MASTER EN INGENIERÍA Y CIENCIA DE DATOS

Curso Académico 2021/2022

Trabajo Fin de Máster

RESOURCES USAGE PREDICTION ON PARALLEL DISTRIBUTED INFRASTRUCTURES

Autor : Antonio Martínez Garín

Tutor : Dr. Agustín Carlos Caminero Herraez

Tutor : Dr. José Manuel Cuadra Troncoso

To my wife Leyre and my son,

This would have never been completed without your love and awesome support.

To my parents,

For all the efforts made, know they were worth it.

Agradecimientos

Cuando empecé con este proyecto, no podía imaginar el esfuerzo que llegaría a costar. Se necesita la guía de personas con experiencia que indiquen los pasos iniciales y puedan corregir el rumbo cuando se desvía. Por ello, me gustaría agradecer al Dr. José Manuel Cuadra Troncoso y al Dr. Agustín Carlos Caminero Herraez por su paciencia e interés a lo largo del desarrollo de este proyecto. Sin desestimar los esfuerzos propios, he de reconocer su labor, que ha sido fundamental para la consecución de este TFM. Gracias por acompañarme y guiarme a lo largo de este viaje.

Resumen

Los contenedores ligeros se están usando de forma extensiva para ejecutar aplicaciones basadas en contenedores llamadas trabajos. Estos trabajos son orquestados por sistemas encargados de administrar clústeres de gran tamaño que contienen cientos de miles de aplicaciones, y destacan por lograr una alta utilización del clúster. Sin embargo, definiciones deficientes de los requisitos de recursos en los contenedores de los trabajos tienen un impacto negativo en la eficiencia general del uso del clúster.

El objetivo principal de este trabajo es encontrar y entrenar modelos para predecir el uso de recursos de los jobs cuando son enviados y analizar su capacidad de predicción. Esto se hará utilizando datos de grandes clústeres de producción que ejecutan contenedores ligeros. Este TFM se encuadra dentro del proyecto del FILE (efFIcient scheduLing of containErs), financiado por la UNED.

Abstract

Lightweight containers are extensively used for running containerized applications as jobs. Jobs are orchestrated by systems which manage large clusters that contain hundreds of thousands of jobs from thousands of applications, and they excel at achieving high utilization of the cluster. However, poor container resource requirements negatively impact the overall efficiency of cluster usage.

The main objective of this work is to find and train models to predict the resource usage of jobs at submission time and analyze their prediction power. This will be done using data from large production clusters running lightweight containers. This thesis is part of the FILE project (efFIcient scheduling of containErs) financed by UNED.

Keywords: Machine Learning, Regression, Lightweight Containers, Tabular Data, Resources Usage, Random Forest.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction and Motivation

Lightweight containers are being used extensively in the last years in the industry, which uses them in production environments with large clusters. Three main features support this widespread, the isolation that they provide, the reproducibility, and the ability to optimize better the resources that an application uses to reduce the memory footprint [Jaramillo et al., 2016]. This has led to the development of open-source container orchestrators like Kubernetes, which is heavily inspired by Google's Borg [Verma et al., 2015, Burns et al., 2016, Blog, 2015].

With the previous ideas in mind, studies have been developed pursuing allocation analysis and optimization in cluster resource usage [Tirmazi et al., 2020a], [Guo et al., 2019] and [Lu et al., 2017]. It would be expected that jobs submitted had been previously assessed for the required resources but, in practice cluster resources are underused.

To solve this issue, we propose to compare the performance of different models over traces of a cluster with the hypothesis that resource requirements can be optimized by predicting resource usage for specific jobs at the time of submission. In order to test this hypothesis, we propose to use resource requests, resource usage, and as many metadata variables are available on published cluster data open datasets.

1.2 Objectives

The principal objectives of this research are:

- Find existing open datasets with production traces of lightweight containers.
- Develop a practical case study of the dataset.

Secondary objectives are likely to arise while developing the thesis, like providing insights on optimal lightweights containers resource requests on shared clusters.

1.3 Problem Formulation

Given a cluster of machines, we define two main resources, *CPU units* (cpu, from now on) and *RAM Memory units* (memory, from now on), which define the capacity of each of the individual machines and can be grouped together to define the capacity of the cluster.

Jobs can be submitted to a cluster with specific cpu and memory requests and will consume some cpu and memory, either on a single machine or several, as jobs are composed of individual tasks. These Jobs will have tasks, and these tasks will consume cpu and memory. If they consume more cpu than requested, and there is free cpu available they can use it, but if more memory than the requested units is consumed the tasks are killed. Predicting the usage of a job is therefore pivotal to ensuring a job can be submitted.

1.4 Memory Structure

The thesis is organized as follows:

- In the first chapter, an introduction to the project is given.
- The second chapter (2) shows related work to the subject.
- Chapter 3 introduces the problem formulation in depth.
- In the fourth chapter (4), the data used for the project is described and analyzed. It includes a thorough description of the data cleaning and pre-processing.
- Chapters 5 and 6 introduce the regression models used in this work.
- Next, the experimentation results are displayed and discussed (Chapter 7).
- Finally, chapter 8 includes the conclusions and future work.

Chapter 2

Previous Work

The research on event data has been extense in fields like anomaly detection, clustering, pattern recognition [Han et al., 2012, Bishop, 2013], where the objective is not to predict but analyze and detect events. Instead, our focus is on predictive analysis, and literature can be found on social media analysis [Kursuncu et al., 2019], banking [Prasad and Madhavi, 2012, Kullaya Swamy and Sarojamma, 2020] or demand [Laptev et al., 2017] events.

Event data is usually found with a timestamp that tracks when the event was generated. Therefore, event data can be considered as a time series. A large portion of literature focuses on analyzing evenly-spaced time series, which has lead to the development of efficient algorithms under that assumption [Eckner, 2012]. This was largely because of limitations in computing resources, which is no longer an issue. In 1.1 we introduced the irregular nature of events in cloud computing and container orchestrators, which leads to unevenly-spaced time series of event data. Approaches [Smedt et al., 2021, Casals et al., 2009] have been described to convert irregular time series into regular ones by applying aggregations of equal time or size, and resampling to obtain even subseries but it comes with an observability loss and negative effect on forecasting accuracy.

Recently, interesting approaches have been published to tackle irregular time series by using Recurrent Neural Networks [Hewamalage et al., 2021] and Transformers. The Temporal Fusion Transformer [Lim et al., 2019] is one of the latest additions in the former category, introducing a new attention-based architecture for multi-horizon forecasting while providing interpretable insights, which is something these kinds of models usually lack. Although they are showing promising results, we are going to discard these models for this thesis.

Concerning regression analysis, [Yan and Su, 2009] describes Linear Regression as the first to be studied in depth, having been widely used in practice. Research in the last 25 years has provided many new models like Random Forest [Breiman, 2001], XGBoost [Chen et al., 2015], or Deep Neural Network architectures [Goodfellow et al., 2016].

Our work focuses on a general approach to lightweight containers, but research is often conducted on general cluster usage, or VM resource allocation [Liu and Cho, 2012] and [Xiao et al., 2012]. Moreover, public datasets of cluster traces from real orchestration production environments are rare, although cloud computing companies like Google [Hellerstein, 2010, Wilkes, 2011, Wilkes, 2020b] and Alibaba [Cheng et al., 2018, Zhang, 2017] have been releasing large cluster traces for scientific research over recent years. This has led to extensive analysis and research on resource efficiency in the clusters [Jajoo et al., 2021, Tirmazi et al., 2020b, Sebastio et al., 2018, Abdul-Rahman and Aida, 2014].

Chapter 3

Problem Formulation

Given a cluster of machines C, we denote each machine as c_i. We define two main resources, CPU units U and RAM Memory M. We can define the capacity of a cluster as a vector $CAP(C) = (\sum U(c_i), \sum M(c_i)) : i = 1, ..., C$.

Jobs J can be submitted to a cluster with specific cpu and memory requests and will consume some cpu and memory, so we denote:

- The requests of a specific Job as $REQ(j_k) = (U(j_k), M(j_k)) : k = 1, ..., J$.
- The resources usage as $RES(j_k) = (U(j_k), M(j_k)) : k = 1, ..., J$.

As we will further discuss in the Data Description 4.1 section, the data will be from the cluster usage Dataset by Google for the year 2019 and only finished jobs are used from Cell E. The units for CPU and Memory are calculated as the average and maximum, respectively, for the execution duration of each job, although it is possible to consider other approaches for CPU as the average of percentiles above 90%.

Chapter 4

Data Description and Analysis

In the previous chapter, it was established that we will not treat the data as time series, so we need to adapt the data to obtain unique entries for each job in the event data.

We will be using three datasets, the first one containing event data for the machines in the cluster, another for the job events, and a final one with resource usage for each job aggregated over time. The resources and requests dataset can be identified as our main dataset, and we expect to extract useful metadata from the collection and job events that will help increase the performance of the models that will be trained.

