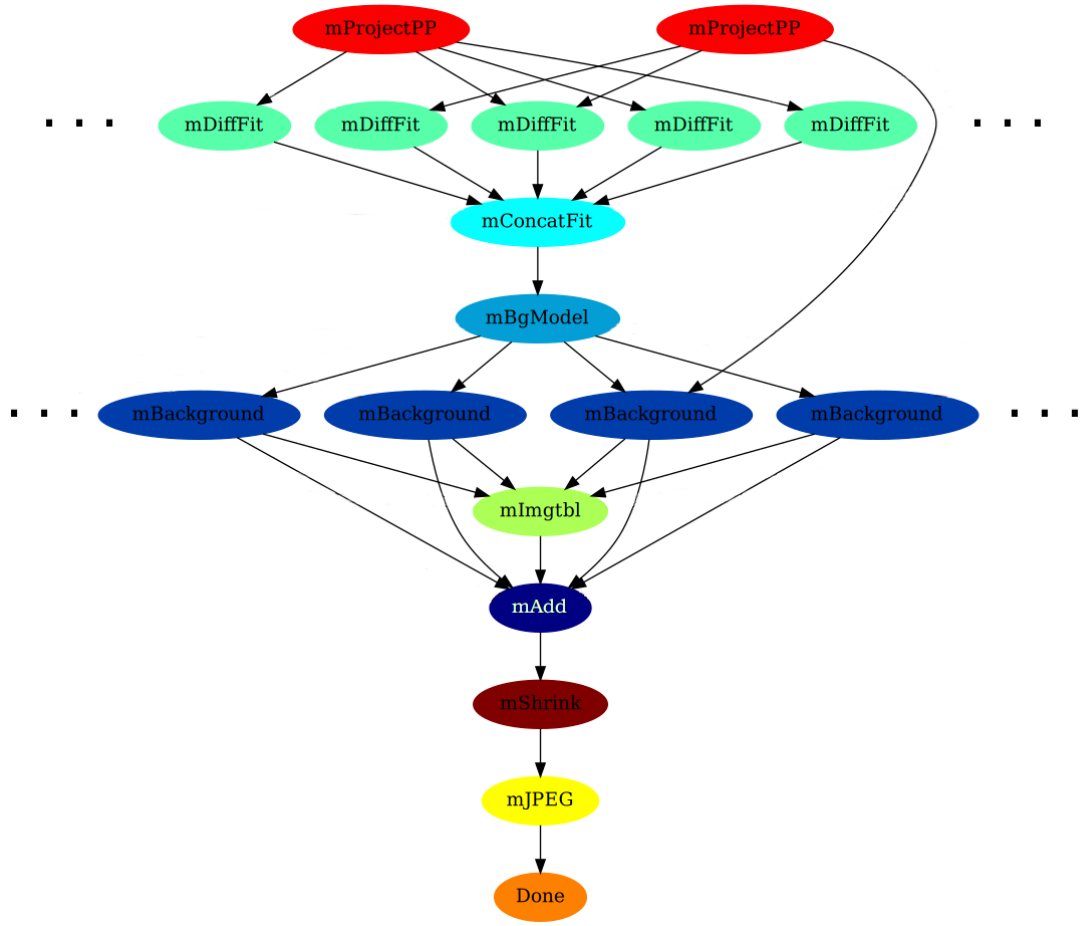

Figure 3.2: Montage workflow graph structure.

SoyKB

Soybean knowledge base[4] is an extensive web platform, established in order to store and integrate the gene, genomics and other relevant genetic data of soybean. The SoyKB workflow is used to process the genetic data. The available workflow[5] port from Pegasus system to Hyperflow was used for the purposes of this study. The structure of the SoyKB workflow graph is shown in Fig. 3.4.

[4] `http://soykb.org/`
[5] `https://github.com/hyperflow-wms/soykb-workflow`

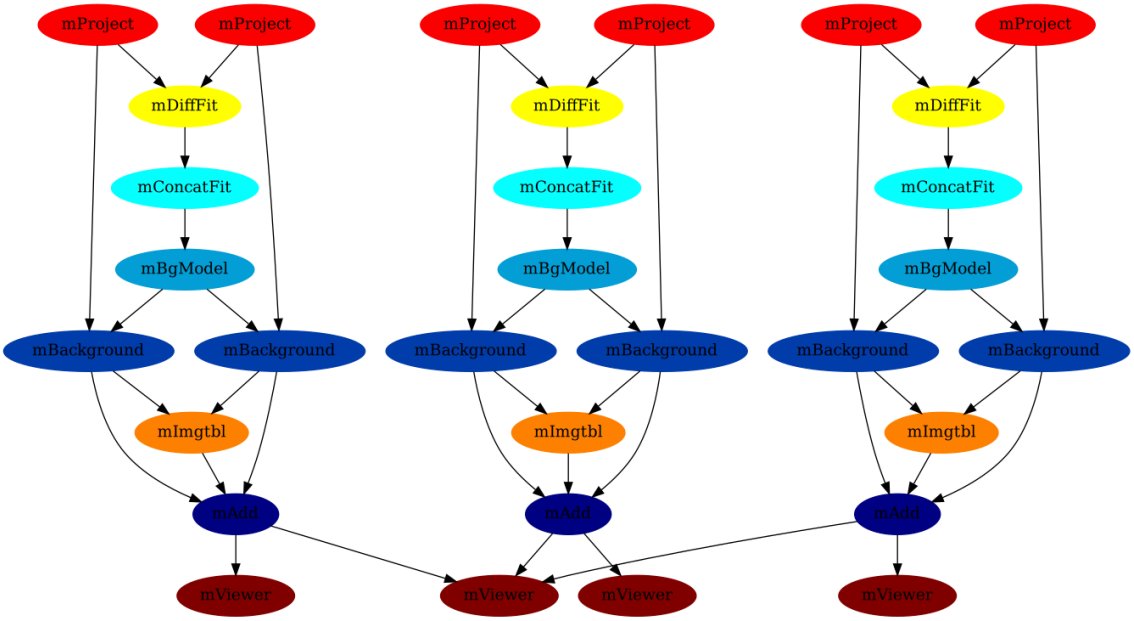

Figure 3.3: Graph of a small Montage2 workflow, with *degree* parameter = 0.01

3.2 Scientific workflow management

3.2.1 HyperFlow

For workflow execution, we used HyperFlow[6], which is an active project, performing a role of scientific workflow management system (WMS), providing simple and robust model of computation and execution engine for complex workflow applications [8]. In HyperFlow, a workflow is defined as a set of *processes* executing *functions* and exchanging *signals*, expressed in a JSON file format. The HyperFlow project provides the user with a multiple available workflow types, converters from other WMSs[7], as well as useful utilities such as log parser[8] and other post-processing workflow tools[9].

HyperFlow allows for workflow execution in various computing infrastructures, including the local machine, either directly which requires that all the WF libraries and executables need to be installed beforehand, or using Docker images, where all necessary software is already installed. HyperFlow also supports distributed computing infrastruc-

[6] `https://github.com/hyperflow-wms/hyperflow`
[7] `https://github.com/hyperflow-wms/pegasus-hyperflow-converter`
[8] `https://github.com/hyperflow-wms/log-parser`
[9] `https://github.com/hyperflow-wms/hflow-tools`

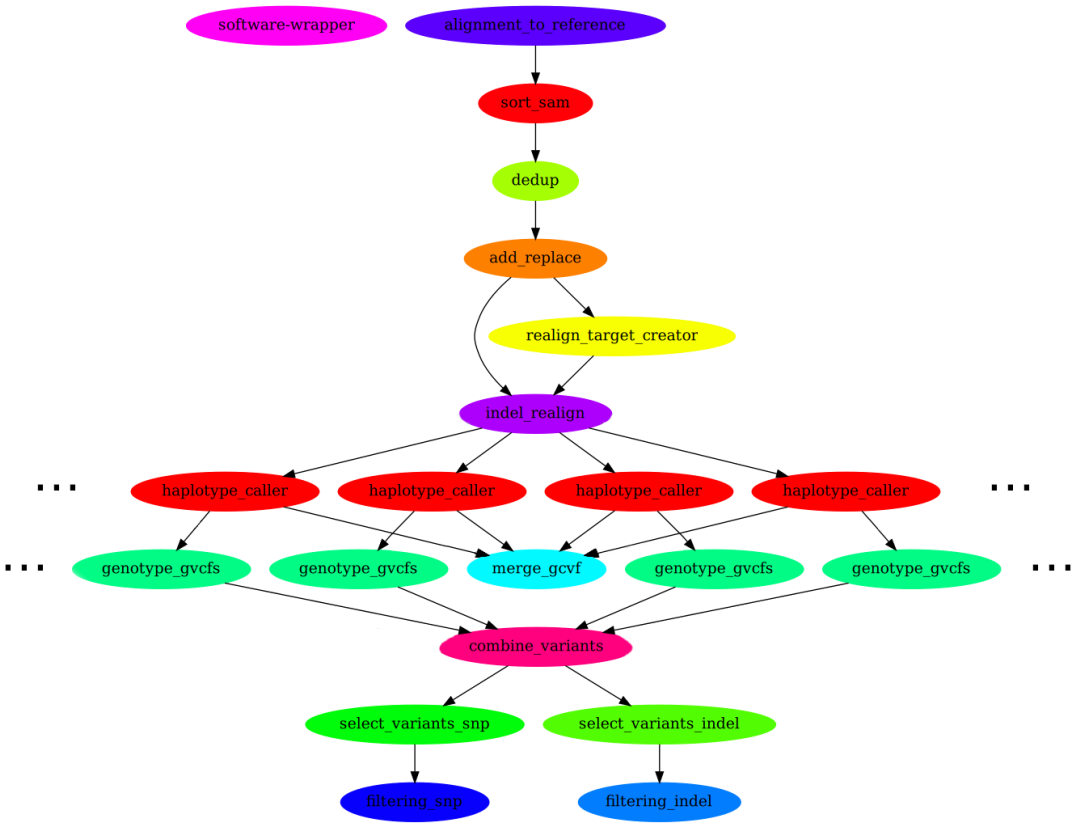

Figure 3.4: Graph of a SoyKB workflow

tures, notably Kubernetes clusters running in the Cloud [30].

3.2.2 Computing infrastructure

In this work, we consider workflows that run in a cluster of nodes (virtual machines) managed by Kubernetes, a platform for containerized applications. Here we briefly present the building blocks of such a computing infrastructure whose architecture is depicted in Fig. 3.5.

Docker

Docker is a technology utilizing OS-level virtualization in order to deliver software in packages called *containers*. Containers provide isolation, bundling of required software, libraries and configuration files, thus allowing for environmental consistency across de-

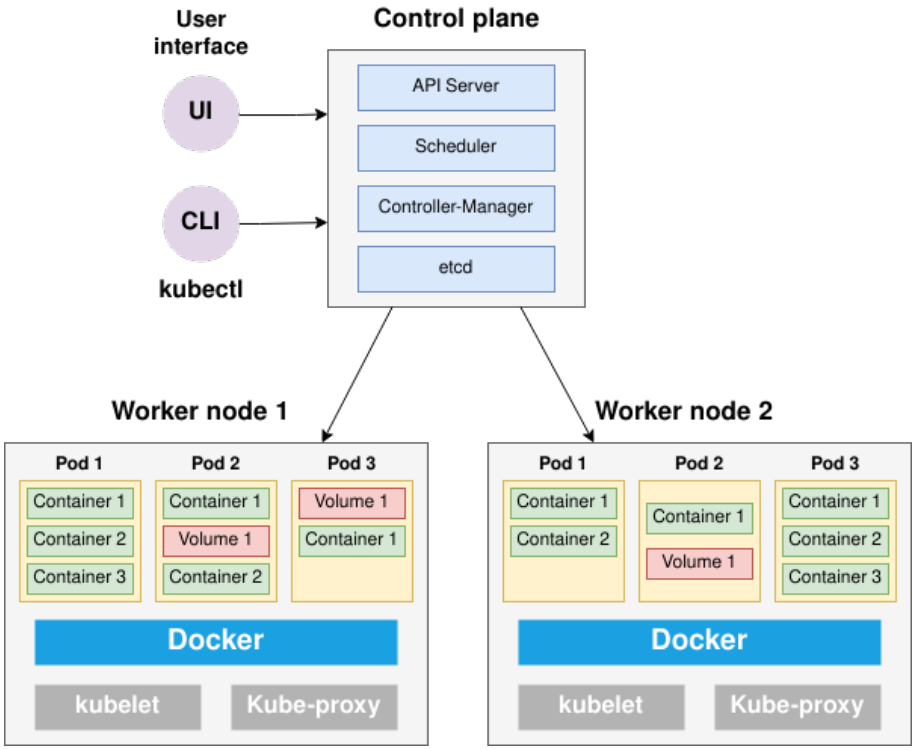

Figure 3.5: Kubernetes architecture overview; figure shows components building k8s on a basic cluster with two worker nodes, each consisting of three pods

velopment, testing, and production, as well as broad portability. Communication between containers is possible through well-defined channels. Finally, containers allow for reduced resources usage compared to virtual machines, since containers are run by a single operating system kernel. An additional advantage compared to virtual machines could also be an application-level management, since using containers raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.

In our work, leveraging Docker images and containers proved to be essential, in order to run a single workflow node in an isolated environment with all the necessary libraries and executables.

Kubernetes

Kubernetes (k8s) is an open-source platform for managing containerized workloads and services. Founded by Google in 2014, Kubernetes has been developed and improved

by great amount since, and is still a large, rapidly growing ecosystem. Its main features include: service discovery, load balancing, storage orchestration, automated rollouts and rollbacks, automatic bin packing, self-healing, secret and configuration management. K8s aims to provide building blocks for developer platforms, whilst maintaining the user choice and high flexibility.

