Analysis and Design of Information Systems

Comparison between Python scaling frameworks for big data analysis and ML

Dimitris Kokkinis, el18896, Spyridon Michail Evangelou el18026, Ignatios Siminis el19821

Abstract—This paper conducts a focused comparative analysis on the scalability and performance of *Ray*, a leading Python scaling framework, in contrast to *Apache Spark*, within the domain of big data analytics and machine learning. Utilizing a multi-node *Ray* cluster, we explore critical aspects such as installation, setup, and performance across diverse cluster configurations. Our study encompasses key operations, including Clustering, Regression, Classification and PageRank. Through meticulous experimentation, we aim to unveil *Ray*'s distinctive features and limitations compared to *Apache Spark* providing valuable insights.

Index Terms—Ray, Apache Spark, big data analytics, machine learning, scalability, performance evaluation.

1 Introduction

In the contemporary landscape of big data analytics and machine learning, the efficiency and scalability of computing frameworks play a pivotal role in extracting insights from vast and complex datasets. *Ray*, an open-source unified computing framework, has emerged as a powerful solution for scaling Data Analytics, Machine Learning, and Artificial Intelligence workloads in Python. With the advent of *Ray* Datasets, a high-performance data processing API catering to large-scale datasets, *Ray* offers unique advantages for optimizing the workflow of data scientists and analysts.

This assignment centers around a comparative exploration of *Ray* with other modern Python scaling frameworks, specifically *Apache Spark*, within the realm of big data analysis and machine learning. The overarching goal is to assess the relative strengths and weaknesses of *Ray* when compared to *Apache Spark*, shedding light on its capabilities in handling diverse analytics and machine learning workloads.

The paper unfolds in three main dimensions, each crucial for a comprehensive evaluation:

1) **Installation and Setup**: Undertaking the installation and setup of both *Ray* and *Apache Spark* on AWS. This ensures a consistent environment for subsequent evaluations.
2) **Data Generation**: The assignment mandates the identification and loading of substantial datasets into both systems, encompassing various characteristics such as size, memory requirements, and the number of records.
3) **Experiment Design and Performance Measurement**: Central to the assessment is the development of experiments targeting diverse machine learning and big data operations. By conducting these experiments across different numbers of nodes/workers and varying input data sizes/types, the assignment delves into the scalability and efficiency of *Ray* and *Apache Spark*. This stage aims to quantify their relative strengths and weaknesses in executing common operations and complex tasks.

The ultimate objective is to provide valuable insights into how *Ray* fares against *Apache Spark*, in the context of scaling analytics code. The subsequent sections will elucidate the methodologies employed in the installation and setup, data generation and loading, and performance measurement, leading to a comprehensive evaluation of the relative merits of *Ray* and *Apache Spark* in the domain of big data analysis and machine learning.

The repo at GitHub repo contains the code developed for the purpose of executing all the steps of this study.

2 Description and Analysis of the System Setup

The topology of the interconnected machines consists of one master and two workers all in the same subnet; the goal was the creation of a shared computational and storage cluster that could perform complex operations on big data. To do so, the *Apache Hadoop*, *Spark* and *Ray* software were utilized, and the details regarding their installations are as follows:

2.1 Initial Setup

The following prerequisites were required for the installation of the aforementioned software:

- Python 3.10: The programming language most of the scripts were developed in
- java-8-openjdk-amd64: The Java JDK used
- Scala-2.12: The programming language that PageRank algorithm for *Spark* was developed with

In order for the VMs to be able to communicate with each other, they were placed in the same subnet: 172.31.16.0/20.

In addition, generated ssh keys were used for each machine, and the /etc/hosts/ file of each VM was modified by including the following:

IP address of master master

IP address of worker1 slave1

IP address of worker2 slave2

Via this manner, the machines could communicate by using their hostnames, which was crucial for *Spark* and Hadoop to work.

2.2 Apache Spark

The setup for *Apache Spark* cluster was done according to the following steps:

1) Download of Spark from the official *Apache Spark* Server[1] (in this case spark-3.5.0-bin-hadoop3.tgz)
2) Decompression of the tar file into /opt/spark
3) Editing the /.bashrc file, appending the appropriate environment variables for Spark to work
4) Copying the ssh public key from master to all VMs with ssh-copy-id, allowing for passwordless ssh login
5) Installation of PySpark with pip in order to develop python programs that work with Spark
6) Installation of the sdk from SDKman[1] and installation of sbt- the build tool used to build the scala PageRank project
7) Launch of Spark with the command start-all.sh

Note: The configuration (memory, cores) of each worker was adjusted by editing the file start-workers.sh of the master, and the spark-defaults.conf file of every worker (including the master). To add the VMs slave1 and slave2 as workers,the workers file was edited.

2.3 Apache Hadoop

The setup for *Apache Hadoop* cluster was done according to the following steps:

1) Creation of the user hdoop on each VM
2) Download of Hadoop from the official *Apache Hadoop Server*[3] (in this case hadoop-3.3.6.tar.gz)
3) Decompression of the tar file into /usr/local/hadoop
4) Change of the owner of the directory to hdoop using the command chown
5) Configuration of the Hadoop environment by setting the variable `JAVA_HOME` to hadoop-env.sh
6) Editing of the configuration files (hdfs-site.xml, core-site.xml, mapred-site.xml), as to specify the directories of namenode and datanodes (in this case /usr/local/`hadoop_space`/hdfs/namenode,datanode repsectively)
7) Formatting of the namenode
8) Copying the ssh public key from master (for user hdoop) to all VMs with ssh-copy-id to enable passwordless ssh login
9) Startup of Hadoop HDFS with the command start-all.sh

Note: The VMs slave1 and slave2 were added as datanodes by editing the workers file.