4.1 Data Description

Here we will present the data that has been used in our work. Previously, we mentioned the three data sets that will be our sources, and what follows is an introduction to the Borg system and the datasets themselves with information extracted from their documentation [Verma et al., 2015, Wilkes, 2020a]. The source is the Google cluster-usage traces v3 [Wilkes, 2020a] which provides trace data from eight Borg cells over the month of May 2019 [Tirmazi et al., 2020b].

A Borg cell is a set of machines, typically all in a single cluster (although it is not a requirement) sharing a common cluster-management system that allocates work to the machines. It defines two main concepts for resource requests that share the same lifecycle:

- **Collections:** They can be jobs, a task, or a set of tasks (instances) specifying computations that a user wants to run.

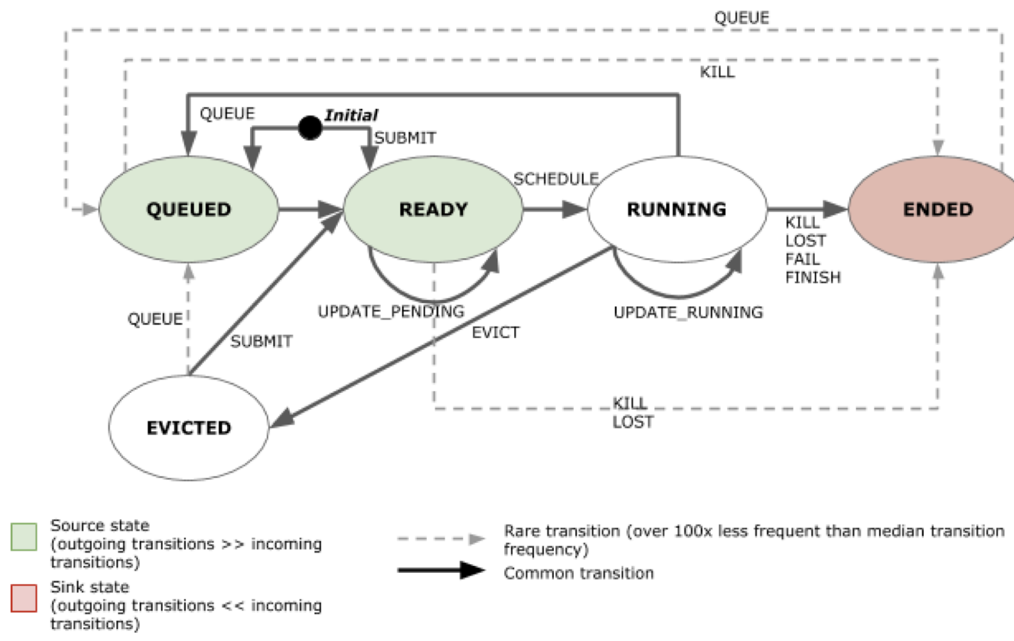

QUEUE
KILL
QUEUE
Initial
SUBMIT
SCHEDULE
QUEUED
READY
RUNNING
ENDED
KILL
LOST
FAIL
FINISH
UPDATE_PENDING
UPDATE_RUNNING
QUEUE
SUBMIT
EVICT
KILL
LOST
EVICTED
Source state (outgoing transitions >> incoming transitions)
Sink state (outgoing transitions << incoming transitions)
Rare transition (over 100x less frequent than median transition frequency)
Common transition

Figure 4.1: Events lifecycle, with the edges reflecting all the available transition types. The only state that marks a collection as successful is FINISH, encoded as 6.[1]

- **Alloc Sets:** Resource reservation made of one or more alloc instances. Jobs can be configured to specifically use these instances.

If no alloc set is specified, then the tasks of a Job will consume resources directly from a machine. The concept of instances is comprised by:

- **Task:** Linux program that can be run on a single machine and is part of a Job.
- **Alloc Instance:** As tasks are tied to Jobs, alloc instances are tied to an alloc set, and they represent a slot reservation inside it.

With the basic concepts around Borg defined, we can now describe in detail the 3 datasets that will be used in this work, all from cell E. All of the datasets include one or more timestamps in milliseconds:

- **CollectionEvents:** Events that describe the life cycle (Figure 4.1) of collections. Each entry represents an event of a collection and the variables of interest for us in this dataset are collection id, user, and collection logical name, which is a hash of the original name of the collection. There are other variables ignored from this dataset because either they

[1]Extracted from Google cluster-usage traces v3.

are replicated on other datasets or offer information about machines and switches. From this da Collections running entirely on dedicated machines are omitted. For each event, it includes a timestamp and type of event (transition).

- **InstanceEvents:** Events describing the life cycle (Figure 4.1) of instances. From this dataset we will take the variables priority (higher is better), scheduling class (categories derived from the priority), alloc collection id (indicates the collection id of the instance, if configured), and resource request information (cpu and memory, normalized). Other variables are ignored because they refer to machines. Instances from collections running entirely on dedicated machines are omitted.

- **InstanceUsage:** This dataset contains usage values from a series of measurement windows, non-overlapping, for each instance. These windows are typically 300 seconds long, but only if the instance is started, stopped, or updated within that period. During the measurement windows, the usage data (cpu units and memory) is sampled roughly every second and then aggregated (average, maximum) and normalized. We will use the start and end time, collection id, instance index, maximum usage memory, average usage cpus, and random sample usage cpus.

As a final note, the CPU units and Memory units are measured internally in Borg as Google Compute units (GCU) and RAM in bytes. Also, CPU requests are measured in GCUs and CPU consumption is measured in GCU-seconds/seconds. The GCU is defined on each machine so that one GCU delivers approximately the same amount of computation on all of them. For RAM, the capacity of a machine with the largest memory size is reported as 1.0, and all the memory sizes are rescaled by dividing them by the maximum machine memory observed across all the traces (Those values can be found on another dataset not used, MachineEvents).

4.1.1 Data Formatting

All the cluster usage data was provided as a JSON compressed with GZIP. There are available 8 folders, named after the 8 cells for which the trace data is provided, and each dataset is stored partitioned in files named after the collection and the partition number as explained in the statistics from Table 4.1. On each partition, each row represents one entry of data.

Dataset	Num. Partitions	Size (json.gz)	Size (.json)
CollectionEvents	1	$\approx$ 254 MB	$\approx$ 2.12 GB
InstanceEvents	51	$\approx$ 27.39 GB	$\approx$ 637.5 GB
InstanceUsage	1302	$\approx$ 699.32 GB	$\approx$ 5.09 TB

Table 4.1: Datasets partitions and size for Cell E.

4.2 Data Cleaning

In this section, we present all the cleaning work pursued for each dataset to create a base dataset for training the models. As we mentioned in previous chapters, we will only extract data from cell E.

In the previous section we talked about the datasets formatting; being provided as json.gzip files, the first stage was to preprocess them and have them converted to a more efficient storage format like parquet (Table 4.1). Otherwise, it would have been impossible to work with this dataset as just loading any partition on *json.gz* took more than 5 minutes on average. This, added to the size of the dataset made it impossible to work with traditional tools like Pandas, so we had to resort to distributed frameworks like Dask[Crist, 2016], which have Pandas-like APIs. Even so, the processing to convert the files took 3 days in total. It should also be noted, that since this data was in JSON, it did not follow a Parquet-compatible format for all the columns, so it was required to flatten the incompatible columns and convert them to accepted data types.

Once we had the datasets converted to parquet and with valid Pandas types, we could proceed with the cleaning:

- **CollectionEvents:** We filtered out events on this dataset, keeping only the ones that had *FINISHED* 4.1. The reason behind this is that we just want to use collections that have their resources requests, among other parameters, correctly defined and do not have any misconfiguration or runtime crashes. This, obviously, creates a bias, but we rather have this bias than use tasks that were never scheduled or crashed because they consumed too many resources or were canceled (because there is no way to guess why they were terminated). Also, we only extracted Jobs, because Alloc Sets are just reservations of resources, but Jobs can be configured to use Alloc Sets. Therefore, if any Job uses an

Alloc Set, we will still have it. The resulting collection ids will be used to filter out collection ids on the other two datasets.

- **InstanceEvents:** Once we have filtered out the unfinished collection ids and the unfinished instance events the resulting dataset was small enough to be loaded as a Pandas DataFrame, 2,5 GB (although loading time is around 4 minutes).

- **InstanceUsage:** Again, we reduced the size of the dataset thanks to the filtering of collection ids to be small enough to fit in memory, 4.1 GB (but the loading time is around 44 minutes)

Finally, we joined all the datasets together. We started by grouping the InstanceUsage dataset to extract for each collection id the minimum start time, the maximum end time, the number of instances, the maximum memory usage, and the average usage cpus. Then, we extracted the unique events from the InstanceEvents dataset to remove duplicates and extract the metadata of the instances. Finally, we extract the user and collection logical name from the CollectionEvents and we joined all the datasets by collection id.
As a result of these operations, we obtained a dataset with the collection ids that finished successfully, their metadata, the requested resources, and the resource usage. It is worth noting that this dataset contains time series identified by the categorical variables *user* and *collection_logical_name*.