A worker machine in Kubernetes is called a *node*. Depending on the cluster, it may be either a virtual or a physical machine. A node is comprised of one or more *pods*. A pod is an elevated-level abstraction grouping of containers and their shared resources, including but not limited to shared storage, networking and additional metadata.

Kubernetes also distinguishes *deployments*. K8s deployment is a set of instructions, describing the desired state of the system, what containers to run, their configuration, metadata, interdependence and context. The controller of the deployment is responsible for changing the actual state to the desired one at a controlled pace.

For the purposes of this study, Kubernetes is used to orchestrate workflow containers, during which we leverage its load balancing and self-healing capabilities.

3.3 Contribution to the HyperFlow project

By the effort of this work, the HyperFlow project was extended with several functionalities, necessary for the data collection process. Following we present an overview of the created features.

Montage2 workflow

We improved the Montage2 workflow port to HyperFlow[10], by creation of Docker images ready for use by worker nodes. The repository was also supplemented with a helper workflow generator, for generation of JSON file representing the workflow graph.

SoyKB workflow

The SoyKB workflow[11] was also improved, similarly to Montage2, by creating Docker images and incorporating a proper workflow generation script.

[10] `https://github.com/hyperflow-wms/montage2-workflow`
[11] `https://github.com/hyperflow-wms/soykb-workflow`

Resource usage logging

We introduced resource usage logging for HyperFlow job executor, representing the worker component running on a Docker container [12].

Kubernetes deployment

The Kubernetes deployment[13] project was extended by several automation scripts and utilities, facilitating deployment of HyperFlow on a Kubernetes cluster on the Google Cloud Platform and on the Amazon AWS cloud.

Log parser

After creation of resource utilization logging, it was necessary to implement a log parser. The parser processes the raw logging data into structured JSON Lines (jsonl) file format, where each line represents single event or metric value. The parser creates three jsonl files:

- *metrics.jsonl* – metrics related to utilization of system resources by the job, e.g. CPU, memory, I/O operations usage,
- *job_descriptions.jsonl* – contains pieces of information related to job execution parameters, e.g. workflow name, arguments list, lists of inputs and outputs and execution time
- *sys_info.jsonl* – contains information regarding the system parameters, e.g. CPU brand, speed, number of cores, total amount of memory

HyperFlow tools

A Docker image for HyperFlow tools[14] was created to facilitate their deployment on Kubernetes.

[12] `https://github.com/hyperflow-wms/hyperflow-job-executor/blob/master/handler.js`

[13] `https://github.com/hyperflow-wms/hyperflow-k8s-deployment`; the project contains configuration files and manuals to set up the HyperFlow environment on a Kubernetes cluster.

[14] `https://github.com/hyperflow-wms/hflow-tools`; the project contains a set of auxiliary scripts and tools for HyperFlow.

3.4 Data collection process

This section covers the reasoning behind the technologies selection as well as logs collection implementation choice. Following is the detailed architecture overview of the data collection system.

3.4.1 Reasoning

The goal of this part of our study was to gather as much pieces of information related to single program execution as possible. That includes common resource utilization metrics, namely CPU, memory used by the process, input-output operations, network statistics, but also metrics like ctime, cache. We also wanted to collect environmental data, which includes system environment variables, host machine components characteristics, meaning i.e. type of CPU, cores count, total memory, and also workflow-related input parameters. Originally we also wanted to collect parameters of the running command, which usually consisted of the filename intended to be read or created, and its size.

In order to easily and effectively collect such data, each command needed to be executed in an isolated, containerized environment, preferably with the necessary libraries and executables on the spot. Such isolation assures that collected data will not be contaminated by any background workloads, as every container will be running solely in purpose of fulfilling a particular task of a given workflow. Additionally, each container will have assigned fixed resources, which assures consistency between multiple executions. Docker containers proved to be suitable solution, since they meet all of these requirements.

As we decided to use workflows for command data population, a workflow management system employing Docker containers was needed. HyperFlow, among other types of execution, allows for running tasks in Docker containers, and comes with a few workflow types available.

Since some workflows may require parallelization on the level of hundreds of jobs running simultaneously, a smart orchestrating system was required to efficiently manage and reuse host machines. Running workflows using Kubernetes proved feasible, although it is not a typical Kubernetes use case. Next, Kubernetes deployments were run on a variety of cloud providers' machine instances types, which together made up for computing clusters.

Finally, the means of usage logs and system information collection, complying with our requirements, was needed to be established. There is a vast number of data collection methods and utilities related to cloud, Kubernetes and Docker environments. Each of them, depending on its assignment, collects data on different levels of abstraction.

Cloud providers equip the client with the high-level of system monitoring, allowing for fast anomaly detection in case of an emergency. On the other hand, Kubernetes and Docker provide the user with a more detailed view of resource utilization, even related to the particular container, however still lack more detailed information related to specific process or some, for the purposes of this study, necessary metrics i.e. IO operations. What is more, all of this common resource usage tools have the highest resolution of data collection of 15 seconds[15], as apparently this is the resolution of metrics calculated within a Kubelet (a primary node agent that runs on each Kubernetes node). Therefore, in order to collect:

- data within the specified, relatively smaller time frames,
- custom metrics,
- usage statistics pertaining single process,

we opted for implementing our own solution, which was integrated into HyperFlow Engine project.

3.4.2 Collected data and logs

This section briefly describes collected data and its post-processing.

As HyperFlow starts running the workflow, so does the log collection mechanism integrated into HyperFlow engine. The logs are created as the result of few JavaScript utilization-related modules[16,17]. Log collection interval is customizable, however in our cases, we settled for 2 seconds.

Logs, alongside HyperFlow specific debug log, are saved at *logs-hf* directory on the NFS server. Text data from this folder is then parsed by integrated logs-parser, creating three files in *jsonl* (JSON lines) format, containing prepared data related to:

[15] `https://github.com/kubernetes-sigs/metrics-server`
[16] `https://www.npmjs.com/package/@stroncium/procfs`
[17] `https://www.npmjs.com/package/systeminformation`

Table 3.1: Job description related parameters

Property	Description
workflowName	string uniquely identifying the workflow
size	input parameter of a workflow, usually related to its size
version	version of the workflow software package
hyperflowId	id of HyperFlow instance
jobId	id of a job
env	k8s environmental variables
podIp	IP address of the pod
nodeName	name of the node
podName	name of the pod
podServiceAccount	service account of the pod
podNamespace	namespace of the pod
executable	name of the executable the job is running
args	command arguments
inputs	input files of the task
outputs	output files of the task
name	name of the job
command	full command string
execTimeMs	execution time of the task

- job execution parameters, e.g. command arguments, inputs and outputs list,
- metrics related to utilization of system resources by the job,
- system parameters, e.g. CPU brand, speed, cores, total memory

Table 3.1 lists the jobs-descriptions-related parameters. Table 3.2 lists system-information-related parameters, namely about CPU and machine memory. Finally, Table 3.3 shows all the utilization and custom metrics.

Table 3.2: System information related parameters

Metric		Description
cpu		CPU information
	manufacturer	e.g. 'Intel(R)'
	brand	e.g. 'Core(TM)2 Duo'
	vendor	vendor ID
	family	processor family
	model	processor model
	stepping	processor stepping
	revision	revision
	voltage	voltage
	speed	in GHz e.g. '3.40'
	speedmin	in GHz e.g. '0.80'
	speedmax	in GHz e.g. '3.90'
	governor	e.g. 'powersave'
	cores	number of cores
	physicalCores	number of physical cores
	processors	number of processors
	socket	socket type
	cache	l1d (data), l1i (instruction), l2, l3 cache size
mem		system memory related parameters
	total	total memory in bytes
	free	not used in bytes
	used	used (incl. buffers/cache)
	active	used actively (excl. buffers/cache)
	available	potentially available (total - active)
	buffers	used by buffers
	cached	used by cache
	slab	used by slab
	buffcache	used by buffers+cache
	swaptotal	-
	swapused	-

swapfree -

Table 3.3: Events and utilization metrics

Metric	Description
event	
handlerStart	start time of job handler
jobStart	start time of the job
jobEnd	end time of the job
handlerEnd	end time of the job handler
cpu	CPU utilization
pid	pid of the running process
memory	memory utilization
ctime	total user + system CPU time in ms
io	CPU info related parameters
read	number of bytes read
write	number of bytes written
readSyscalls	number of read syscalls
writeSyscalls	number of write syscalls
readReal	number of bytes read which were really fetched from storage layer
writeReal	number of bytes written which were really sent to storage layer
writeCancelled	number of bytes process caused to not be written
network	network related metrics
name	interface name
rxBytes	number of received bytes
rxPackets	number of received packets
rxErrors	total number of receive errors
rxDrop	number of received packets dropped
rxFifo	number of receive FIFO buffer errors
rxFrame	number of packet framing errors
rxCompressed	number of compressed packets received
rxMulticast	number of multicast frames received
txBytes	transmitted bytes
txPackets	transmitted packets
txErrors	total number of transmit errors

txDrop	number of transmit packets dropped
txFifo	number of transmit FIFO buffer errors
txColls	number of collisions detected on the interface
txCarrier	number of carrier losses detected on the interface
txCompressed	number of compressed packets transmitted

3.5 Experimental setup

This section describes the deployment architecture of the system used to run workflows and collect experimental data.

For the purposes of this study we used the services of two cloud providers, namely Google Cloud Platform (GCP) and Amazon Web Services (AWS). Tab. 3.4 shows an overview of used compute machine instances.

Table 3.4: Overview of used compute instance types from GCP and AWS cloud platforms

Compute machine type	vCPUs	Total memory [GB]	Usage/Characteristics
e2-standard-4	4	16	cost-optimized
n1-standard-4	4	15	first generation general-purpose
n2-standard-4	4	16	second generation general-purpose
n2d-standard-4	4	16	second generation AMD EPYC Rome processor
c2-standard-4	4	16	latest generation Intel Scalable Processors (Cascade Lake)
t3.large	2	8	burstable general-purpose
t3.2xlarge	8	32	burstable general-purpose

The dataset includes also several arbitrary machine types, which will later on be marked as *unassigned*.

A Kubernetes cluster consisted of between 3 to 6 compute machine instances, each forming a k8s node. On each cluster, HyperFlow Kubernetes deployment[18] was placed. In our case, typically cluster was divided into two node pools: *hfmaster*, where HyperFlow engine was run, and *workers*, which were the instances that jobs were executed on.

[18] `https://github.com/hyperflow-wms/hyperflow-k8s-deployment`

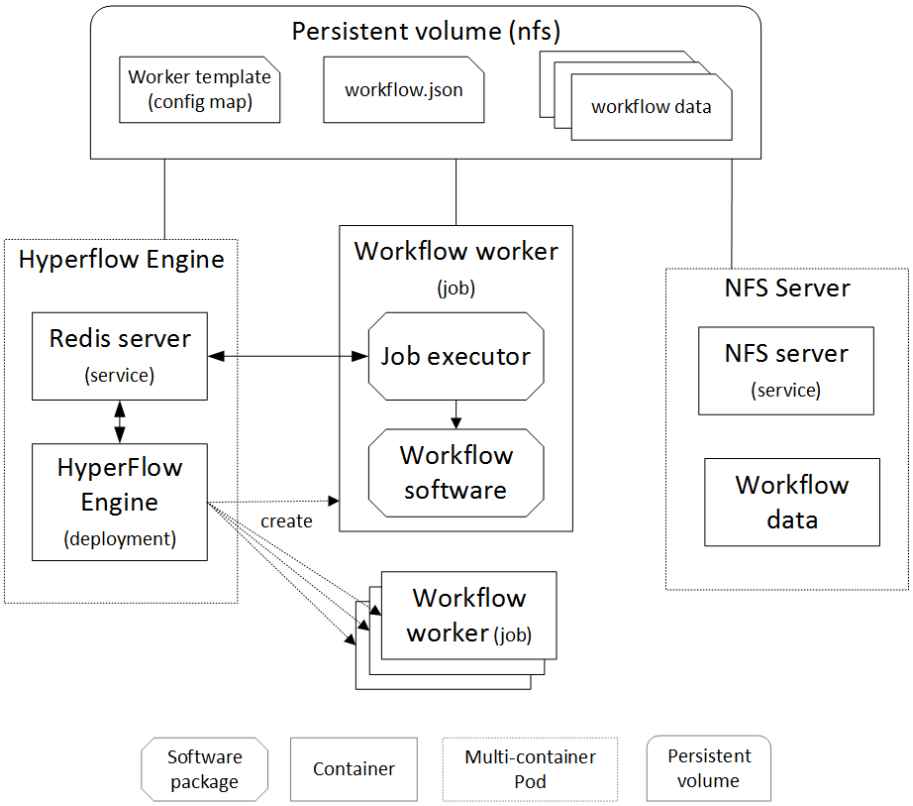

Figure 3.6: Hyperflow Kuberentes deployment architecture overview. Source: HyperFlow k8s deployment repository: `https://github.com/hyperflow-wms/hyperflow-k8s-deployment`

Fig. 3.6 depicts HyperFlow's k8s deployment architecture. It consists of several sub-modules. First one comprises of HyperFlow Engine container, running HyperFlow WMS executable, and Redis dictionary database server container, used for exchanging data between worker containers and HyperFlow Engine. Together, they create single multi-container k8s pod. Second sub-module is a persistent volume, storing:

- file *workflow.json*, representing the workflow structure,
- workflow-specific data,
- worker container template in YAML format.