2.4 Ray Framework

Ray was installed as described below:

1) Usage of the command pip install -U "ray[data,train,tune,serve]"[4]
2) Editing of /.bashrc PATH variable to include /home/ubuntu/.local/bin/ in order for the *Ray* binary to be readily accessible
3) Starting *Ray* on all machines with ray start

2.5 Additional installations

The experiments that will be mentioned in this report required additional Python libraries/scala packages to be installed in the VMs:

- *Python*
  1) **scikit-learn**: A key Machine Learning library, providing a wide variety of ML models and evaluators for those models, and further utilized by the *Spark* and *Ray* frameworks?. It aided in the creation of datasets for Regression, Classification and Clustering.
  2) **xgboost-ray**: This library was needed for the Regression and Classification experiments with *Ray*.
  3) **networkx**: A Python library used to generate Graphs from data and perform operations on them.
  4) **fastparquet**: A library used to write the generated datasets to parquet files.
- *Scala*
  1) **graphframes**: A scala package that was used to perform graph operations including PageRank.

3 Infrastructure and Software description

3.1 Infrastructure

For the purposes of this project, three laptops were used. The specifications of each laptop are detailed below:

Laptop 1: Windows 11, 8 GB RAM, 6 cores

Laptop 2: Windows 10, 8 GB RAM, 6 cores

Laptop 3: Windows 11, 16 GB RAM, 8 cores

After much effort, it was decided that other machines were in need, therefore a different path was opted for. The initial machines were dealing with a lot of issues, causing the results to be merely factual and the genuine time results of the machine learning tasks to be obscured. To the best of our knowledge, the only solution was to perform the metrics on other machines with the same potential. As a result, virtual machines provided by Amazon Web Servises (AWS) were selected, with the below specifications:

The same configuration was used on all 3 VMs: Ubuntu Jammy 22.04 AMD64 Server – 16GB RAM – 4vCPUs (2 cores)

3.2 Software

Moving forward, the frameworks to be compared on ML tasks, were *Apache Spark* and *Ray*.

Apache Spark

Apache Spark is an open-source, distributed processing system used for big data workloads. It utilizes in-memory caching, and optimized query execution for fast analytic queries against data of any size. It provides development APIs in Java, Scala, Python and R, and supports code reuse across multiple workloads—batch processing, interactive queries, real-time analytics, machine learning, and graph processing. It was created to address the probelm that Hadoop MapReduce faced of the latency of I/O operations, by:

- In-memory processing

- Reducing the number of steps
- Data reuse across multiple parallel operations

Spark reuses data by utilizing an in-memory cache so as to accelerate machine learning tasks that repeatedly call a function of the same dataset. This is plausible with dataframes, an abstraction of Resilient Distributed Dataset (RDD), which is a collection of which is a collection of objects that is cached in memory, and reused in multiple *Spark* operations.

Ray

Ray is an open-source framework designed to enable the development of scalable and distributed applications in Python. It provides a simple and flexible programming model for building distributed systems, making it easier to leverage the power of parallel and distributed computing. Some key features and capabilities of the *Ray* framework include:

- Task parallelism
- Distributed computing
- Remote function execution
- Distributed data processing
- Reinforcement learning support

At first, one our laptops was using an Ubuntu Linux virtual machine with 8 GB of memory and 4 cores. The other two were using Windows operating system, so we had to set up *Spark* and *Ray* in Windows at first.

After a few tests, we noticed that the difference between the operating systems may interfere with the actual performance of the frameworks. Consequently, we had to test our theory by using Ubuntu Linux virtual machines for the other two as well, with 4GB of memory and 4 cores.

Although, the results seemed to improve, there was an underlying issue with one of our machines that kept undermining the overall progress.

As our access in *Okeanos* was permanently denied, external virtual machines was our last stronghold. Thus, as mentioned above, we selected *AWS* machines to carry the experiments and our theory proved to be right. The capacity and efficiency of the machines let the performance of the frameworks be truly equal. As a result, we were able to test and compare the two frameworks in a well-matched manner.

4 Description of the Experiments

The experiments deemed most representative for the fair comparison of *Apache Spark* and *Ray* were the following:

4.1 Classification

First, a dataset had to be created, using `make_classification` function of sklearn. Each sample of the dataset consisted of 60 features, only 20 of which were informative. The number of classes was 2 (binary classification), and the dataset was created in chunks (smaller files) with 400000 rows each.

This successfully generated about 8GB of training data and 2GB of test data, which was split into 20 and 5 parquet files respectively (2:1 compression achieved). The dataset was also imputed with some null values (100 for each chunk) and was then stored on the HDFS filesystem.

After generating the dataset, testing scripts were programmed as to evaluate the performance of each framework. For both *Ray* and *Spark*, the machine learning pipeline consisted of the following steps:

1) Impute the null values with a Simple Imputer using 'mean' as strategy
2) Train an XGBoost Classifier with 100 estimators and a 0.01 learning rate
3) Make predictions on the test dataset
4) Evaluate the predictions using the metric accuracy

4.2 Regression

The regression dataset was generated similarly to classification one, with the difference being that the sklearn function used here was `make_regression`. Again, the samples consisted of 60 features (20 informative) and some gaussian noise was also included by setting the noise parameter equal to 1. The data was split in training and test with 23 and 6 parquet files (40000 samples in each parquet) respectively, totaling 5.5GB (11GB uncompressed). The dataset was also imputed with some null values (100 for each chunk) and stored in the HDFS filesystem.

The machine learning pipeline used here is similar to that of classification and is listed below:

1) Impute the null values with a Simple Imputer using 'mean' as strategy
2) Train an XGBoost Regressor with 100 estimators and a 0.01 learning rate
3) Make predictions on the test dataset
4) Evaluate the predictions using the metric root mean square error (RMSE)

4.3 Clustering

For this experiment, three smaller evaluation sets were created. Each one was built with the function `make_blobs` of sklearn. Every sample consists of 40 features, belonging to 1 out of 5 clustering centers. The samples were created with a standard deviation of 0.3. Each dataset consists of 5 parquet files (500000 sample in each parquet), 1GB in total for a single dataset and 3GB for the 3 datasets (about 6GB uncompressed).

Noting, the initial methodology was to create a single dataset that could be used for clustering. However, in contrast to the XGBoost model, which is built-in for the package `xgboost_ray` and can distribute the data between workers, the sklearn library (KMeans) does not automatically perform data distribution, in *Ray*, with the task of clustering only assigned to one node at a time.

