

Performance Predictions of Spark Jobs with Machine Learning Tasks Using Various Artificial Intelligence Models

Eleni Boukouvala

School of Electrical and Computer Engineering

National Technical University of Athens

Athens, Greece

el15741@mail.ntua.gr

Spiros Diochnos

School of Electrical and Computer Engineering

National Technical University of Athens

Athens, Greece

el15727@mail.ntua.gr

Abstract—As data continues to grow faster than chips continue to shrink the need to divide the work in multiple CPUs rises. And as workloads and users continue to demand responsiveness and low latency, memory based distributed computing technologies are going to become even more mainstream. One such technology is Apache Spark and with the multiple possible parameters to play with, tuning it exactly to your needs requires a specialist. We aim to make an AI tool for those specialists that predicts the CPU and memory requirements of a Spark job depending on the initial parameters and also a few snapshots of the system after the execution has already begun. In this paper the focus is on machine learning and AI workloads. And yes we are using AI to predict for how long an AI will run. Our models can reach accuracy over 95% with precision of 1 second.

Keywords—Apache Spark, Performance, Configuration Setting, Machine learning, prediction of execution time, Cloud computing

I. Introduction

The analysis of huge datasets is the main objective and at the same time the biggest challenge of research in the majority of academic and entrepreneurial fields. The available raw data not only need to be gathered and stored efficiently but also be analyzed in a meaningful way, so that patterns and insights can be identified. The provided knowledge has the potential to create new, experimental implementations in the specific field. In this context, Machine Learning (ML) methods are being utilised to properly extract useful insights from the massive amount of data. As feature extraction and pattern recognition are tedious and time-consuming procedures that with the slightest of deviance can lead to inaccurate results, data analysis using ML algorithms should be thoughtful and precise. The researches rely on working experience to access the time involved in their experiments. In order to fully explore the characteristics of the data, the utility of the algorithms used and the properties of the platform hosting the data analysis, a considerable effort and time must be put by the individuals at hand. To support the usage of Machine Learning techniques that automatically identify patterns and select features from large scale data, several distributed parallel computing platforms were developed. One of the most exceptional is Apache Spark, which is an open-source, general-purpose distributed processing system used for big data workloads. It utilises in-memory caching, and optimised query execution for fast analytic queries against data of any size. Except for the core engine, it supports the modules Spark SQL and Spark MLlib. Computations of high complexity are being assigned to machines and scheduled according to the current workload, minimising operational cost and time. However, the execution performance can be improved by taking into consideration the specific type of

application to be optimised. This can be done through

1. choosing the appropriate configuration settings of the Spark clusters like the number of machines and the resources available on each machine.
2. choosing the right settings for each machine, such as number of nodes, the number of executors per node
3. efficient scheduling Stages/Tasks on the nodes of a cluster
4. optimising Stages/Tasks (physical execution plan) of a Spark job.

The knowledge of the execution time and memory usage is useful on multiple occasions [1],[2],[5]. The more humane one is that the users of the application are informed on the running time of their experiments and they do not have to wait excessively for it to end. On the other hand, knowing the execution time beforehand helps with the scheduling of big tasks and minimises the overall workload.

In the present assignment, we attempt to predict the execution time of ML operators/algorithms (specifically implemented in PySpark and Spark MLlib) over a distributed computing system (Spark) by applying AI regression models. We chose 5 ML algorithms :

1. Random Forest Classification
2. Decision Tree Classification
3. Decision Tree Regression
4. Naive Bayes
5. Logistic Regression with ElasticNet

Our process can be described in the following steps :

- the generation of the data, the data features are the input parameters and metrics of the chosen ML operators
- the training of regression models on the induced dataset and their evaluation
- the application of the best model to predict the execution time/memory usage of ML tasks.

The body of the paper is organised in three parts. First we explain the configuration of our cloud environment including the configuration of the spark system. The second part is a brief overview of the produced data and finally a detailed explanation of the ML and AI models.

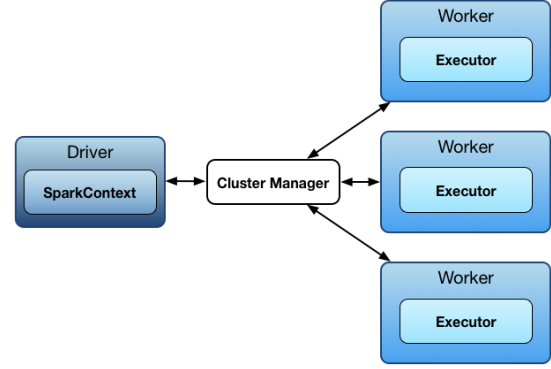
II. System Specifications and Configuration Overview

A. Hadoop Spark and Cloud Environment Configuration

For our cloud provider we had several options. Google Cloud, Microsoft Azure and AWS, which are market leaders, had compelling free tiers, but we decided to use the 3 VMs provided at Okeanos-Knossos mainly to remove some of the complexity and limitations the other cloud platforms.

<input type="checkbox"/>	OS	Name	Flavor	Status
<input type="checkbox"/>	Ubuntu 18.04	master	4 CPU, 8192MB, 30GB	Running
<input type="checkbox"/>	Ubuntu 18.04	slave1	4 CPU, 8192MB, 30GB	Running
<input type="checkbox"/>	Ubuntu 18.04	slave2	4 CPU, 8192MB, 30GB	Running

As mentioned before we had 3 VMs with each allocated 4 out of 20 threads E5-2650 v3 @ 2.30GHz and 8GB DDR4 RAM and 30GB of HDD storage. We installed HDFS and Apache Spark on all three nodes in the configuration shown below.



Since we had only three VMs the master node was both a driver and a worker. The Spark configuration varied between tests to have a more diverse dataset to work with. The cores per executor varied between one and four, the executor memory was integer multiple of 1GB up to 7 GB, and the executors were from 3 to 12. For easier development we used the Visual Studio Code SSH Server, which made file transferring and debugging much easier. Also it allowed us to forward ports to our local machines and therefore to use the Web UI for Hadoop and Spark.

B. Synthetic Data

file_100000.txt	244.25 MB	9/10/2022, 7:17:13 PM	2	64 MB	file_534800.txt
file_100001.txt	244.25 MB	9/10/2022, 7:17:14 PM	2	64 MB	file_540000.txt
file_100002.txt	244.25 MB	9/10/2022, 7:17:15 PM	2	64 MB	file_545000.txt
file_100003.txt	244.25 MB	9/10/2022, 7:17:16 PM	2	64 MB	file_550000.txt
file_100004.txt	244.25 MB	9/10/2022, 7:17:17 PM	2	64 MB	file_555000.txt
file_100005.txt	244.25 MB	9/10/2022, 7:17:18 PM	2	64 MB	file_560000.txt
file_100006.txt	244.25 MB	9/10/2022, 7:17:19 PM	2	64 MB	file_565000.txt
file_100007.txt	244.25 MB	9/10/2022, 7:17:20 PM	2	64 MB	file_570000.txt
file_100008.txt	244.25 MB	9/10/2022, 7:17:21 PM	2	64 MB	file_575000.txt
file_100009.txt	244.25 MB	9/10/2022, 7:17:22 PM	2	64 MB	file_580000.txt

For better results in our research and to remove as much bias as possible we run all the algorithms with the same synthetic data. The files are arrays with as many rows as the name of the file (e.g. file_582000.txt has 582000 lines) and 500 columns. In each column 80-100 values are 1 and the rest are 0 chosen randomly. This means our tests were made with 3 million to 0.3 billion data points.

C. ML Operators / Algorithms

The operators we chose were based on MLlib [7] which is Apache Spark's scalable machine learning library. And the majority of our testing was done on the following:

"random_forest_classifier_example.py"
 "decision_tree_classification_example.py"
 "decision_tree_regression_example.py"
 "naive_bayes_example.py"
 "logistic_regression_with_elastic_net.py".

We concluded that these are a good representation of the Library and with slight modification we could use the same input data for all of them.