4.3 Data Analysis

Outliers and distribution of the data can heavily impact the prediction accuracy of models that assume a Normal distribution of the data. Through this section, we will carry out an in-depth analysis of the data to identify characteristics that can be used later to benefit our models. This section also includes the analysis of the relations between the target and independent variables and the distribution of the events in time. Also, it is worth mentioning that there are no missing values in the final dataset, as shown in Tables 4.2 and 4.3.

	start_time	end_time	priority	scheduling_class	num_instances	alloc_collection_id	resource_request_cpus	resource_request_memory	maximum_usage_memory	average_usage_cpus	random_sample_usage_cpus
count	529311	529311	529311	529311	529311	529311	529311	529311	529311	529311	529311
Mean	1.306453e+12	1.307947e+12	128.876863	0.933321	8.563082e+01	0.000297	0.007862	0.002927	2.123000e-03	0.002110	0.002159
Std	7.840218e+11	7.837372e+11	66.482851	0.802010	4.909463e+03	0.017220	0.018208	0.005390	4.500872e-03	0.007818	0.007966
Min	3.000000e+08	6.120000e+08	0	0	1e+00	0	0	0	9.536743e-07	0	0
1st Qtl.	6.244855e+11	6.270680e+11	103	0	3e+00	0	0.000880	0.000408	3.328323e-04	0.000225	0.000139
Median	1.232059e+12	1.233313e+12	115	1	4e+00	0	0.001320	0.000986	8.392334e-04	0.000392	0.000355
3rd Qtl.	1.969502e+12	1.970470e+12	200	2	6e+00	0	0.008102	0.002762	2.037048e-03	0.000814	0.000943
Max	2.678989e+12	2.678993e+12	360	3	1.357094e+06	1	0.518555	0.390625	3.085938e-01	0.327791	0.328583

Table 4.2: Statistics of the numerical variables including the quantiles.

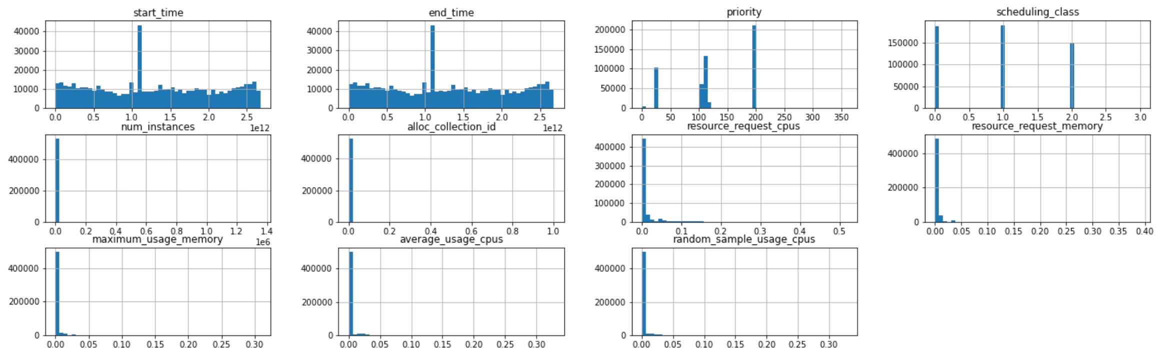
	user	collection_logical_name
count	529311	529311
unique	1738	18190
top	F2+Gv53Pxd4KDRb[...]	qYyuDw/A+bLNI4+[...]
freq	95898	54030

Table 4.3: Statistics of the categorical variables showing the number of categories for each one and the most frequent.

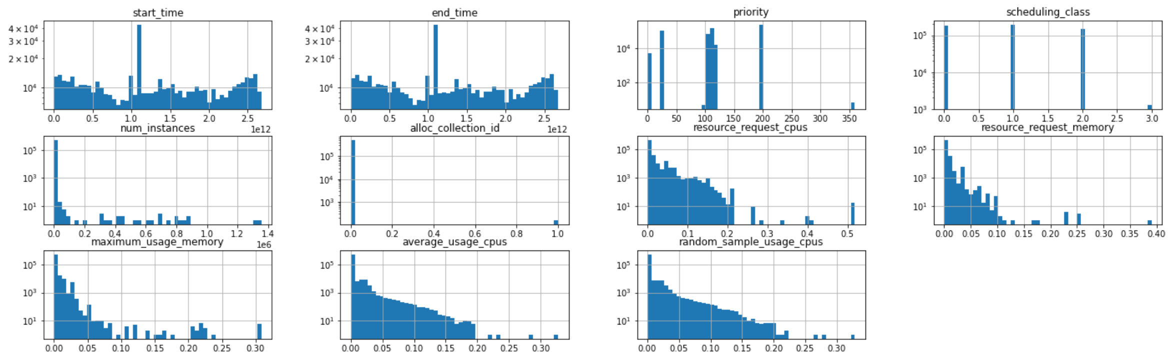
4.3.1 Data Distributions and outliers

The main descriptive statistics introduced in Tables 4.2 and 4.3 are complemented by the distributions represented in Figure 4.2. Looking first at the start_time, Table 4.3 shows the original number of jobs at a minute level, with an unusual number of jobs in the first minute, and the result of removing the outliers. These outliers are there due to how the trace was recorded, leaving the first minute with 893 jobs, being impossible to guess which of the events on that second happened before the start of the trace. The final minutes of the trace are not affected, but to keep the trace with complete days, and since the trace starts on minute 5, for training the models, we will adjust the dataset to start on minute 5 of day 0 (skipping the first second, as it is where the outliers of minute 5 lie) until day 31 minute 5 second 0.

Continuing with the histograms of the other continuous variables, they show right-skewed heavy tail distributions, which is confirmed by calculating the skewness 4.4. These outliers can affect model predictions so it will be needed to apply robust scalers for these variables before training them. This will be further discussed in Chapters 5 and 6.

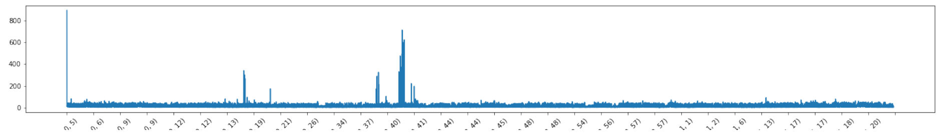


(a) Unscaled

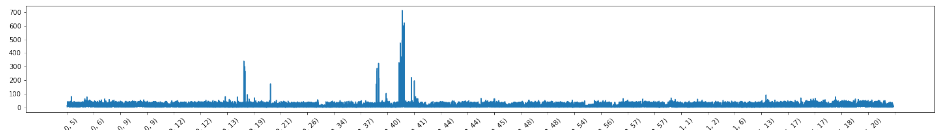


(b) Log Scale

Figure 4.2: Features distributions, (a) showing the unscaled histograms and (b) showing the log scale histograms.

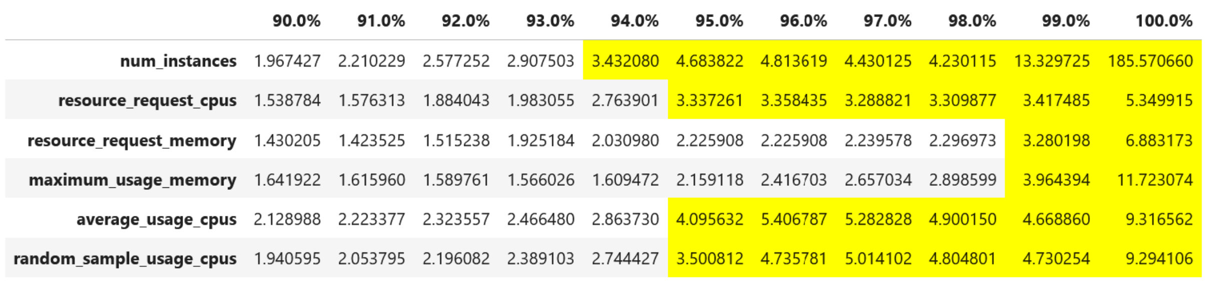


(a) Original



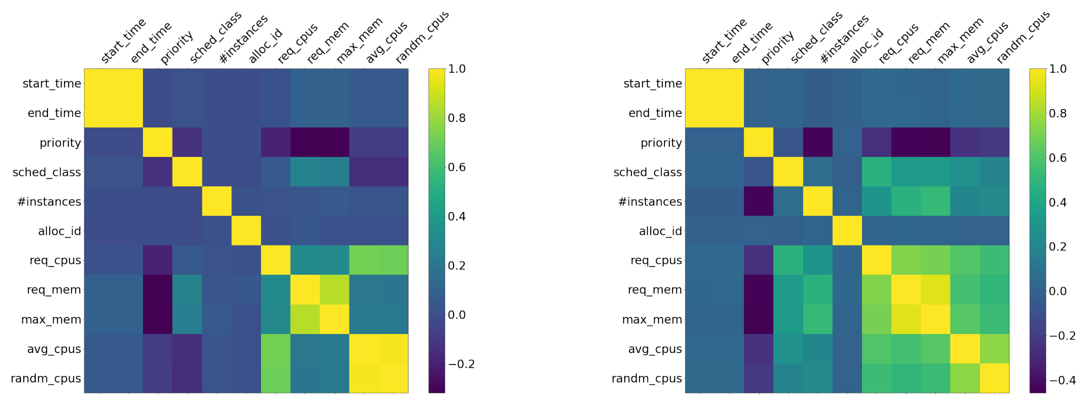
(b) Adjusted

Figure 4.3: Comparison showing the number of jobs grouped by day, hour, and minute in the trace with (a) showing the original dataset and (b) showing the result of removing outliers.