Therefore, the code provided by the Github repository ray-mapreduce-kmeans was used, offering a KMeans clustering implementation with *Ray*, which supports parallelism. Unfortunately, even though the necessary modifications to the code were made in order to work with the data given, its scalability was poor. To tackle that, three smaller datasets were generated (cluster1, 2 and 3), then processed simultaneously by creating a different Pipeline actor for each one.

The clustering performed on each dataset was also distributed in actors as to increase the parallelization of tasks. This was achieved through a MapReduce algorithm; a number of Mappers (5 in the experiment), which are responsible for assigning samples to clusters, were selected, and Reducers (same number as clusters, i.e. 5) which are responsible for updating the centers after the assignment step of each iteration.

For Spark, the built-in KMeans model was used, combined with threading so that the 3 datasets could be processed simultaneously in parallel (1 thread for every dataset). A Standard Scaler was introduced as well, dealing with problems that could arise due to the large dimensionality of the data.

4.4 PageRank

The final conducted experiment was the execution of the PageRank Algorithm on a set of custom undirected graphs. The graphs were created via a script which generated 5000000 edges for 100000 vertices (with labels from 0 to 99999). This was done with the help of the random module in python; for each edge, a random tuple of vertices was selected. However, because this procedure imposes the risk of generating multiple identical edges, libraries were used to parse the edges into the graphs which resulted in final graphs with no duplicate edges. A total of 9 graphs were created and their edges were stored in separate txt files, totaling 500MB (about 56 MB for each file). The files were then stored in the HDFS filesystem.

The execution of PageRank in *Spark* was done through the programming language Scala and the GraphFrames package. The edges and vertices were loaded into two dataframes and were then passed as arguments to the graph constructor. Then, by using the built in pagerank method, the PageRank of the graph was calculated, with reset probability equal to 0.15 and tolerance equal to 0.01. With the help of threading in Scala, 3 graphs were parsed and the PageRank was calculated, all simultaneously.

In *Ray*, execution of the algorithm proved to be more complex than expected. To help create the graph and perform some basic operations, the Python module networkx was utilized. Although it offers a built in PageRank implementation, the algorithm was executed by only one node and, as a result, the work was not distributed. Ultimately, this created biased results, not allowing for the fair comparison of *Ray* and *Spark*, since in *Spark*, the algorithm was parallelized. To overcome this pitfall, a parallel version of PageRank was implemented using Python- illustrated by the image above.

In this experiment's implementation, the values of $x_v^{(k+1)}$ and δv were calculated in parallel by two or three separate actors. Similarly to *Spark*, actors were used as to be able to process three graphs at the same time. In short, although the graph was created with networkx, the code for the parallel PageRank algorithm was written by us. For the work to be evenly distributed between the nodes, 'SPREAD' was opted for as a scheduling strategy in *Ray*. The reset probability and tolerance values were the same as in the case of *Spark*.

Input: graph $G = (\mathcal{V}, \mathcal{E})$, α, ϵ
Output: PageRank $\mathbf{x}$

Initialize $\mathbf{x} = (1-\alpha)\mathbf{e}$
while true **do**
  for $v \in \mathcal{V}$ **do**
    $x_v^{(k+1)} = \alpha \sum_{w \in \mathcal{S}_v} \frac{x_w^{(k)}}{|\mathcal{T}_w|} + (1-\alpha)$
    $\delta_v = |x_v^{(k+1)} - x_v^{(k)}|$
  end for
  if $\|\boldsymbol{\delta}\|_\infty < \epsilon$ **then**
    break;
  end if
end while
$\mathbf{x} = \frac{\mathbf{x}}{\|\mathbf{x}\|_1}$

Fig. 1. Pagerank Algorithm

5 Results and Performance Analysis

In the following section, the results of the different configurations used for the experiments along with relative time plots and screenshots of the peaks of memory usage will be presented. Then, the observations and comments for each experiment will be described, comparing the performances of *Apache Spark*, and *Ray*.

The time plots were created based on data obtained through the scripting and execution of the experiments located at GitHub folder */Results/Stats*, while the data for memory plots were collected from screenshots while the programs executed located at GitHub folder `/Results/Memory_Screenshots`.

The configurations marked as representative for every experiment were the following:

- Cluster with 3 workers, 4 vCPU cores each
- Cluster with 3 workers, 2 vCPU cores each
- Cluster with 2 workers, 4 vCPU cores each

No limitations on the memory usage were imposed, and the same configurations for both *Spark* and *Ray* were used.

For some experiments not all configurations could be tested, due to the cluster experiencing crashes as the workload was too memory heavy. These cases are noted in the analysis below, and are often a key point of comparison.

The analysis begins with two classic machine learning tasks, namely Classification and Regression. For both experiments the XGBoost model was favored over classic models due to *Ray* offering a built-in implementation of distributed XGBoost for both tasks, that performs the distribution of dataset between nodes. During the model selection phase some alternatives were taken into consideration, such as sklearn models, or pytorch/tensorflow implementations that *Ray* supports. However, in the end, both were dismissed: The first option, as traditional sklearn models do not work distributely with *Ray* and one node is assigned the total workload, and the second as it was considered of the scope of this assignment. The XGBoost model is relatively easy to manage in terms of code and can allow for a similar implementation of the experiment in *Ray* and *Spark*.

Before presenting the results, it also has to be noted that although both in *Spark* and *Ray* the same model (XGBoost) was used, *Ray* is more flexible in configuring distributed

parameters. Specifically, although both frameworks offer the capability of selecting the number of workers for the XGBoost pipeline, in *Ray* the number of CPU/GPU cores per actor is also customizable. In our experiments for *Ray*, it was decided that only half the CPU cores of each worker will be utilized for training and prediction and the rest will handle the supporting elements of the pipeline and code. No GPU was used in any experiment.

5.1 Classification

For Classification, Accuracy was selected as the metric for evaluating the results of the ML Pipelines.

