Assignment 2: Dask vs PySpark vs Koalas vs Modin *

Emanuel Tomé †, João Ferreira ‡, Ricardo Faria §, Vânia Guimarães ¶

DCC-FCUP, July 2021

1 Introduction

Several parallel DataFrame systems alternatives to Pandas DataFrame are available nowadays. However, the differences between the available libraries is not always clear in which situations one is preferable over the others. Moreover, the cost of transition from Pandas to another library can be different not only for the need (or not) of learning a new syntax, but also in the compatibility of those libraries with others largely used by Data Scientists such as Scikit-Learn and Numpy. In this context, the main objectives of this work are:

- Identify performance bottlenecks when using a specific library.
- Identify syntactic differences among the different libraries.
- Identify operations that are best suited to one particular library.
- Get acquainted with these libraries and knowing what is supported from Pandas, Scikit-learn and Numpy.

2 Brief background on PySpark, Dask, Modin, JobLib, Rapids and Koalas

2.1 Dask

Libraries such as NumPy, Pandas and Scikit-learn, are designed to run on a single core. Therefore, all the data will be temporarily loaded onto the RAM of our local system. However, if we deal with extreme large datasets, we will inevitably problems to run it. Dask appeared mainly to solve this problem. Dask [1] is a parallel computing library that works by distributing larger computations and breaking it down into smaller computations through a task scheduler and task workers. Designed to parallelize in python ecosystems, Dask is suited to solve a wide variety of problems including structured data analysis, large-scale simulations used in scientific computing and general-purpose [2]. However, for data that fits into RAM, Pandas can often be faster and easier to use than Dask DataFrame. The popularity of the Dask is due to the union of the power of distributed computing for data science with the good integration to common Python data tools. Another advantage is that without change the interface, the users can run on clusters with multiple cores or on a common machine using a single process. Otherwise, the users don't really need to worry about the low-level internals, Dask provides several collections for wrapping low-level tasks into high-level workflows.

2.2 Modin

Modin [3] uses Ray or Dask to provide an effortless way to speed up pandas notebooks, scripts and libraries. The authors claim that Modin is able to scale the pandas workflow by changing a single line of code. That is, to use Modin and take advantage of its speedup, the user only has to change the line of code which imports pandas, `import pandas as pd`, to `import modin.pandas as pd` and continues using their previous pandas notebooks.

*This work was submitted on the framework of the course Big Data and Cloud Computing of the Master in Data Science.

†Emanuel Tomé is a student at the Faculty of Sciences of the University of Porto. Currently, he is enrolled in the 1^{st} year of the Master's in Data Science (e-mail: up200702634@edu.fc.up.pt).

‡João Ferreira is a student at the Faculty of Sciences of the University of Porto. Currently, he is enrolled in the 1^{st} year of the Master's in Data Science (e-mail: up202004115@edu.fc.up.pt

§Ricardo Faria is a student at the Faculty of Sciences of the University of Porto. Currently, he is enrolled in the 1^{st} year of the Master's in Data Science (e-mail: up202004105@edu.fc.up.pt

¶Vânia Guimarães is a student at the Faculty of Sciences of the University of Porto. Currently, she is enrolled in the 1^{st} year of the Master's in Data Science (e-mail: up200505287@edu.fc.up.pt).

Moreover, to use Modin the user does not have to know how many cores their system has and does not need to specify how to distribute the data. Data Scientists spend their time extracting value from their data than on tools that extract data.

2.3 JobLib

Joblib provides solution for several activities, such as loading up large Numpy arrays, persisting python object or performance of custom python function, with the help of parallel computing, memoization and caching mechanism. Joblib [4] provides a set of tools for making the pipeline lightweight to a great extent in Python [5], without dependency on other libraries. Joblib allows to use cache which avoids recomputation of some of the steps and execute parallelization to fully utilise all the cores of CPU/GPU. So, function called with same argument will not be re-compute, instead, output loads back from cache using memmapping. It also provides a compressor during persistence for large data, to save space on disk. A fantastic library that became popular because of its optimized time-complexity feature.

2.4 Rapids

Rapids [6] is a suite of open source software libraries and APIs that provides the ability to execute end-to-end data science and analytics pipelines entirely on GPUs. Rapids utilises NVidia Cuda primitives for low-level compute optimisation, and exposes GPU parallelism and high-bandwidth memory speed through user-friendly Python interfaces. It also focuses on common data preparation tasks for analytics and data science, and includes support for multi-node, multi-GPU deployments, enabling vastly accelerated processing and training on much larger dataset sizes.

2.5 PySpark

Apache Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters [7]. Spark supports multiple widely used programming languages, such as Python, Java, Scala and R, includes libraries for diverse tasks ranging from SQL to streaming and machine learning and runs anywhere (from a laptop to a cluster of thousands of servers).

PySpark [8] is an interface for Apache Spark in Python. It allows to write Spark applications using Python APIs and also provides the PySpark shell for interactively analyse data in a distributed environment. PySpark supports most of Spark's features such as Spark SQL, DataFrame, Streaming, MLlib (Machine Learning) and Spark Core.

2.6 Koalas

Pandas syntax and PySpark syntax differ to a considerable degree because PySpark has been notably influenced by SQL syntax. Regular pandas users will argue that it is much less intuitive. This is where Koalas enters the picture. Koalas is a data science library that implements the Pandas APIs on top of Apache Spark so, data scientists can use their favourite APIs on datasets of all sizes [9], and be more productive. Pandas does not scale well to big data since it was designed for small data sets. Using Koalas it is possible make the transition from a single machine to a distributed environment without needing to learn a new framework.

3 Materials and methods

3.1 Machines used and their characteristics

As required, we started by creating a cluster similar to i3.4xlarge AWS and a machine similar to i3.16xlarge, on Databricks and Dataproc. However, trial accounts for both platforms do not allow work with such power machines. This information is not provided to us, therefore after we started the experiments, both counts were cancelled. To solve the problem, we asked for a machine at DCC. With advanced project, the server stopped to work. The next alternative was create a single node on Dataproc with 8 CPUs and 32 Gb of memory with 300 dollars, the coupon given by Google for each new member. Although, there was some incompatibilities with Dask and Koalas, that made it impossible to continue the execution of the assignment.

The finally alternative was to create two Virtual Machines. The experiments which required Pandas, Dask, Modin, Koalas and Joblib were executed in a Virtual Machine of Google Cloud Platform with 16 CPUs (Intel Cascade Lake) and 64 GB of memory. Selected zone was us-central1-a.

Another Virtual Machine was created to run the experiments with Rapids. This machine contains 8 CPUs, 30 GB of memory and 1 GPU Nvidia Tesla T4. Selected zone was us-central1-b.

3.2 Datasets description

As we mentioned before, we started with all dataset required. On each new machine created, the data was decreasing, according to the power of the respective machine. At the end, both experiments and the Machine Learning task were executed with 2009, January of Yellow Taxi Trip Records from NYC Taxi and Limousine Commission (TLC) Trip Record Data.

4 Experiment #1: repeat NYC taxi driver dataset study

In this section we present the results obtained for one month of the NYC taxi driver dataset study [9], namely for data from January 2009. Note that only a month of data was considered due to limitations of the available computational capacity. We present the computation times using the following libraries: Pandas (sequential), Joblib, Dask, Modin (with Dask), Koalas (PySpark) and Rapids.

In Figures 1 and 2 are presented the obtained computation times of the standard operations. The first thing to notice is that Rapids has an outstanding performance for all the operations considered. With the exception of operation "count index length" where Pandas is the faster (note that all data fit in memory), Rapids is the faster library. In the opposite side, Joblib seems to be generally the library with worst performance, having poor performance in the majority of operations and the highest total execution time (see Figure 4). However, we should remark that the parallelization with Joblib was made using Pandas dataframes, which should not be the best choice for this library since the Pandas dataframes are stored in columns and not in rows, as for example the Numpy arrays. However, this choice was made since our objective was compare the performance of different libraries when using dataframes.

It is also possible to notice that Modin, followed by Koalas, has by far the worst computation time for the operations "complex arithmetic ops" and "mean of complex artithmetic ops". Indeed, Modin has the second worst total execution time of all benchmarks (see Figure 4).

Considering the geometric mean of the execution times of the standard operations (Figure 4), one can state that Joblib had the worst geometric mean (2.92s) and Rapids and Pandas the best ones. The low computation times for Pandas is only possible since the dataframe fits in memory. We can also notice that although Dask as a lower total computation time than Koalas (Figure 3), the obtained geometric mean for both libraries is similar.

It should be remarked that care should be take when comparing the different computation times since they could not be completely fare or even comparable. For instance, Dask has the lower computation time for reading data. However, in Dask the reading is lazy, which means that the computation time for reading files will be the same regardless the size of the files.

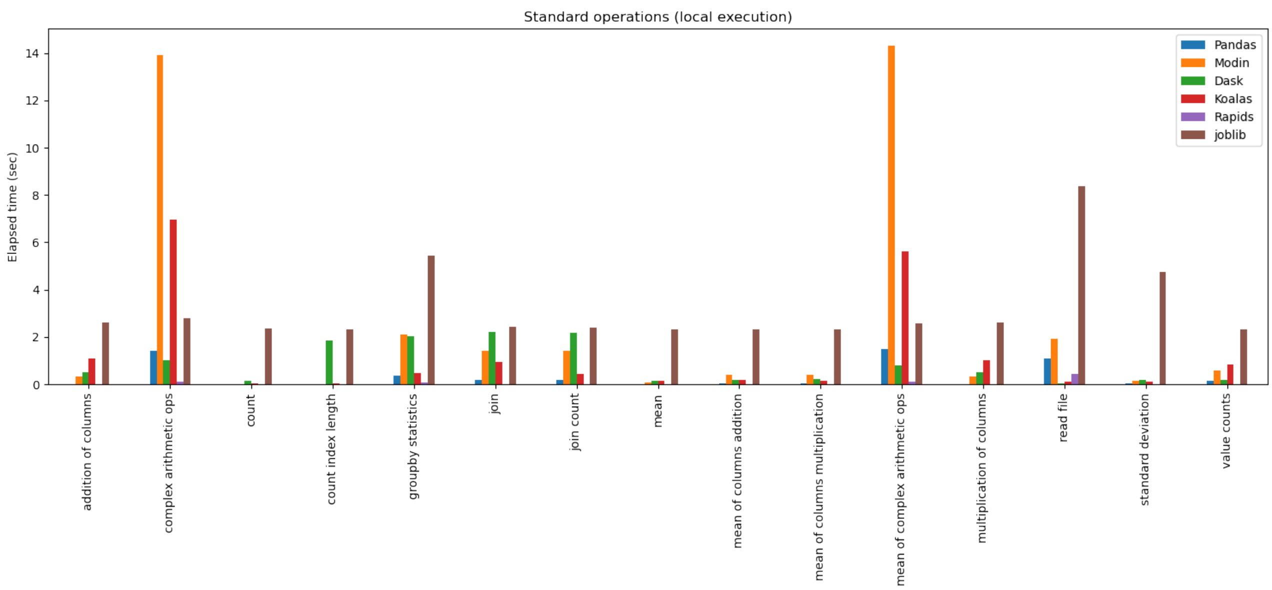

Figure 1: Computation times of the standard operations.