D. Input Parameters and Metrics

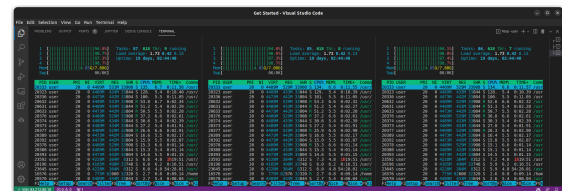
At the time of writing we have more than 165 thousand files with metrics. So to focus on results: from all the metrics we selected to focus only on the following 6, `"*.executor.runTime"`, `"*.executor.jvmCPUtime.csv"`, `"*.executor.CPUtime.csv"`, `"*.driver.BlockManager.memory.maxOnHeapMem_MB.csv"`, `"*.jvm.non-heap.used.csv"`, `"*.jvm.heap.used.csv"`. Three of them are related to execution time and the rest of them are about memory. Notice that storage is suspiciously absent, this is by design since we wanted to focus on tasks that run mainly on memory[11],[12].

Metrics	Definition
sumCPU (s)	This value represent the sum of the CPU time of all the executors
maxHeap (MB)	This value represent the max of the on heap memory of all the executors
maxNonHeap(MB)	This value represent the max of the off-heap memory of all the executors
sumRun(s)	this value represent the sum of the run time of the executors
sumjvmCPU(s)	This value represent the sum of the CPU time of the Java Virtual Machine
maxdrheap(MB)	This value represent the

	maximum of the driver memory
Input Parameters	Definition
spark.executor.instances	Number of executors
spark.app.name	The algorithm used
spark.executor.cores	Number of threads per spark executor
spark.executor.memory(GB)	Memory per spark executor
input.size (1000 lines)	The number of lines in the input files (in thousands)

We collected metrics every second with the intention to make predictions in real time. This is not yet implemented, for now we have sampled the metrics 4 times in the execution and one at the end. So in the collected data we will find metrics in this form `"sumRun(s 3/5)"` that represent the run time of all the executors at three fifths of the run. Of course in reality time only goes one way so we cannot know when this moment will come, but in this way we can still make predictions during the run even if it is a little cheating.

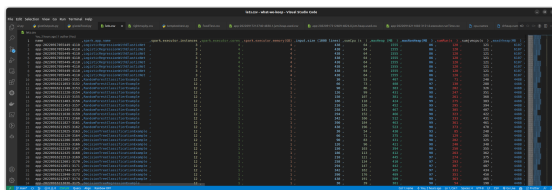
After the configuration was complete the servers lifted their sleeves and started working and as seen below working they did.



E. Collected Measurements

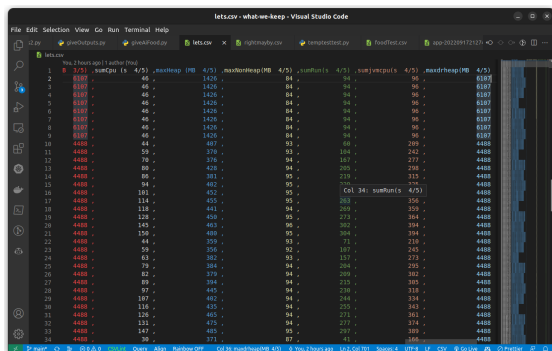
The default naming convention for spark is `app-<timestamp>-<sequential number>`. So

based on our last run we have 7,664 entries. It turns out that was a lot and we lost some data from our measurements due to some rookie mistakes in storage management. All the graphs on this paper are from the entries 3100 to 4900. Using the Spark build-in CSV reporting tool we had all our measurements on the master node, this filled the disk and some of the tests were not saved. When we collected all the CSV files and we decided on what metrics we thought were important we had the following file for the AI models. This file has one line for each entry and 3 categories of columns. The first is the input parameters, the second is the final metrics we chose, and the last is the partial metrics up to n fifths of the run, for n between 1 and 4.



The screenshot shows a CSV file with multiple columns. The first column is an index from 1 to 34. The next column is 'input_size'. The following columns are grouped by 'n' (1, 2, 3, 4) and contain metrics like 'sumRun', 'sumjvmCPU', 'maxHeap', 'maxNonHeap', 'sumCPU', and 'maxHeap'. The values are numerical and vary across the rows.

So when we train the AI models to predict the result we can give it different subsets of the columns to simulate partial completion of the run.

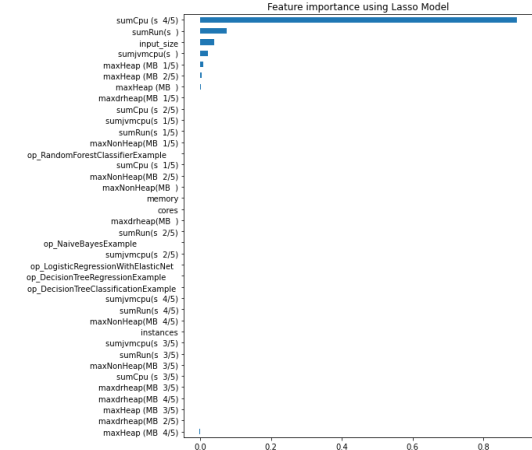
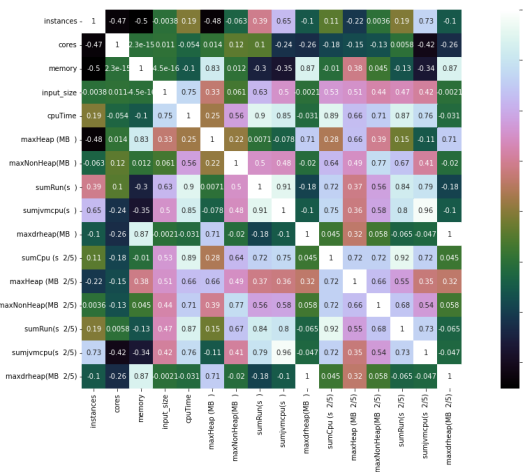
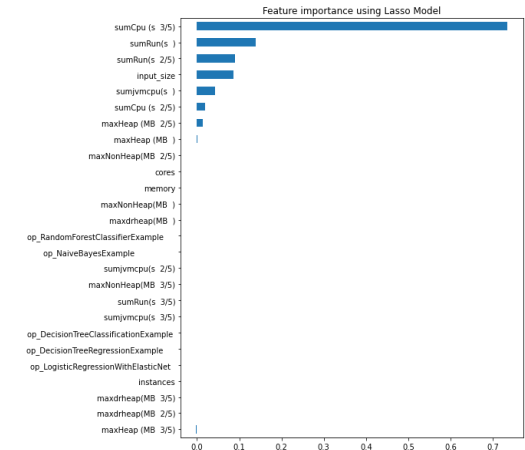
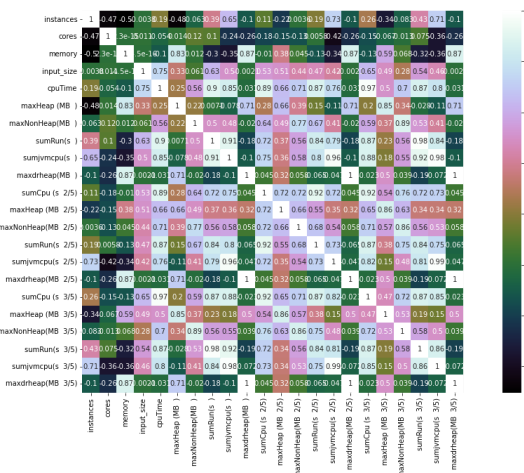
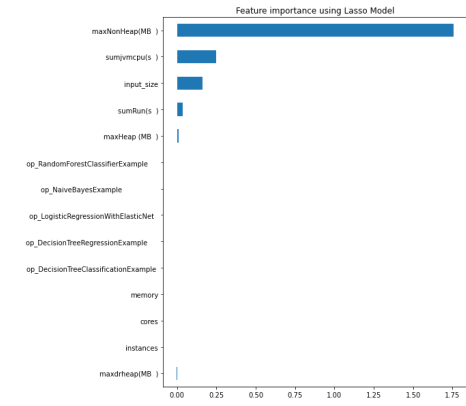
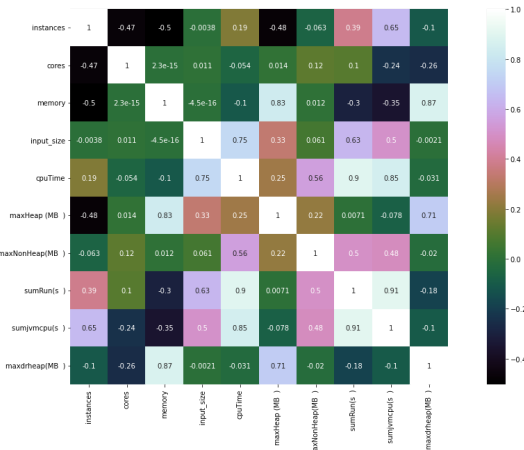