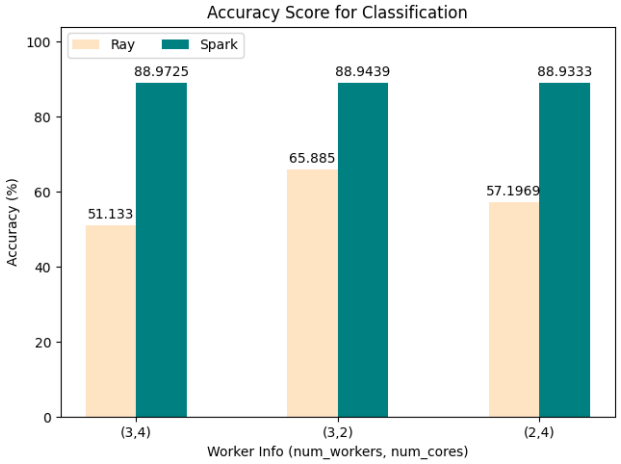

Fig. 2. Accuracy Score Figure

Spark is dominating *Ray*'s performance in the classification task and the graph above clearly shows the superior results of *Spark*. In the case of *Ray*, the performance is generally poor. Especially concerning is the case of 3 nodes and 4 cores/node for *Ray*, as the accuracy result does not different much from pure guessing the target labels. This could mean one of the following things: Either the data processing and training was not benefited by the bigger number of nodes and cores, either there was too much overfitting, either the training was not optimally performed. The results are somewhat better for the other configurations, but still they are not stable as in the case of *Spark*, where the Accuracy changes only by some decimal digits. Accuracy of *Ray* experiments seems to be dependent on the number of workers and cores used or it is just that the model cannot adapt to the particular dataset.

It is also worth noting, that even though the dataset was created through a simple script with the `make_classification` function of scikit learn, meaning that it contains no false information, outliers or human errors and lack of information, the highest accuracy score is not very close to 100% (about 90% accuracy). Before selecting XGBoost as our model we experimented with other classifiers such us Random Forest Classifiers, which achieved a greater score, around 98%. This could imply that the dataset generated by our script is not so suitable for training with XGBoost.

We observe that *Spark* does not have much fluctuation in execution time: In *Spark*, execution time undergoes a smooth

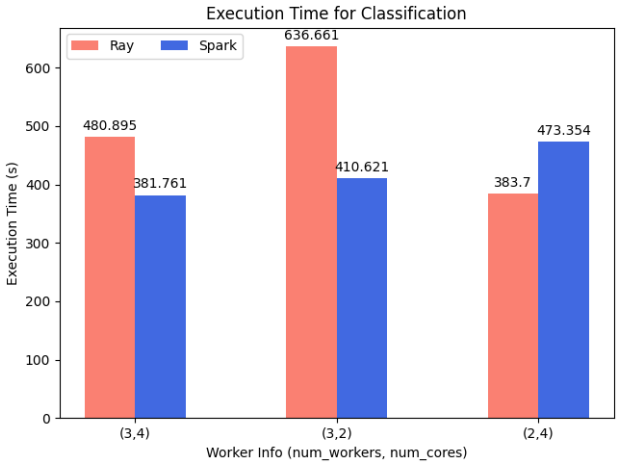

Fig. 3. Execution Time figure

increase when configurations change, which is rational, due to the cluster having lower CPU cores (case (3,4) to (3,2)) or workers (case (3,4) to (2,4)). However, in any case these changes are small. Therefore a simple conclusion we can reach is that spark can be characterized rather stable for the task of Classification, distributing work efficiently among nodes.

As for *Ray*, from the graph above it is clear that the number of cores has a great impact on the performance of the classification task. As the number of cores decreases, the execution time increases to approximately 1.4 the time compared to the double amount of cores. Here, it should be reminded that when designing the python script for *Ray* Classification, we decided that half the cores for each configuration will be used for training the XGBoost model and predicting the labels. That could partially explain the time differences.

Now, comparing *Spark* with *Ray*, we can see that for three workers *Spark* is able to execute the task faster than *Ray*, but the difference between the frameworks is that *Ray* 's performance, contrary to *Spark*, increases as the number of workers decreases. A possible explanation is that *Spark* can handle multiple workers better than *Ray*, distributing the required work much efficiently. *Ray*'s parallelized work for the task of Classification is collapsing as the number of nodes increases, as though many workers can not collaborate better than a small amount of workers.

As far as peak memory consumption is concerned, we can see that although the two frameworks use about the same memory size in the case of 3 workers, *Ray* does not escalate so well when reducing the number of available workers. Although, as also noted before for the time plot, peak memory consumption of *Spark* does not greatly fluctuate (we see a rational increase from 7.6 to 8.2 GB per worker), in the case of *Ray*, we notice a 57.7% increase in active memory used (from 7.8 to 12.3 GB per worker).

A key point that can be made by the above observation, is that *Spark* handles dataset distribution and (de)materialization better than *Ray* for the task of XGBoost Classification, leading to relatively stable memory consumption.

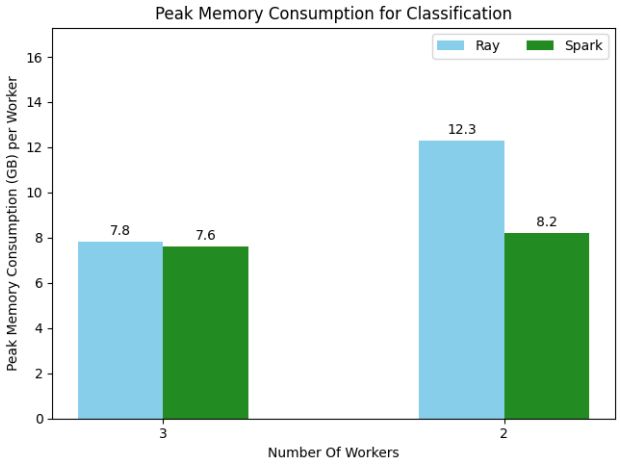

Fig. 4. Memory consumption figure

5.2 Regression

Before analyzing the results, it is highlighted that *Ray* could only handle the workload of the generated dataset in the case of three available nodes. When the number of nodes decreased to 2, one worker would crash, ending abruptly the experiment script. The reason that this did not happen before in the case of Classification is that we chose to use a slightly larger dataset here (11 GB for Regression, 10 GB for Classification), to be able to assess the performance of the two frameworks on bigger data.