Considering now the computation times obtained for the filtered dataset, that is, for a small dataset, the obtained results are presented in Figures 5 and 6. One can noticed that now Dask has the worst computation times for almost all operations. This confirms what is stated in the documentation of Dask, where the developers confirm that for small datasets does not worth using Dask. Rapids for this smaller dataset is still the faster

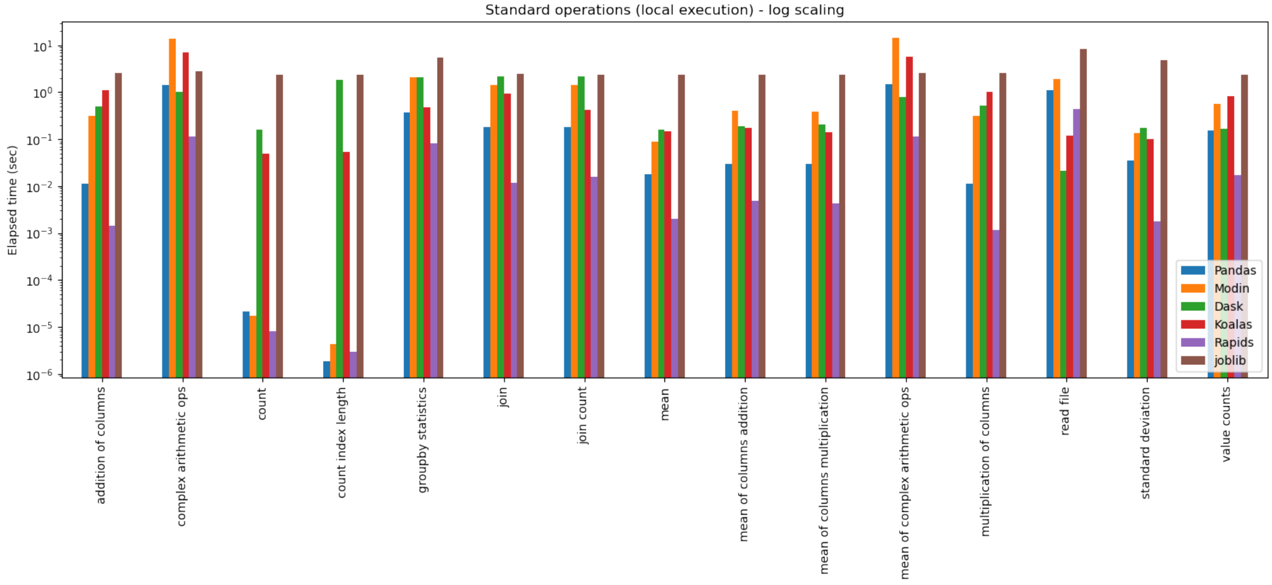

Figure 2: Computation times of the standard operations - logarithmic scale.

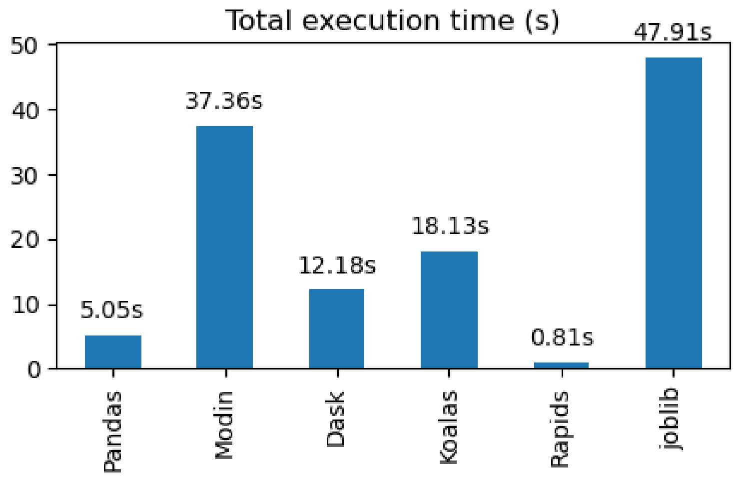

Figure 3: Total execution time of the standard operations

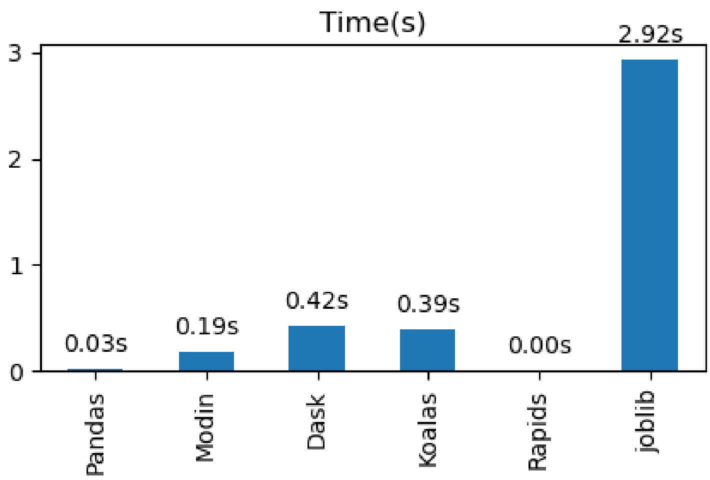

Figure 4: Geometric mean of the execution times of the standard operations.

library closely followed by Pandas. Modin and Koalas have similar total computation times, while Joblib has slightly lower total computation time (see Figure 7.

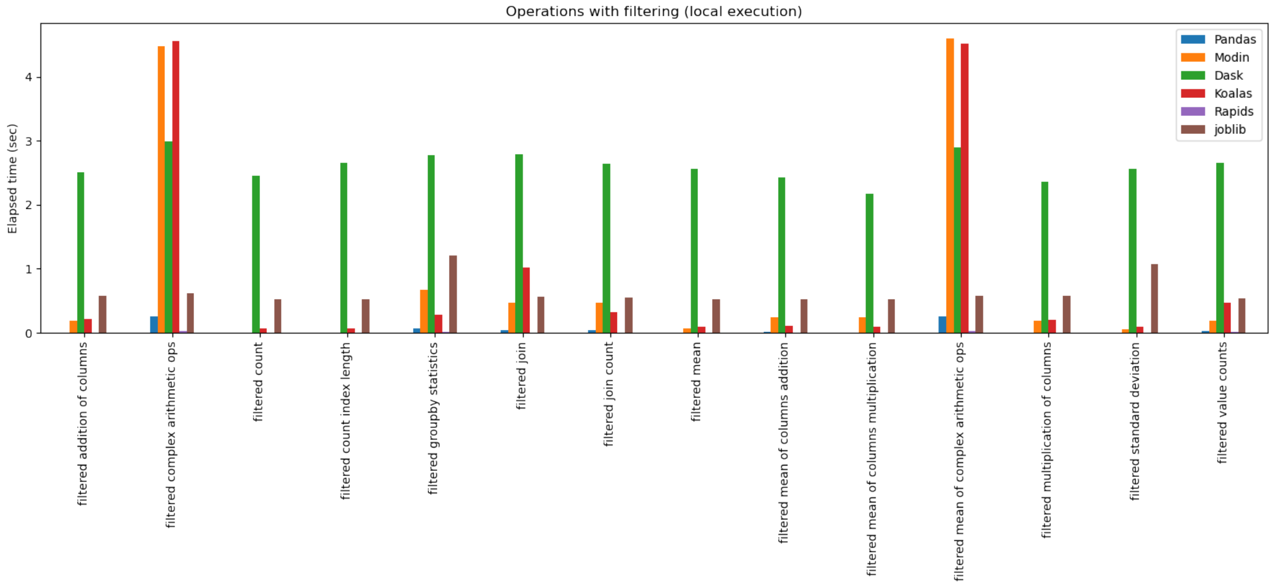

Figure 5: Computation times of the standard operations with filtering.

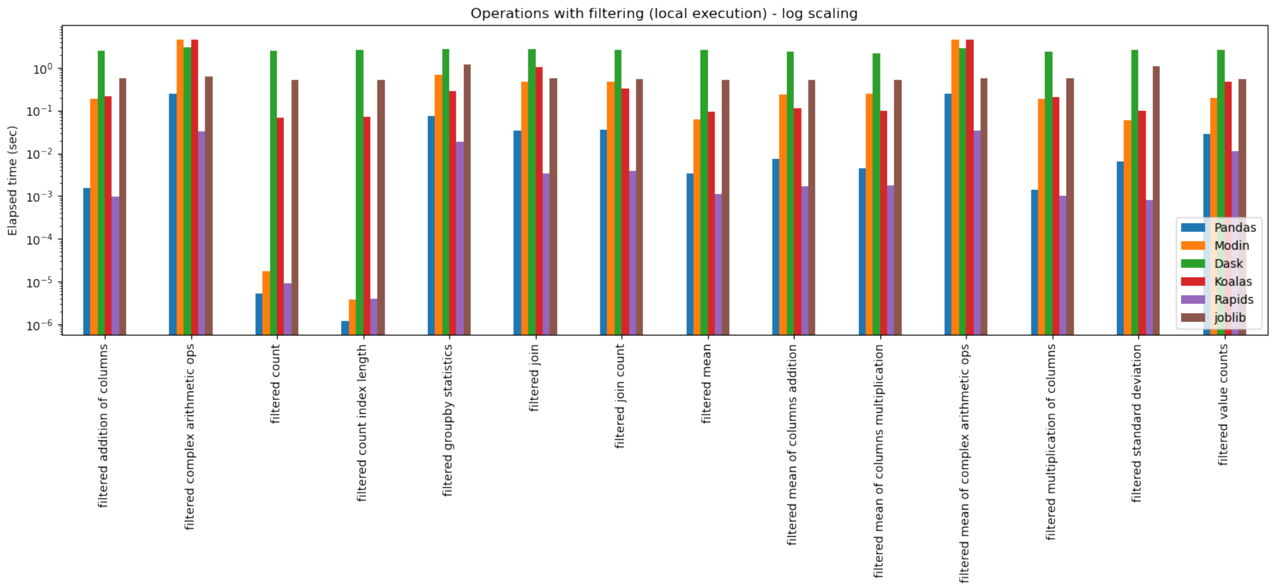

Figure 6: Computation times of the standard operations with filtering - logarithmic scale.