Next is NFS server container, mounting to the persistent volume, allowing for data sharing and maintaining consistency between all the containers. Last is a *worker* container, running with the specialized workflow software in place. Single worker container is re-

sponsible for executing single command, denoting one node in workflow's graph. In a majority of cloud providers cases, it is possible to establish different node pools in a single cluster. We leveraged this functionality to divide our deployment into two separate node pools: one denoted solely to workflow management, second to running only workflow worker containers. The first node pool uses normal computing machine instances, which are utilized for running HyperFlow engine, persistent volume and NFS server. For the second node pool we used the so-called *preemptible instances*, which grant reduced availability in exchange for less than half the original price. The rationale for this is that, depending on the workflow, it may be required to run thousands of worker jobs, which are expected to be running for a relatively short period of time, therefore the reduced availability will not be an issue in this case. The division of node pools was primarily driven by the purpose of utilizing cheaper instances, thus saving resources.
The procedure of running a workflow is as follows:

1. Create k8s capable cloud cluster
2. Establish two different node pools: hf-master and workers
3. Deploy HyperFlow k8s deployment on the cluster
4. Wait until NFS server is mounted on the HyperFlow container
5. Start running workflow
6. Wait until workflow finishes
7. Start post-processing logs
8. Wait until post-processing finishes
9. Send post-processed data to the external bucket storage

Chapter 4

Prediction methodology

The purpose of this chapter is to describe the techniques employed in this work during prediction model training. Firstly, we will outline our study design. Secondly, we will characterize distinguished datasets. Finally, the usage of dimension reduction methods and binary classification, as well as various regressors types will be discussed.

4.1 The study design

This work focuses on developing a resource utilization prediction mechanism, which is supposed to assist a resource allocator to provision resources at an optimal level. Collected dataset, which shall be used to train various prediction models, contains pieces of information related to system, events, utilization and workflow job description. Utilization and event metrics have been collected with 2 s interval during running of each task. Using collected metrics, we will establish a two-step prediction system, able to predict execution time of a particular task. In the first step we will perform binary classification using logistic regression, thus estimating if a specific resource was intensively utilized or not. The threshold value for the intensity labeling process was assumed as a mean value of a resource from the logs in scope of the entire dataset. Then, based on these models and system configuration data from jobs, we will predict execution time of a job, leveraging multiple implementations of regression. Lastly, we shall compare results and distinguish the method yielding best results.

4.2 Prepared data sets

Data from three types of workflows, totalling to 262 runs has been collected. From these workflows, we can distinguish 28 different types of jobs. Summing up all jobs, there are a total of 185079 job runs, each represented by information about system, events, utilization, collected in the interval of 2 seconds. All of those metrics add up to 34 different features[1] in our dataset. Workflows have been run on 7 different machine instances. The distribution of workflow runs of different types of workflows is shown in Fig. 4.1. The *unassigned* cases are a collection of runs deployed on machine instances whose type is unknown.

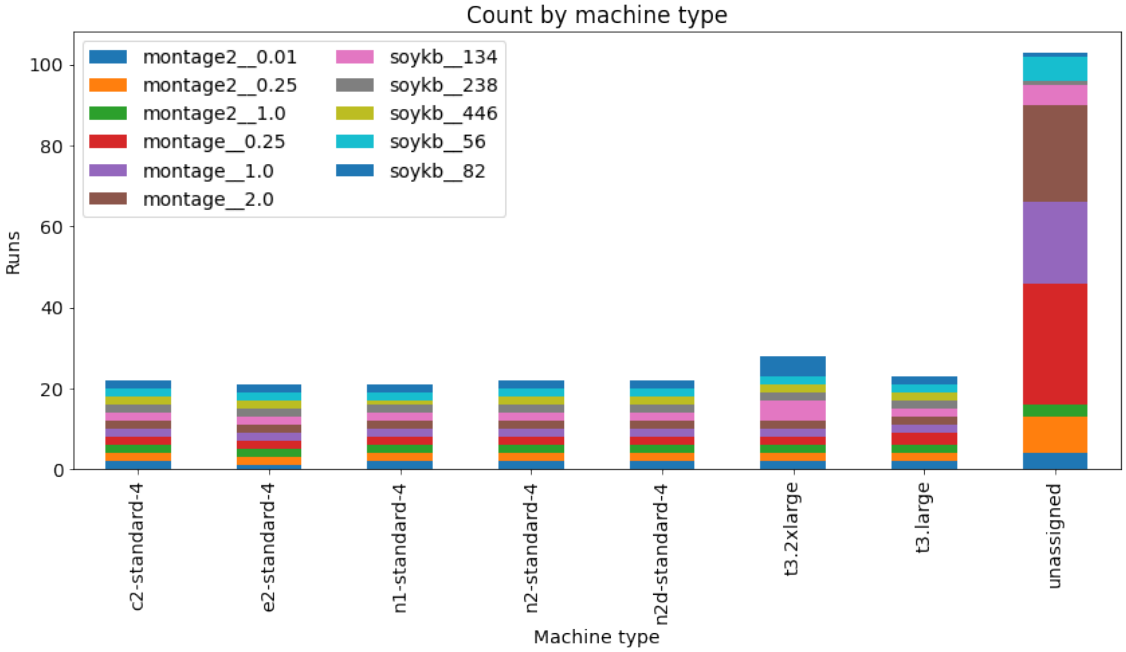

Figure 4.1: Summary of the collected workflows grouped by machine instance type

The occurrence frequency of specific job instances, shown in Fig. 4.2, is unevenly distributed in our dataset, which was expected, given the structure of the workflow graphs. The least frequent task is *merge_gcvf* from the SoyKB workflow (about 100 instances), while the most frequent job is Montage's *mDiffFit* (about 100,000 instances).

Another important characteristic while analyzing the dataset is the execution time of jobs. Fig. 4.3 presents histogram of the execution time of jobs in the collected dataset.

[1]Several metrics were dropped because of redundancy

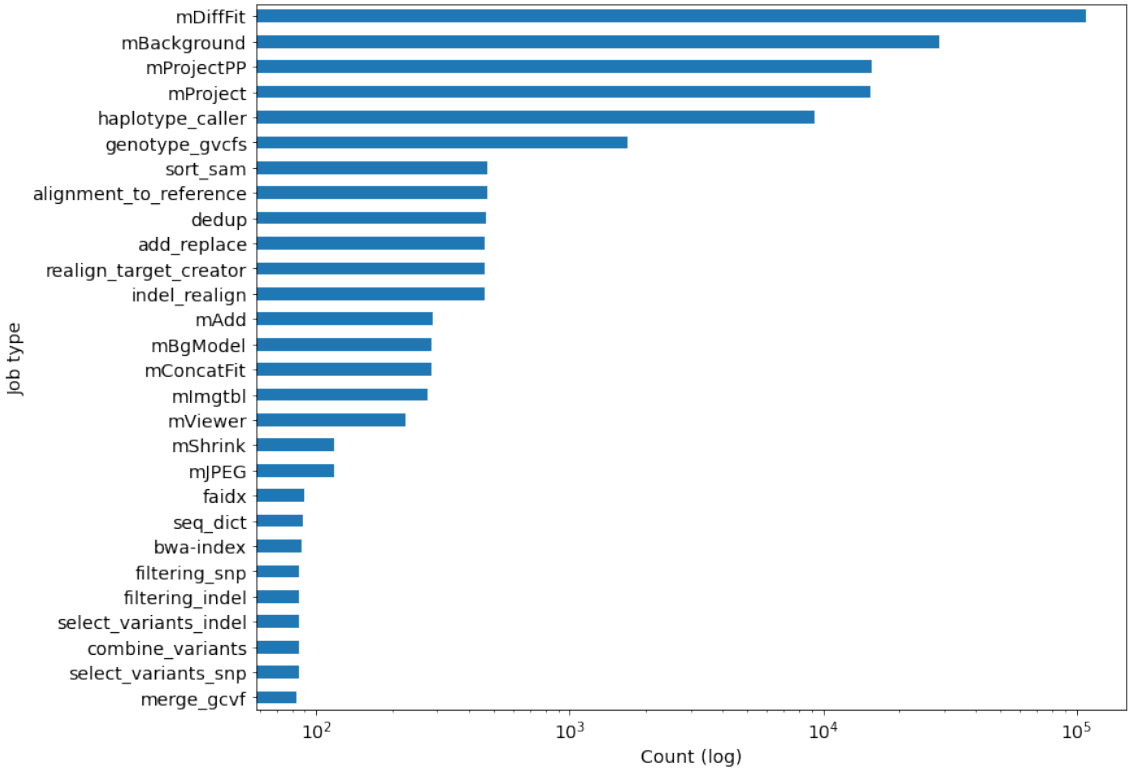

Figure 4.2: Count of job instances in the dataset by job type. The camel-cased names denote jobs from Montage workflows, whereas the snake-cased ones are from SoyKB workflow.

As we can see, the majority of jobs have execution time of less than 1 second. These are mainly the tasks from Montage workflows, e.g. the aforementioned *mDiffFit*.

Fig. 4.4 presents the comparison of average task execution time and machine instance type, distinguished by CPU brand, speed, number of cores and total memory. The values generally follow an intuitive pattern: machines with faster CPUs performed significantly faster. What is more, regardless of the number of cores and total memory instances with the same CPU speeds share average execution time on a similar level. In fact, in some cases higher total memory indicated on longer execution times.

Given the nonuniform job-type-to-count distribution in our dataset, it was considered appropriate to divide it and distinguish 4 subsets representing tasks with similar context, such as the execution time or job counts. This division will later on allow us to compare effectiveness of different classification and prediction methods of different dataset types. It may also be considered as a form of manual clusterization, aiming to reduce drastically

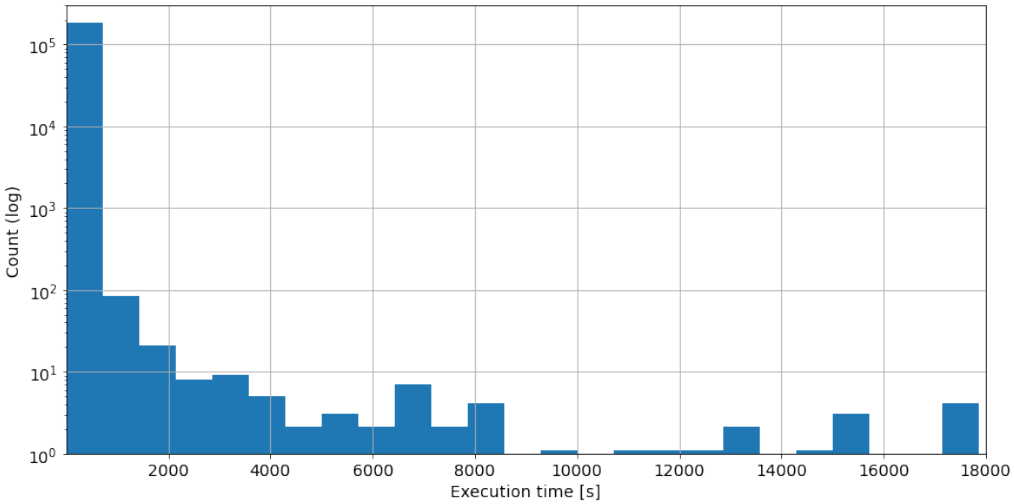

Figure 4.3: Distribution of job execution time in the collected dataset.

outlying values e.g. for the dataset containing 10^5 runs of short jobs, a few jobs lasting several houres might be considered a deviation. We intended to pre-isolate such cases, with the premise of better results.

With the above in mind, we differentiated the following 4 datasets:

1. All types of jobs run on all types of machines
2. Jobs with small execution time, lower than 3s [2]
3. Jobs with average execution time, between 4000ms and 25000ms[3]
4. Most frequently occurring jobs, with count greater than 3000[4]

Fig. 4.5 shows a comparison of histograms of execution time of jobs in the distinguished 4 datasets.

In order to provide suitable training data, each of the 4 datasets has been shuffled and uniformly distributed in the context of a job type, resulting in datasets that have the same amount of jobs of a given type. Fig. 4.6 shows a summary of occurrences of jobs in each dataset.

For the used machine learning methods, all categorical features have been transformed using the *Min-Max scaler*, so that each of them has a normalized value between 0 and 1.

[2]Threshold values were selected experimentally, based on the collected data.
[3]Refer to 2
[4]Refer to 2

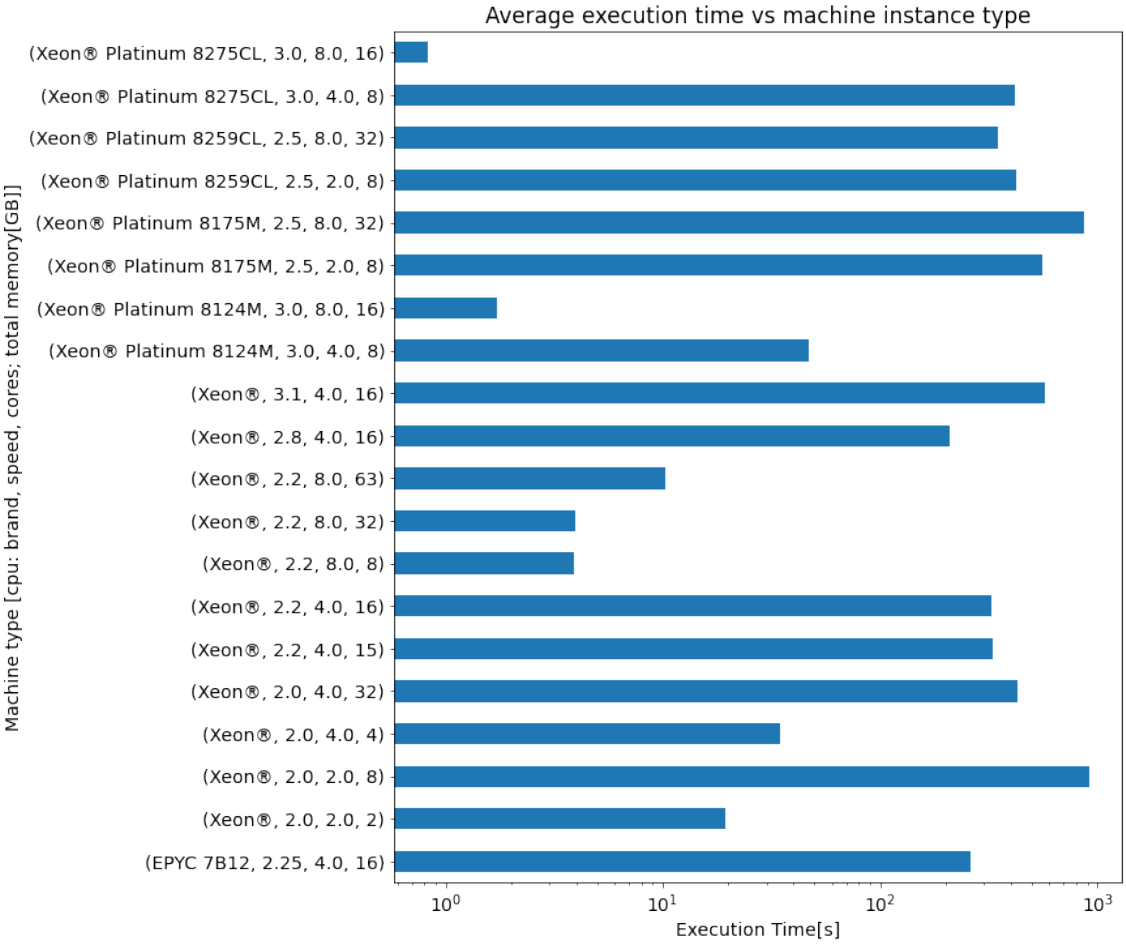

Figure 4.4: Comparison of average job execution time per machine instance type, characterized by its CPU brand, speed, number of cores and total memory.