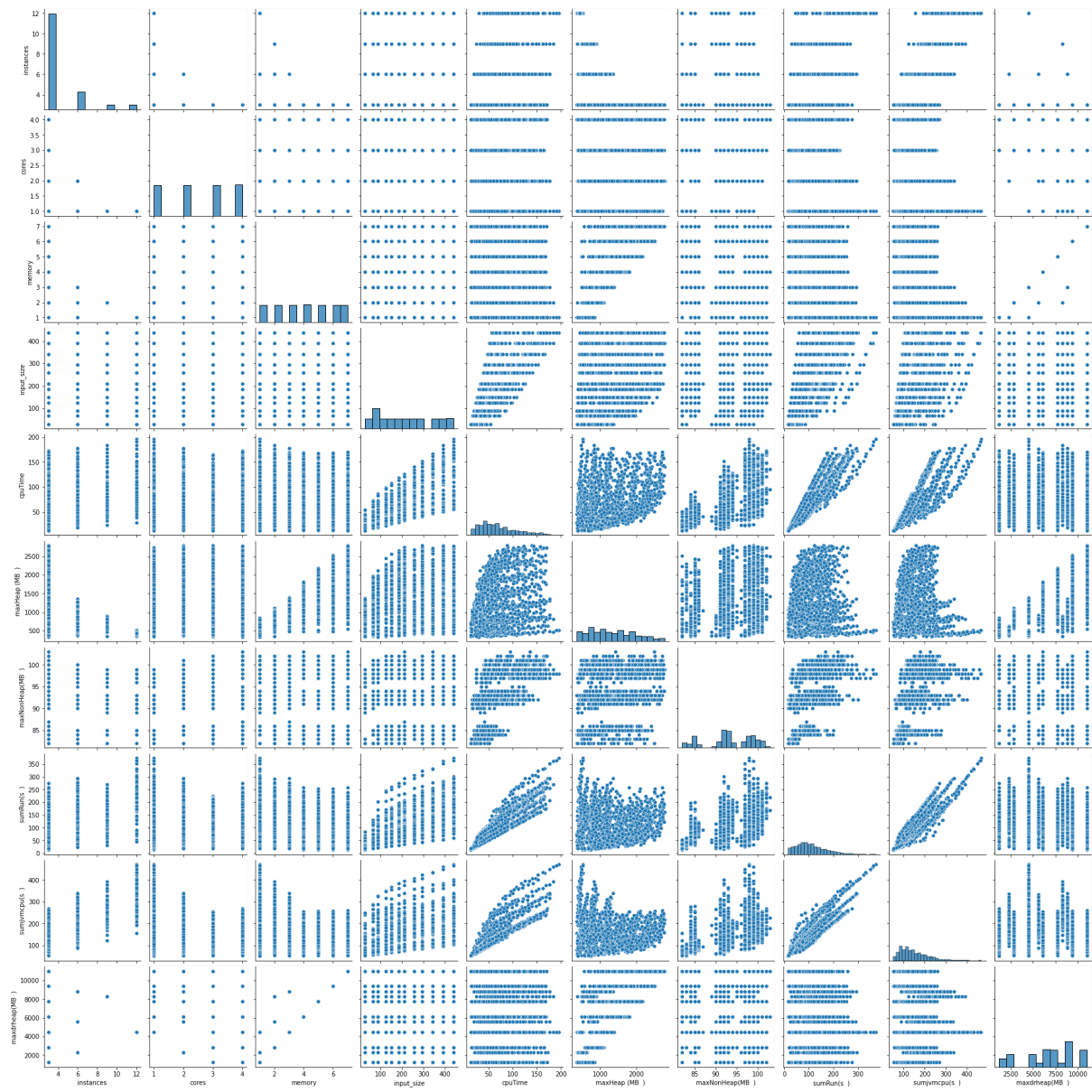
The screenshot shows a CSV file with multiple columns. The first column is an index from 1 to 34. The next column is 'input_size'. The following columns are grouped by 'n' (1, 2, 3, 4) and contain metrics like 'sumRun', 'sumjvmCPU', 'maxHeap', 'maxNonHeap', 'sumCPU', and 'maxHeap'. The values are numerical and vary across the rows.

III. Exploratory Data Analysis

As described in the previous section, we collected the input parameters and the metrics of the ML operators runtime from Spark metrics in order to create a dataset for the regressors training. We perform exploratory data analysis to receive insights from the extracted data and to inspect their usefulness. In the figures below, we present the correlation matrices and the results of LassoCV for feature selection of the original dataset and two other versions. The other versions differ in the number of features. The first one is has 11, the second has 23 and the third (original) has 35 including the target. We tested various versions of the dataset to observe the predictive performance of the regression models and we chose these 3 as the best examples.

In the three different datasets, a different feature is the most valuable. However, as it is expected the highest importance are have the metrics 'sumRun' (total run time), 'sumjvmCPU' (total time of jvm machine) , 'maxHeap', 'maxNonHeap', sumCPU of instances and the input parameter of 'input_size'.





IV. Machine Learning Models

The Spark features and the collected metrics are used to train a machine learning model (regressor algorithm) that later will be used for predicting the execution time of unseen ML computing events. The functionality of the statistical machine learning models is to derive the relationship between the set of output metrics (independent variables) and input features (dependent variable). In our study, we choose to focus on the prediction of mainly execution times but also memory usage. Moreover, the model has to be able to generalise for different datasets, using the same learning algorithm under the same computational environment.

As it has been shown in the literature that regression models have been successful in the prediction of execution time of various applications. In this section of the paper, we investigate the effectiveness of seven regression models :

- Linear Regression
- Linear Regression with Regularisation
- Polynomial Regression
- Lasso Regression with Regularisation
- Lasso Regression with Polynomial Regularisation
- Ridge Regression with Polynomial Regularisation
- Gradient Boosting Regression

We chose a variety of regression models from the simplest to more complex ones in order to have a wide range of models to compare and extract conclusions from. Furthermore, they have been shown to be efficient in predicting execution times in previous works. Some honourable mentions are the Polynomial Regression models with Lasso (SPORSE) and the Boosting algorithms [3],[4],[6],[8]. The training of the learning algorithms involve the dataset, which was created from as described in the previous Section. Before continuing with the training of the models, we made sure

to clean the data of null values, duplicates or other outliers that would throw off the training process. In the generated dataset, there is only one categorical feature that of the ML algorithm name.

One-Hot encoding

For our models to be able to understand the different categories, we use the One-Hot encoding technique to convert each categorical value into a new categorical column and assign a binary value of 1 or 0 to those columns. Thus, five new columns are created.

Data Augmentation - Smogn

The size of our dataset is quite small and we are not fully capable of knowing or understanding the correlation between the features collected. A common technique to ease the problem of overfitting is data augmentation. However, it is especially difficult to perform augmentation on numerical data for regression as clear boundaries between data do not exist, as in classification. To address this problem we used the python library smogn [9],[10]. Smogn is a Python implementation of Synthetic Minority Over-Sampling Technique for Regression with Gaussian Noise (SMOGEN). It conducts the Synthetic Minority Over-Sampling Technique for Regression (SMOTER) with traditional interpolation, as well as with the introduction of Gaussian Noise (SMOTER-GN). It can perform oversampling and undersampling of the minority or majority "classes". Contrary to our prior assumptions in our case, the extension downsized our dataset, probably due to low relevance of the data. We observed less overfitting and better accuracy metrics while using it.