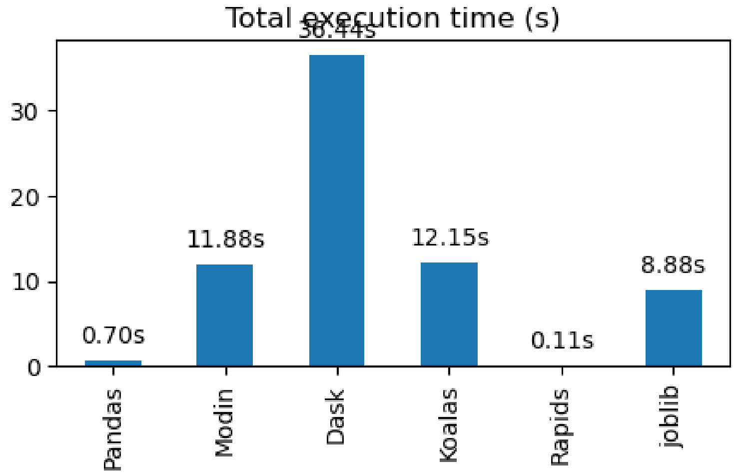

Figure 7: Total execution time of the filtered standard operations.

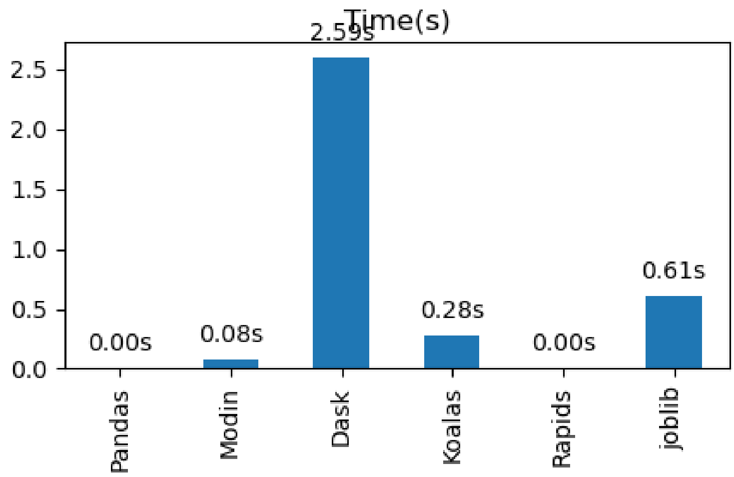

Figure 8: Geometric mean of the execution times of the filtered standard operations.

5 Experiment #2

5.1 Description of the experiment

The aim of Experiment #2 is to build a full machine learning pipeline and predict the target variable "fare_amount" using **Dask**, **Joblib**, **Koalas** and **Rapids**.

A full machine learning pipeline has the following steps:

- Reading the data
- Pre-processing
- Training
- Testing

5.1.1 Read Data

Due to problems of budget and computing resources, the dataset used in the pipeline corresponds to only one month of the NYC taxi driver dataset. The chosen month was January of 2009. With exception of Joblib, which reads the data stored as a Pickle file, all the other libraries read files in the parquet format.

5.1.2 Pre-processing

Depending on the type of dataframe used, the pre-processing may vary but these steps are common to all libraries. Using the time in "trip_pickup_datetime" it was added two new columns: "dayofweek" (0 to 6 according to the day of the week) and "hour". A new variable, "weekday" was created with dayofweek (0 if workday, 1 if weekend). Variable "trip_duration" was created by subtrating "trip_pickup_datetime" from "trip_dropoff_datetime". In "payment_type" some categories had different values with the same mean so these cases were recoded. All values with "CASH" recoded to "Cash" and "CREDIT" recoded to "Credit". Lines with abnormal values were removed according to this selection:

- "fare_amt" > 0
- "trip_distance" > 0
- "trip_duration" > 0
- "tip_amt" ≥ 0

An One Hot Enconding was applied to the categorical variables 'vendor_name' and 'payment_type'. Several variables with no interest were removed from the dataset, some of them are not related with the target, and others had no information. These variables are: 'index', 'trip_pickup_datetime', 'trip_dropoff_datetime', 'rate_code', 'mta_tax','store_and_forward', 'vendor_name' and 'payment_type'. The 'total_amt' was also removed since the 'fare_amt' can be easy computed just to subtracting in the 'total_amt' the variables 'tols_amt' and 'tip_amt'. Finally, it was applied a MinMaxScaler to all numeric variables including the target variable.

5.1.3 Traning

After preprocessing, randomSplit was applied to the dataset:

- 70% Train
- 30% Test

The model used by most libraries was XGBRegressor (Extreme Gradient Boosting) of Scikit-learn. We only setup parameter "objective='reg:squarederror'", which is the loss function that is minimized during the training of the model.

XGBRegression is a decision-tree-based ensemble Machine Learning algorithm that apply the principle of boosting. A decision tree is the weak learner, the resulting algorithm is called gradient boosted trees. The "eXtreme" refers to speed enhancements obtained by parallel computation.

For the experiments with Koalas, the Extreme Gradient Boosting was also tested[10] with Koalas but there were some errors that could not be solved. As described in PySpark Example, the spark-xgboost was installed and imported. However, the function XGBRegressor could not be loaded. Further, inspecting the package installed[11], several Python versions up to 3.6 are supported. Although, it was not possible work with XGBRegressor API even using Python 3.5 and 3.6.

Instead of XGB, the model used was GBTRegressor (Gradient-Boosted Trees), which is worse in terms of computational time, since it is not optimised with parallel processing. The loss function to be minimised the same as the XGB models.

5.2 Obtained metrics

Because this is a regression problem, in order to rank the performance of the models that were tested above, in the different pipelines, the metrics used were:

- Mean squared error - MSE.
- Root mean squared error - RMSE.
- Mean absolute error - MAE.
- Max error - ME.
- Coefficient of determination - R^2.

The MSE was the metric used to fit the models. The results obtained are shown in Table 1.

Table 1: Performance metrics of the fitted models on the different libraries.

	MSE	RMSE	MAE	Max error	R^2	Pipeline execution time (s)
Rapids	0.000244	0.015611	0.013758	0.954854	0.799853	97.75
Koalas	0.000089	0.009446	0.003511	0.989797	0.926805	461.52
Dask	0.000075	0.008655	0.002869	0.992073	0.938329	385.35
Joblib	0.000028	0.005292	0.001278	0.792122	0.977025	3422.39

As we can visualise, the library that obtained the best results (even if the differences are almost insignificant), was the *Joblib*, having the best results among the other ones. The rapids library was the one with the worse results and one possible reason for this was a limitation of the cuML library, which doesn't currently have an XGBRegressor available implemented. To circumvent this, the one used was XGBRegressor from Python XGBoost package, applied on the dataframe that was saved in GPU.

One additional limitation was the holdout performed on the GPU dataframe, since there isn't currently available a test_train_split function to perform the split of the dataframe on the GPU, so the way this was surpassed was by using the indices of the dataframe corresponding to x% of the whole data, without shuffling the data. As a consequence, the method decreased the independence of the data, and increased the bias. Regarding Koalas, and as already explained before, since we couldn't use the XGBRegressor model with that library, the GBTRegressor was used. Therefore, care should be taken when comparing the obtained metrics since the models are not exactly the same over the different libraries.

Overall, all the models had a similar performance independently of the library used, however, the execution time of the pipeline is quite different along the librarys. We can see that the Joblib was by far the one that had the longest execution time of them all, counting a total of 3422 seconds (57 minutes). Then, we had the Koalas library, that counted a total of 462 seconds (approximately 8 minutes), followed by the Dask library, which had took about 368 seconds (6 minutes, approximately), and finally, rapids which took only 85 seconds (1 minute and 25 seconds). This comes to reinforce the fact that using GPU for machine learning tasks that deal with large datasets is highly efficient, lowering the execution times of almost every standard operation we could think of.

5.3 Profiling

Regarding the profiling of the pipelines, for the majority of the libraries the pipeline part that takes more time to compute is the training of the machine learning model. The exception was Koalas, where more computation time was spent in predicting and computing the evaluation metrics. Joblib was by far the library that took more time to conclude the computation time. However, it should be noted that this is not directly related with Joblib, but with the library/model used to build the model, even though the used model and library trains the model in parallel. If only the pre-processing and "prediction and testing" steps of the pipeline are considered, Joblib has a very good performance.

With exception of Koalas, where the second bottleneck was the training of the machine learning model, the second botleneck for all libraries was pre-processing of the data. Indeed, this is a step in the pipeline where several computations take place, such as transformation of the variables, scaling, drop of variables, computation of new variables from others, merging of dataframes, etc. For this reason, it was expected that this stage of the pipeline was one of the stages with higher computation time. The framework with higher computation time was Dask. One possible reason for that is the need to create indices in the dataframe of numeric, categorical and target variables, in order to guarantee the correct concatenation of the three. Operations like "set_index" and 'merge/join' are harder to do in a parallel or distributed setting than if they are in-memory on a single machine. Shuffling operations that rearrange data become much more communication intensive.