This seems to imply that at least for the tasks of Classification and Regression where the same model is used (XGBoostClassifier,Regressor) *Spark* can handle memory and data distribution more efficiently than *Ray* resulting in *Spark* being able to process larger data with the same resources as *Ray*.

The metric for evaluating the performance of the two ML Pipelines is Root Mean Square Error (RMSE).

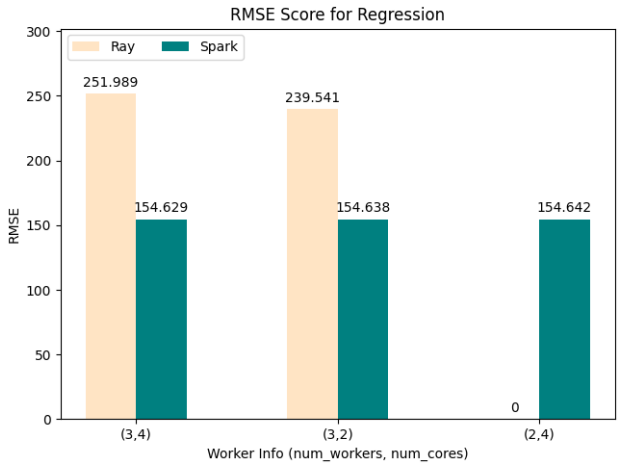

Fig. 5. RMSE Score for Regression

As it held true for the case of Classification, here, it can also be noticed that *Spark* achieves a far better RMSE score than *Ray* with about 100 units in difference. This, could mean that *Spark* can learn the dataset better than *Ray*, resulting in better performance. As mentioned before, the generation of Regression and Classification datasets was similar (`make_regression`, `make_classification`), so we can assume that *Spark* XGBoost performs better on the data generated through sklearn. Again, the RMSE of spark is stable (only the decimals change between experiments). In the case of *Ray*, however, there is some fluctuation of RMSE in the range of 10 units, although not as volatile as Accuracy in the Classification task.

It seems that as far as Regression is concerned, Ray XGBoost performs better than on Classification, still though much worse than *Spark*.

Fig. 6. Execution Time for Regression

The execution times for *Spark* follow a similar pattern to those of the Classification task before, with the only difference being the number of cores affecting the time performance less than on Classification. The number of workers on the other hand plays a greater role. As could be expected, a decrease on number of workers or cores leads to increased execution times.

As for *Ray*, the execution time gets a significant increase of 150% as the cores go from four to two.

In contrast to classification however, here we can notice a better workload management of *Ray* against *Spark* in the case of 3 workers with 4 cores each, with approximately 21% faster execution time. However, the penalty *Ray* suffered from decreasing the number of cores lead to *Spark* taking the lead in the case of 3 workers with 2 cores each.

All in all, we could say that one major difference between the two frameworks for both Regression and Classification tasks on XGBoost, seems to be that *Ray*'s performance is greatly dictated by the number of cores used, while *Spark* even though it sees a slight increase, it is not much affected.

As for the peak memory usage, *Ray* uses a greater amount of memory resources compared to *Spark*. When reducing the number of workers from 3 to 2, the increase of memory in *Ray* is so big, that the distributed workload can no longer fit in the memory of the workers and results in the ML Pipeline crashing. On the other hand, *Spark* seems to handle the decrease in workers very smoothly, with the peak memory use only reaching 8.8 GB (0.7 GB more than in the case of 2 workers).

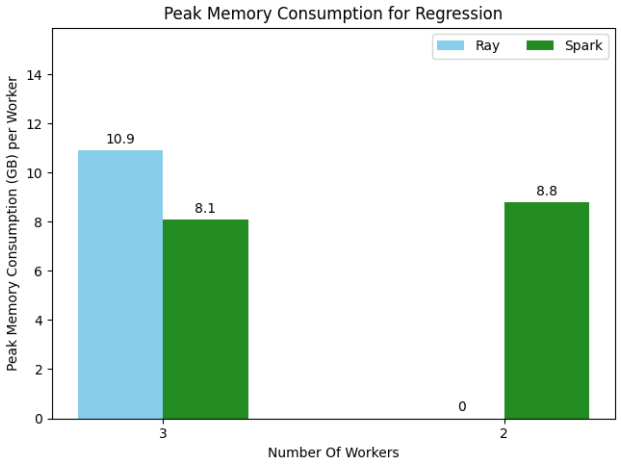

Fig. 7. Memory Consumption for Regression

This proves that for tasks like Regression and, as noted before, Classification, *Spark* is more memory efficient than *Ray* and can handle larger datasets.

5.3 Clustering

For this task, the two frameworks were assigned the execution of the KMeans algorithm on 3 different clustering datasets. As we could not find a built in *Ray* model to perform this task, we resorted to the aforementioned gihtub implementation[5] of parallelized distributed KMeans.

In the case of 3 workers the maximum number of iterations for the KMeans algorithm was selected to be 6. However, as in *Ray*, the memory requirements exceeded the available resources in the case of 2 workers, the maximum number of iterations was reduced to 2 for that case.

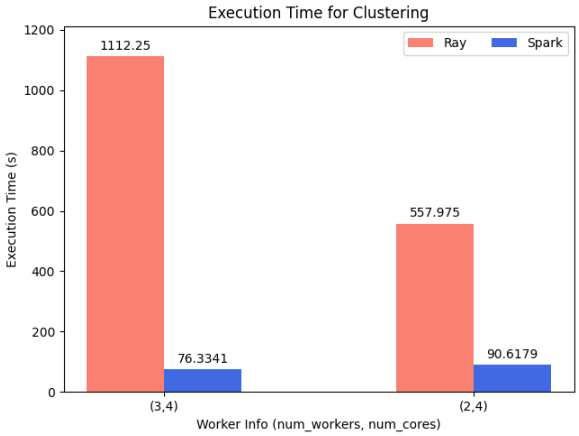

Fig. 8. Execution Time for Clustering

After examining the Execution Time of both experiments, the following observations can be made:

In *Ray*, the execution time is heavily dependent on the number of iterations of the KMeans algorithm. As it is evident here, by reducing maximum iterations from 6 to 2, the execution time dropped to 50% of its original value. Accounting for the timing overhead introduced by using one less worker, our clustering algorithm on *Ray* could have an almost linear dependence on the number of iterations.