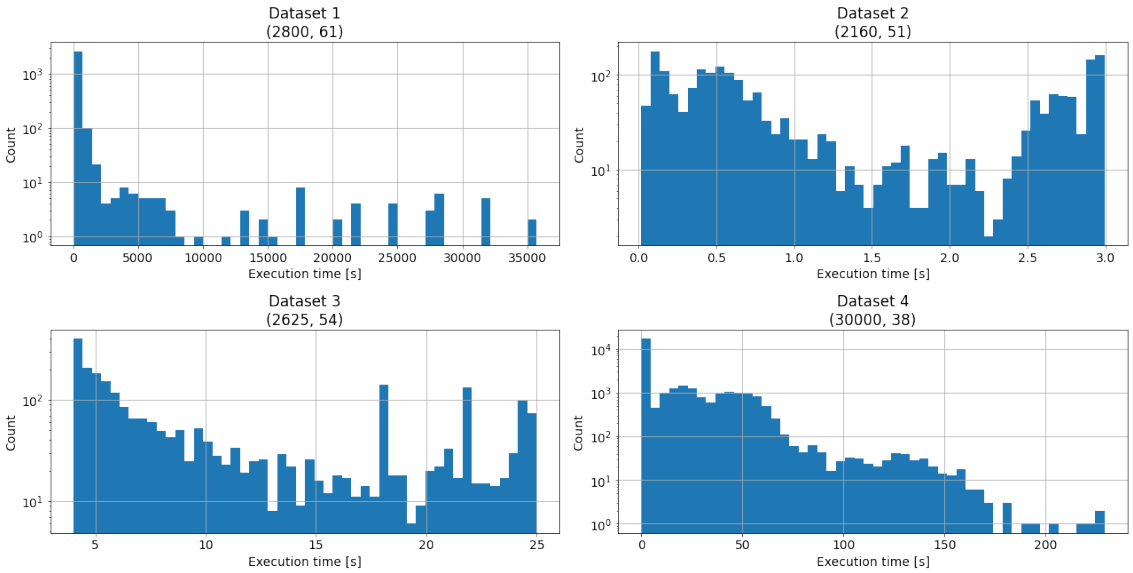

Figure 4.5: Comparison of histograms of execution time of jobs for all 4 datasets. The two-item tuple below the title indicates the shape of a dataset (rows, columns)

This is the reason why not all datasets have the same column size, since Min-Max scaler populates categorical data values into columns. Several metrics pertaining to runtime utilization statistics of a task have also been removed from the datasets, as these would not be known prior to tasks execution, hence are useless for a prediction model training. Also, some highly correlated features were reduced as well.

4.3 Dimension reduction methods

This section covers dimension reduction methods used in data analysis. Dimension reduction is the method of transforming high-dimensional data into a low-dimensional space, ideally so that the new representation retains relevant properties of the original data. Such transformation is usually beneficial, since processing high-dimensional data is more compute-intensive and it is difficult to visualize. Dimension reduction can also be used for noise reduction, cluster analysis or as an intermediate step into further analyses. Methods for dimension reduction commonly are divided into linear and non-linear approaches, whereas those can be subdivided into feature selection and feature projection (also called feature extraction) methods.

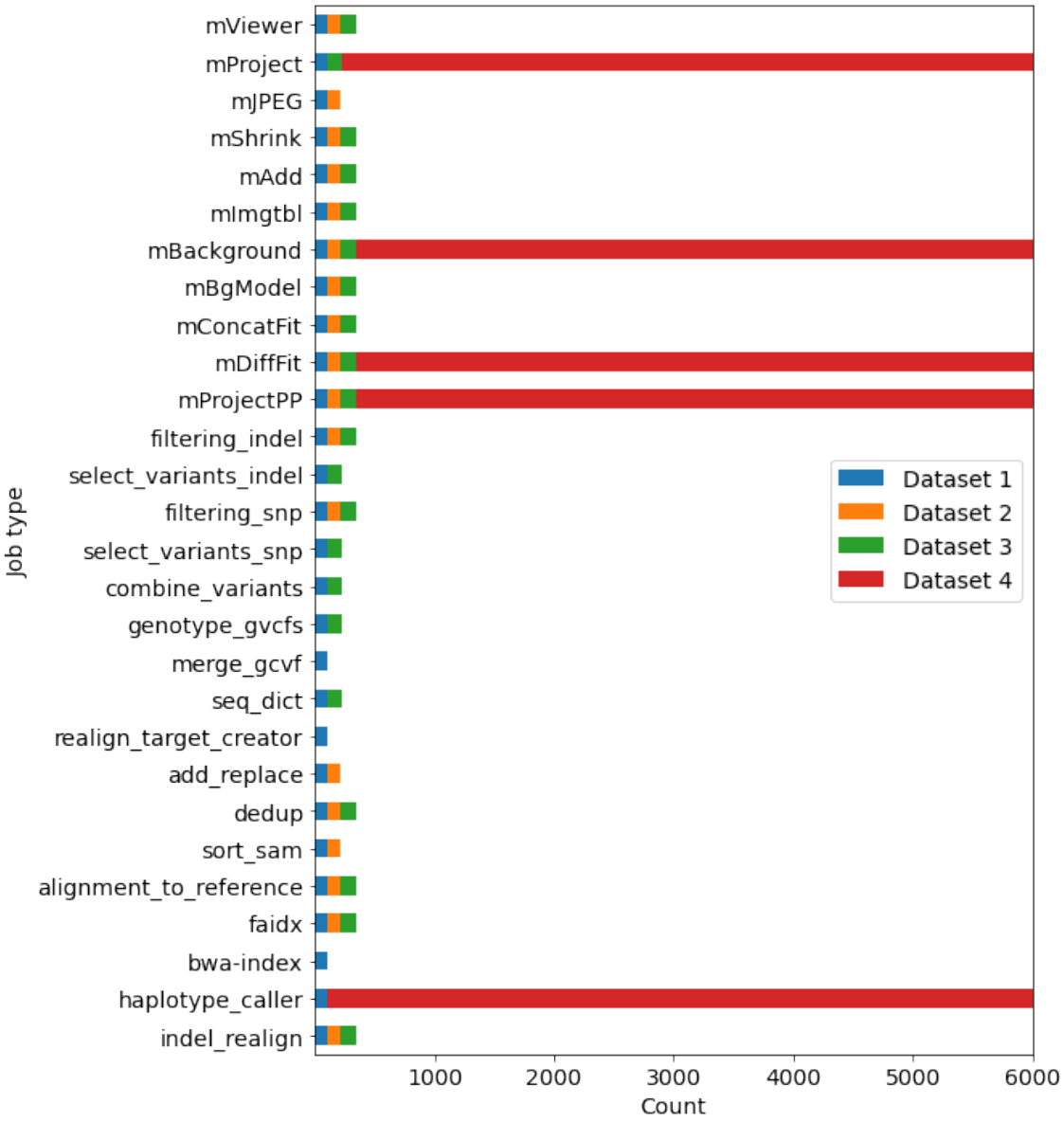

Figure 4.6: Job instances distribution for all 4 datasets.

In our case, dimension reduction has been used for data visualization purposes. For comparison, two different methods for data visualization of a dimensionally reduced data have been used: PCA and t-SNE.

4.3.1 PCA

Principal Component Analysis (PCA) [5] is the most common linear technique used for dimensionality reduction. Similarly, it can also be used for noise filtering, visualization of high-dimensional data and feature selection. Because of its versatility and straightforward interpretation, PCA has proven useful in a vast variety of disciplines. It works by finding orthogonal projection vectors that account for the maximum amount of *variance*[5] in the data. PCA usually utilizes singular value decomposition (SVD) in order to find the eigenvectors that capture the inherent patterns of the dataset. The eigenvectors corresponding to eigenvalues are used in counting of the explained variance ratio. Principal components are then ranked in order of their explained variance ratios. Found principal components can then be used to reconstruct an essential fraction of the variance of the original data.

4.3.2 t-SNE

T-distributed Stochastic Neighbor Embedding (t-SNE), contrary to PCA, is a non-linear dimensionality reduction technique, also useful for visualization of high-dimensional datasets. Introduced in [24], the technique is becoming increasingly popular in the field of machine learning and data science, due to its ability to create plausible two or three dimensional mapping of a data characterized by a high-dimensional space. The method adapts to the underlying data and performs different transformations on different regions. The algorithm comprises of two main stages. In the first stage, t-SNE creates a probability distribution over high-dimensional objects so that similar objects get assigned a higher probability, whereas dissimilar ones are assigned a lower probability. In the second stage, a similar probability distribution over the points in the low-dimentional map is defined. Finally, t-SNE minimizes the Kullback-Leiber divergence [6] between those two

[5] Variance is a measure of how far a set of numbers are spread out from their average value.

[6] Kullback–Leibler divergence is a measure of how one probability distribution is different from a second, reference probability distribution.

distributions with respect to the locations of the points in the mapping.

t-SNE is strongly influenced by its parameters, thus, in order to appropriately assess the meaning of the visualizations, proper fine-tuning of parameters is needed. Perhaps the most important tuneable parameter of t-SNE is *perplexity* which defines the extent of ”nearest neighborhood” of points in which the similarity is calculated. In the original t-SNE paper [24] it is suggested to set the perplexity values between 5 and 50. In our case, we set it to 40.

4.4 Binary intensity classification

Our first-step prediction consists in estimating the *intensity* of a particular resource in a job. We decided to adopt a binary intensity scale, namely *intensive* and *non-intensive*. Such classification can be achieved using statistical binary classification, a method widely used in medical testing, social sciences, quality control as well as machine learning. There are many methods commonly used for binary classification which include neural networks, random forests, decision trees, support vector machines (SVMs) and logistic regression. In our case, we decided to train the prediction models based on logistic regression. Logistic regression [18], contrary to linear regression, is applied to estimate a number of discrete classes rather than continuous variables. The input values are combined linearly using coefficient values, while the output values are binary.

The name of the method comes from the logistic function, by which the method is characterized. This function models the conditional probability of the dependent variables Y by independent variables X. The logistic function is an example of a sigmoid function, which is an S-shaped curve, that takes any real-valued numbers and maps them into a value greater than 0 and lower than 1. The logistic function is defined by the formula:

$$f(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}.$$

The function converts the logarithm of the odds (*log-odds*) to probability. Assuming some base threshold, the probabilities can be evaluated to binary values.

In order to feed the training data to the logistic regression model, data firstly needed to be labeled. For this purpose, we used the mean value of a resource usage metric over the entire dataset as a threshold value for labelling jobs as either *intensive* or *non-intensive*.

Dataset prepared in such a way was then used to train the model, with the train split of 75% of the data, and the remaining 25% were used for testing purposes. The training had set maximum of 1000 iterations and used the *lbfgs* algorithm in the optimization problem.

4.5 Execution time prediction

The second-step prediction estimates the execution time of a job. In this part, features present in our dataset, supplemented with binary-classified resource intensity information, are fed into training of several multivariable regression models, described below.

4.5.1 Regression analysis

Regression analysis [12] is a collection of statistical processes used to estimate the relationship between a dependent variable (*outcome variable*) and one or more independent variables (also called *predictors* or *features*). This type of analysis is used mainly for two distinct reasons. The first one is predicting and forecasting which is the reason why regression is important in the field of machine learning. Second, regression analysis can be used in certain cases to derive causal relationships between the dependent and independent variables. For the purposes of this work, regression analysis has been used for its predicting properties, aiming to estimate the execution time of a job.

Regression models incorporate the following components:

- the **independent variables**, also called **features** in machine learning studies, are observed in data
- the **dependent variables**, also called **target** variables or **label** attributes, which are also observed in data and are the values expected to change when the independent variables are manipulated
- the **unknown parameters** of a model, which are calculated during the training process
- the **error terms**, representing random statistical noise or a bias

There is already a plethora of regression analysis methods, among which we can distinguish linear regression, polynomial regression, logistic regression, nonlinear regression, non-parametric regression and many others.

The simplest variant of regression is the linear regression, where the target variable is a linear combination of the parameters related to the observed features. Typically, this comes down to finding the line that most closely fits the data according to a particular mathematical criterion. In the case of linear regression, this task is usually achieved with the usage of *ordinary least squares* method which minimizes the sum of the squares of the differences between measured target variables in the given dataset, and those predicted by the linear function. When the relationship is not linear, we are dealing with nonlinear regression, and the sum of squares has to be minimized by an iterative process.