The operation related to import data, Dask was faster than Rapids that use a GPU. This is due to the fact that one of the features of Dask is lazy execution. That means it loads and then processes the data in chunks, so that only a subset of the data needs to be in memory at any given time. However, cuDF stores Dataframes in GPU memory and uses the GPU to perform computations. Regarding Joblib, this was the library with the worst performance. However, it should be stated that for this library the data was readed from a pickle file because the library documentation refers that this is the more efficient way to load data to joblib. If the dataframe was read using, for instance, a lower computation time for this step would be obtained.

Table 2: Line profiling on each of the pipeline's operations.

	Import data	Pre-processing	Training	Predicting and Testing	Pipeline
Rapids	0.47	2.50	94.16	0.63	97.75
Koalas	2.7	19.04	391.50	443.08	856.34
Dask	0.02	23.92	348.12	13.27	385.35
Joblib	9.51	18.07	3388.54	6.26	3422.39

6 Main difficulties and challenges

In this work we faced many difficulties and challenges. The main ones are listed and described bellow:

- **Rapids limitations** - The lack of some vital functions on the cuDF and cuML libraries made some tasks not as easy as they should. For example, the train_test_split function was not implemented and it was needed to implement some kind of algorithm that would split the data into different parts. The problem here was to maintain efficiency, which was not possible due to the sequential programming used to perform this task.
- **XGBRegressor with Koalas** - as already referred, although all the attempts, it was not possible to use the XGBRegressor with Koalas/PySpark. As an alternative, we used the GBTRegressor algorithm.
- **Configuration of the Clusters** - In the begging of this assignment we tried to configure a cluster in the Google Cloud Platform (GCP) using Databricks. However, our account was cancelled two times due to the limitation of GCP described in the next point.
- **Limitation in the GCP of using only a CPU with no more than 8 cores** - when using the educational vouchers, it is not possible to use virtual machines with more than 8 cores. If the user does that, the billing and voucher are cancelled.
- **Necessity of changing platform several times** - We start by trying to do our experiments with Databricks. However, in the beginning our account was cancelled and countless options were tried. We then moved to a DCC server that was made available to the authors. However, when we started to implement our experiments related with the machine learning part, the kernel was always stopping. We then moved to a single node in the Dataprocs of the GCP where we were not able to run all the libraries. Namely, we where not able to read the parquet files that was created before using koalas. We then finally configured two virtual machines in GCP and these was our final solution. Note that each platform has their own characteristics which requires configuring the system, libraries dependencies and sometimes the code.
- **Scarce resources for a so big dataset** - Besides we have to find ways to load data related to 4 years of records and convert them to parquet format, deal with a system that goes down frequently or fit the data to our current resources without lose the purpose of the project was a big challenge. Additionally, the lack of credits in GCP made it difficult to setup a proper machine to run the tasks.
- **Impossibility of doing distributed execution** - As mentioned on the guidelines of the assignment, we tried to implement a distributed execution on Databricks, tirelessly looking for information that would guide us in this direction as we couldn't associate the cluster with the Dask client. After a day of intense search, we were informed that it is not possible to do it and that it was no longer necessary to apply the distributed execution in our experiments.
- **Interpretation of the results obtained using the Yappi profiler** - The output from the Yappi profiler is not as simple as the line profiler. While the line profiler outputs the time spent in each command, the yappi profiler outputs the time spent in each operation which makes it hard to identify, without knowing the meaning of each operation, where the program is spending more time. Although yappi may be harder to analyse, it is possible to profile each thread in details.

- **Time limitation -** Despite the extension of the deadline for the assignment, the difficulties encountered along the way were mainly due to the infrastructure needed to execute the experiments. This situation took our time in the analysis of the core of the project.

7 Discussion and Conclusions

In this work we applied different libraries that allow the parallelization of operations in DataFrames and data science pipelines. The obtained results in the benchmarks showed that the use of Rapids/GPUs outperforms by far the results using the other libraries. Dask and Koalas had similar performances and Joblib was the one with the worst computation times. However, it should be state that we used Joblib with dataframes, while it is more suited to use Numpy arrays, which are stored in memory in rows, while the dataframes are stored in columns. Moreover, the kind of operations done in our experiments may not be the ones for which this library is more suited for.

Rapids had an outstanding performance since it used an Nvidia GPU to perform all of the dataframe's operations as well all the machine learning tasks, having some limitations as previously said. Overall, the use of Rapids had an enormous impact in the execution times, decreasing it in relation to the other libraries tested. This was possible because the used GPU made possible to process multiple computations simultaneously, as it has a large number of cores, which allows for better computation of multiple parallel processes. This can be more specifically explained by the fact that GPUs are bandwidth optimised while CPUs are latency optimised, i.e the GPU can fetch much more memory at once, while CPU's can only fetch small amounts of memory quickly, having to perform a lot of memory operations to achieve the same goal. When you add this up to thread parallelism, GPUs get a lot of advantage from the fact that now it can perform large amounts of data operations, simultaneously and multiple times.

Koalas has the disadvantage to be the library with the more distinct syntax of the ones tested, specially when it is necessary to work in PySpark dataframes. Indeed, the Koalas syntax is similar to pandas syntax. However, there was the need of working directly with PySpark since there are functions that are not implemented in Koalas. Even though, since this is a brand new technology this may improve over the time and more functionalities and documentation may be provided.

Regarding the line profiling, we concluded that, at least for the model used, for the majority of the libraries the biggest part of the pipeline computation time is spent training the model. The exception was the Koalas library, which has more computation time in prediction and testing. The prediction and testing step was computed again, but this time only for predicting and it's computation time was 0.38s. The prediction time is low but the time used to compute RegressionEvaluator for each metric is quite large.

Finally, it should be referred that a bigger dataset may have been used to due a more fair comparison between the different libraries. Indeed, a data set of about 2.5GB may not be enough to evaluate the potential of the studied libraries. Therefore, this study may be complemented in the future with other analysis, operations and a bigger datasets.

References

[1] Dask Development Team. *Dask: Library for dynamic task scheduling.* 2016. URL: https://dask.org.

[2] J. Daniel. *Data Science with Python and Dask.* Manning Publications, 2019. ISBN: 9781638353546. URL: https://books.google.pt/books?id=KTkzEAAAQBAJ.

[3] Devin Petersohn et al. "Towards Scalable Dataframe Systems". In: *CoRR* abs/2001.00888 (2020). arXiv: 2001.00888. URL: http://arxiv.org/abs/2001.00888.

[4] Joblib Development Team. *Joblib: running Python functions as pipeline jobs.* 2020. URL: https://joblib.readthedocs.io/.

[5] Pratik Gandhi. *Using joblib to speed up your Python pipelines.* Sept. 2020. URL: https://towardsdatascience.com/using-joblib-to-speed-up-your-python-pipelines-dd97440c653d.

[6] Rapids Development Team. *About Rapids.* 2021. URL: https://rapids.ai/about.html.

[7] B. Chambers and M. Zaharia. *Spark: The Definitive Guide: Big Data Processing Made Simple.* O'Reilly Media, 2018. ISBN: 9781491912300. URL: https://books.google.pt/books?id=oitLDwAAQBAJ.

[8] *PySpark Documentation¶.* URL: https://spark.apache.org/docs/latest/api/python/index.html.

[9] *How fast Koalas and PySpark are compared to Dask - The Databricks Blog.* Apr. 2021. URL: https://databricks.com/blog/2021/04/07/benchmark-koalas-pyspark-and-dask.html.

[10] *Databricks XGBoost Example.* URL: https://databricks.com/notebooks/xgboost/xgboost4j-spark-example.html.

[11] *Github Spark-xgboost*. URL: https://github.com/sllynn/spark-xgboost/blob/master/setup.py.

Appendixes

A Detailed description of the used Machines

Machine 1 - No GPU
CPU:

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 16
- On-line CPU(s) list: 0-15
- Thread(s) per core: 2
- Core(s) per socket: 8
- Socket(s): 1
- NUMA node(s): 1
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 85
- Model name: Intel(R) Xeon(R) CPU
- Stepping: 7
- CPU MHz: 3100.240
- BogoMIPS: 6200.48
- Hypervisor vendor: KVM
- Virtualization type: full
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 1024K
- L3 cache: 25344K
- NUMA node0 CPU(s): 0-15

RAM: 64GB

Machine 2 - with GPU

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 8
- On-line CPU(s) list: 0-7
- Thread(s) per core: 2

- Core(s) per socket: 4
- Socket(s): 1
- NUMA node(s): 1
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 63
- Model name: Intel(R) Xeon(R) CPU @ 2.30GHz
- Stepping: 0
- CPU MHz: 2299.998
- BogoMIPS: 4599.99
- Hypervisor vendor: KVM
- Virtualization type: full
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 46080K
- NUMA node0 CPU(s): 0-7

RAM: 30GB
GPU: NVIDIA Tesla T4 (16GB)

B Benchmark results

C Source code of the setup functions for JobLib

```
import os
import joblib
from joblib import dump, load
from joblib import Parallel, delayed
import pandas as pd
import numpy as np
from tqdm import tqdm
import functools
# import databricks.koalas as ks
import time
import math

def get_results(benchmarks):
    """Return a pandas DataFrame containing benchmark results."""
    return pd.DataFrame.from_dict(benchmarks)

def benchmark(f, n_jobs=-1, df=None, benchmarks=None, name=None, **kwargs):
    """Benchmark the given function against the given DataFrame.

    Parameters
    ----------
    f: function to benchmark
    df: data frame
    benchmarks: container for benchmark results
    name: task name

    Returns
    -------
    Duration (in seconds) of the given operation
    """
    start_time = time.time()
```

Table 3: Computation times of the benchmark operations (in seconds).