K-fold Cross Validation

Once the model is created using the data, it is necessary to produce evidence that the regression models are able to generate accurate predictions. An approach for this is to use Kfold Cross Validation. Generally, Cross Validation is a resampling method that uses different portions of the data to test and train a model on different iterations. For k-fold CV, the training set is split into k smaller sets. The performance measure reported by k-fold cross-validation is the average of the values computed in the iterations. We chose not to perform a more complex CV method like Grid Search as it will be too time and cost consuming for seven regression models. Taking into consideration that for the purpose of predicting the performance of a ML program while it's still running, we will run 6 different versions of our dataset. This decision was made to deepen our understanding of which parameters and techniques lead to better results. It is important to mention that we tested multiple sees numbers and we observed similar performances. We don't expand on that aspect of the training phase as it does not provide further insights.

Optimization of regression model parameters

All models were optimised to perform to the best of their ability over the specified datasets. The optimization of some regressors like Ridge or GBR would be too time and resources consuming as the bare multiple parameters. For the sake of focusing equally on all predictive models, we did not continue with an in depth hyper-parameterization.

V. Experiments

A. Experimental Setup

All experiments regarding the training of the AI models were performed

- **Datasets Versions**

To predict the execution time or the memory usage of one ML operator, we created multiple experiments that will differ in the dataset being used

1. First, we train the regressors on the whole dataset which includes the input parameters of Spark configuration and the collected metrics. The metrics collect five instances of 6 features (sumCPU, maxHeap, maxNonHeap, sumRun, sumjvmCPU, maxdrheap).
2. Then, we train the models using the input parameters and the total (5/5) values of these 6 features.
3. Next, we try training including only the first instance of the features (1/5)
4. We repeat that for the rest of the instances.

After running these experiments, we chose the best dataset to continue with further comparison of the regression models.

- **Evaluation Metrics**

To measure the accuracy of our prediction models, we use the R-Squared R^2 , Adjusted R-Squared R^2 , Mean Squared Error (MSE), Root Mean Squared Error (RMSE) and the Mean Absolute Error (MAE). R^2 is calculated as follows :

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

where y_i is the actual measured execution time, \hat{y}_i is the predicted

execution time, and \bar{y}_i is the mean of all actual execution times. The Adjusted R^2 metric which adjusts the number of predictors relative to the number of samples can be calculated

$$\text{Adjusted } R^2 = 1 - \frac{(n-1)}{(n-p-1)} * (1 -$$

where p is the number of features in the model and n is the number of samples. The closer value to 1 indicates a better fit of model. The mean square error (MSE), when closer to zero indicates a better accuracy. The mean absolute error follows a similar pattern. However, it is more robust to

outliers. It is important to be mentioned that in our code, we use the implementations of the library sklearn in which the metrics are found as follows

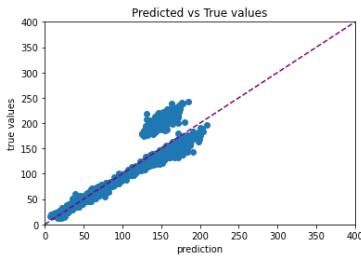
`'neg_mean_squared_error', 'r2', 'neg_root_mean_squared_error', 'neg_mean_absolute_error'`. The `neg` in the prefix means that the unified API returns the negative values of the scores. In detail, the unified scoring API always maximises the score, so scores which need to be minimised are negated in order for the unified scoring API to work correctly. The score that is returned is therefore negated when it is a score that should be minimised and left positive if it is a score that should be maximised.

B. Experimental Results

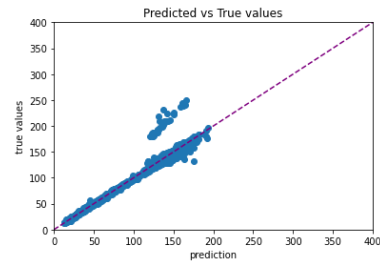
We test the prediction accuracy of models by running the experiments described below. The shown results are mainly about the prediction of execution time. However, we have implemented the code for the prediction of the other metrics as well.

Case 1 : Dataset with 35 features

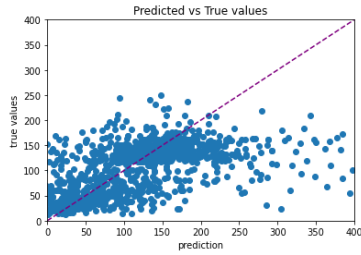
In the first experiment, we trained the 7 regression models using the fully-sized, extracted dataset of Section 3 which consists of 35 features. The features include 5 input parameters and values of 5 instances of 6 metrics. The idea behind this selection of data is that by including different values of the metrics at specific times ($\frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}$ and total time of experiment duration), the model will be able to find unique correlations of between the duration of execution time duration or memory usage and the appointed time stamps. In Fig. 5.1 we plot the predicted vs the actual execution times for each regression model and in Table 5.1 we showcase the evaluation metrics.



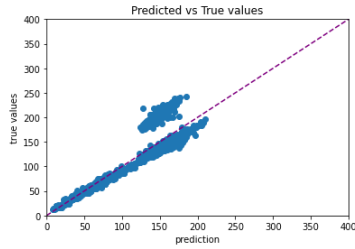
(a) Linear Regression



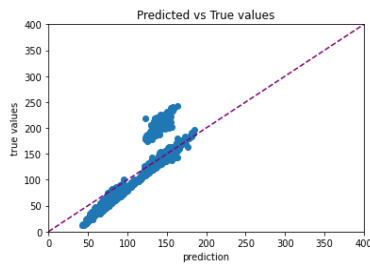
(b) Linear Regression with Regularisation



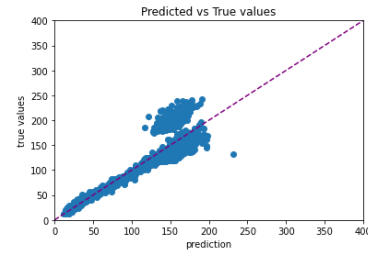
(c) Polynomial Regression



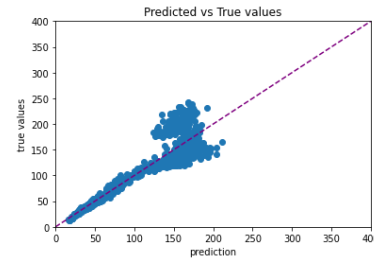
(d) Lasso Regression with Regularisation



(e) Lasso Regression with Polynomial Regularisation



(f) Ridge Regression



(h) Gradient Boosting Regression

Figure 5.1 : Predicted vs actual execution time of Case 1

	Linear	Linear Reg	Poly	Lasso Reg	Lasso Poly	Ridge	GBR
r^2	0.8256	0.8246	-2.5906	0.8239	0.826	0.8272	0.8325
Adj_ r^2	0.8208	0.8198	-2.6885	0.819	0.8212	0.8224	0.8279
mse	-550.6987	-552.2092	-7179.8169	-556.7915	-549.7801	-545.0602	-527.4602
rmse	-23.3557	-23.3854	-50.5377	-23.4792	-23.3241	-23.2526	-22.8725
mae	-16.3871	-16.4774	-50.5377	-16.3257	-16.325	-15.8405	-15.6366

Table 5.1 : Evaluation metrics of Case 1

The performance of the models is similar with accuracy of 82%. The only exception is the Polynomial regression model which due to the big number of features overfits. All models are presented with high error, so we can conclude that the selected features are not a good fit for training.

Case 2 : Dataset with 11 features

In the second experiment, we trained the 7 regression models using the 5 input parameters and the 6 metrics of the final instances (total time). In Fig. 5.2 we plot the predicted vs the actual execution times for each regression model and in Table 5.2 we showcase the evaluation metrics.