	90.0%	91.0%	92.0%	93.0%	94.0%	95.0%	96.0%	97.0%	98.0%	99.0%	100.0%
num_instances	1.967427	2.210229	2.577252	2.907503	3.432080	4.683822	4.813619	4.430125	4.230115	13.329725	185.570660
resource_request_cpus	1.538784	1.576313	1.884043	1.983055	2.763901	3.337261	3.358435	3.288821	3.309877	3.417485	5.349915
resource_request_memory	1.430205	1.423525	1.515238	1.925184	2.030980	2.225908	2.225908	2.239578	2.296973	3.280198	6.883173
maximum_usage_memory	1.641922	1.615960	1.589761	1.566026	1.609472	2.159118	2.416703	2.657034	2.898599	3.964394	11.723074
average_usage_cpus	2.128988	2.223377	2.323557	2.466480	2.863730	4.095632	5.406787	5.282828	4.900150	4.668860	9.316562
random_sample_usage_cpus	1.940595	2.053795	2.196082	2.389103	2.744427	3.500812	4.735781	5.014102	4.804801	4.730254	9.294106

Figure 4.4: Skewness coefficients calculated for the percentiles 90th to 100th of features displaying a heavy tail distribution. Highlighted are the $(percentile, variable)$ cells above 3.



(a) Pearson Correlation Matrix

(b) Spearman Correlation Matrix

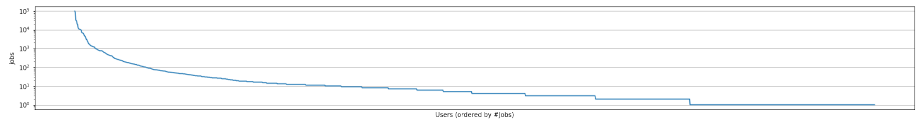
Figure 4.5: Correlation matrices side by side. Matrix (a) shows linear relationships, whereas (b) highlights monotonic relationships. Names are shortened for visualization purposes.

4.3.2 Variables Correlation

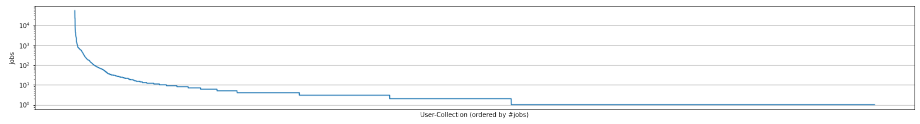
Figure 4.5 shows an evaluation of the correlation between variables, and a strong correlation between *start_time* and *end_time* can be observed, as well as between requests and usage of cpu and memory respectively. To avoid multicollinearity on models like Logistic Regression, we must remove one of the timestamp variables. Request and resource usage for cpu and memory also show a high correlation, and there is also some correlation between the memory and cpu variables. Looking closer at *num_instances* and the priority variables we can see some correlation with the resources and requests, especially with memory. The *alloc_id* variable does not seem to have much correlation with any of the other variables.

	Users			Collections			User-Collection		
	First	Last	Count	First	Last	Count	First	Last	Count
Min	3.060e+08	4.260e+08	1	3.060e+08	4.260e+08	1	3.060e+08	4.260e+08	1
1st Qtl.	5.732e+11	1.085e+12	2	6.594e+11	1.082e+12	1	6.594e+11	1.082e+12	1
Median	1.005e+12	1.091e+12	4	1.077e+12	1.090e+12	2	1.077e+12	1.090e+12	2
3rd Qtl.	1.0817e+12	1.117e+12	14	1.090e+12	1.305e+12	4	1.090e+12	1.305e+12	4
Max	2.646e+12	2.679e+12	95887	2.654e+12	2.679e+12	54019	2.654e+12	2.679e+12	54019

Table 4.4: Statistics of the different time series based on grouping by the categories, showing the first and last event start time and the number of events.



(a) Length of time series for users.



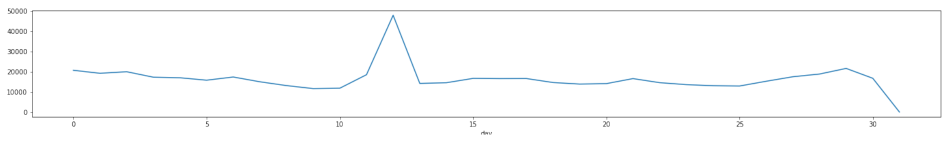
(b) Length of users-collection names time series.

Figure 4.6: Time series length of the different available time series combinations. Each data point represents one series. Fig (a) shows an aggregation by users, (b) by collection names, and (c) by a combination of user and collection name.

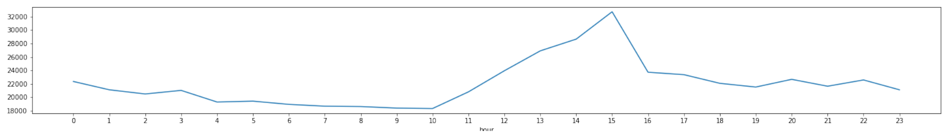
4.3.3 Time Series Study

The dataset can be grouped into several time series, so let's look at the distribution and number of the timestamps for each grouping. Table Figure 4.6 shows the length of the different available groupings for time series, and Table 4.4 shows the *users* series for time and number of observations. They show that the number of events is not regular, the length of the series differs greatly between categories in each grouping, and the time stamps are not aligned.

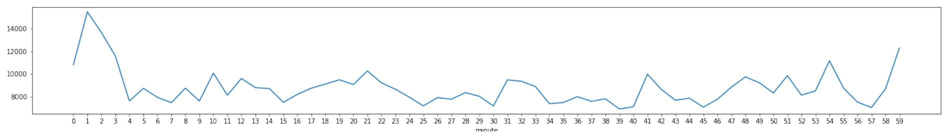
Grouping the number of jobs by periods of day, hour, and minute (Table 4.7), we observe peaks of job requests on days 6, 11, and 12. On an hour level, jobs are requested more frequently between 11 and 16 hours, and keep decreasing from there until 10 hours. On a minute level, some peaks are observed; and we see jobs are requested more around minutes 59 to 2.



(a) Jobs by Day



(b) Jobs by Hour



(c) Jobs by Minute

Figure 4.7: Number of jobs aggregated by (a) Days, (b) Hours, (c) and Minutes to show seasonality.

As the data shows, we have hundreds of very irregular series; they are not aligned on time or number of observations, displaying large differences in the number of observations between series, and where series on Q3 have 14 or fewer jobs, whereas for series above Q3 the number keeps exponentially increasing until reaching 95,887 on the largest series.

Chapter 5

Regression Models

This section will focus on describing three supervised learning regression models and regression metrics that will be used in our work: Linear Regression, Random Forest, and Deep Neural Networks.

5.1 Random Forest (RF)

This ensemble method for regression obtains results by building a set of decision trees during the training phase. Random Forest was initially developed in 1995 with the idea of growing trees with a random selection of features [Ho, 1995] and lately extended [Breiman, 2001] to add the idea of bagging (bootstrap aggregating) to keep the variance of the "forest" of trees under control. For regression, the average of the individual trees is returned as the prediction.

Random Forest has a tendency to overfit the training set. This can be controlled by tuning the maximum number of features that are randomly chosen to grow each tree [Hastie et al., 2009]. This happens because each tree has high variance but low bias, averaging moderates the variance while keeping the bias low. However, interpretability provided by decision trees is lost.

Data scaling is not required as this is a decision tree-based algorithm and, in practice, they have shown to be robust to outliers; essentially, the effect of the outliers is diminished because of the averaging between all the trees.

5.2 Linear Regression (LR)

For n observations of t continuous, dependent, variables $\boldsymbol{Y}$ a multiple linear regression tries to establish a relation between $\boldsymbol{Y}$ and the explicative variables $\boldsymbol{X} = (1, X_1, X_2, \dots, X_p)$ with a function $\boldsymbol{Y} = f(X) + \varepsilon$. This function can be expressed as $f(x) = \boldsymbol{X\beta}$, leaving us with the formal definition:

$$\boldsymbol{Y} = \boldsymbol{X\beta} + \boldsymbol{\varepsilon} \tag{5.1}$$

where $\boldsymbol{Y} \in \mathcal{M}_{nxt}, \boldsymbol{X} \in \mathcal{M}_{nx(p+1)}, \boldsymbol{\beta} \in \mathcal{M}_{pxt}$ and $\boldsymbol{\varepsilon} \in \mathcal{M}_{nxt}$. An extra requirement is that $(n > p)$ which means that we need more data than explicative variables [James et al., 2013].

The only problem with this first definition is that it will not fit a nonlinear relationship in the data. To solve this, polynomial regression exists, which is a slight variation of linear regression:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + ... + \beta_m x_i^m + \varepsilon_i (i = 1, 2, ..., m) \tag{5.2}$$

Even with this modification, the model is still linear in terms of the unknown parameters β by treating $x, x^i, ..., x^m$ as independent variables, so it is still a linear regression.