	Pandas	Modin	Dask	Koalas	Rapids	Joblib
read file	1.097639	1.933903	0.021552	0.116975	0.440296	8.386324
count	0.000021	0.000017	0.161452	0.048637	0.000008	2.341892
count index length	0.000002	0.000004	1.84509	0.053336	0.000003	2.322545
mean	0.017941	0.090053	0.163021	0.144577	0.00205	2.328575
standard deviation	0.035824	0.136144	0.173782	0.102439	0.001807	4.737141
mean of columns addition	0.029497	0.395677	0.19121	0.172799	0.004861	2.321197
addition of columns	0.011179	0.315962	0.493689	1.09793	0.001438	2.600978
mean of columns multiplication	0.029217	0.384626	0.202049	0.141332	0.004349	2.319647
multiplication of columns	0.011427	0.315781	0.508529	1.015454	0.001162	2.606427
value counts	0.156386	0.576157	0.168916	0.835458	0.016973	2.336677
mean of complex arithmetic ops	1.504419	14.310768	0.784181	5.617018	0.112237	2.557495
complex arithmetic ops	1.431074	13.929066	1.032044	6.947288	0.114976	2.783355
groupby statistics	0.372646	2.109774	2.039868	0.481217	0.082091	5.423327
join count	0.177912	1.429141	2.177256	0.420335	0.015542	2.403003
join	0.177711	1.428062	2.215343	0.932832	0.011545	2.440177
filtered count	0.000005	0.000017	2.459231	0.069556	0.000009	0.520608
filtered count index length	0.000001	0.000004	2.660978	0.069706	0.000004	0.519908
filtered mean	0.003317	0.061901	2.558253	0.095373	0.001124	0.523736
filtered standard deviation	0.006458	0.058423	2.55637	0.09805	0.000816	1.067333
filtered mean of columns addition	0.00735	0.241343	2.422405	0.113718	0.00173	0.525848
filtered addition of columns	0.001515	0.185699	2.506612	0.218986	0.000979	0.577525
filtered mean of columns multiplication	0.004525	0.245141	2.175726	0.100009	0.001773	0.525475
filtered multiplication of columns	0.00138	0.191738	2.362341	0.208008	0.001014	0.576129
filtered mean of complex arithmetic ops	0.252189	4.605596	2.900001	4.521427	0.033652	0.578335
filtered complex arithmetic ops	0.250223	4.477522	2.987706	4.55545	0.032899	0.619596
filtered value counts	0.027867	0.192825	2.652701	0.468087	0.011148	0.532077
filtered groupby statistics	0.073863	0.677037	2.770664	0.283749	0.018422	1.203191
filtered join count	0.03491	0.474413	2.639927	0.328058	0.00389	0.544891
filtered join	0.033922	0.470783	2.7875	1.020233	0.003434	0.561686

```python
    ret = func_parallel(f, n_jobs, df, **kwargs)
    benchmarks['duration'].append(time.time() - start_time)
    benchmarks['task'].append(name)
    print(f"{name} took: {benchmarks['duration'][-1]} seconds")

    return benchmarks['duration'][-1]


def func_parallel(f, n_jobs=-1, df=None, **kwargs):
    n_jobs = n_jobs if n_jobs>0 else os.cpu_count()

    if df is not None:
        dim0 = df.shape[0] # numero de linhas
        chunk = math.ceil(dim0/n_jobs) #if n_jobs else math.ceil(dim0/os.cpu_count()) # tamanho do chunk
        nb_chunks = math.ceil(dim0/chunk)

        index_list = [(i*chunk,min(chunk*(i+1),dim0)) for i in range(nb_chunks)]


    if f.__name__ in ['standard_deviation']:
        average = func_parallel(mean, n_jobs=n_jobs, df=df)

        res_list = Parallel(n_jobs=n_jobs)(delayed(f)(df.iloc[index[0]:index[1],:],average, **kwargs) for index in index_list)

        res = (np.sum([res1[0] for res1 in res_list])/(np.sum([res1[1] for res1 in res_list])-1))**(1/2)

    elif f.__name__ in ['write_file_joblib']:
        Parallel(n_jobs=n_jobs)(delayed(f)(df.iloc[index[0]:index[1],:], i, pickle_file=kwargs['pickle_file']) for i,index in enumerate(index_list))
        res = 1

    elif f.__name__ in ['read_file_parquet']:
        res_list=Parallel(n_jobs=n_jobs)(delayed(f)(pickle_file) for pickle_file in kwargs['files_list'])
        res = pd.concat(res_list)

    else:
        res_list = Parallel(n_jobs=n_jobs)(delayed(f)(df.iloc[index[0]:index[1],:], **kwargs) for index in index_list)

        if f.__name__ in ['count', 'count_index_length']:
            res = sum(res_list)

        elif f.__name__ in ['mean','mean_of_sum','mean_of_product','mean_of_complicated_arithmetic_operation']:
            res = np.sum([res1[0] for res1 in res_list])/np.sum([res1[1] for res1 in res_list])

        elif f.__name__ in ['sum_columns','product_columns','complicated_arithmetic_operation']:
            res = pd.concat(res_list)

        elif f.__name__ in ['value_counts']:
            res = functools.reduce(lambda a, b: a.add(b, fill_value=0), res_list)

        elif f.__name__ in ['groupby_statistics']:
            # mean computation
            res = functools.reduce(lambda a, b: a.add(b, fill_value=0), res_list)
            mean_fare = pd.DataFrame(res.fare_amt['sum']/res.fare_amt['count'], columns=['fare_amt_mean'])
            mean_tip = pd.DataFrame(res.tip_amt['sum']/res.tip_amt['count'], columns=['tip_amt_mean'])

            # std computation
            def groupby_statistics_2(df, mean_fare, mean_tip):
                gb = df.groupby(by='passenger_count')
                df_2 = pd.DataFrame()
                for i, data in gb:
                    df_2.loc[i,'fare_amt_std'] = sum((data['fare_amt'] - mean_fare.loc[i,'fare_amt_mean'])**2)
                    df_2.loc[i,'tip_amt_std'] = sum((data['tip_amt'] - mean_tip.loc[i,'tip_amt_mean'])**2)
                return df_2
```

```
            res_list_std = Parallel(n_jobs=n_jobs)(delayed(groupby_statistics_2)(df.iloc[index[0]:index[1],:], mean_fare, mean_tip) for index in index_list)
            res_std = functools.reduce(lambda a, b: a.add(b, fill_value=0), res_list_std)
            res_std['fare_amt_std']=(res_std['fare_amt_std']/(res['fare_amt']['count']-1))**(1/2)
            res_std['tip_amt_std']=(res_std['tip_amt_std']/(res['tip_amt']['count']-1))**(1/2)

            res = pd.concat([mean_fare, res_std['fare_amt_std'], mean_tip, res_std['tip_amt_std']],axis=1)
            res.index.name= 'passenger_count'

        elif f.__name__ in ['join_data', 'join_count']:
            res = res_list[0]

    return res
```

D Source code of standard operations for JobLib

```
def write_file_joblib(df, i, pickle_file):
    with open(pickle_file+'_'+str(i), 'wb') as f:
        dump(df, f)#, compress='zlib')

def read_file_parquet(pickle_file):
    with open (pickle_file, 'rb') as f:
        df = load(f)
    return df

def count(df=None): # DONE
    return len(df)

def count_index_length(df=None): # DONE
    return len(df.index)

def mean(df): # DONE
    return (df.fare_amt.sum(), df.shape[0])

def standard_deviation(df, average): # DONE
    res = (df.fare_amt-average)**2
    return (res.sum(), res.shape[0])

def mean_of_sum(df): # DONE
    return ((df.fare_amt + df.tip_amt).sum(), df.shape[0])

def sum_columns(df): # DONE
    x = df.fare_amt + df.tip_amt
    return x

def mean_of_product(df): # DONE
    return ((df.fare_amt * df.tip_amt).sum(), df.shape[0])

def product_columns(df): # DONE
    x = df.fare_amt * df.tip_amt
    return x

def value_counts(df): # DONE
    val_counts = df.fare_amt.copy().value_counts()
    return val_counts

def complicated_arithmetic_operation(df): # DONE
    theta_1 = df.start_lon
    phi_1 = df.start_lat
    theta_2 = df.end_lon
    phi_2 = df.end_lat
    temp = (np.sin((theta_2 - theta_1) / 2 * np.pi / 180) ** 2
           + np.cos(theta_1 * np.pi / 180) * np.cos(theta_2 * np.pi / 180) * np.sin((phi_2 - phi_1) / 2 * np.pi / 180) ** 2)
    ret = np.multiply(np.arctan2(np.sqrt(temp), np.sqrt(1-temp)),2)
    return ret

def mean_of_complicated_arithmetic_operation(df): # DONE
    theta_1 = df.start_lon
    phi_1 = df.start_lat
    theta_2 = df.end_lon
```

```
    phi_2 = df.end_lat
    temp = (np.sin((theta_2 - theta_1) / 2 * np.pi / 180) ** 2
           + np.cos(theta_1 * np.pi / 180) * np.cos(theta_2 * np.pi / 180) * np.sin((phi_2 -
    phi_1) / 2 * np.pi / 180) ** 2)
    ret = np.multiply(np.arctan2(np.sqrt(temp), np.sqrt(1-temp)),2)
    return (ret.sum(), ret.shape[0])

def groupby_statistics(df): # DONE
    gb = df.groupby(by='passenger_count').agg(
      {
        'fare_amt': ['sum', 'count'],
        'tip_amt': ['sum', 'count']
      }
    )
    return gb

def join_count(df, **kwargs): # DONE
    return len(pd.merge(df, kwargs['other'], left_index=True, right_index=True))

def join_data(df, **kwargs): # DONE
    return pd.merge(df, kwargs['other'], left_index=True, right_index=True)
```