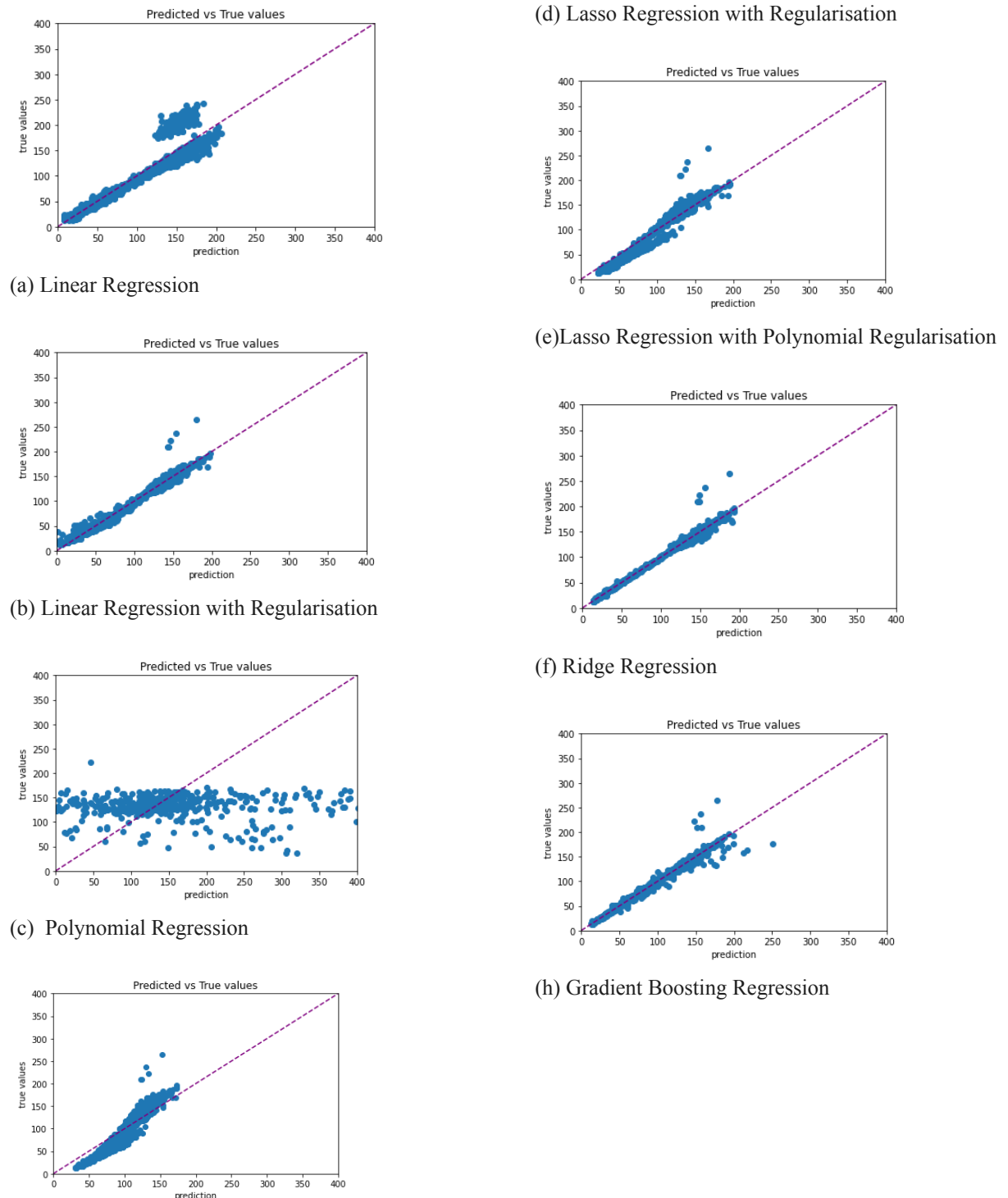


Figure 5.2 : Predicted vs actual execution time of Case 2

	Linear	Linear Reg	with Poly	Lasso Reg	Lasso Poly	Ridge	GBR
--	--------	------------	-----------	-----------	------------	-------	-----

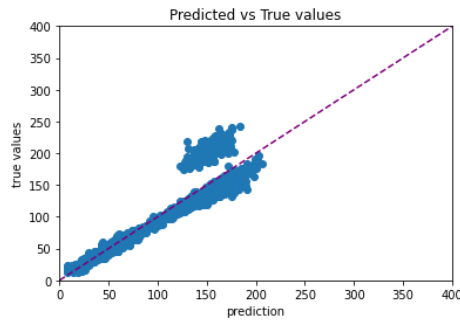
r^2	0.9747	0.9747	0.9846	0.9711	0.9256	0.9871	0.9797
Adj_ r^2	0.9745	0.9745	0.9844	0.9708	0.9248	0.9869	0.9794
mse	-54.1468	-54.1498	-33.4912	-61.4099	-157.1267	-28.2795	-42.9917
rmse	-7.1423	-7.1424	-5.4284	-7.6527	-12.4427	-4.8664	-6.1694
mae	-4.5412	-4.5418	-2.8083	-5.3167	-9.7189	-2.3333	-2.8395

Table 5.2 : Evaluation metrics of Case 2

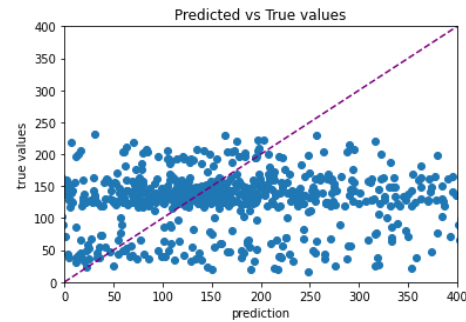
The performance of the models is similar with accuracy of 97%. The only exception is the Lasso regression with Polynomial Regularisation model. There is an obvious decrease in error in this experiment in contrast to Case 1. The Polynomial regression model performs optimally with the present small dataset.

Case 3 : Dataset with 17 features

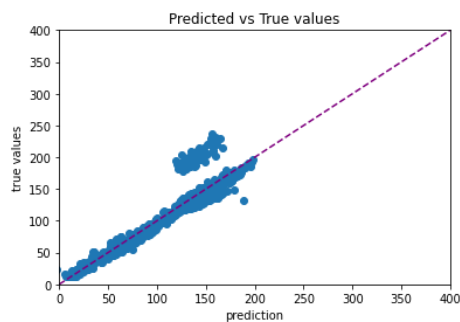
In the third experiment, we trained the 7 regression models using the 5 input parameters and the 6 metrics of the final instances (total time) and the instances of the first quarter. In Fig. 5.3 we plot the predicted vs the actual execution times for each regression model and in Table 5.3 we showcase the evaluation metrics.