5.3 Deep Neural Networks (DNN)

These kinds of models were inspired by the networks of biological neurons found in our brains. Deep Neural Networks can be described, essentially, as Neural Networks with a large number of layers. This makes it hard to train them because of the large number of parameters to be learned; nowadays, Deep Neural Networks can be trained in a reasonable amount of time thanks to the tremendous increase seen since the 1990s in computing power [Géron, 2019]. Each neuron can be considered as linear regression with, typically, a nonlinear activation function (although we can have neurons without any activation function). The weights of each neuron are learned by a method called backpropagation [Goodfellow et al., 2016], which allows to calculate and attribute the error associated with each neuron. By using this error, it is possible to adjust and fit the weights of each neuron. Therefore we can define a neural network model as the combination of the network architecture and the weights learned by each neuron.

Several techniques have appeared to speed-up training times and reduce overfitting, allowing the networks to train faster and better [Ioffe and Szegedy, 2015a, Goodfellow et al., 2016, Srivastava et al., 2014]. For this work, we will try building a model using:

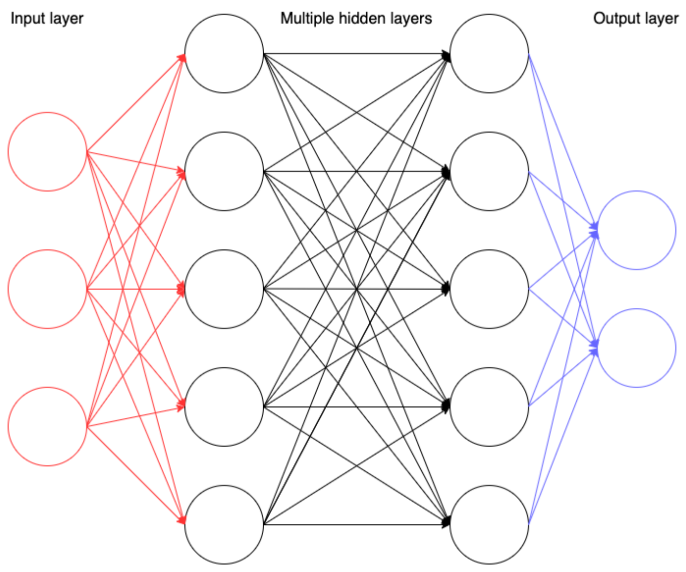


Figure 5.1: Example of a Neural Network with fully connected layers. It is composed of 3 neurons in the input layer, 2 hidden layers with 5 neurons each and 2 neurons in the output layer.

- Batch Normalization: This technique stabilizes the learning process and reduces drastically the number of epochs required for training deep neural networks. It does it by standardizing the activations of a previous layer so that the following layer will not make dramatically different assumptions about the inputs when the weights are updated.

- Dropout: Reduces overfitting and improves regularization by disconnecting, or dropping out, randomly the outputs of a layer. This has the effect of creating layers with different numbers of neurons and connectivities between layers so that the training becomes noisy and forces different nodes to learn more about specific inputs.

- Mini-batch Gradient Descent: This variation of the gradient descent algorithm consists in splitting the training dataset into small batches. The default is usually 32, but some recent work has proven better results when training with smaller mini-batches, as small as 2 or 4 [Masters and Luschi, 2018].

As the loss function, we will use *MSLE (Mean Squared Logarithmic Error)*, the measure of the percentual difference between the true and the predicted value of the dependent variable. The main benefit of this metric is that it penalizes more underestimates than overestimates (which introduces an asymmetry in the error curve), and since it is a relative difference, it does not penalize more large differences on large values, which makes it robust to outliers. It is described with the formula:

$$MSLE = \frac{1}{n}\sum_{i=1}^{n}(log_e(1+y_i) - log_e(1+\hat{y}_i))^2 \tag{5.3}$$

5.4 Scoring Metrics

The following metrics will be used for measuring the performance of the algorithms:

- R^2 *(Coefficient of determination):* Proportion of variation in the dependent variable that is predictable from the independent variables. The best possible value is 1, and 0 is equivalent to just returning the average of the dependent variable without considering the inputs. It can also have negative values. It is obtained by dividing the *residual sum of squares* between the *total sum of squares*:

$$R^2 = 1 - \frac{\sum_i e_i^2}{\sum_i (y_i - \overline{y})^2} \tag{5.4}$$

- *MAE (Mean Absolute Error)*: Measure of errors between all the observations. It is less affected by outliers than mean squared error, as it penalizes less large errors. It is obtained by dividing the sum of absolute errors by the number of observations:

$$MAE = \frac{\sum_{i=1}^{n} |e_i|}{n} \tag{5.5}$$

Chapter 6

Model Training and Hyperparameter Tuning

In this chapter, we will present the features preprocessing required, architectures, training, and hyperparameter tuning for the different models. For all the models, the dataset will be previously adjusted based on the *start_time* as explained in 4.3.1. Once the hyperparameter tuning of all the models

6.1 Categorical and Numerical Features

In regard to feature transformation and scaling it is required to process numerical features scaling them so that they can be comparable, and encode categorical features into a numeric format before feeding them to the algorithms. As we will see on Random Forest, this is not required for that specific model, but for Neural Networks and Linear Regression, it is fundamental [Bishop et al., 1995]. Therefore, we will clip the values of the numerical variables described on Figure 4.4 by setting a maximum value for each variable using the largest non-highlighted quantile value. We expect to reduce the number of outliers that can affect the training.

The main reason for scaling the data on Linear Regression is that the coefficients can be extremely different for data on different scales, reducing interpretability. As an example, let's consider two dummy features a, which is always positive, and b, which is always negative, with different scales. Centering them around 0 by standardizing them allows the coefficients to be able to compare directly units in both features, keeping coefficients on the same scale.

Categories	oh_1	oh_2	oh_3	b_1	b_2
'cat_a'	1	0	0	0	1
'cat_b'	0	1	0	1	0
'cat_c'	0	0	1	1	1

Table 6.1: One Hot encoding and Binary encoding for a categorical variable. The first column represents a categorical column, the next three columns are the one-hot encoding, and the last 2 represent the binary encoding.

This reasoning also applies to neural networks, but an even more important reason for scaling features is that it boosts gradient descent to converge faster [Ioffe and Szegedy, 2015b].

6.1.1 Features Engineering and Selection

We added two new features, $duration = end\_time - start\_time$, and $burstable$, which is a binary feature defined as $burstable = resource\_request\_cpus == 0$. The reasoning behind this new feature is that jobs where $resource\_request\_cpus = 0$ use free cpu capacity for processing. This way the models might benefit from this information.

Features selection is the process of selecting a subset of relevant variables. The main benefits are reducing training time, avoiding overfitting, and obtaining better predictions. For this reason, we removed $start\_time$ and end_time as they showed no correlation to the target variables 4.5.

The three chosen models require that we encode categorical features as float numbers (although Random Forest also accepts integers). We will experiment with:

- *one-hot encoding:* Consists of encoding k categories of a feature as a sparse matrix $\mathcal{M}_{nxk}$ where n is the number of statistical units, or observations.
- *binary encoding:* Encoding with an approach similar to one-hot encoding, but instead of creating one column per category, it encodes each category as an ordinal number and then encodes each of those numbers with their binary representation, with as many columns as bits required.

An example of both encodings can be seen in Table 6.1.

6.2 Training, Validation, and Test Sets

Before training, we need to split the data first into 3 datasets randomly sampled [James et al., 2013] which, later, will be split into features and targets:

- *Training dataset:* This dataset will be used to fit the models (75%).
- *Validation dataset:* This dataset will be used while tuning each model hyperparameters to provide an unbiased evaluation of the model that was fit with the training data. When the hyperparameter tuning is done, it will be used together with the training dataset to fit a model. (12.5%)
- *Test dataset:* This dataset will be used for unbiased evaluation of the different models after having fit them with the combination of the training and validation datasets. (12.5%)

6.2.1 K-Fold Cross Validation

Above, we introduced the validation dataset usage and its impact on the model bias, but it has limitations [James et al., 2013] such as having no randomness because the validation set is constant. To reduce even further overfitting, *k-fold cross-validation* can be applied. This technique consists of "randomly dividing the train set of observations into k groups, or folds, of approximately equal size. The first fold is treated as a validation set, and the method is fit on the remaining $k - 1$ folds" [James et al., 2013]. Then, we execute rounds until all the folds are evaluated, and combine the results, usually, by averaging them.

6.3 Random Forest (RF)

In the models section 5.1 we explained that RF does not require feature scaling, but the categorical features need to be encoded. We will try both representations, evaluating against the validation set two models trained with the same hyperparameters and proceed with the one that has the best performance.

The main hyperparameters for RF are [Owen, 2022]:

- *n_estimators:* Number of decision trees to be utilized to build the forest. In general, the larger the number, the better. The main trade-off is the increasing computational cost. Also, at a certain threshold, the number of trees no longer increases performance significantly (or even decreases, due to overfitting).