E Source code of Experiment #2

E.1 Dask

```
# Read data
import databricks.koalas as ks
def read_data():

    t0 = time.time()
    dask_data = dd.read_parquet('../../data/2009-01/', index_col='index')
    t1 = time.time()
    print(f'[INFO] Computation time: {t1-t0}s')

    return dask_data

def pre_processing(df):

    # Create variables dayofweek and hour
    df=df.assign(dayofweek = df.trip_pickup_datetime.dt.dayofweek,
                 hour = df.trip_pickup_datetime.dt.hour)

    # Creation of the variable weekend (0-weekend, 1-work day)
    df['weekend'] = (df['dayofweek'] < 5).astype(int)

    # Creation of the variables trip_duration
    df['trip_duration'] = (df['trip_dropoff_datetime'] - df['trip_pickup_datetime'])/timedelta
    (minutes=1)
    #df['trip_duration'] = df['trip_duration']/timedelta(seconds=1)

    # Drop variables with no interest
    df = df.drop(['trip_pickup_datetime',
              'trip_dropoff_datetime',
              'rate_code', 'mta_tax',
              'store_and_forward','dayofweek', 'total_amt'], axis = 1)


    # Corretion of the classes of the variable payment_type
    df = df.categorize(columns=['vendor_name', 'payment_type'])
    df['payment_type'] = df['payment_type'].replace('CASH', 'Cash')
    df['payment_type'] = df['payment_type'].replace('CREDIT', 'Credit')


    # Remove lines with abnormal values
    df = df[df['fare_amt']>0]
    df = df[df['trip_distance']>0]
    df = df[df['trip_duration']>0]
    df = df[df['tip_amt']>=0]

    # Min-max scaling of numerical variables
    numeric_variables = df[['passenger_count', 'trip_distance', 'start_lon', 'start_lat', '
    end_lon',
                          'end_lat', 'surcharge', 'tip_amt', 'tolls_amt',
```

```
                            'trip_duration', 'fare_amt']]

    scaler = MinMaxScaler()
    scaler.fit(numeric_variables)
    num_data = scaler.transform(numeric_variables)
    num_data = num_data.assign(idx=1)
    num_data = num_data.set_index(num_data.idx.cumsum()-1)
    num_data = num_data.drop('idx', axis=1)

    # One hot encondig of variables vendor_name and payment_type
    categorical_variables = df[['vendor_name', 'payment_type']]
    cat_data = dd.get_dummies(categorical_variables)
    cat_data = cat_data.assign(idx=1)
    cat_data = cat_data.set_index(cat_data.idx.cumsum()-1)
    cat_data = cat_data.drop('idx', axis=1)

    weekend = df[['weekend']]
    weekend = weekend.assign(idx=1)
    weekend = weekend.set_index(weekend.idx.cumsum()-1)
    weekend = weekend.drop('idx', axis=1)

    # Concatenate numerical variables and categorical variables
    num_cat_data = dd.merge(num_data, cat_data, left_index=True, right_index=True)
    dask_data = dd.merge(num_cat_data, weekend,  on = 'idx')#left_index=True, right_index=True) #

    return dask_data

def training(dask_data):
    # Split into training and testing data
    train, test = dask_data.random_split([0.7, 0.3], random_state=0)

    # Separate labels from data
    y_train = train.fare_amt
    y_test = test.fare_amt

    del train['fare_amt']  # remove informative column from data
    del test['fare_amt']  # remove informative column from data

    model = XGBRegressor(objective='reg:squarederror')

    t0=time.time()

    model.fit(train, y_train)

    t1 = time.time()
    print(f'[INFO] Computation time: {t1-t0}')

    return model, train, y_train, test, y_test

folder_out = '2009-01/'

def save_model(model, model_name):
    io.save_obj(model,model_name)

#save_model(model, folder_out + 'Dask_model'+ datetime.now().strftime("%H%M%S"))

# Prediction and test
from dask_ml.metrics import mean_absolute_error

def pred_test(model, test, y_test):
    # Predictions
    prediction = model.predict(test)

    # Evaluation metrics computation
    y_test = np.asarray(y_test)
    prediction = np.asarray(prediction)

    # Metrics
    df_metrics = pd.DataFrame()
    df_metrics.loc[0,'max_error'] = metrics.max_error(y_test, prediction)
    df_metrics.loc[0,'R^2'] = metrics.explained_variance_score(y_test,prediction)
    df_metrics.loc[0,'MAE'] = metrics.mean_absolute_error(y_test,prediction)
    df_metrics.loc[0,'MSE'] = metrics.mean_squared_error(y_test,prediction)
    df_metrics.loc[0,'RMSE'] = np.sqrt(metrics.mean_squared_error(y_test,prediction))

```

```
    return df_metrics


def pipeline_dask(n_jobs=-1):
    t0 = time.time()
    # Read data
    print('[INFO] Reading data...')
    df = read_data()
    print('[INFO] Data loaded.')

    # Pre-processing
    print('[INFO] Pre-processing')
    dask_data = pre_processing(df)

    #Train the model
    print('[INFO] Training the model...')
    t2 = time.time()
    model, train, y_train, test, y_test = training(dask_data)
    t3 = time.time()
    print(f'[INFO] Model trained in {t3-t2} s.')

    # Save the model
    print('[INFO] Saving the model...')
    save_model(model, folder_out + 'Dask_model'+ datetime.now().strftime("%H%M%S"))
    print('[INFO] Model saved.')

    # Predict and model metrics
    print('[INFO] Predicting and computing model metrics...')
    df_metrics = pred_test(model, test, y_test)

    # Save metrics
    io.save_obj(df_metrics,folder_out + 'Dask_model_metrics'+ datetime.now().strftime("%H%M%S"
))
    t1 = time.time()
    print('[INFO] All metrics computed.\n[INFO] End of the pipeline.')
    print(f'[INFO] Computation time for training the model: {t1-t0}s')
```