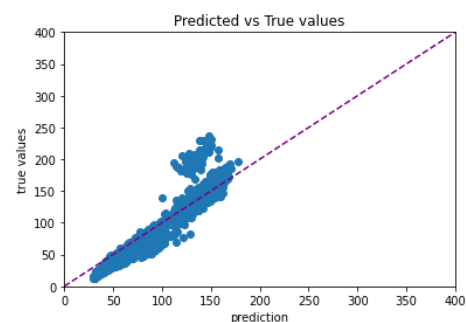
(a) Linear Regression



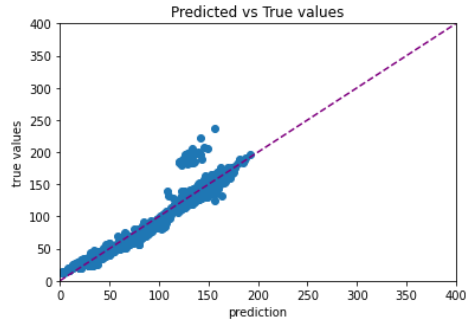
(c) Polynomial Regression



(b) Linear Regression with Regularisation

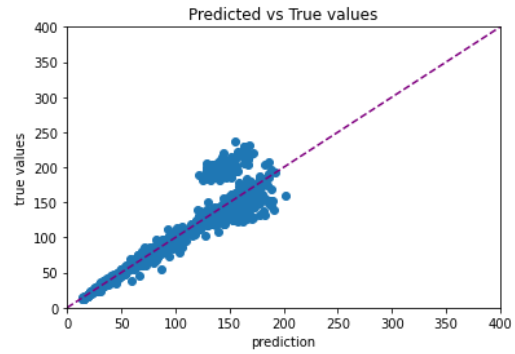


(d) Lasso Regression with Regularisation



(e) Lasso Regression with Polynomial Regularisation

(f) Ridge Regression



(h) Gradient Boosting Regression

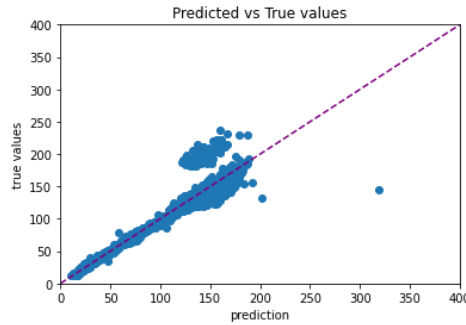


Figure 5.3 : Predicted vs actual execution time of Case 3

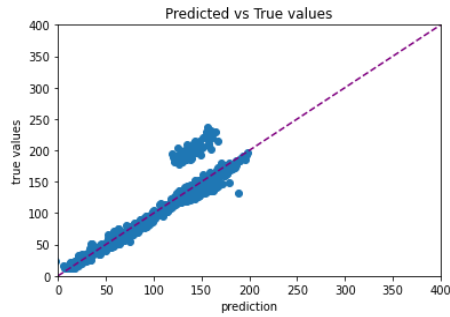
	Linear	Linear Reg	with Poly	Lasso Reg	Lasso Poly	Ridge	GBR
r^2	0.8899	0.8898	-2.8628	0.8855	0.8687	0.8889	0.8879
Adj_r^2	0.8883	0.8883	-2.9034	0.8838	0.8668	0.8873	0.8863
mse	-281.6894	-281.7729	86978728053	-293.3001	-336.4636	-285.0084	-284.9554
rmse	-16.6693	-16.6723	294921.56	-17.0044	-18.2055	-16.7008	-16.8044
mae	-9.0181	-9.0263	-16.6723	-9.5617	-10.4569	-8.7046	-9.3589

Table 5.3 : Evaluation metrics of Case 3

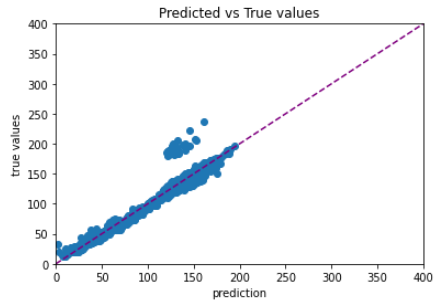
The performance of the models is close to 88%. We increased the number of features which led to overfitting. The values of the first quarter do not seem to provide enough variance to help the models' prediction. There is an increase in error in this experiment compared to Case 2. The Polynomial regression model displays commensurable behaviour with Case 1, as it has high error value.

Case 4 : Dataset with 17 features

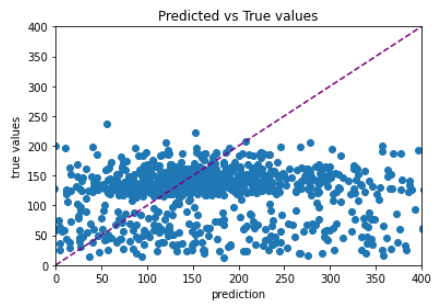
In the fourth experiment, we trained the 7 regression models using the 5 input parameters and the 6 metrics of the final instances (total time) and the instances of the second quarter. In Fig. 5.4 we plot the predicted vs the actual execution times for each regression model and in Table 5.4 we showcase the evaluation metrics.



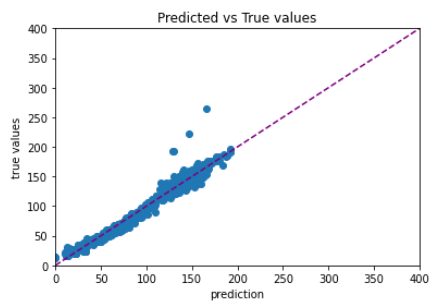
(a) Linear Regression



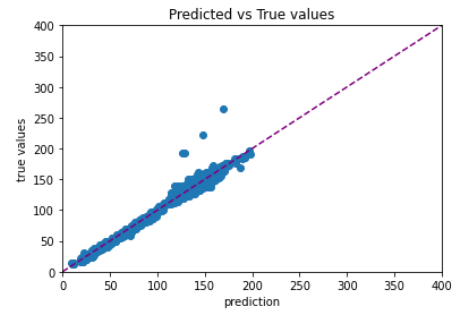
(b) Linear Regression with Regularisation



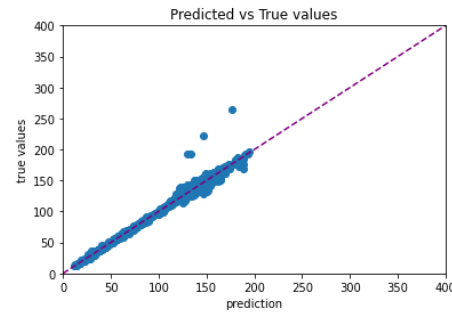
(c) Polynomial Regression



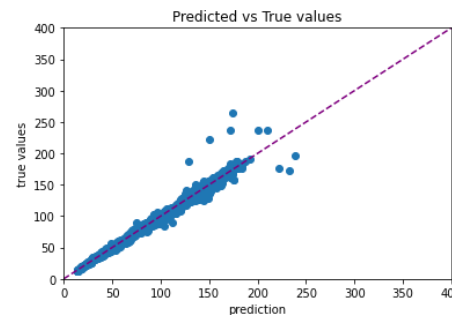
(d) Lasso Regression with Regularisation



(e) Lasso Regression with Polynomial Regularisation



(f) Ridge Regression



(h) Gradient Boosting Regression

Figure 5.4 : Predicted vs actual execution time of Case 4

	Linear	Linear Reg with Poly	Lasso Reg	Lasso Poly	Ridge	GBR	
r^2	0.9304	0.9304	-8.2086	0.9242	0.8932	0.9378	0.9265
Adj_r^2	0.9294	0.9294	-8.325	0.9231	0.8916	0.9368	0.9254
mse	-161.6265	-161.7225	-1.79E+21	-176.5086	-248.106	-144.5947	-169.6935

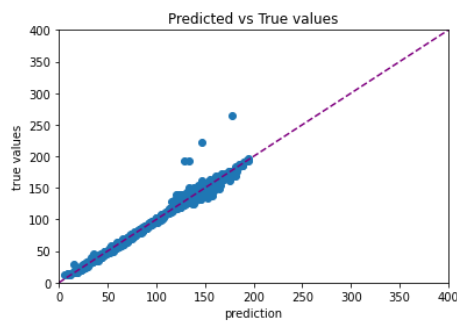
rmse	-12.5894	-12.5932	-1339447018 3	-13.1544	-15.6477	-11.9144	-12.9285
mae	-6.2628	-6.2629	-1416829250	-7.2932	-9.9176	-5.7488	-6.5595

Table 5.4 : Evaluation metrics of Case 4

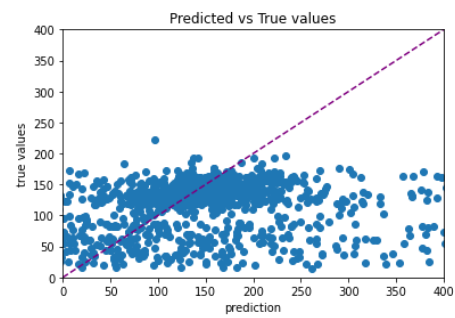
Most of the models show accuracy close to 93%. We increased the number of features which led to overfitting in the case of Polynomial regression. The errors of the rest regressors decreased in comparison to Case 3 for which we can derive that the values of the second quarter are more relevant to the prediction of our target.

Case 5 : Dataset with 17 features

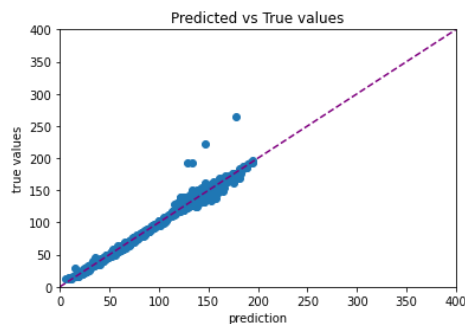
In the fifth experiment, we trained the 7 regression models using the 5 input parameters and the 6 metrics of the final instances (total time) and the instances of the third quarter. In Fig. 5.5 we plot the predicted vs the actual execution times for each regression model and in Table 5.5 we showcase the evaluation metrics.