Regression analysis can be realized using a variety of different methods. In this study, we performed training of the regression models based on four different techniques, which are described in the following subsections.

4.5.2 Decision tree

Decision tree is a fairly simple and easily interpretable algorithm used to solve a wide variety of problems. They are commonly used in machine learning as well as for statistical purposes. Decision trees perform well while dealing with large datasets, and usually do not require complex data preparation. They can be used for decision or regression analysis, as well as classification.

The technique at its core relies on the decision tree in the process of training its predictive models. As the method iterates over the dataset, data is being broken down into smaller subsets. Simultaneously, the corresponding decision tree structure is being created. It consists of *decision* and *leaf nodes*. The first one can only have more than two branches, each reflecting values for the tested attribute. The second kind of nodes is a representation of a decision on a target value. For regression trees, output variables take continuous values.

In our model, we used default parameters as defined by sklearn[7] platform. The function measuring the quality of a split was *mean squared error* (MSE). The split required minimum of 2 samples and was chosen by the best value of the criterion (MSE function).

[7] `https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html`

4.5.3 Random forest

Random forest is an ensemble, supervised learning method, used for classification and regression. Ensemble methods utilize the combined possibilities of multiple machine learning algorithms, in order to obtain more accurate predictions, compared to any individual model alone. We can distinguish two different types of ensemble learning: *boosting* and *bootstrap aggregation*.

Boosting leverages weighted averages to make poorly performing methods into more performant ones. Each model creates the chain of algorithms, where each one dictates the next one what features it should focus on. Bootstrap aggregation (*bagging*), on the other hand, is another set of meta-algorithms. It runs each model independently and individually. At the end of training process, bagging aggregates the outputs without preference to any method. Such approach can also significantly reduce the variance, which is a desired feature when dealing with decision trees.

Random forest is a method belonging to the family of bagging techniques. It relies on creating multiple decision trees during the training and outputting the mean prediction of the individual trees. The random forest technique is an improvement to decision trees method, which is prone to overfitting to the training set.

The advantages of random forest method include efficient execution on large datasets, handling thousands of input variables without variable reduction, being effective for estimating missing data and generating an internal unbiased estimate of the generalization error.

Regarding the parameters of the model used, the number of decision trees to be created was set to 100. As to the parameters of decision trees, they were set the same as in the previous method, with quality of split function as MSE.

4.5.4 *k*-nearest neighbors

k-nearest neighbors (*k*-NN) is a non-parametric algorithm used commonly in statistical estimation and pattern recognition. It can be utilized for regression as well as classification. For the regression purposes, the input of the model involves the k nearest training examples in the feature space, whilst the output is the average of the values of k nearest neighbours. Assigning weight to the contribution of the neighbours can be a useful technique both in classification and regression. Weights are calcuated with respect to the

inverse of the distance of the k nearest neighbours.

k-NN is an example of instance-based learning, where new problem instances are compared with the ones seen in training, stored in memory, instead of performing explicit generalization. It is also a type of lazy learning, where all computation is deferred until function evaluation.

Since k-NN is approximated only locally, and the algorithm relies on distance for classification, it is extremely sensitive to the local structure of the data and can provide better accuracy if the training data is normalized, which is what we incorporated into our learning process.

Regarding the hyperparameters of the model, based on the trial and error method, we decided to set the number of neighbours parameter k to 10, and to count weight points by the inverse of their distance, instead of the uniform weighing, where all points are weighted equally in each neighbourhood.

4.5.5 Artificial neural network

Artificial neural networks, commonly called just neural networks (NNs) are methods at its core inspired by the biological neural networks of the human brain. As brain consists of a set of connected nodes, called neurons, that can transmit signals to other neurons, an ANN has artificial neurons. Each neuron, upon receiving the signal in a form of a real number, processes it and sends appropriate information to other neurons connected to it. The connections between the nodes are called *edges*. Both neurons and edges usually characterized by weight factor associated to them, which is adjusted during the learning process.

The neurons are arranged in a structure of layers, where each one may perform different transformations on its input. First layer is called the *input layer* and this is the layer where the initial signals are travelling from. The last layer, responsible for producing the final result, is commonly referred to as the output layer. In between these two layers there can also be additional *hidden layers*. A neural network equipped with more than one hidden layer is known as a deep neural network (DNN). Hidden layers increase the complexity of the model but may also potentially improve prediction performance.

The hyperparameters, meaning the parameters characterizing the learning process, of ANN include learning rate, the number of hidden layers, batch size, activation function

for the hidden layers, loss function and the solver for weight optimization.

Since neural networks are able to reproduce and model nonlinear processes, its applications are immense. Those include: cruise control for autonomous cars, trajectory prediction, pattern recognition, medical diagnosis of many kinds, data mining, visualization, filtering, cybersecurity and an abundance of others.

Similarly to the previous methods described, ANNs, among many other usage examples, are capable of performing classification and regression analysis. For our case, we utilized a deep neural network with two hidden layers, which is shown in Fig. 4.7. Both input and the first hidden layer consists of 50 neurons, whereas the second hidden layer has 25 neurons. The output layer is just one neuron. For all first three layers, we decided to use activation function described as the rectified linear unit (ReLU), which compared to traditionally used sigmoid function, or other similar activation functions, allows for faster and more efficient training of DNNs on large and composite datasets.

In order to prevent our model from over-fitting, we introduced a dropout of 20% of neurons in the learning process. Dropout is a technique used for regularization of ANNs that helps to reduce interdependent neuronal learning, by randomly ignoring a certain amount of neurons during the learning process. As a loss function, we used the mean absolute percentage error, which will be explained in more detail in the next chapter.

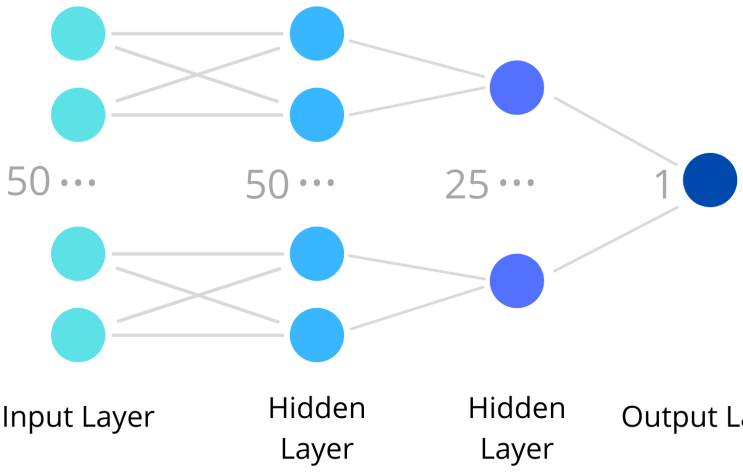

Figure 4.7: Visualization of artificial neural network used. The ANN consisted of 50 neurons in the input layer, another 50 in first hidden layer, next 25 in the following hidden layer and finally single artificial neuron in the output layer.

4.6 Implementation

For the purposes of training the previously mentioned models, and using all statistics and machine learning tools, a Colab Notebook[8] was created, available at the *matplinta/execution-time-prediction* github repository[9].

For the purposes of implementation of this study, the following libraries have been used, all derived from Python 3 programming language:

- matplotlib[10], for plots and visualizations,
- pandas[11], for data storage and manipulation in a form of DataFrames,
- scikit-learn[12] for scaling and dividing data, applying various regression analysis variants to the dataset, and for PCA and t-SNE classification visualization,
- TensorFlow[13] for building and training the neural network model

[8] https://colab.research.google.com/

[9] https://github.com/matplinta/execution-time-prediction/blob/master/execTimePredict.ipynb

[10] https://matplotlib.org/

[11] https://pandas.pydata.org/pandas-docs/stable/index.html

[12] https://scikit-learn.org/stable/

[13] https://www.tensorflow.org/

Chapter 5

Evaluation of the results

5.1 Visualization of dimensionally-reduced data

In this section, we analyze the visualizations of each dataset, reduced to two-dimensional space. We will compare two methods, namely PCA and t-SNE.

Before attempting to classify the intensity of resources of tasks, it was important to establish whether or not the experimental data exhibits some correlation between its features and labeled intensity parameters. Four parameters have been considered: *cpu*, *memory*, *io.read* and *io.write*.

Fig. 5.1 shows the results of PCA and t-SNE dimension reduction on dataset 1, containing all types of jobs. For the *cpu* parameter, both PCA and t-SNE failed to distinguish *intensive* and *non-intensive* cases. The created shapes mostly overlap with each other. For *memory* parameter, the shapes still overlap for *intensive* and *non-intensive* cases, however there are definitely a few clusters of *non-intensive* cases only. Some of them perhaps would benefit from noise removal using outlier detection, as we only see single points of *intensive* cases in them. Parameters *io.read* and *io.write* exhibit similar structure, with the better distinction on the *io.write* side, which is especially visible on the PCA chart.

Fig. 5.2 presents visualisation of dimensionally-reduced dataset 2, containing only the short jobs, of execution time lower than 3 s. For *cpu*, PCA results clearly distinguish between two values, along the vertical axis. t-SNE results, on the other hand, similarily to previous dataset overlap in a majority of points, however we can pinpoint several clusters of data points of the same value. For parameter *memory*, PCA fails to clearly distinguish

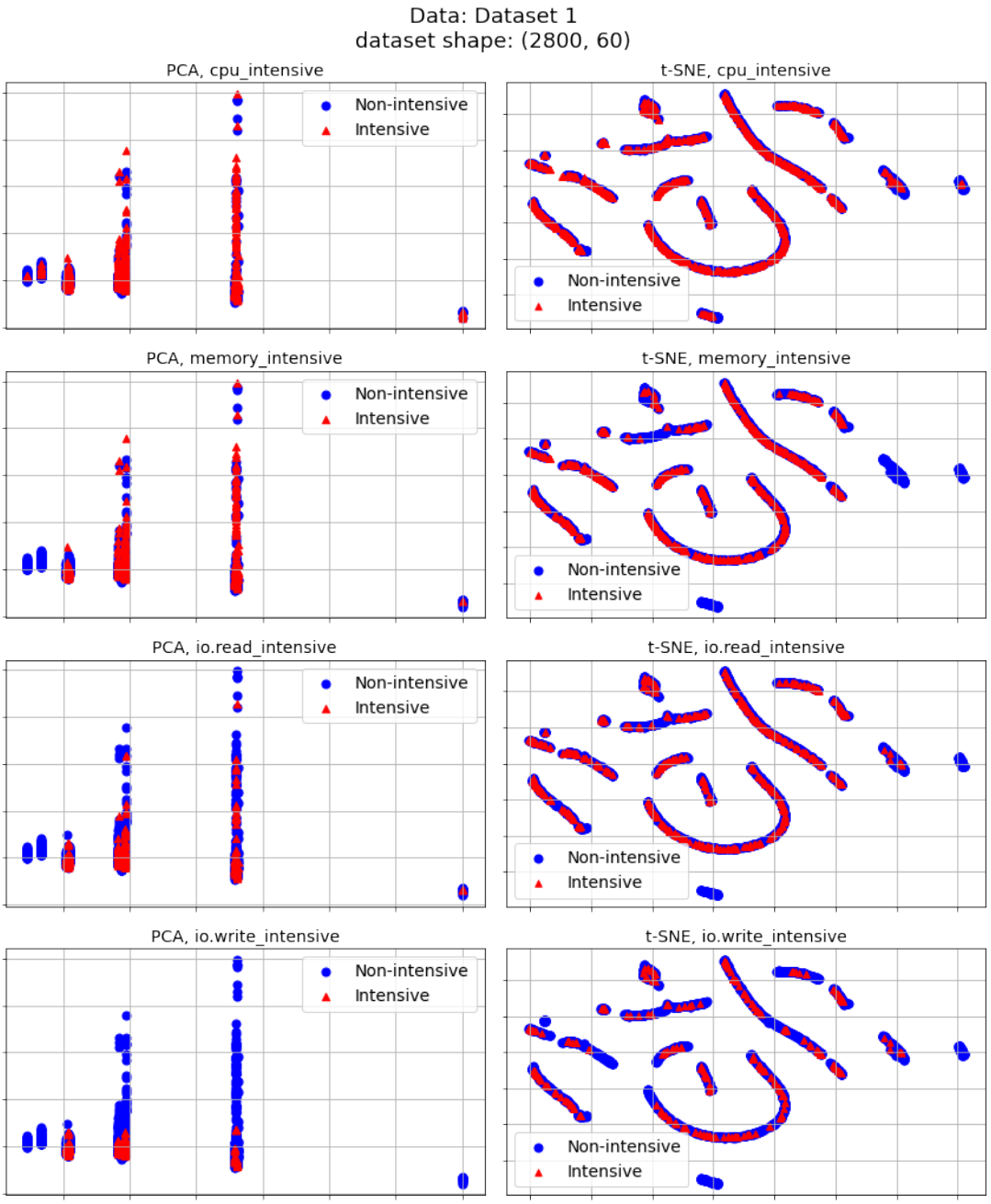

Figure 5.1: PCA and t-SNE projection to 2D for dataset 1. t-SNE created with parameter *perplexity* = 40