E.2 Joblib

```
def func_parallel(f, n_jobs=-1, df=None, **kwargs):
    n_jobs = n_jobs if n_jobs>0 else os.cpu_count()

    if df is not None:
        dim0 = df.shape[0] # numero de linhas
        chunk = math.ceil(dim0/n_jobs) #if n_jobs else math.ceil(dim0/os.cpu_count()) #
tamanho do chunk
        nb_chunks = math.ceil(dim0/chunk)

        index_list = [(i*chunk,min(chunk*(i+1),dim0)) for i in range(nb_chunks)]


    if f.__name__ in ['standard_deviation']:
        average = func_parallel(mean, n_jobs=n_jobs, df=df)

        res_list = Parallel(n_jobs=n_jobs)(delayed(f)(df.iloc[index[0]:index[1],:],average, **
kwargs) for index in index_list)

        res = (np.sum([res1[0] for res1 in res_list])/(np.sum([res1[1] for res1 in res_list])
-1))**(1/2)

    elif f.__name__ in ['write_file_joblib']:
        Parallel(n_jobs=n_jobs)(delayed(f)(df.iloc[index[0]:index[1],:], i, pickle_file=kwargs
['pickle_file']) for i,index in enumerate(index_list))
        res = 1

    elif f.__name__ in ['read_file_parquet']:
        res_list=Parallel(n_jobs=n_jobs)(delayed(f)(pickle_file) for pickle_file in kwargs['
files_list'])
        res = pd.concat(res_list)

    else:
        res_list = Parallel(n_jobs=n_jobs)(delayed(f)(df.iloc[index[0]:index[1]], **kwargs)
for index in index_list)

        if f.__name__ in ['count', 'count_index_length']:
```

```
            res = sum(res_list)

        elif f.__name__ in ['mean','mean_of_sum','mean_of_product','mean_of_complicated_arithmetic_operation']:
            res = np.sum([res1[0] for res1 in res_list])/np.sum([res1[1] for res1 in res_list])

        elif f.__name__ in ['sum_columns','product_columns','complicated_arithmetic_operation']:
            res = pd.concat(res_list)

        elif f.__name__ in ['value_counts']:
            res = functools.reduce(lambda a, b: a.add(b, fill_value=0), res_list)

        elif f.__name__ in ['groupby_statistics']:
            # mean computation
            res = functools.reduce(lambda a, b: a.add(b, fill_value=0), res_list)
            mean_fare = pd.DataFrame(res.fare_amt['sum']/res.fare_amt['count'], columns=['fare_amt_mean'])
            mean_tip = pd.DataFrame(res.tip_amt['sum']/res.tip_amt['count'], columns=['tip_amt_mean'])

            # std computation
            def groupby_statistics_2(df, mean_fare, mean_tip):
                gb = df.groupby(by='passenger_count')
                df_2 = pd.DataFrame()
                for i, data in gb:
                    df_2.loc[i,'fare_amt_std'] = sum((data['fare_amt'] - mean_fare.loc[i,'fare_amt_mean'])**2)
                    df_2.loc[i,'tip_amt_std'] = sum((data['tip_amt'] - mean_tip.loc[i,'tip_amt_mean'])**2)
                return df_2

            res_list_std = Parallel(n_jobs=n_jobs)(delayed(groupby_statistics_2)(df.iloc[index[0]:index[1],:], mean_fare, mean_tip) for index in index_list)
            res_std = functools.reduce(lambda a, b: a.add(b, fill_value=0), res_list_std)
            res_std['fare_amt_std']=(res_std['fare_amt_std']/(res['fare_amt']['count']-1))**(1/2)
            res_std['tip_amt_std']=(res_std['tip_amt_std']/(res['tip_amt']['count']-1))**(1/2)

            res = pd.concat([mean_fare, res_std['fare_amt_std'], mean_tip, res_std['tip_amt_std']],axis=1)
            res.index.name= 'passenger_count'

        elif f.__name__ in ['join_data', 'join_count']:
            res = res_list[0]

    return res

def read_file_parquet(pickle_file):
    with open (pickle_file, 'rb') as f:
        df = load(f)
    return df

def mean(df): # DONE
    return (df.sum(), df.shape[0])

# Read data
def read_data():
    folder_in_out = '2009-01/'
    folder_data = '../../data/joblib_taxi_pickles/' + folder_in_out
    files_list = ['part__0', 'part__1', 'part__2', 'part__3', 'part__4', 'part__5', 'part__6', 'part__7']
    files_list =[folder_data+file for file in files_list]

    t0 = time.time()

    joblib_data = func_parallel(read_file_parquet, n_jobs=-1, df=None, name='read file', files_list=files_list)

    t1 = time.time()
    print(f'[INFO] Computation time: {t1-t0}s')

    return joblib_data

# Pre-processing
```

```python
from datetime import timedelta
from sklearn.preprocessing import MinMaxScaler

def pre_processing_1(df):
    # Create variables dayofweek and hour
    df=df.assign(dayofweek = df.trip_pickup_datetime.dt.dayofweek,
                 hour = df.trip_pickup_datetime.dt.hour)

    # Creation of the variable weekend (0-weekend, 1-work day)
    df['weekend'] = (df['dayofweek'] < 5).astype(int)

    # Drop of variable weekend
    df = df.drop(['dayofweek'], axis=1)

    # Creation of the variables trip_duration
    df['trip_duration'] = (df['trip_dropoff_datetime'] - df['trip_pickup_datetime'])/timedelta(seconds=1)

    # Remove lines with abnormal values
    df = df[df['fare_amt']>0]
    df = df[df['trip_distance']>0]
    df = df[df['trip_duration']>0]
    df = df[df['tip_amt']>=0]

    # One hot encondig of variables vendor_name and payment_type
    cat_data = pd.get_dummies(df[['vendor_name', 'payment_type']])

    # Target
#     y = df['fare_amt']

    # Drop variables with no interest
    df = df.drop(['index','trip_pickup_datetime',
                  'trip_dropoff_datetime',
                  'rate_code', 'mta_tax',
                  'store_and_forward',
                  'vendor_name', 'payment_type'], axis = 1)

    # Concatenate dataframes
    df = pd.concat([df, cat_data], axis=1)

    return df


def pre_processing_2(df, scaler, numerical_variables):
    # Min-max scaling of numerical variables
#     numerical_variables=['passenger_count', 'trip_distance', 'start_lon', 'start_lat', 'end_lon',
#                          'end_lat', 'surcharge', 'tip_amt', 'tolls_amt', 'total_amt',
#                          'trip_duration','hour']

#     scaler = MinMaxScaler()
#     scaler.fit(numeric_variables)

    # Concatenate numerical variables and categorical variables
    df = pd.concat([pd.DataFrame(scaler.transform(df[numerical_variables]), columns=numerical_variables),
                    df.drop(numerical_variables, axis=1).reset_index(drop=True)], axis=1)

    return df


# Training
from sklearn.model_selection import train_test_split
from xgboost import XGBRegressor

def training(X, y):
    # Hold out of the data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=0)

    # Model training
    model = XGBRegressor(objective='reg:squarederror')

    # Model training
    model.fit(X_train, y_train)

    return model, X_train, y_train, X_test, y_test
```

```python


def save_model(model, model_name):
    io.save_obj(model,model_name)

# Prediction and test

def pred_test(model, X_test, y_test, mean_test):
    # Predictions
    prediction = model.predict(X_test)

    # Evaluation metrics computation
    df_error = y_test-prediction

    # Metrics
    df_metrics = pd.DataFrame()
    df_metrics.loc[0, 'max_error'] = max(df_error)
    df_metrics.loc[0, 'R^2'] = sum((y_test-mean_test)**2)
    df_metrics.loc[0, 'MAE'] = sum(abs(df_error))
    df_metrics.loc[0, 'MSE'] = sum(df_error**2)
    df_metrics.loc[0, 'size'] = df_error.shape[0]

    return df_metrics


def pipeline_joblib(n_jobs=-1):
    t0 = time.time()
    # Read data
    print('[INFO] Reading data...')
    df = read_data()
    print('[INFO] Data loaded.')

    # Pre-processing 1 (parallel)
    tp0 = time.time()
    print('[INFO] Pre-processing 1')
    n_jobs = n_jobs if n_jobs>0 else os.cpu_count()
    dim0 = df.shape[0] # numero de linhas
    chunk = math.ceil(dim0/n_jobs) #if n_jobs else math.ceil(dim0/os.cpu_count()) # tamanho do chunk
    nb_chunks = math.ceil(dim0/chunk)
    index_list = [(i*chunk,min(chunk*(i+1),dim0)) for i in range(nb_chunks)]

    res_list = Parallel(n_jobs=n_jobs)(delayed(pre_processing_1)(df.iloc[index[0]:index[1],:]) for index in index_list)
    df = pd.concat(res_list)
#     X = pd.concat([r[0] for r in res_list])
#     y = pd.concat([r[1] for r in res_list])

    tp1 = time.time()
    print(f'[INFO] End of pre-processing 1 in {tp1-tp0}s.')

    tp0 = time.time()
    print(f'[INFO] Fit of the scaller.')
    # Reset index (serial)
    df = df.reset_index(drop=True)
#     y = y.reset_index(drop=True)

    # Fit of the Min-Max scaller (serial)
    numerical_variables=['passenger_count', 'trip_distance', 'start_lon', 'start_lat', 'end_lon',
                         'end_lat', 'surcharge', 'tip_amt', 'tolls_amt', #'total_amt',
                         'trip_duration','hour','fare_amt']
    scaler = MinMaxScaler()
    scaler.fit(df[numerical_variables])

    tp1 = time.time()
    print(f'[INFO] End of the fit of the scaller in {tp1-tp0}s.')

    # Pre-processing 2 (parallel)
    tp0=time.time()
    print('[INFO] Pre-processing 2')
    dim0 = df.shape[0] # numero de linhas
    chunk = math.ceil(dim0/n_jobs) #if n_jobs else math.ceil(dim0/os.cpu_count()) # tamanho do chunk
    nb_chunks = math.ceil(dim0/chunk)
    index_list = [(i*chunk,min(chunk*(i+1),dim0)) for i in range(nb_chunks)]
```

```
    res_list = Parallel(n_jobs=n_jobs)(delayed(pre_processing_2)(df.iloc[index[0]:index[1],:],
     scaler=scaler, numerical_variables=numerical_variables) for index in index_list)
    df = pd.concat(res_list)
    tp1 = time.time()
    print(f'[INFO] End of pre-processing 2 in {tp1-tp0}s.')

    # Training (serial function, but XGBR training is parallel)
    print('[INFO] Training the model...')
    tp0 = time.time()
    model, X_train, y_train, X_test, y_test = training(df.drop(['fare_amt'],axis=1), df['
    fare_amt'])
    tp1 = time.time()
    print(f'[INFO] Model trained in {tp1-tp0} s.')

    # Save the model (serial)
    print('[INFO] Saving the model...')
    tp0 = time.time()
    save_model(model, folder_out + 'Joblib_model'+ datetime.now().strftime("%H%M%S"))
    tp1 = time.time()
    print('[INFO] Model saved.')

    # Predict and model metrics (parallel)
    tp0 = time.time()
    print('[INFO] Predicting and computing model metrics...')
    mean_test = func_parallel(mean, n_jobs=-1, df=y_test)

    dim0 = X_test.shape[0] # numero de linhas
    chunk = math.ceil(dim0/n_jobs) #if n_jobs else math.ceil(dim0/os.cpu_count()) # tamanho do
     chunk
    nb_chunks = math.ceil(dim0/chunk)
    index_list = [(i*chunk,min(chunk*(i+1),dim0)) for i in range(nb_chunks)]


    res_list = Parallel(n_jobs=n_jobs)(delayed(pred_test)(model,
                                                         X_test.iloc[index[0]:index[1],:],
                                                         y_test.iloc[index[0]:index[1]],
    mean_test) for index in index_list)
    res = functools.reduce(lambda a, b: a.add(b, fill_value=0), res_list)
    res = res/res['size'].values[0]
    res['max_error'] = max([x['max_error'].values[0] for x in res_list])
    res['R^2'] = 1 - sum([x['MSE'].values[0] for x in res_list])/sum([x['R^2'].values[0] for x
     in res_list])
    res.loc[0,'RMSE'] = (res.loc[0,'MSE'])**(1/2)

    res = res.drop(['size'],axis=1)

    # Save metrics
    io.save_obj(res,folder_out + 'Joblib_model_metrics'+ datetime.now().strftime("%H%M%S"))
    tp1 = time.time()
    print(f'[INFO] Metrics computed and saved in {tp1-tp0} s.')

    t1 = time.time()
    print(f'[INFO] All metrics computed.\n[INFO] End of the pipeline in {t1-t0} seconds.')
```

E.3 Koalas

```

import pandas as pd
import numpy as np
import databricks.koalas as ks
from datetime import timedelta, datetime
from utils import io
import time

import matplotlib.pyplot as plt

from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import MinMaxScaler
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.regression import GBTRegressor
from pyspark.ml.linalg import Vectors
from pyspark.sql.functions import udf
from pyspark.sql.types import DoubleType
from pyspark.sql.functions import monotonically_increasing_id
```

```python

###########################################################################
########################### Read data ###################################
###########################################################################
def read_data():
    koalas_data = ks.read_parquet('../../data/2009-01', index_col='index')
    return koalas_data


###########################################################################
######################### Pre-processing ################################
###########################################################################
def pre_processing(koalas_data):

    koalas_data=koalas_data.assign(pdayofweek = koalas_data.trip_pickup_datetime.dt.dayofweek,
                                   phour = koalas_data.trip_pickup_datetime.dt.hour)

    # Creation of the variable weekend (0-weekend, 1-work day)
    koalas_data['WEEKDAY'] =  koalas_data['pdayofweek'] //5

    # Drop of variable weekend
    koalas_data = koalas_data.drop(['pdayofweek'], axis = 1)

    # Creation of the variables trip_duration
    koalas_data['trip_duration'] = koalas_data['trip_dropoff_datetime'] - koalas_data['
    trip_pickup_datetime']

    # Recode same variables with different names
    koalas_data['payment_type'] = koalas_data['payment_type'].mask(koalas_data['payment_type'
    ]=='CASH', 'Cash')
    koalas_data['payment_type'] = koalas_data['payment_type'].mask(koalas_data['payment_type'
    ]=='CREDIT', 'Credit')

    # Remove lines with abnormal values
    koalas_data = koalas_data[koalas_data['fare_amt']>0]
    koalas_data = koalas_data[koalas_data['trip_distance']>0]
    koalas_data = koalas_data[koalas_data['trip_duration']>0]
    koalas_data = koalas_data[koalas_data['tip_amt']>=0]

    # One hot encondig of variables vendor_name and payment_type
    cat_data = ks.get_dummies(koalas_data[['vendor_name', 'payment_type']])

    # Transform Categorical Variables in Koalas Dataframe to Pyspark Dataframe
    pycat = cat_data.to_spark()

    # Drop variables with no interest
    num_data = koalas_data.drop(['index', 'trip_pickup_datetime', 'trip_dropoff_datetime'
                                 , 'rate_code', 'mta_tax','store_and_forward', 'total_amt',
                                 'vendor_name', 'payment_type'], axis = 1)

    # Transform Numerical Variables Koalas Dataframe to Pyspark Dataframe
    pynum = num_data.to_spark()

    # Adding all values exept "fare_amt" to an array called "features"
    vectorAssembler_num = VectorAssembler(inputCols =['passenger_count', 'trip_distance', '
    start_lon', 'start_lat',
                                                      'end_lon', 'end_lat', 'surcharge', 'tip_amt'
    , 'tolls_amt',
                                                      'phour', 'trip_duration', 'WEEKDAY'],
                                          outputCol = "num_features")
    pynum2 = vectorAssembler_num.transform(pynum)

    # Apply MinMaxScaler to num_features and return scaledFeatures column
    scaler = MinMaxScaler(inputCol="num_features", outputCol="scaledFeatures")
    scalerModel = scaler.fit(pynum2)
    scaledData = scalerModel.transform(pynum2)

    # Transform fare_amt to a vector to apply MinMaxScaler on target variable
    vectorAssembler_fare = VectorAssembler(inputCols =['fare_amt'], outputCol = "fare_amt_f")
    scaledData2 = vectorAssembler_fare.transform(scaledData)

    # Apply MinMaxScaler to fare_amt_f and return s_fare_amt column
    scaler2 = MinMaxScaler(inputCol="fare_amt_f", outputCol="s_fare_amt")
    scalerModel2 = scaler2.fit(scaledData2)
    scaledData3 = scalerModel2.transform(scaledData2)