- *max_features:* The maximum number of randomly sampled features used by each tree to choose a splitting point. The squared root of the number of features is a nice default to start searching.
- *min_samples_split:* This specifies the minimum number of samples required for a tree to be able to split a node. Higher values help prevent overfitting.
- *max_depth:* The maximum depth that each tree can grow. Limiting it reduces overfitting. In practice, leaving the trees to grow to maximum length has no major impact other than longer computing time if we tune *min_samples*.
- *bootstrap:* Whether bootstrap samples are used when building trees instead of the whole dataset.

The hyperparameters will be selected by applying 5-fold crossvalidation 6.2.1.

6.4 Linear Regression (LR)

For linear regression to work properly, it is needed to do a feature selection first to ensure no correlation between variables, which can negatively affect the predictions. Moreover, numerical features need to be scaled or normalized and categorical features, encoded. As we described in Preprocessing 6.1, the values of numerical data will be clipped, so we are able to apply standardization, to scale to unit variance.

We will try two linear regressions with default hyperparameters, one regular linear regression (LR), and a polynomial regression (PR), and check the performance of one-hot encoding and binary encoding on them. Then, with the best, the following hyperparameters could be tuned:

- *fit intercept (LR and PR):* Whether to calculate the intercept of the linear regression.
- *polynomial features degree (PR only):* Maximal degree of the polynomial features.
- *use interaction only (PR only):* Whether to use interaction features only.

- *include bias (PR only):* Whether to include the bias column.

The hyperparameters will be selected by comparing each model score against the validation set.

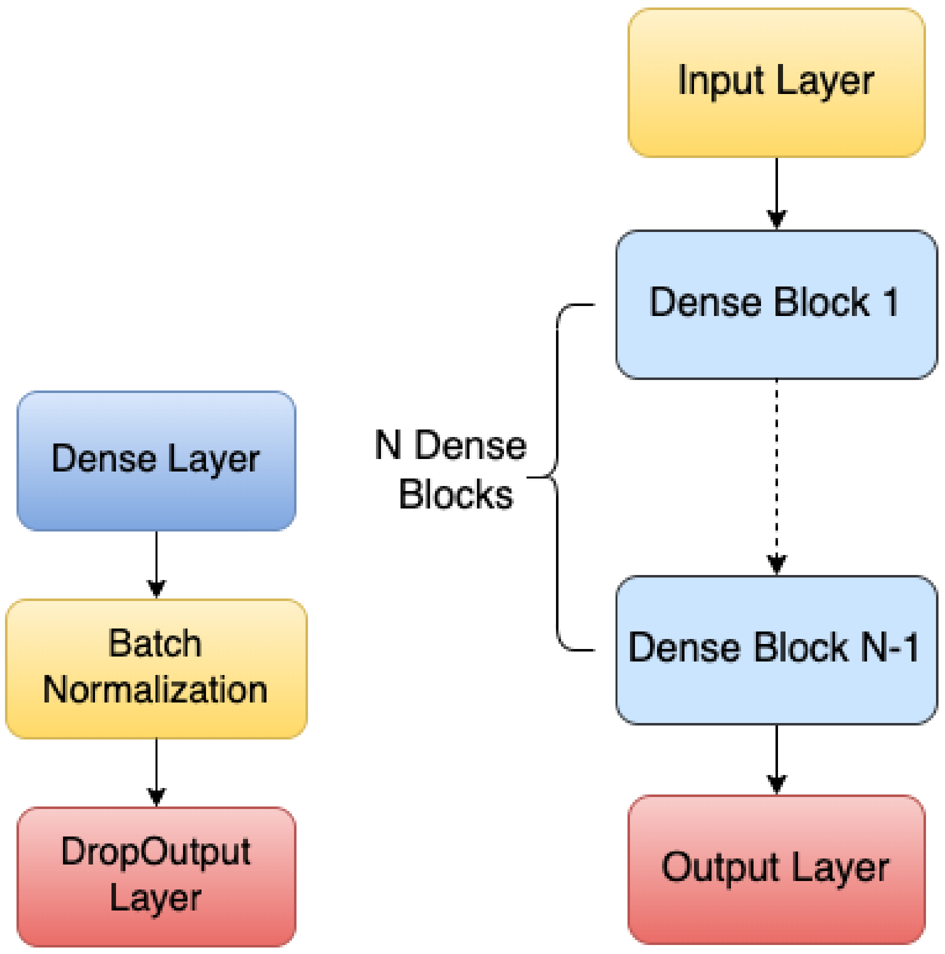
6.5 Deep Neural Network (DNN)

The features preprocessing required for DNN will be the same as for LR 6.4, so we will apply the same as the final one chosen for LR. For the architecture, we will try different setups combining dense layers with *relu* as activation function, batch normalization, and dropout. Finally, the output layer will be a dense layer with three units, and without any activation function. To find a correct setup we will train and validate with a reduced sample from the training dataset. The hyperparameters for the DNN will be:

- *width:* Number of units per dense layer.
- *depth:* Number of dense blocks 6.1, which are composed of a dense, normalization, and dropout layer.
- *batch_size:* The number of observations to include in each batch for training.

The number of units per dense layer, the number of layers (or depth), and the batch size used for training the model.

The hyperparameters will be tuned by comparing the fitted models against the validation set.



(a) Dense Block (b) Network Architecture

Figure 6.1: Details on how the network architecture is built, and a dense block composition.

Chapter 7

Results and Discussion

Through this chapter, the results of the experiments will be shown, and the following questions will be answered:

1. Which model performs better?
2. Which models are more complex in terms of training time, prediction time, and hyperparameter tuning?
3. Do we overpredict more than underpredict?
4. How much do the independent variables explain the dependent variables?

We expect that the answers to these questions will help us evaluate if these results help in the selection of better resource request values for lightweight containers.

7.1 Results

In this section, we will show the results of the experiment, including final hyperparameters, model performance, and DNN architecture.

7.1.1 Feature Engineering, Scaling, and Importance

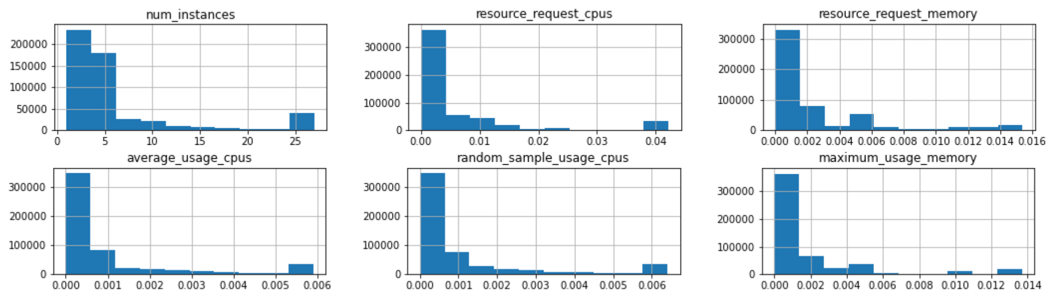
Feature engineering was required to train the models. Table 7.1 shows the effect that encoding and scaling of the data have on the predictive power. Looking at the impact that the clipped

Model	Validation Score (R^2)	Training Time
RF Binary Unclipped	0.92	0:00:10
RF Binary Clipped	**0.93**	**0:00:09**
RF One-Hot Unclipped	0.92	1:06:24
RF One-Hot Clipped	0.93	0:11:14
LR Binary Unclipped	0.65	**0:00:05**
LR Binary Clipped	0.74	0:00:05
LR One-Hot Unclipped	0.84	0:00:08
LR One-Hot Clipped	**0.92**	0:01:22
PR Binary Unclipped	0.85	0:01:16
PR Binary Clipped	**0.88**	**0:01:10**

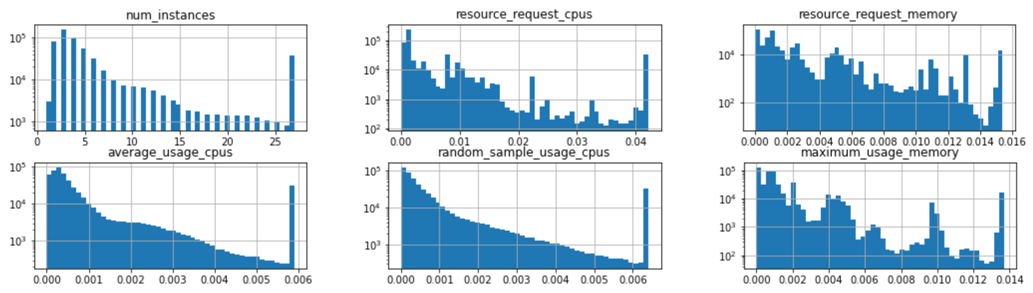
Table 7.1: Binary and One-Hot encoding with Standard and Robust scaling (Clipped data is scaled with Standard scaler, Unclipped data with Robust scaler) comparison, with training time rounded to the nearest second. One-Hot for Polynomic Regression is missing due to R2 being 0 for both Standard and Robust scaling. In bold is the best score and training time. All the models are trained with the default hyperparameters.

dataset distribution 7.1 has, for Random Forest (RF) (For reference, Figure 4.3.1 shows the original distribution), the scores are all very similar, although the best score and training time are obtained when the numeric features are clipped and the numeric features are binary encoded. For Linear Regression (LR) the score and training times are inversely proportional, the best model in terms of scoring is obtained when numeric features are clipped and categorical features are encoded with one-hot encoding but show the largest training time. Finally, for Polynomial Regression (PR) the best score and training time are obtained by clipping and binary encoding. The results show that the PR performs worse than LR in terms of scoring while having a similar training time. The models trained with clipped numeric data obtained better validation scores.