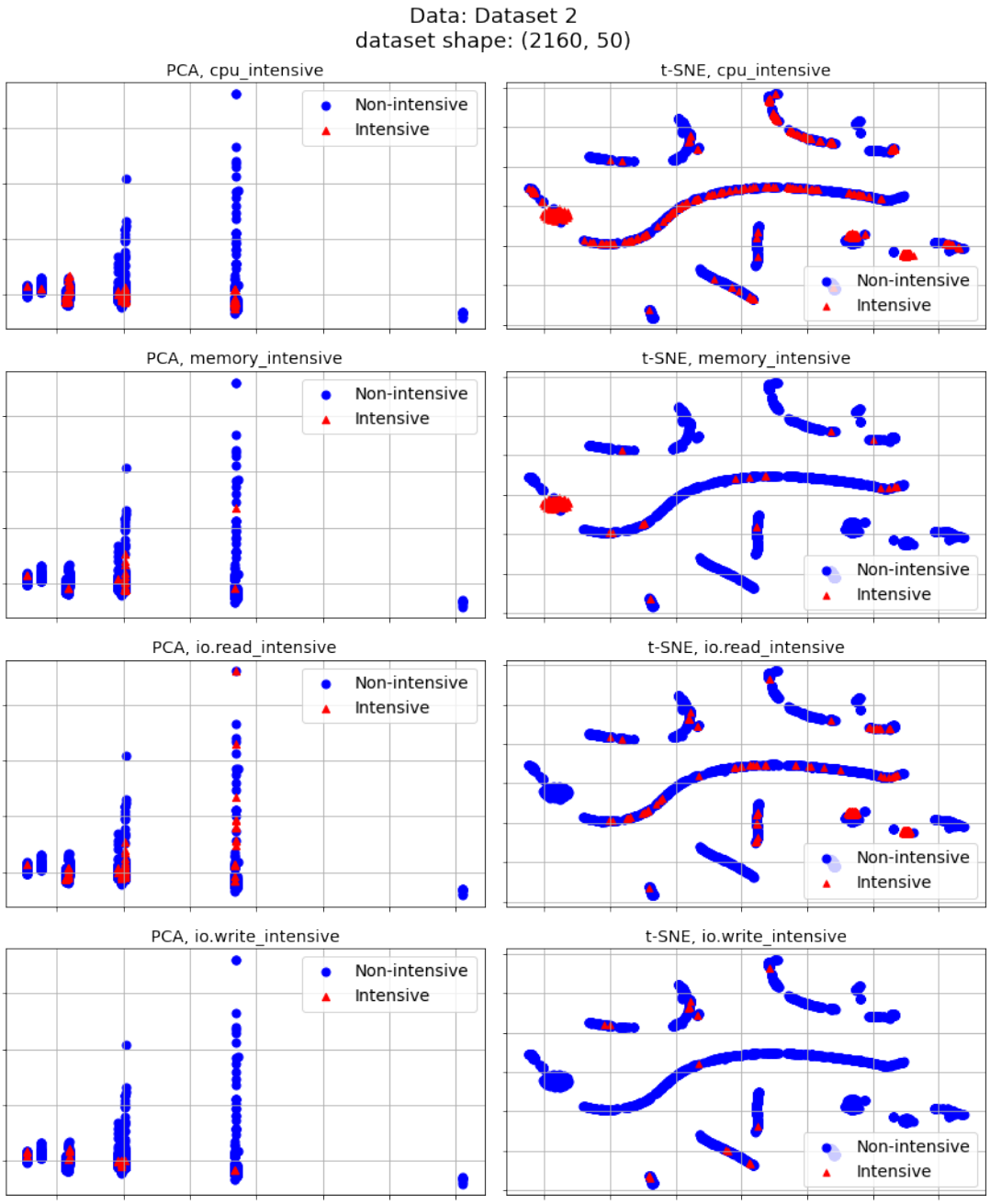

Figure 5.2: PCA and t-SNE projection to 2D for dataset 2. t-SNE created with parameter *perplexity* = 40

the two cases. t-SNE in this case does a better job, with a clear cluster of *intensive* points on the left side of the graph, and the rest of the clusters with *non-intensive* cases with occasional noise. PCA for *io.read* again failed to distinguish the two cases. As for t-SNE, we can clearly spot several clusters of *non-intensive* cases. For *io.write* in dataset 2 the *non-intensive* values dominate, which is visible on the graphs. PCA looks similar to *cpu* parameter's graphs, with vertical distinction between the two cases. t-SNE is outbalanced by clusters of *non-intensive* cases, with scarce amounts of *intensive* points in between.

Fig. 5.3 shows PCA and t-SNE graphs for dataset 3, representing jobs with mediocre execution time. This time, in the case of *cpu*, PCA were able to differentiate some small clusters of *intensive* and *non-intensive* points, however, data points generally overlap. The t-SNE equivalent plots also show this relationship, however there are clearly established two clusters of *non-intensive* and *intensive* cases respectively, on the opposite sides of the graph. One can also spot several small *intensive* clusters throughout the whole visualization. In the case of *memory*, the previous statements still hold true, although even more distinctively: on the PCA side, we can see some dispersion in the middle, mostly for *intensive* points, whereas for t-SNE there are even more clusters of *non-intensive* points, with a few *intensive* outliers and one relatively large cluster of those values. As for *io.read* and *io.write*, PCA still looks the most separated, compared to other studied resources. On the other hand, t-SNE graphs, especially for *io.write*, show great domination of *non-intensive* clusters, only with some outlier cases and a single cluster of *intensive* points on the far-right part of the chart. The graph for *io.read* looks somewhat like the opposite of *io.write*, with the main formed clusters of the opposite values. For the rest of the plot the majority of points overlap.

Fig. 5.4 pictures visualizations for dataset 4, constituting only few, most occuring jobs of our dataset. Larger size of this dataset allowed for seemingly better results, which is especially visible on the t-SNE parts with the bigger clusters. For parameter *cpu*, in the PCA chart, we can see clearly dominance of *intensive* cases, especially on the middle main line, where in the upper part of the chart there are only these cases. Several *non-intensive* points can be spotted overlapping at the bottom of the chart. t-SNE, on the other hand, while still having a majority of overlapping points and clusters, we can spot some distinctive clusters of *non-intensive* points in the middle and on the far-right side of the graph. For the *memory* label, PCA graphs clearly divide to *non-intensive* cases on the sides, and similarly *intensive* points in the upper middle. On the t-SNE chart, there

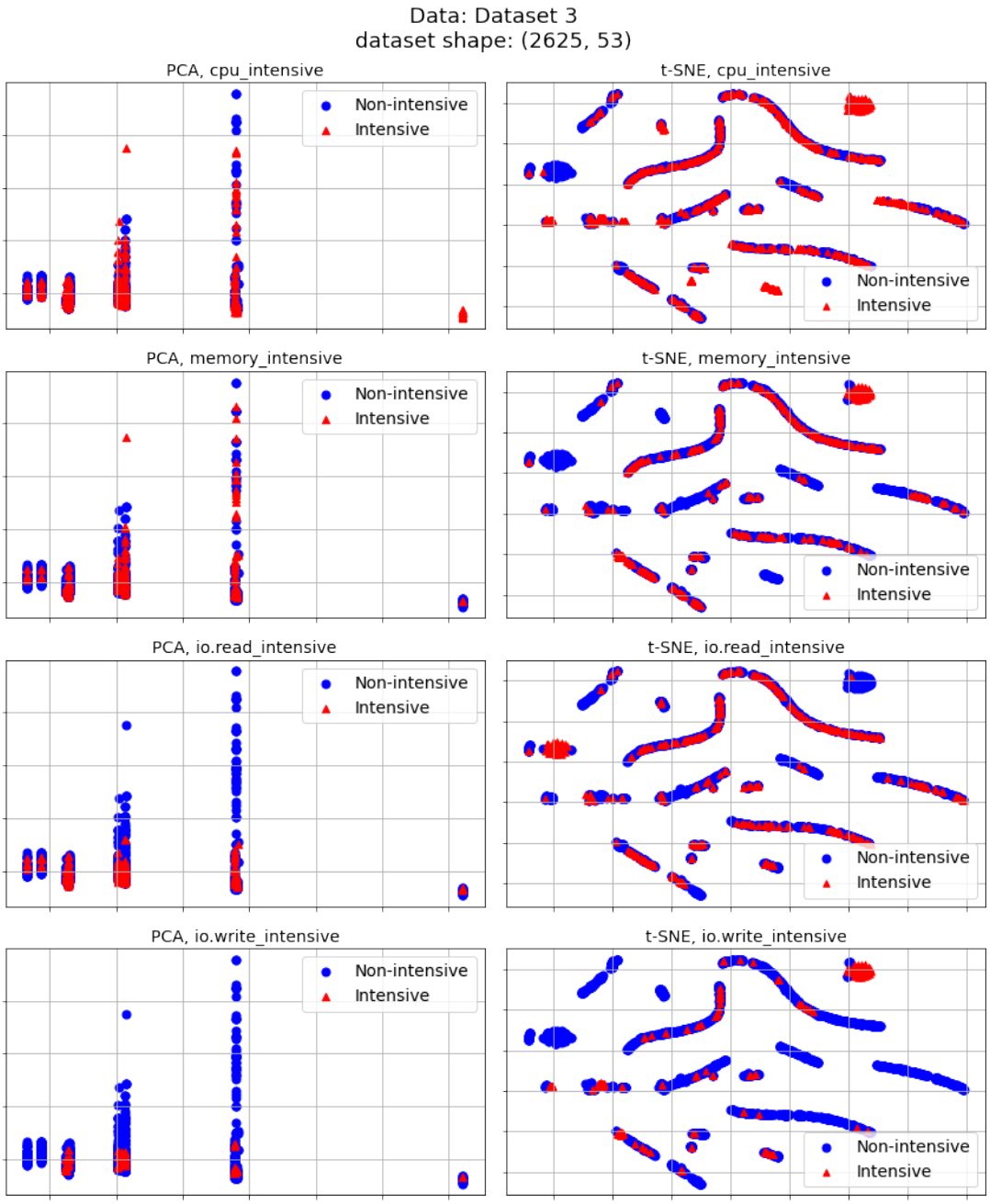

Figure 5.3: PCA and t-SNE projection to 2D for dataset 3. t-SNE created with parameter *perplexity* = 40

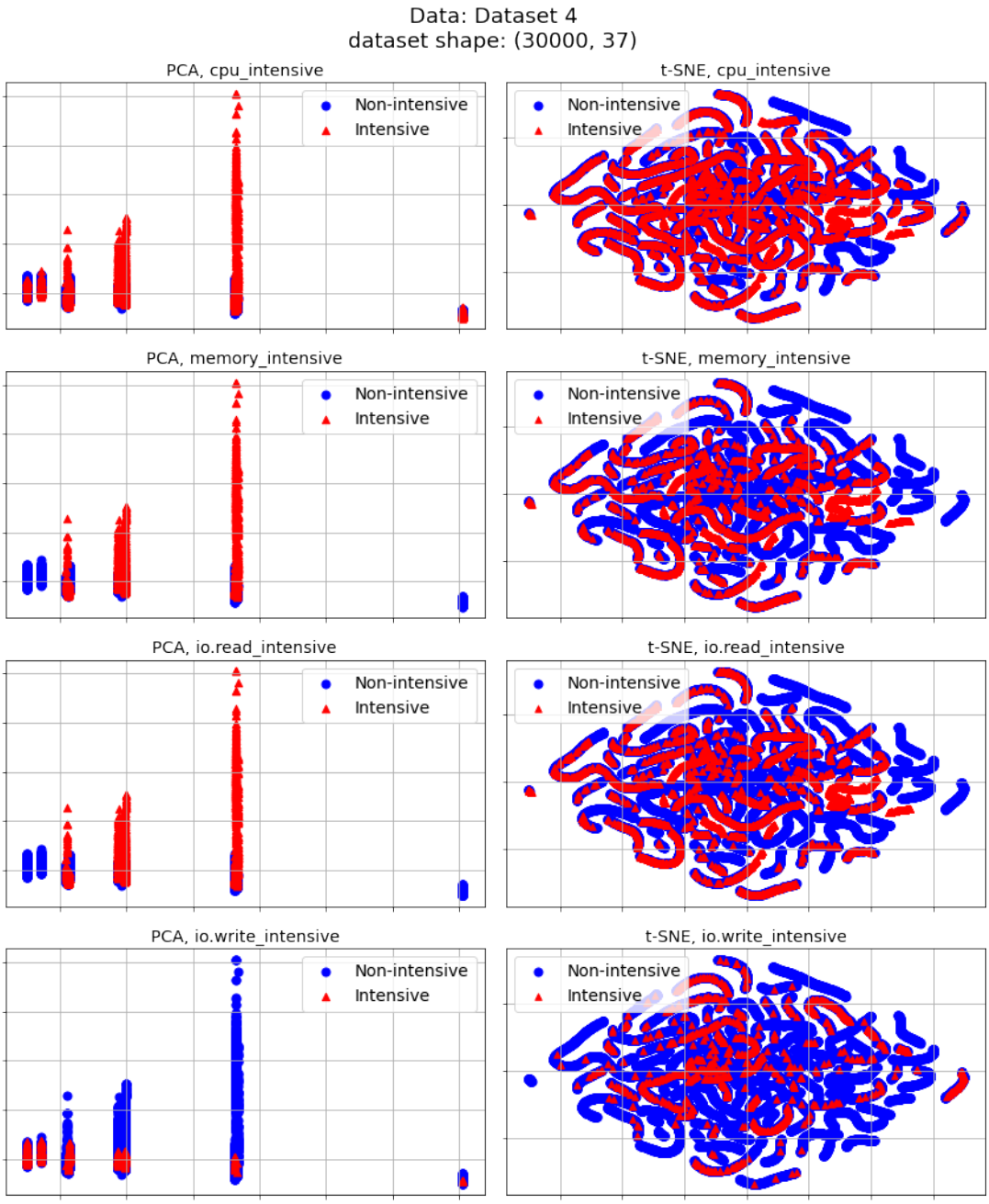

Figure 5.4: PCA and t-SNE projection to 2D for dataset 4. t-SNE created with parameter *perplexity* = 40

are noticeably bigger *non-intensive* regions in the middle-left and right part. We can also recognise a small cluster of only *intensive* values in the middle of the right part of the plot. In the remaining clusters, cases overlap. As for *io.read*, it resembles *memory* graphs to the great extent, considering both dimension reduction techniques. The reason for this might be the strict correlation between system reading resources and its memory usage. Finally, the *io.write* parameter is, in the PCA case, almost the opposite of the *io.read* graph, with many *non-intensive* values concentrated in the center. As for t-SNE projection, again, there is single main cluster of *non-intensive* values, with only few single values and small clusters of *intensive* points.