Spark on the other hand, does not seem to be affected by the chosen number of maximum iterations. The time it took to perform the clustering was a bit higher in the case of 2 workers than that of three workers, which implies that KMeans algorithm in *Spark* is optimized on the number of iterations. Consequently, the only factor that could play a major role here, is the available parallelization/processing power which is dictated by the number of workers.

Comparing the two frameworks, it is evident that execution times for *Spark* are vastly better than that of *Ray*. To our knowledge, this could be explained by the fact that for *Spark*, a built in library was used (pyspark.ml.clustering.KMeans) where as in *Ray* a custom solution was applied. Here we have to note that the original KMeans implementation in [5] was created bearing in mind that the input data only had 2 feature dimensions. So, even though the implementation used a python compiler to communicate with C functions in order to speed up the process of clustering, the increase of feature space from 2 to 40 dimensions and reasons relating to code modifications for adjusting our experiment with the code provided seemed not to escalate optimally, resulting in much higher execution times than *Spark*.

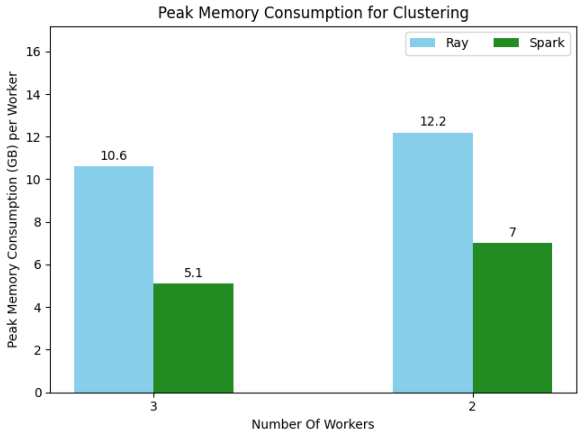

Fig. 9. Memory Consumption for for Clustering

The above diagram highlights that *Spark* outperforms *Ray* not only in terms of time but also in peak memory usage. Although both frameworks share a similar memory increase when decreasing the available workers, spark seems to use half the memory of *Ray*. Consequently the implementation of KMeans on *Spark* is more memory efficient than the custom implementation on *Ray*.

The above analysis reinforces the point, that the lack of extensive *Ray* libraries and packages for many ML tasks, due to it being still relatively new compared to *Spark*, makes *Spark*, at least for the time being, more suitable for performing ML tasks like clustering, especially when simplicity is opted for. Alternative solutions like the use of PyTorch and TensorFlow could be employed in the case of *Ray*. These solutions are also supported by the framework. However, developing ML pipelines on these packages is still more complex than using a simple KMeans model which is more than enough for the present task and out of the scope of this report.

5.4 PageRank

The final experiment conducted was the execution of PageRank on 9 undirected graphs. For *Ray* and *Spark* the configuration in the case of 3 workers was the same. However, when using 2 nodes in *Ray*, in order for our script to function correctly we decided to give each node 6 cores, totalling once more 12 cores. For *Spark*, we kept the number of cores at 4.

We assume that this change will not notably affect our analysis, as the VMs only posses 4vCPUs each, mapping to 2 physical cores in total. As a result, the '6 cores' assigned to each node are more like an internal *Ray* configuration that does not correspond to physical benefits or more processing power.

Results Assessment

The resulting PageRank values computed during execution where similar on both *Ray* and *Spark* with the top scoring nodes being the same and only differing in some decimal digits (probably due to slight differences in the implementation of the algorithm) .

For example, for a single graph the results compared side by side were the following:

```
9127,1.0727525733069476
64136,1.0690052830424093
20806,1.0652604805354924
36279,1.0646651991861786
94901,1.0637393049052892
63791,1.0637008504922303
76942,1.0630585002423256
71595,1.0625660891725328
47947,1.060691359717272
85009,1.0605782884454598
```

```
,node,score
0,9127,1.066663533638138
1,64136,1.0643460437424073
2,20806,1.060559166705525
3,36279,1.0601112307553848
4,94901,1.059193240044071
5,63791,1.059158574203627
6,76942,1.0585417889746462
7,71595,1.0580723944377235
8,47947,1.0561561170075786
9,85009,1.0560862380942038
```

Fig. 10. (Left) Spark, (Right) Ray

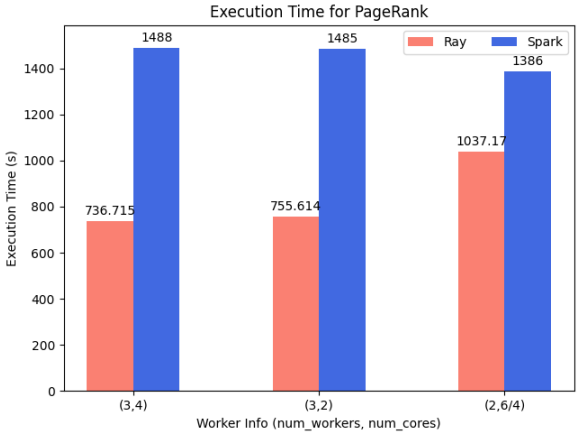

Fig. 11. Execution Time for PageRank

Here, we can observe the two frameworks showing the exact opposite behaviour.

Ray behaves as expected, with the fastest time achieved at the best configuration (3 workers, 4 cores each), with a slight time increase when halving the cores, and a significant one when reducing the workers by one. This is to be expected and it seems that the parallelization achieved in the case of *Ray* is very beneficial to the distributed execution of the PageRank algorithm on the graphs.

In the case of *Spark*, communication between different workers seems to not be so optimal. When executing the experiment we noticed a spike in memory consumption in one worker, while the others consumed significantly less memory, This means that workload was not evenly split between the workers and explains the reduce in execution time when the number of workers was reduced.