Figure 7.2 shows that *resources request* features, *user* and *duration* are the most important features on the hyperparameter-tuned Random Forest.



(a) Normal Scale



(b) Log Scale

Figure 7.1: Effect on the distributions of the features when clipping is applied.

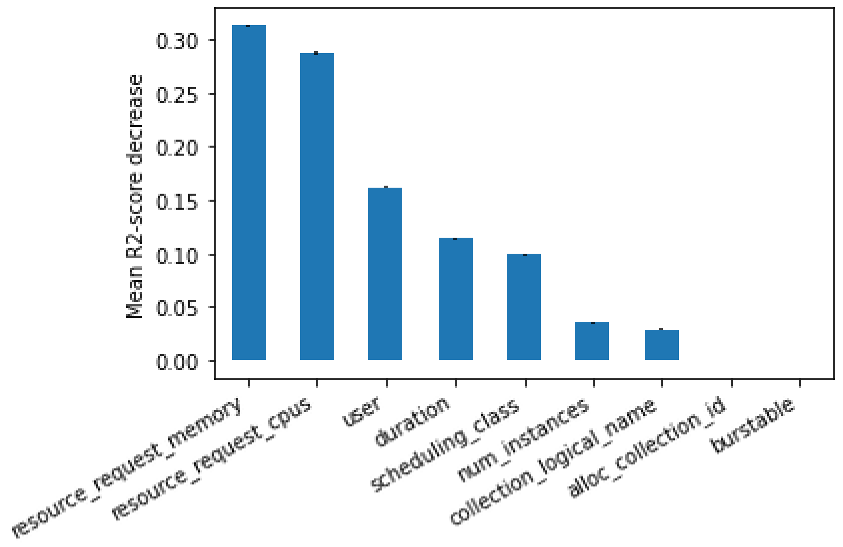


Figure 7.2: Features importance extracted from the Random Forest model.

Model	Hyperparameter	Values Explored
RF	bootstrap	True, False
	max_features	3, 4, 5, 6, 7
	min_samples_split	2, 3, 4, 5
	n_estimators	[10, 150]
LR	fit_intercept	True, False
DNN	batch_size	32, 64, 128, 1024, 2048
	width	32, 64, 128, 256, 512, 1024
	depth	[2, 10]

Table 7.2: Explored hyperparameters for each model Values are expressed as comma-separated lists or as intervals.

7.1.2 Hyperparameters Tuning

Based on the previous results, Linear Regression was trained without polynomial features. Table 7.2 shows all the hyperparameters and the range of values explored for each of them. The Deep Neural Network (DNN) has the larger number of possible permutations, 60, and LR the lowest, with 2. The neural network obtained with the selected hyperparameters has 1,075,715 trainable and 5,120 non-trainable parameters.

7.1.3 Models Performance

Table 7.4 shows a summary of training and prediction times. The Deep Neural Network is the model that takes longer to train. On predictions, 5 predictions were generated by each of them and the mean time for these predictions is displayed, with Linear Regression being the faster. Looking at Table 7.5, the model with the best performance is Random Forest, followed closely by Linear Regression. All the models are negatively biased on cpu, but only Random Forest and Linear Regression have a positive bias on memory.

Model	Hyperparameter	Final Values
RF	bootstrap	False
	max_features	5
	min_samples_split	3
	n_estimators	80
LR	fit_intercept	False
DNN	batch_size	128
	width	512
	depth	5

Table 7.3: Final hyperparameters found for each model.

Model	Train Time	5-Predictions Mean Time
RF	02:26.39	00:03.75
LR	02:08.81	00:00.37
DNN	23:45.64	00:03.80

Table 7.4: Duration, for the final model training and the average of 5 predictions.

Model	Target	R^2	MAE	Bias	Agg. R^2	Agg. MAE	Agg. Bias
RF	max_mem	0.99	6.94e-5	3.09e-07	0.92	2.01e-4	-4.48e-7
	avg_cpu	0.96	1.44e-4	-2.65e-07			
	rnd_cpu	0.88	2.75e-4	-1.39e-06			
LR	max_mem	0.99	9.01e-5	7.83e-08	0.92	2.22e-4	-2.24e-7
	avg_cpu	0.94	1.83e-4	-3.59e-07			
	rnd_cpu	0.92	2.47e-4	-3.90e-07			
DNN	max_mem	0.82	9.81e-4	-3.07e-4	0.78	7.67e-4	-2.12e-4
	avg_cpu	0.81	5.67e-4	-1.59e-4			
	rnd_cpu	0.74	6.73e-4	-1.68e-4			

Table 7.5: Comparison of the scores and bias on the test dataset.

7.2 Discussion

Having reviewed the results of all the experiments executed, now, we will analyze them.

In regards to the best model in terms of score, Random Forest has shown the best results. We did not limit the depth of the individual trees, so they could have easily overfit the dataset, but since we limited the number of features randomly chosen by tuning this hyperparameter, we prevented the overfitting while keeping the bias low and reducing the variance, because the results of the trees are averaged [Breiman, 2001]. Effectively, each tree is working with a subset of the data, so they can be uncorrelated from each other. Interestingly, bootstrapping was disabled when we found the best hyperparameters. This can be explained by the large number of samples in our dataset, which makes resampling data with replacement unnecessary.

If we look at the complexity of the models, the Deep Neural Network has proven to be the hardest to train and configure. Training time is large, and choosing the correct architecture requires large experimentation; the network has many hyperparameters that can be tuned, which depend on the architecture chosen. Others like the activation function used on each layer, or whether an activation function is used at the outputs layer, or the loss metric, can be narrowed into a smaller selection, but they need to be tried nevertheless. The batch size has also proven to affect the training speed, which compared to the other models is dramatically higher. This is inherent to the great number of parameters that the network needs to learn. Dropout also increases the number of required parameters, as it effectively disconnects parts of the network during training to prevent neurons from co-adapt [Hinton et al., 2012].

Feature engineering and scaling have proven to be key for linear regression and the neural network. When feeding clipped vs unclipped we saw that it has a dramatic impact on the performance of these two models. In a neural network, if the data has outliers or the features are on very different scales, the larger values can greatly impact how the weights are adjusted, making the training much more unstable, or even preventing it from converging. Linear regression is also very sensitive to outliers as these data points do not follow the general trend of the rest of the data, and this is exactly what linear regression tries to achieve. Random Forest proved to be robust to outliers, no feature scaling was required and although clipping improved the scoring, the impact was not huge. Higher dimensionality on the categorical data impacted negatively the training time for Random Forest, as the makes it less likely that numerical features are chosen, so the trees have fewer options for splitting the data, which induces sparsity in the trees.

Analyzing the aggregated metrics, we see they are always penalized because the random usage cpus consistently have lower scores on R^2 and MAE, which can be explained by the fact that random usage cpus express the average of random samples of cpu usage sampled each second during the execution of a job. This approximates the average cpu usage for long-running jobs, but for tasks that run in periods smaller than the maximum 300 seconds window, this metric is unreliable. The high bias (Table 7.5) of the neural network indicates that we might need to reduce the dropout. For LR and RF, if we go back to the numeric statistics (Table 4.2) the average memory usage is 2.12e-3, which means we have low positive bias +3.09e-7, average usage cpu, and random usage cpu with low negative bias. This indicates that we tend to slightly overpredict the memory usage and slightly underpredict the cpu usage with RF and LR, which is good, as maximum memory is a hard limit (if a job goes over the memory limit it is killed), but cpu usage is not (a job not having sufficient cpu will simply throttle).

Finally, Figure 7.2 shows that for Random Forest the *alloc_collection_id* and *burstable* variables do not influence the dependent variables, whereas the resource requests are the most important. Looking at the coefficients of determination in Table 7.5 we see that almost all *max_memory* and *average_usage_cpu* variation can be predicted with the independent variables.

Chapter 8

Conclusions and Future Work

From an objectives perspective, if we recall what we defined at the beginning of the thesis, 1.2, find existing open datasets with production traces of lightweight containers, and develop a practical case study of the dataset, we can consider both of them achieved.

Finding open datasets for lightweight containers proved to be a hard task, as only two were found, Google's Cluster Data [Wilkes, 2020b] and Alibaba [Zhang, 2017]. Most often, datasets found either did not contain container usage data or were too old. Also, We managed to obtain a model that allows us to predict the usage of memory and cpu. Random Forest has proven to be easy to train; even with the default parameters and no feature scaling performed well on tabular data. Hyperparameter tuning is relatively easy compared to other complex models like Deep Neural Networks, and little feature engineering is required. In fact, only the categorical variables need to be encoded.