To sum up, we used the two-dimension reduction techniques for the purpose of determining whether or not features of prepared datasets exhibit some distinctive relations between labeled *intensive* and *non-intensive* values. While results have not been clearly separate clusters, in almost all cases there were some small distinctive clusters of single values, indicating the possibility of a feasible binary classification.

5.2 Intensity classification

After the promising results from dimension reduction methods, we performed a binary classification on four resources types: *cpu*, *memory*, *io.read*, *io.write*. The classification was performed using the logistic regression method. For evaluation purposes, we used two metrics: Score and F1.

Score is the mean accuracy of the classification. This value is normalized where 1.0 denotes the best result. The **F1 score** (also F-measure) is another measure of test's accuracy, calculated from the *precision* and the *recall* of the test. Precision, also referred to as positive predictive value, determines what fraction of values are actually correct, and is given by the formula:

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

whereas recall, also known as *sensitivity*, determines what fraction of positive values was correctly classified. Recall is given by the following formula:

$$recall = \frac{TP}{TP+FN}$$

where: TP – True positives, FP – False positives, FN – False negatives. Based on those measures of relevance, the F1 score is calculated as such:

$$F1 = \frac{2 * (precision * recall)}{precision + recall}$$

Tab. 5.1 shows the results of binary classification, compared in the 4 datasets. It is worth pointing out that the F1 score was calculated with regard to the *intensive* label (meaning that the positive class was *intensive*). For nearly all cases, the classification yields very good results, nearly perfect in some cases. Both the score and the F1 score metrics almost never fall lower than 0.82, which means that in most cases the models output correct results, and are characterized by high precision and recall values. The only exception is be the parameter *io.write* of the second and fourth dataset, which returned F1 score of value 0 and 0.242, respectively. A low F1 score usually indicates poor values of both precision and recall. This may be explained by a highly imbalanced data, which is the case in the both those datasets. Fig. 5.5 shows this in more detail; it presents the distribution of intensity of the labelled data of each resource. For *io.write*, there are considerably more *non-intensive* jobs than *intensive* ones for both sets of data. As a result, the classifier did not learn how to spot positive cases (*intensive*) effectively. In the case of dataset 2, both precision and recall are equal to 0. For the dataset 4, $precision = 0.74$ and $recall = 0.16$, meaning that only a small fraction of positive values was correctly classified. The score metric is still very high, since for most cases it correctly classifies jobs as *non-intensive*, because of the mentioned disproportion in the data.

Table 5.1: Results of binary classification of four resources: cpu, memory, io.read, io.write

Dataset	Parameters	Score	F1
1	cpu	0.926	0.923
	memory	0.971	0.952
	io.read	0.984	0.966
	io.write	1.0	1.0
2	cpu	0.981	0.969
	memory	0.991	0.933
	io.read	0.976	0.839
	io.write	0.994	0
3	cpu	0.938	0.939
	memory	0.997	0.993
	io.read	1.0	1.0
	io.write	1.0	1.0
4	cpu	0.991	0.989
	memory	1.0	1.0
	io.read	1.0	1.0
	io.write	0.955	0.242

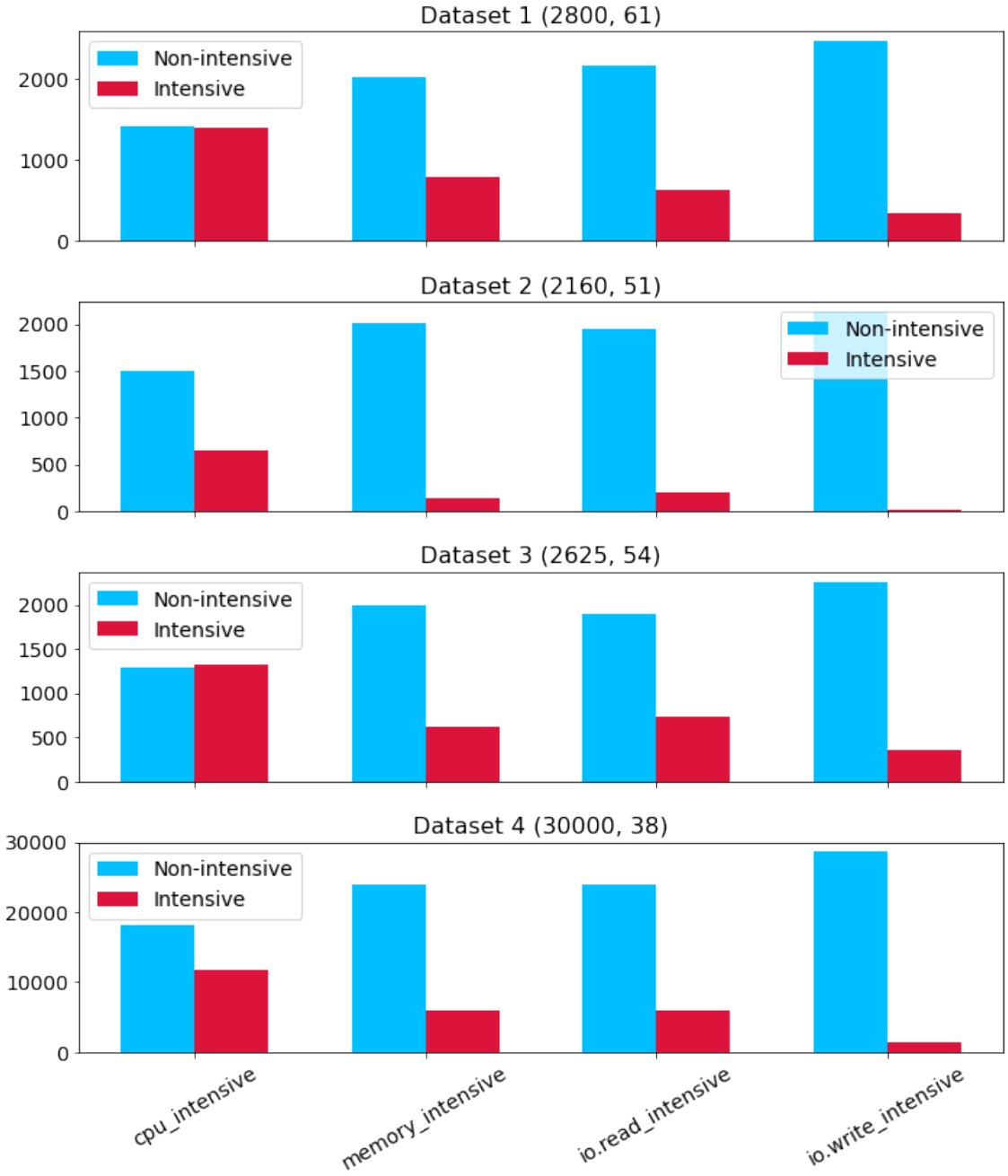

Figure 5.5: The distribution of intensity in labelled data, per dataset

5.3 Execution time prediction

5.3.1 Evaluation metrics

With the system configuration and job description data, complemented by labelled intensity derived from classifiers, we were able to apply regression analysis using several different methods, in order to estimate a task's execution time. During assessment of the said methods, we opted for three metrics, namely: *the coefficient of determination (R^2)*, *mean absolute percentage error*, *mean absolute scaled error* and *relative absolute error*.

The **coefficient of determination** (R^2), used in statistics, constitutes to the proportion of the variance in the target (dependent) variable that is predictable from the independent variables. In regression the R^2 is a statistical indicator of how closely the regression predictions match the actual data points. R^2 value might be interpreted as the amount of variance explained by the model. The coefficient of determination normally ranges from 0 to 1, but it can take negative values, when the chosen model does not follow the trend of the data. An R^2 value of 1 indicates that the regression predictions are perfectly fitted to the data. The coefficient R^2 is defined as:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where SS_{res} is the residual sum of squares $\sum_i (y_i - f_i)^2$ and SS_{tot} is the total sum of squares $\sum_i (y_i - \bar{y})^2$, y_i is an actual value, f_i is the predicted value, and $\bar{y}$ is the mean of the observed data.

The **mean absolute percentage error** (MAPE, also mean absolute percentage deviation) is a popular measure of prediction accuracy of a forecasting system, used in statistics. It also commonly finds usages in machine learning as a loss function for regression problems. It is given by the formula:

$$MAPE = \frac{1}{n} \sum_{t=1}^{n} \left| \frac{A_t - F_t}{A_t} \right| * 100\%$$

where A_t is the actual value, F_t is the forecast value, n is the number of predictions. When multiplied by 100%, it is expressed as a percentage error. MAPE, however, can be problematic as it is known to cause several issues in practical application:

- causes division by zero if there are zero values
- it puts heavier penalty on negative errors, as such, particularly small values bias the MAPE
- for too low forecasts the error cannot exceed 100%, but there is no upper limit to the percentage error for predictions which are too high.

In order to mitigate these drawbacks, literature proposes some other measures, one of which is the **mean absolute scaled error**, which we also employed as a metric of our prediction models. The mean absolute scaled error (MASE) is yet another measure of accuracy for forecasts. It has been deemed as the metric with favorable properties, compared to other methods, among other things, due to easy interpretation, symmetry, predictable behaviour and scale invariance. MASE is estimated by:

$$MASE = \frac{\frac{1}{J}\sum_{j=1}^{J}\left|e_j\right|}{\frac{1}{T-1}\sum_{t=2}^{T}|A_t - A_{t-1}|}$$

where in the numerator e_j is the estimation error, defined as $e_j = A_j - F_j$, A_j being the actual value, and F_j being the estimate, and J being the number of estimates, and the denominator as the mean absolute error of the one step naive forecast method, on the training set, in the T. A_t is the actual value, A_{t-1} is the value from the prior period. MASE takes values from 0 to 1, and the lower the value, the better the estimate.

Another metric used for evaluation purposes is the **relative absolute error** (RAE) [16], given by:

$$RAE = \frac{\sum_{i=1}^{n}|r_i - e_i|}{\sum_{i=1}^{n}\left|r_i - \frac{1}{n}\sum_{i=1}^{n} r_i\right|}$$

where n is the number of predictions, r_i is the actual value and e_i is the estimate. Smaller values of RAE indicate better prediction accuracy.

Moreover, empirical study of different metrics [6] suggests using RAE over other types of metrics. Additionally, RAE has been used by previous works [31] for execution time prediction of computational tasks.

Table 5.2: Results of task's execution time prediction, comparison of 4 different methods used, per dataset

Dataset	Method	RAE	R^2 Score	MAPE	MASE
1	DecisionTree	0.05	0.97	32.95	0.06
	RandomForest	0.04	0.99	37.14	0.04
	kNN	0.14	0.79	56.2	0.15
	Neural Network	0.45	0.1	27.48	0.48
2	DecisionTree	0.06	0.98	14.51	0.05
	RandomForest	0.07	0.98	16.51	0.06
	kNN	0.07	0.97	16.74	0.06
	Neural Network	0.25	0.89	20.72	0.21
3	DecisionTree	0.2	0.81	12.74	0.16
	RandomForest	0.19	0.86	13.09	0.15
	kNN	0.13	0.92	8.27	0.1
	Neural Network	0.47	0.62	22.79	0.36
4	DecisionTree	0.13	0.93	42	0.11
	RandomForest	0.13	0.94	40.89	0.11
	kNN	0.11	0.95	35.91	0.09
	Neural Network	0.22	0.73	23.62	0.18

5.3.2 Results

Tab. 5.2 shows results of task's execution time prediction, distinguished over selected datasets and different methods. We decided to evaluate the models using 4 distinct metrics, which were described in the previous section.

Dataset 1 represents all types of jobs and all types of machines, giving the most generalized view of the collected data. Here, the best method, by the RAE metric, was Random Forest (0.04), which is also reflected in the best score of R^2 (0.99) and MASE (0.04). Almost ideal R^2 value suggests that nearly all the variance is explained by the model. Only slightly worse results were given by the Decision Tree method, which is not surprising, since Random Forest is an ensamble method, improving upon decision trees. k-NN method performed considerably worse, however still giving the accuracy on an acceptable level. On the other hand, considering the MAPE metric, the Neural Network returned best

results (27.48), 5% less than Decision Tree method, and 10% less than Random Forest, the overall best method when considering other metrics. Outside of MAPE, Neural Network performed substantially worse than other methods.

The reason for the overall best value of MAPE for Neural Network is probably because MAPE was used as a loss function during the training process. Thus, the network arranged itself in such a way as to minimize this value, with no regard for other metrics.

Dataset 2 contains the shortest jobs, with the threshold value of 3s. For this set, the Decision Tree method gave best results (regarding RAE, R^2 and MASE), quite similar to those of the dataset 1. The second best method, Random Forest performed considerably poorer. k-NN however, gave significantly better results than for dataset 1, and only slightly worse then the previous methods. As for the ANN, again, the model performed substantially worse, as with dataset 1, compared to other methods. With regard to MAPE metric, results for all methods (except k-NN) are significantly better. The biggest difference can be observed at k-NN side, with a 40% lower error than in the previous dataset.

Dataset 3 represents jobs with average execution time, between 2000ms and 25000ms. The best methods here were k-NN, Random Forest and Decision Tree respectively. The values of these errors in some parts nearly doubled or tripled for RAE and MASE, and were only slightly poorer for R^2 score, while making a comparison to datasets 1 and 2. With respect to ANN, there is a substantial improvement of R^2 score over dataset 1, but a degradation when comparing with the dataset 2.