Overall, *Ray* seems to perform the PageRank task 50% faster than *Spark* in the case of 3 workers and 25% faster in the case of 2 workers. It can scale faster and more efficiently when the number of workers and processing power increase. *Spark*, on the other hand, appears more stable, but does not scale very well. One other reason that makes the better performance of *Ray* even more significant, is the fact that the PageRank program for *Spark* was developed in Scala, a language that relies on Java Virtual Machine and is much faster than Python, which was the programming language the PageRank algorithm for *Ray* was developed in.

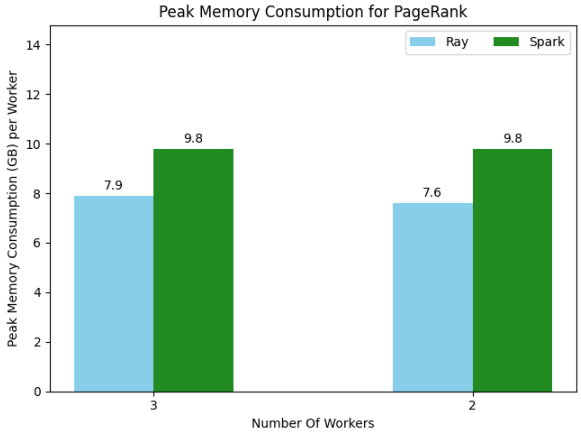

Fig. 12. Memory Consumption for for PageRank

By examining the memory plot, we can notice a rather steady peak memory consumption between the two frameworks. However, *Ray* consumes about 28% less memory than *Spark*, making it the best option between the two.

As a total statement regarding the PageRank algorithm, we have to denote that the dataset used for this operation was not very big (only 500MB). However the graph representation and processing requirements led to this high peak memory usage of the two frameworks. Considering the size of the dataset, we can only assume that the difficulty of this task lies in computational rather than data-intensive complexity. It is probably the only experiment where the biggest challenge was the execution of the algorithm itself and not so the handling of big data. This fact, combined with the better performance of *Ray* (remember that both frameworks provide similar outputs) leads us to the conclusion that *Ray* is more suitable for applications that are computationally heavy while *Spark* can handle data-intensive applications better.

6 Overall Comparison

In this section we present a thorough and comprehensive overall comparison between the two frameworks, providing

graphical analysis and additional information for each one.

6.1 Spark Conclusions

Below, where the bar is zero it means that there was no measurement.

6.1.1 Execution Time

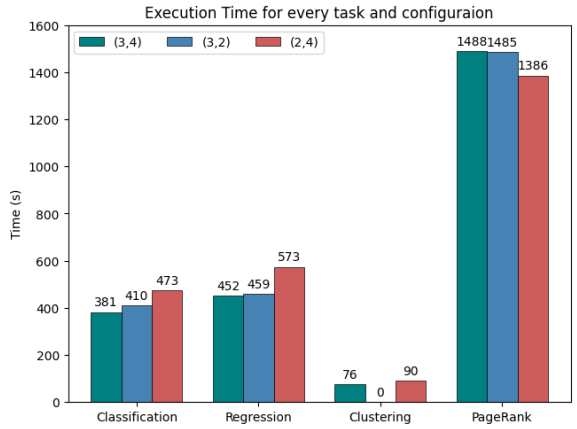

Fig. 13. Execution times for Spark

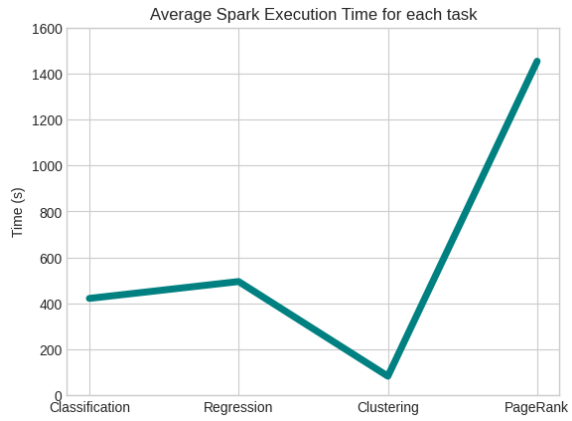

Fig. 14. Average Spark execution time

Using the above graphical analysis, it is pretty clear the tasks where *Spark* shines and the task (PageRank) where it hinders.

6.1.2 Memory Consumption

As for memory consumption, *Spark* has more stability with less worker nodes than with more workers.

6.2 *Ray* Conclusions

6.2.1 Execution Time

It is noticeable that from the average execution time graph of *Ray* for each machine learning task, that although it is not as fast as *Spark*, it is well-balanced keeping a steady high average time for every task of about 650 seconds.

6.2.2 Memory Consumption

Memory consumption using *Ray* is a lot higher than *Spark*.