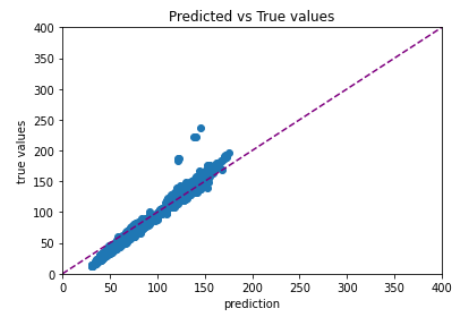
(a) Linear Regression



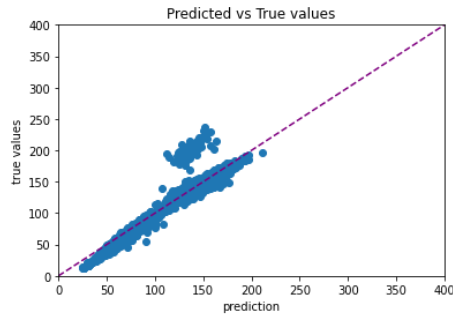
(c) Polynomial Regression



(b) Linear Regression with Regularisation

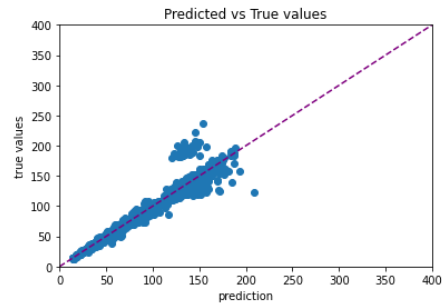


(d) Lasso Regression with Regularisation



(e) Lasso Regression with Polynomial Regularisation

(f) Ridge Regression



(h) Gradient Boosting Regression

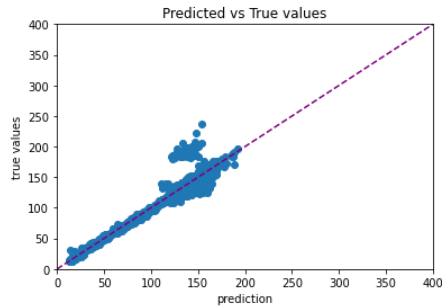


Figure 5.5 : Predicted vs actual execution time of Case 5

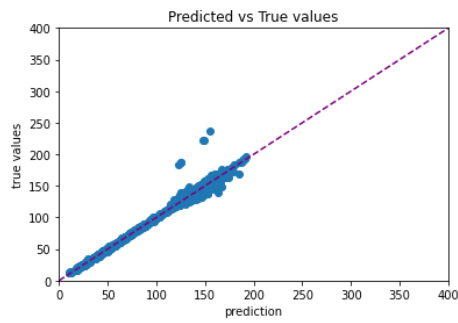
	Linear	Linear Reg with Poly	Lasso Reg	Lasso Poly	Ridge	GBR
r^2	0.9864	0.9865	0.9852	0.9766	0.9826	0.9888
Adj_r^2	0.9863	0.9863	-1.0133	0.9762	0.9823	0.9886
mse	-28.7219	-28.7221	-31.2081	-49.0842	-36.6035	-31.4768
rmse	-5.0611	-5.0607	-5.3138	-6.8224	-5.7782	-5.3119
mae	-2.7871	-2.7833	-2.9753	-4.5941	-3.6846	-2.2894

Table 5.5 : Evaluation metrics of Case 5

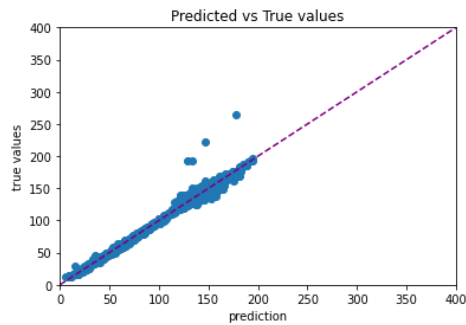
All seven regressors show accuracy close to 98%. Even the Polynomial regression model fits without overly overfitting. In comparison to the previous cases, we observe low errors coupled with high accuracy values while increasing the feature size, so the features selected were beneficial to the training process.

Case 6 : Dataset with 23 features

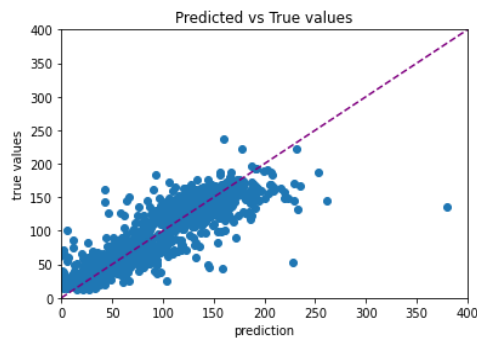
In the sixth experiment, we trained the 7 regression models using the 5 input parameters and the 6 metrics of the final instances (total time), the instances of the second and third quarters. In Fig. 5.6 we plot the predicted vs the actual execution times for each regression model and in Table 5.6 we showcase the evaluation metrics.



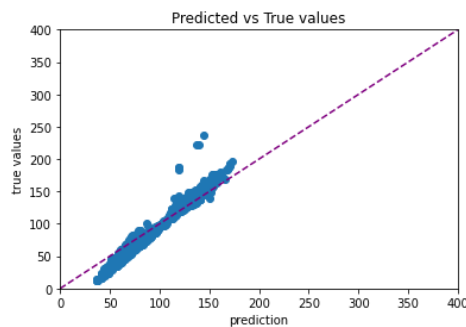
(a) Linear Regression



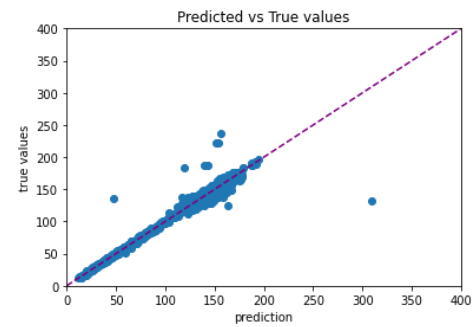
(b) Linear Regression with Regularisation



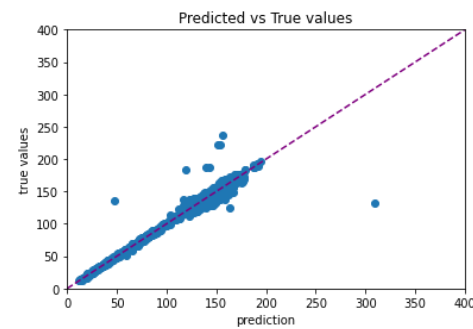
(c) Polynomial Regression



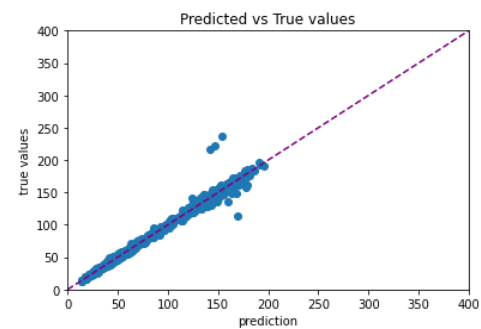
(d) Lasso Regression with Regularisation



(e) Lasso Regression with Polynomial Regularisation



(f) Ridge Regression



(h) Gradient Boosting Regression

Figure 5.5 : Predicted vs actual execution time of Case 5

	Linear	Linear Reg	Poly	Lasso Reg	Lasso Poly	Ridge	GBR
r^2	0.9848	0.9838	0.9826	0.9829	0.9808	0.9747	0.9823

Adj_r^2	0.9843	0.9835	0.9822	0.9824	0.9802	0.974	0.9819
mse	-31.49	-35.0862	-37.5976	-35.33	-39.6565	-52.188	-38.1637
rmse	-5.254	-5.6075	-5.8386	-5.6278	-6.038	-6.652	-5.9034
mae	-2.4704	-2.8807	-3.3056	-3.1291	-3.6959	-2.9213	-3.4036

Table 5.5 : Evaluation metrics of Case 5

All seven regressors show accuracy close to 98%. Even the Polynomial regression model fits without overly overfitting. In comparison to the previous cases, we observe low errors coupled with high accuracy values while increasing the feature size, so the features selected were beneficial to the training process.