As to the dataset 4, constituting most frequently occurring jobs, the method yielding the best results was k-NN with values of RAE (0.11), R^2 Score (0.95) and MASE (0.09) meeting in between the best values from datasets 1 and 3. Decision Tree and Random Forest also performed well with almost exact same scores. ANN again performed the most poorly (except for MAPE, which returned the best score, lower by at least 12% compared to other techniques), however, compared to the previous sets of data, this time ANN returned considerably better values for RAE, R^2 and MASE error metrics, similar to those of dataset 2.

It should be pointed out that, as mentioned previously, that MAPE metric is biased, and, when dealing with small values (typically less than 1) it is prone to larger percentage errors [25]. As it happens in dataset 1 and 4 (refer to Fig. 4.5) there are many very short jobs, higher values of the MAPE metric in these cases are to be expected.

To summarize, the most versatile technique, with regard to all 4 distinct datasets,

turned out to be Random Forest, yielding best and very accurate results in nearly all cases. The Decision Tree method performed slightly worse, but also with fairly reasonably fitted results. The third rank goes to the *k*-Nearest Neighbours method which achieved best results for datasets 3 and 4. This might indicate that with relatively bigger sets of data and with more evenly distributed dependent variable, it could be beneficial to incorporate *k*-NN method over Random Forest or Decision Tree methods. Finally, the Neural Network approach gave the worst results. While minimizing the MAPE metric in the learning process, in most cases ANN failed to output reliable and accurate predictions, which is reflected in the other metrics. Only the largest dataset allowed ANN to improve its results, which may imply that the method requires significantly larger data source in order to train an accurate model.

Chapter 6

Discussion

This chapter focuses on interpretations of the results and their implications. We also discuss the limitations of the research, as well as provide recommendations for future studies.

Interpretations

Overall we were able to achieve accurate estimates based on the proposed two-step prediction process. Firstly, the classifier leveraging logistic regression method achieved an average accuracy score of 98%, whether the job instance was characterized by an intensive or non-intensive resource utilization, for CPU, memory, IO read and IO write operations. Secondly, building upon these predictions, we utilized regression analysis in an effort to estimate the length of job's execution time. We compared 4 different machine learning techniques, namely Decision Tree, Random forest, k-Nearest Neighbours an Artificial, Deep Neural Network.

Both classification and execution time prediction were performed on four distinctive data set types, each representing kinds of jobs with similar characteristics. Such division allowed us to evaluate how different regression methods performed on each dataset. Whereas the classification achieved relatively similar results over all four sets, for regression analysis, the error metrics varied significantly, depending on the method used and the kind of the dataset. The results indicate that for the majority of job types, the Random Forest method is the most versatile one, yielding highly accurate predictions. This especially holds true for datasets with small to average sizes. For larger datasets, the k-Nearest

Neighbours technique should be preferred.

Although the comparison of our results to other works can not be done explicitly, because of different datasets and workflow applications used in the experiments, we can compare values from the Montage workflow presented by Pham *et. al.* [31] from their two-stage approach to our model from dataset 1, containing all types of jobs. In this case, our approach decreased the average error by over 55% (RAE of 0.0897 vs 0.04). Both results were achieved using the Random Forest method. Additionally, our data covered a higher count of machine instance types and included more information regarding machine and job parameters.

Implications

The results collected build on existing evidence of utilizing history data of machine, workflow and task's parameters and machine learning techniques into accurate prediction of resource utilization level and runtime prediction. Accurate estimates allow for improving of executing multitude of tasks in a cloud. Trained models from this work could be incorporated into prediction system, used for optimization of resource allocators in the cloud environment, also those particularly leveraged for workflow execution purposes. Improving upon those allocators entails maximization of performance and minimization of costs for the cloud provider.

Limitations

Although promising, the achieved results were obtained using a relatively small number of different job types from Montage, Montage2 and SoyKB workflows. Consequently, the results cannot be generalized to other workflows. Moreover, our work excluded network parameters from analysis of resources utilization, as the monitoring data on network usage was not reliable. Finally, we considered only execution prediction parameters of individual workflow jobs, while parameters of the entire workflows (e.g. size) could also be considered.

Future research

Our study does not exhaust the presented topic. Several recommendations directions for further studies can be investigated. For one, more experimental data could be collected

from executions of more workflow types on a variety of different machine instances, acquired from different cloud providers, leading to increased generality of the study. Data collection itself could be enhanced, e.g. by the missing network usage metrics. A larger dataset could also benefit from introducing noise removal or implementing some outlier detection methods in data preparation process. Additionally, several other machine learning methods could be evaluated. Finally, regression analysis could be used for more detailed resource usage prediction.

Chapter 7

Conclusion

The main goal of this Thesis was to develop and experimentally validate machine learning approach to resource utilization intensity and execution time prediction of a computational, workflow job. To achieve this, we automated and enhanced HyperFlow logs collection process, which allowed us to collect valuable data, used for the creation of prediction models purposes. We then analyzed multitude of job types, collected from 3 distinct scientific workflows. Using the gathered pieces of information regarding utilization of the basic resources, computing machine instance parameters, workflow's input parameters and individual task's parameters, including the size of the input and output argument list, we produced a prediction process yielding promising results.

The prediction process consisted of two stages: classification of job runs by their resource utilization intensity, then using the results of this classification for complementing the training data of regression analysis using various machine learning methods. The results show a highly accurate execution time prediction, outperforming previous studies. From tested methods, we distinguished Random Forest as the most prominent one.

The proposed model and method for prediction of runtime characteristics of scientific workflow jobs could find several uses, including the enhancement of resource allocation and scheduling algorithms used in workflow management systems. Such optimizations could lead to cost reduction and performance improvement.

While our work is limited to only tasks from several workflow types, future studies should consider other types of jobs, workflows and possibly aim to predict usage of different resources. The diversity of machine types and cloud providers could also be extended.

Despite the limitations, the proposed method was successfully validated, therefore fulfilling the main objective of this Thesis.

Bibliography

[1] 11 Advantages of Cloud Computing and How Your Business Can Benefit From Them. URL https://www.skyhighnetworks.com/cloud-security-blog/11-advantages-of-cloud-computing-and-how-your-business-can-benefit-from-them/.

[2] 451 Research: 69% of enterprises will have multi-cloud/hybrid IT environments by 2019, but greater choice brings excessive complexity. URL https://451research.com/images/Marketing/press_releases/Pre_Re-Invent_2018_press_release_final_11_22.pdf.

[3] Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper. URL https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html.

[4] Exponential growth of supercomputing power as recorded by the TOP500 list. URL https://www.top500.org/statistics/perfdevel/.

[5] Abdi H., Williams L.J.: Principal component analysis. In: *Wiley interdisciplinary reviews: computational statistics*, vol. 2(4), pp. 433–459, 2010.

[6] Armstrong J.S., Collopy F.: Error measures for generalizing about forecasting methods: Empirical comparisons. In: *International journal of forecasting*, vol. 8(1), pp. 69–80, 1992.

[7] Atkinson M., Gesing S., Montagnat J., Taylor I.: Scientific workflows: Past, present and future, 2017.

[8] Balis B.: HyperFlow. In: *Future Gener. Comput. Syst.*, vol. 55(C), p. 147–162, 2016. ISSN 0167-739X. URL http://dx.doi.org/10.1016/j.future.2015.08.015.

[9] Berriman G., Good J., Laity A., Bergou A., Jacob J., Katz D., Deelman E., Kesselman C., Singh G., Su M.H., et al.: Montage: A grid enabled image mosaic service for the national virtual observatory. In: *Astronomical Data Analysis Software and Systems (ADASS) XIII*, vol. 314, p. 593. 2004.

[10] Bittencourt L.F., Goldman A., Madeira E.R., da Fonseca N.L., Sakellariou R.: Scheduling in distributed systems: A cloud computing perspective. In: *Computer science review*, vol. 30, pp. 31–54, 2018.

[11] Caron E., Desprez F., Muresan A.: Forecasting for grid and cloud computing on-demand resources based on pattern matching. In: *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pp. 456–463. IEEE, 2010.

[12] Chatterjee S., Hadi A.S.: *Regression analysis by example*. John Wiley & Sons, 2015.

[13] Da Silva R.F., Juve G., Rynge M., Deelman E., Livny M.: Online task resource consumption prediction for scientific workflows. In: *Parallel Processing Letters*, vol. 25(03), p. 1541003, 2015.

[14] Dell'Amico M., Carra D., Pastorelli M., Michiardi P.: Revisiting size-based scheduling with estimated job sizes. In: *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, pp. 411–420. IEEE, 2014.

[15] Duan R., Nadeem F., Wang J., Zhang Y., Prodan R., Fahringer T.: A hybrid intelligent method for performance modeling and prediction of workflow activities in grids. In: *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 339–347. IEEE Computer Society, 2009.

[16] Frank E., Hall M.A.: *Data mining: practical machine learning tools and techniques*. Morgan Kaufmann, 2011.

[17] Gawali M.B., Shinde S.K.: Task scheduling and resource allocation in cloud computing using a heuristic approach. In: *Journal of Cloud Computing*, vol. 7(1), p. 4, 2018.

[18] Hosmer Jr D.W., Lemeshow S., Sturdivant R.X.: *Applied logistic regression*, vol. 398. John Wiley & Sons, 2013.

[19] Iosup A., Ostermann S., Yigitbasi M.N., Prodan R., Fahringer T., Epema D.: Performance analysis of cloud computing services for many-tasks scientific computing. In: *IEEE Transactions on Parallel and Distributed systems*, vol. 22(6), pp. 931–945, 2011.

[20] Juve G., Deelman E.: Scientific workflows in the cloud. In: *Grids, clouds and virtualization*, pp. 71–91. Springer, 2011.

[21] Kecskemeti G., Nemeth Z., Kertesz A., Ranjan R.: Cloud workload prediction based on workflow execution time discrepancies. In: *Cluster Computing*, vol. 22(3), pp. 737–755, 2019.

[22] Liu X., Ni Z., Yuan D., Jiang Y., Wu Z., Chen J., Yang Y.: A novel statistical time-series pattern based interval forecasting strategy for activity durations in workflow systems. In: *Journal of Systems and Software*, vol. 84(3), pp. 354–376, 2011.

[23] Liu Y., Khan S.M., Wang J., Rynge M., Zhang Y., Zeng S., Chen S., dos Santos J.V.M., Valliyodan B., Calyam P.P., et al.: PGen: large-scale genomic variations analysis workflow and browser in SoyKB. In: *BMC bioinformatics*, vol. 17, pp. 177–186. BioMed Central, 2016.

[24] Maaten L.v.d., Hinton G.: Visualizing data using t-SNE. In: *Journal of machine learning research*, vol. 9(Nov), pp. 2579–2605, 2008.

[25] Makridakis S.: Accuracy measures: theoretical and practical concerns. In: *International journal of forecasting*, vol. 9(4), pp. 527–529, 1993.

[26] Mao L., Qi D., Lin W., Zhu C.: A self-adaptive prediction algorithm for cloud workloads. In: *International Journal of Grid and High Performance Computing (IJGHPC)*, vol. 7(2), pp. 65–76, 2015.

[27] Marston S., Li Z., Bandyopadhyay S., Zhang J., Ghalsasi A.: Cloud computing—The business perspective. In: *Decision support systems*, vol. 51(1), pp. 176–189, 2011.

[28] Miu T., Missier P.: Predicting the execution time of workflow activities based on their input features. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 64–72. IEEE, 2012.

[29] Nadeem F., Fahringer T.: Predicting the execution time of grid workflow applications through local learning. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, p. 33. ACM, 2009.

[30] Orzechowski M., Baliś B., Słota R.G., Kitowski J.: Reproducibility of computational experiments on Kubernetes-managed container clouds with HyperFlow. In: *International Conference on Computational Science*, pp. 220–233. Springer, Cham, 2020.

[31] Pham T.P., Durillo J.J., Fahringer T.: Predicting workflow task execution time in the cloud using a two-stage machine learning approach. In: *IEEE Transactions on Cloud Computing*, 2017.

[32] Poehlman W.L., Rynge M., Balamurugan D., Mills N., Feltus F.A.: OSG-KINC: High-throughput gene co-expression network construction using the open science grid. In: *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pp. 1827–1831. IEEE, 2017.

[33] Qin J., Fahringer T.: *Scientific Workflows: Programming, Optimization, and Synthesis with ASKALON and AWDL.* Springer Science & Business Media, 2012.

[34] Rehr J.J., Vila F.D., Gardner J.P., Svec L., Prange M.: Scientific computing in the cloud. In: *Computing in science & Engineering*, vol. 12(3), pp. 34–43, 2010.

[35] Rugwiro U., Gu C., Ding W.: Task Scheduling and Resource Allocation Based on Ant-Colony Optimization and Deep Reinforcement Learning. In: *Journal of Internet Technology*, vol. 20(5), pp. 1463–1475, 2019.

[36] Rupp K.: 40 Years of Microprocessor Trend Data. URL `https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/`.

[37] Taylor I.J., Deelman E., Gannon D.B., Shields M., et al.: *Workflows for e-Science: scientific workflows for grids*, vol. 1. Springer, 2007.

[38] Trott O., Olson A.J.: AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. In: *Journal of computational chemistry*, vol. 31(2), pp. 455–461, 2010.

[39] Verboven S., Hellinckx P., Arickx F., Broeckhove J.: Runtime Prediction Based Grid Scheduling of Parameter Sweep Jobs. In: *2008 IEEE Asia-Pacific Services Computing Conference*, pp. 33–38. 2008.

[40] Wierman A., Nuyens M.: Scheduling despite Inexact Job-Size Information. In: , vol. 36(1), p. 25–36, 2008. ISSN 0163-5999. URL `http://dx.doi.org/10.1145/1384529.1375461`.

[41] Yan C., Luo H., Hu Z., Li X., Zhang Y.: Deadline guarantee enhanced scheduling of scientific workflow applications in grid. In: *Journal of Computers*, vol. 8(4), pp. 842–850, 2013.