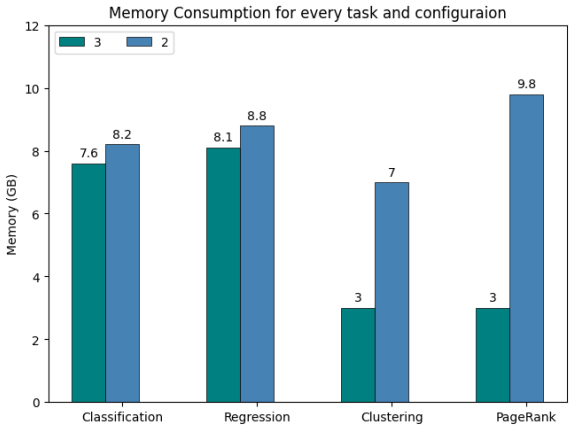

Fig. 15. Memory Consumption of Spark

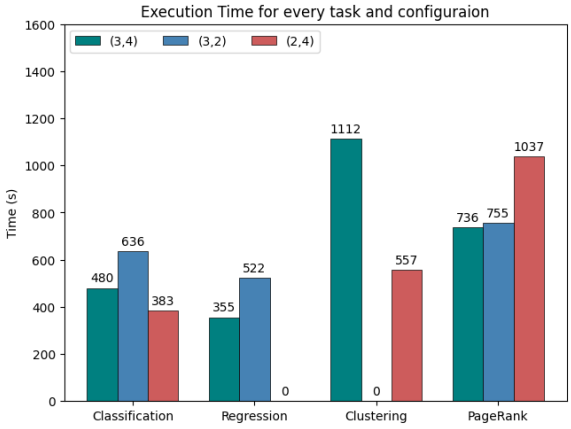

Fig. 16. Execution times for Ray

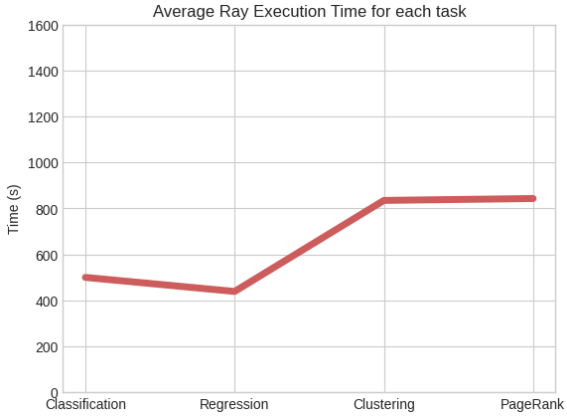

Fig. 17. Average Ray execution times for each task

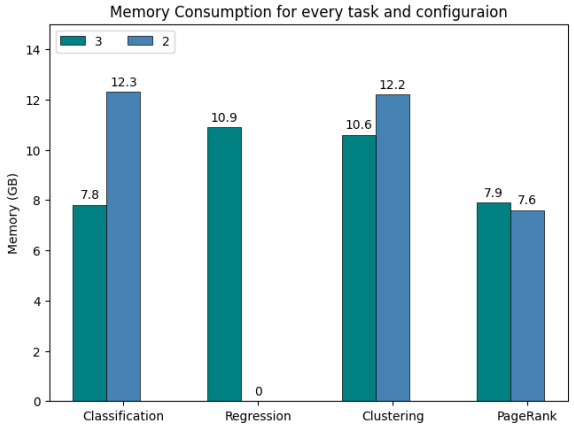

Fig. 18. Memory Consumption of Ray

6.2.3 Final Results

At this point, we provide an overall concise presentation of the strengths and weaknesses of the two frameworks, as established by the experiments conducted:

- ❍ **Setup**: The setup of the distributed cluster is easier on *Ray*, which just requires a Python environment. *Spark* on the other hand, due to its Java dependencies proves more difficult to set up for the first time. After the initial setup, the clusters are easy to start and stop on both frameworks
- ❍ **Configuration**: On *Spark* one can configure each worker's available CPU cores and memory that can be used while executing a task. On *Ray*, one can only assign logical (virtual) resources to each node, meaning that it is harder to impose physical restrictions on the amount of memory or CPU cores each worker is allowed to use.
- ❍ **Resource Distribution**: When designing a distributed program/task, *park* takes care for the allocation and distribution of resources between workers. *Ray* on the other hand, as observed in our experiments, offers fine-grained control on the amount of worker CPUs and other resources each task requires. This makes possible to assign different tasks to different cluster nodes, depending on their processing power, something difficult to perform on *Spark*.
- ❍ **Stability**: Our results seem to indicate that *Spark* is more stable in execution time and memory consumption than *Ray*, which experiences time fluctuations when the configuration of the cluster changes.
- ❍ **Scalability**: As implied by our previous analysis, *Ray* seems to perform and scale better on Computational-Heavy tasks, while *Spark* on Data-Intensive tasks.
- ❍ **Built-in packages and Features**: At the time of writing, *Ray* does not seem to offer so much built-in content and documentation as *Spark*, one of the reasons being that *Spark* is an established player on the market, while *Ray* is only yet emerging. That does not however limit the capabilities and potential of *Ray* as a framework, as its simple integration with many popular Python libraries (like PyTorch, XGBoost and TensorFlow), makes it a great option for developing distributed ML applications.

Finally a more succinct presentation of advantages and disadvantages:

Spark

- ❒ More builtin models libraries than ray
- ❒ Stable memory usage
- ❒ Superior on Classification, Regression, Clustering

Ray

- ❒ More fine-grained configuration of resources
- ❒ Straightforward setup
- ❒ Can be used along common python libraries but the details need to be taken care by the programmer
- ❒ Better on PageRank task

7 Conclusion

Upon considering all the facts, thoroughly examining the numerical analysis conducted for all experiments, we can now proceed to a final conclusion for both of the frameworks. The evaluation encompassed various aspects such as execution time, resource utilization, accuracy.

Machine Learning has seen unprecedented progress in the last few years and one reason for this advancement is the availability of numerous data. In the realm of Big Data, *Apache Spark* and *Ray* are considered to be a lighthouse for every pioneer that wishes to tame this massive amount of available data. Both frameworks, through extensive experimentation have demonstrated strengths and limitations in different scenarios.

Apache Spark exhibited robust performance for certain machine learning tasks. Its distributed computing capabilities proved advantageous for large-scale datasets and complex computations. In our experiments, *Apache Spark* clearly outperformed *Ray* in the majority of the tasks. However, it also showed some limitations in terms of adaptability to specific ML algorithms, such as PageRank. On the other hand, *Ray*, with its emphasis on flexible and scalable computation, showcased promising results.

The choice between *Apache Spark* and *Ray* depends on specific use case requirements, but in overall *Apache Spark* is undoubtedly a strong candidate. The decision should be guided by the specific needs of the application, the scale of the data, and the nature of the machine learning tasks at hand.

This report serves as a valuable reference for practitioners seeking insights into the comparative performance of these frameworks in real-world machine learning applications.

Acknowledgments

This work was conducted as part of the course: "Analysis and Design of Information Systems" of the 9^{th} semester of the faculty of Electrical and Computer Engineering at the National Technical University of Athens.

REFERENCES

[1] https://spark.apache.org/downloads.html
[2] https://sdkman.io/
[3] https://spark.apache.org/downloads.html
[4] https://docs.ray.io/en/latest/ray-overview/installation.html
[5] https://github.com/EthanWng97/ray-mapreduce-kmeans
[6] https://www.cs.utexas.edu/inderjit/public_papers/scalable_pagerank_europar15.pdf