After experimenting with other combinations of the time instances (of the metrics), we came to the conclusion to limit our experiments to the six above, as they delivered the best result.

C. Algorithm Comparison

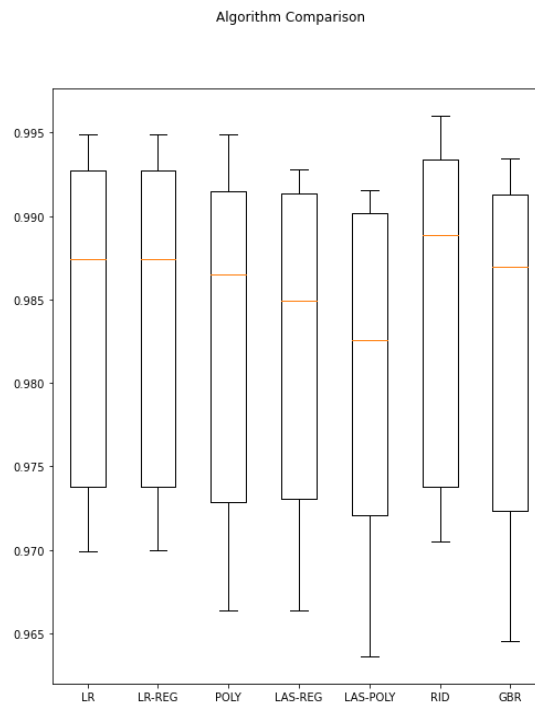


Figure 5.6 : Algorithm Comparison graph for Case 1

The above plot showcases the comparison between the accuracy evaluation metrics for the dataset of Case 1. Due to the extremely low value of Polynomial regression (POLY: -847884038.411777) , it was removed from the graph.

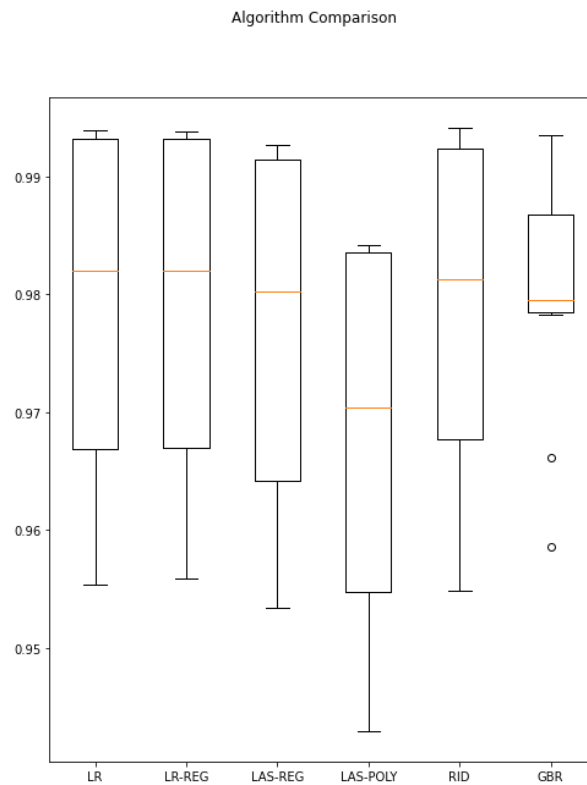


Figure 5.7 : Algorithm Comparison graph for Case 2

The above plot showcases the comparison between the accuracy evaluation metrics for the dataset of Case 2. As the data consist of few features, simpler models seem to perform better.

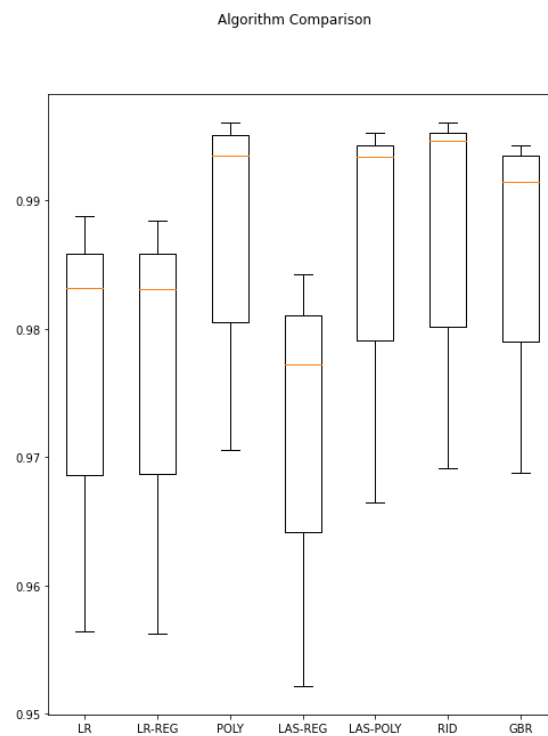


Figure 5.8 : Algorithm Comparison graph for Case 6

The above plot showcases the comparison between the accuracy evaluation metrics for the dataset of Case 6. As the data consist of more features, more complex models seem to perform better.

VI. CONCLUSION

In this paper we explored multiple ways to predict the computational resources required for executing some of the most common machine learning and artificial intelligence tasks using Apache Spark on a small cluster of servers. With the dataset we achieved on the one hand great accuracy, in most cases better than 0.95, but on the other hand it was to the detriment of precision. With additional time and resources we can collect more data for a more extensive, accurate and diverse dataset including more operators and tighter control of Spark's use. Also we could make it respond in real time taking the metrics directly from spark and giving the expected results to the user with a CLI tool.

REFERENCES

- [1] Dirk Tetzlaff, Sabine Glesner, "Intelligent Prediction of Execution Times", ISBN: 978-1-4673-5256-7/13/
- [2] Sara Mustafa *, Iman Elghandour, Mohamed A. Ismail, "A Machine Learning Approach for Predicting Execution Time of Spark Jobs", Alexandria Engineering Journal (2018) 57, 3767–3778
- [3] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, Mayur Naik, "Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression", "Advances in Neural Information Processing Systems", pp. 883-891 ,
- [4] Rattan Priyaa, Bruno Feres de Souzaab, Andre L.D. Rossi ´ b and Andre C.P.L.F. de Carvalho ´ b, "Predicting execution time of machine learning tasks for scheduling", International Journal of Hybrid Intelligent Systems 10 (2013) 23–32
- [5] Nhan Nguyen, Mohammad Maifi Hasan Khan, Yusuf Albayram, Kewen Wang, "Understanding the Influence of Configuration Settings: An Execution Model-driven Framework for Apache Spark Platform", 2017 IEEE 10th International Conference on Cloud Computing
- [6] Lu Dong, Peng Li, He Xu, Baozhou Luo, Yu Mi, "Performance Prediction of Spark Based on the Multiple Linear Regression Analysis", Springer Nature Singapore Pte Ltd. 2017, G. Chen et al. (Eds.): PAAP 2017, CCIS 729, pp. 70–81, 2017
- [7] Machine Learning Library (MLlib) Guide, <https://spark.apache.org/docs/latest/ml-guide.html>
- [8] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, Mayur Naik, "Predicting execution time of computer programs using sparse polynomial regression", Advances in neural information processing systems, Vol. 23
- [9] Torgo, Luís & Ribeiro, Rita & Pfahringer, Bernhard & Branco, Paula. (2013). SMOTE for Regression. 8154. 378-389. 10.1007/978-3-642-40669-0_33.
- [10] Synthetic Minority Over-Sampling Technique for Regression with Gaussian Noise, <https://github.com/nickkunz/smogn>
- [11] SparkMeasure is a tool for performance troubleshooting of Apache Spark jobs,

<https://github.com/LucaCanali/sparkMeasure#sparkmeasure>

- [12] Performance Analysis of a CPU-Intensive Workload in Apache Spark,
<https://db-blog.web.cern.ch/blog/luca-canali/2017-09-performance-analysis-CPU-intensive-workload-apache-spark>