Finally, when it comes to the predictions of resource usage, the memory predictions are positively biased and memory usage has a very small overprediction. As long as the predicted memory is lower than the requested memory, the jobs will not be terminated and the memory usage on the cluster can be reduced, resulting in more memory available and capacity for more jobs. As for the cpu usage, it is slightly underpredicted, but there exist tasks that set cpu request as 0, so they use free capacity. This is because cpu is not such a critical resource, so under, and although there can be peaks of usage, cpu can be freed during the execution of a task, while memory is not. If a job needs to ensure the cpu and memory availability using a high priority, as monitoring jobs do, will ensure it has the resources always available during the execution.

8.1 Future Work

We propose some approaches related to this work:

- An interesting approach would be to take the results from this work and try to apply them to the other cells provided in the original dataset. Also, it could be interesting to obtain trace data from a production Kubernetes cluster or from other datasets that contain lightweight containers with equivalent metadata.

- The data could be processed differently to obtain other kinds of jobs, like failed jobs, and try to predict which job definitions are more likely to not finish so that the user can be warned at submission time.

- From a time series point of view, Google's Cluster Data dataset [Wilkes, 2020a] presented here could be used to fit a model to forecast resource usage for specific priorities, like forecasting the number of production tier jobs that will be requested, or free capacity. The latter approach could be used to optimize node provisioning for the cluster by planning to allocate other lower priority jobs on the forecasted free capacity so that the cluster is not underused.

Appendix A

APPENDIX A: Converting JSON GZIP data to Parquet

An example of how to read and transform large datasets by using Dask will be available on `https://github.com/Hijus22/cluster-usage-forecasting`.

Bibliography

[Abdul-Rahman and Aida, 2014] Abdul-Rahman, O. A. and Aida, K. (2014). Towards understanding the usage behavior of Google cloud users: the mice and elephants phenomenon. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 272–277, Singapore.

[Bishop, 2013] Bishop, C. (2013). *Pattern Recognition and Machine Learning: All "just the Facts 101" Material*. Information science and statistics. Springer (India) Private Limited.

[Bishop et al., 1995] Bishop, C. M. et al. (1995). *Neural networks for pattern recognition*. Oxford university press.

[Blog, 2015] Blog, K. (2015). Borg: The predecessor to kubernetes. Kubernetes Blog. Posted at https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/. (accesed on 2021-04-16).

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

[Burns et al., 2016] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes. *ACM Queue*, 14:70–93.

[Casals et al., 2009] Casals, J., Jerez, M., and Sotoca, S. (2009). Modelling and forecasting time series sampled at different frequencies. *Journal of Forecasting*, 28:316 – 342.

[Chen et al., 2015] Chen, T., He, T., Benesty, M., Khotilovich, V., Tang, Y., Cho, H., Chen, K., et al. (2015). Xgboost: extreme gradient boosting. *R package version 0.4-2*, 1(4):1–4.

[Cheng et al., 2018] Cheng, Y., Chai, Z., and Anwar, A. (2018). Characterizing co-located datacenter workloads: An alibaba case study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–3.

[Crist, 2016] Crist, J. (2016). Dask & numba: Simple libraries for optimizing scientific python code. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2342–2343. IEEE.

[Eckner, 2012] Eckner, A. (2012). Algorithms for unevenly-spaced time series: Moving averages and other rolling operators. In *Working Paper*.

[Géron, 2019] Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.".

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

[Guo et al., 2019] Guo, J., Chang, Z., Wang, S., Ding, H., Feng, Y., Mao, L., and Bao, Y. (2019). Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, New York, NY, USA. Association for Computing Machinery.

[Han et al., 2012] Han, J., Kamber, M., and Pei, J. (2012). 12 - outlier detection. In Han, J., Kamber, M., and Pei, J., editors, *Data Mining (Third Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 543–584. Morgan Kaufmann, Boston, third edition edition.

[Hastie et al., 2009] Hastie, T., Tibshirani, R., Friedman, J. H., and Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer.

[Hellerstein, 2010] Hellerstein, J. L. (2010). Google cluster data. Google research blog. Posted at `http://googleresearch.blogspot.com/2010/01/google-cluster-data.html`. (accesed on 2020-04-12).

[Hewamalage et al., 2021] Hewamalage, H., Bergmeir, C., and Bandara, K. (2021). Recurrent neural networks for time series forecasting: Current status and future directions. *International Journal of Forecasting*, 37(1):388–427.

[Hinton et al., 2012] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors.

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

[Ioffe and Szegedy, 2015a] Ioffe, S. and Szegedy, C. (2015a). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

[Ioffe and Szegedy, 2015b] Ioffe, S. and Szegedy, C. (2015b). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

[Jajoo et al., 2021] Jajoo, A., Hu, Y. C., Lin, X., and Deng, N. (2021). The case for task sampling based learning for cluster job scheduling. *Computing Research Repository*, abs/2108.10464.

[James et al., 2013] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning*. Springer (New York).

[Jaramillo et al., 2016] Jaramillo, D., Nguyen, D. V., and Smart, R. (2016). Leveraging microservices architecture by using docker technology. In *SoutheastCon 2016*, pages 1–5.

[Kullaya Swamy and Sarojamma, 2020] Kullaya Swamy, A. and Sarojamma, B. (2020). Bank transaction data modeling by optimized hybrid machine learning merged with arima. *Journal of Management Analytics*, 7(4):624–648.

[Kursuncu et al., 2019] Kursuncu, U., Gaur, M., Lokala, U., Thirunarayan, K., Sheth, A., and Arpinar, I. B. (2019). *Predictive Analysis on Twitter: Techniques and Applications*, pages 67–104. Springer International Publishing, Cham.

[Laptev et al., 2017] Laptev, N., Yosinski, J., Li, L. E., and Smyl, S. (2017). Time-series extreme event forecasting with neural networks at uber. In *International conference on machine learning*, volume 34, pages 1–5. sn.

[Lim et al., 2019] Lim, B., Arik, S. O., Loeff, N., and Pfister, T. (2019). Temporal fusion transformers for interpretable multi-horizon time series forecasting.

[Liu and Cho, 2012] Liu, Z. and Cho, S. (2012). Characterizing machines and workloads on a google cluster. In *2012 41st International Conference on Parallel Processing Workshops*, pages 397–403. IEEE.

[Lu et al., 2017] Lu, C., Ye, K., Xu, G., Xu, C.-Z., and Bai, T. (2017). Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892.

[Masters and Luschi, 2018] Masters, D. and Luschi, C. (2018). Revisiting small batch training for deep neural networks. *CoRR*, abs/1804.07612.

[Owen, 2022] Owen, L. (2022). *Hyperparameter Tuning with Python*. Packt Publishing.

[Prasad and Madhavi, 2012] Prasad, U. D. and Madhavi, S. (2012). Prediction of churn behavior of bank customers using data mining tools. *Business Intelligence Journal*, 5(1):96–101.

[Sebastio et al., 2018] Sebastio, S., Trivedi, K. S., and Alonso, J. (2018). Characterizing machines lifecycle in google data centers. *Performance Evaluation*, 126:39 – 63.

[Smedt et al., 2021] Smedt, J. D., Yeshchenko, A., Polyvyanyy, A., Weerdt, J. D., and Mendling, J. (2021). Process model forecasting using time series analysis of event sequence data. In *International Conference on Conceptual Modeling*, pages 47–61. Springer.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

[Tirmazi et al., 2020a] Tirmazi, M., Barker, A., Deng, N., Haque, M. E., Qin, Z. G., Hand, S., Harchol-Balter, M., and Wilkes, J. (2020a). Borg: the next generation. In *EuroSys'20*, Heraklion, Crete.

[Tirmazi et al., 2020b] Tirmazi, M., Barker, A., Deng, N., Haque, M. E., Qin, Z. G., Hand, S., Harchol-Balter, M., and Wilkes, J. (2020b). Borg: the Next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*, Heraklion, Greece. ACM.

[Verma et al., 2015] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys'15)*, Bordeaux, France.

[Wilkes, 2011] Wilkes, J. (2011). More Google cluster data. Google research blog. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html. (accesed on 2022-09-09).

[Wilkes, 2020a] Wilkes, J. (2020a). Google cluster-usage traces v3. Technical report, Google Inc., Mountain View, CA, USA. Posted at https://github.com/google/cluster-data/blob/master/ClusterData2019.md. (accesed on 2022-09-12).

[Wilkes, 2020b] Wilkes, J. (2020b). Yet more Google compute cluster trace data. Google research blog. Posted at https://ai.googleblog.com/2020/04/yet-more-google-compute-cluster-trace.html. (accesed on 2022-09-11).

[Xiao et al., 2012] Xiao, Z., Song, W., and Chen, Q. (2012). Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE transactions on parallel and distributed systems*, 24(6):1107–1117.

[Yan and Su, 2009] Yan, X. and Su, X. (2009). *Linear regression analysis: theory and computing*. World Scientific.

[Zhang, 2017] Zhang, Z. (2017). Alibaba/clusterdata: Cluster data collected from production clusters in alibaba for cluster management research.