```

```python
    # Create id column on both categorical and numerical dataframes and merge by id
    df1 = scaledData3
    df2 = pycat

    df1 = df1.withColumn("id", monotonically_increasing_id())
    df2 = df2.withColumn("id", monotonically_increasing_id())

    df3 = df2.join(df1, "id").drop("id")

    # Add categorical values and scaledFeatures into new column features
    vectorAssembler = VectorAssembler(inputCols =['vendor_name_CMT', 'vendor_name_DDS', 'vendor_name_VTS',
                                                  'payment_type_Cash', 'payment_type_Credit',
                                                  'payment_type_Dispute', 'payment_type_No Charge',
                                                      'scaledFeatures'],       outputCol = "features")
    df4 = vectorAssembler.transform(df3)

    # Convert vector s_fare_amt to Double scaled_fare_amt
    unlist = udf(lambda x: float(list(x)[0]), DoubleType())
    pyko2 = df4.withColumn("scaled_fare_amt", unlist("s_fare_amt"))

    return koalas_data, pyko2


########################################################################
############################# Training #################################
########################################################################

def train(pyko2):
    # Hold out of the data
    splits = pyko2.randomSplit([0.7,0.3])
    train_df = splits[0]
    test_df = splits[1]

    # Model training
    gbt = GBTRegressor(featuresCol="features", labelCol="scaled_fare_amt", lossType = "squared")

    # Model training
    gbt_model = gbt.fit(train_df)

    return gbt_model, train_df, test_df


########################################################################
####################### Prediction and test ############################
########################################################################

def pred_test(gbt_model, test_df):
    # Predictions
    gbt_predictions = gbt_model.transform(test_df)

    # Root Mean Square Error
    gbt_evaluator_rmse = RegressionEvaluator(labelCol="scaled_fare_amt", predictionCol="prediction", metricName="rmse")
    gbt_rmse = gbt_evaluator_rmse.evaluate(gbt_predictions)

    # Mean Square Error
    gbt_evaluator_mse = RegressionEvaluator(labelCol="scaled_fare_amt", predictionCol="prediction", metricName="mse")
    gbt_mse = gbt_evaluator_mse.evaluate(gbt_predictions)

    # Mean Absolute Error
    gbt_evaluator_mae = RegressionEvaluator(labelCol="scaled_fare_amt", predictionCol="prediction", metricName="mae")
    gbt_mae = gbt_evaluator_mae.evaluate(gbt_predictions)

    # R2
    gbt_evaluator_r2 = RegressionEvaluator(labelCol="scaled_fare_amt", predictionCol="prediction", metricName="r2")
    gbt_r2 = gbt_evaluator_r2.evaluate(gbt_predictions)

    # Kolas Dataframe with predictions
    koalas_predict = ks.DataFrame(gbt_predictions)
```

```

    # Metrics
    df_metrics = pd.DataFrame()
    df_metrics.loc[0,'max_error'] = ((koalas_predict['prediction']-koalas_predict['
    scaled_fare_amt']).abs()).max()
    df_metrics.loc[0,'R^2'] = gbt_r2
    df_metrics.loc[0,'MAE'] = gbt_mae
    df_metrics.loc[0,'MSE'] = gbt_mse
    df_metrics.loc[0,'RMSE'] = gbt_rmse

    return koalas_predict, df_metrics

########################################################################
############################## Pipeline ################################
########################################################################

def pipeline_koalas():
    t0 = time.time()

    # Read data
    tr0 = time.time()
    print('[INFO] Reading data...')
    koalas_data = read_data()
    tr1 = time.time()
    print(f'[INFO] Data loaded in {tr1-tr0} s.\n')

    # Pre processing
    tp0 = time.time()
    print('[INFO] Pre-processing')
    koalas_data, pyko2 = pre_processing(koalas_data)
    tp1 = time.time()
    print(f'[INFO] Pre-processing done in {tp1-tp0} s.\n')

    # Traning
    tt0 = time.time()
    print('[INFO] Training the model...')
    gbt_model, train_df, test_df = train(pyko2)
    tt1 = time.time()
    print(f'[INFO] Model trained in {tt1-tt0} s.\n')

    # Predict and model metrics
    tm0 = time.time()
    print('[INFO] Predicting and computing model metrics...')
    koalas_predict, df_metrics = pred_test(gbt_model, test_df)
    tm1 = time.time()
    print(f'[INFO] Predictions and all metrics computed in {tm1-tm0} s.\n')

    # Save metrics
    io.save_obj(df_metrics, 'Koalas_model_metrics'+ datetime.now().strftime("%H%M%S"))

    t1 = time.time()
    print(f'[INFO] Pipeline computed in {t1-t0} s.\n[INFO] End of the pipeline.')
    return koalas_predict, df_metrics

########################################################################
############################## Yappi ###################################
########################################################################

import yappi

# Yappi Read
yappi.clear_stats()
yappi.start()
koalas_data = read_data()
yappi.stop()
stats = yappi.get_func_stats(filter_callback=lambda x: x.ttot>0.5)#.print_all()
stats.save('../outputs/callgrind.kread.prof', type = "ystat")

# Yappi Pre Processing
yappi.clear_stats()
yappi.start()
koalas_data, pyko2 = pre_processing(koalas_data)
yappi.stop()
stats = yappi.get_func_stats(filter_callback=lambda x: x.ttot>0.5)#.print_all()
stats.save('../outputs/callgrind.kprepros.prof', type = "ystat")

```

```python
# Yappi Train
yappi.clear_stats()
yappi.start()
gbt_model, train_df, test_df = train(pyko2)
yappi.stop()
stats = yappi.get_func_stats(filter_callback=lambda x: x.ttot>0.5)#.print_all()
stats.save('../outputs/callgrind.ktrain.prof', type = "ystat")

# Yappi Pred
yappi.clear_stats()
yappi.start()
koalas_predict, df_metrics = pred_test(gbt_model, test_df)
yappi.stop()
stats = yappi.get_func_stats(filter_callback=lambda x: x.ttot>0.5)#.print_all()
stats.save('../outputs/callgrind.kpred.prof', type = "ystat")
```

E.4 Rapids

```python
def import_data(path = None):#"../../data/ks_taxi_parquet"
    t0 = time.time()

    try:
        rapids_data = cudf.io.parquet.read_parquet(path)
        global train_ind
        train_ind = round(len(rapids_data)*0.7)
    except:
        print("No path provided.")
    t1 = time.time()

    print(f'[INFO] Computation time for loading the file: {t1-t0}s')
    return rapids_data


def preprocessing(rapids_data):
    t0 = time.time()

    rapids_data = rapids_data.assign(pdayofweek = rapids_data.trip_pickup_datetime.dt.
    dayofweek,
                                    phour = rapids_data.trip_pickup_datetime.dt.hour)

    # computation of the variable weekend
    #zero weekend, 1 week
    rapids_data['weekend'] = (rapids_data['pdayofweek'] < 5).astype(int)

    # computation of the variable trip_duration
    rapids_data['trip_duration'] = rapids_data['trip_dropoff_datetime'] - rapids_data['
    trip_pickup_datetime']
    rapids_data['trip_duration'] = rapids_data['trip_duration']/timedelta(seconds=1)

    # Corretion of the classes of the variable payment_type
    rapids_data['payment_type'] = rapids_data['payment_type'].mask(rapids_data['payment_type'
    ]=='CASH', 'Cash')
    rapids_data['payment_type'] = rapids_data['payment_type'].mask(rapids_data['payment_type'
    ]=='CREDIT', 'Credit')

    # Drop concatentated timestamp columns
    rapids_data = rapids_data.drop(['trip_pickup_datetime', 'trip_dropoff_datetime', '
    rate_code', 'mta_tax','store_and_forward','pdayofweek'], axis = 1)

    # Remove lines with abnormal values
    rapids_data = rapids_data[rapids_data['fare_amt']>0]
    rapids_data = rapids_data[rapids_data['trip_distance']>0]
    rapids_data = rapids_data[rapids_data['trip_duration']>0]
    rapids_data = rapids_data[rapids_data['tip_amt']>=0]
    rapids_data = rapids_data.drop('total_amt', axis = 1)

    rapids_data = rapids_data.iloc[:,1:].reset_index()

    # One hot enconding of variables vendor_name and payment_type
    rapids_data["vendor_name"] = rapids_data["vendor_name"].astype("category")
    rapids_data["payment_type"] = rapids_data["payment_type"].astype("category")

    categorical_variables = rapids_data[['vendor_name', 'payment_type']]

    # encoder = OneHotEncoder(sparse=False)
```

```python
    # encoder.fit(categorical_variables)
    # cat_data = encoder.transform(categorical_variables).compute()

    cat_data = cudf.get_dummies(categorical_variables)

    # Min-max scaler of numerical variables
    numeric_variables = rapids_data[['passenger_count', 'trip_distance', 'start_lon', 'start_lat', 'end_lon',
                                     'end_lat', 'surcharge', 'tip_amt', 'tolls_amt', 'trip_duration', 'fare_amt']]
    scaler = MinMaxScaler()
    scaler.fit(numeric_variables)
    num_data = scaler.transform(numeric_variables)

    num_data = num_data.rename(columns={0: 'passenger_count', 1: 'trip_distance', 2: 'start_lon', 3: 'start_lat', 4: 'end_lon', 5: 'end_lat',
                                        6: 'surcharge', 7: 'tip_amt', 8: 'tolls_amt', 9: 'trip_duration', 10: 'fare_amt'})

    rapids_data = cudf.concat([num_data, cat_data], axis=1) #rapids_data['fare_amt']]

    t1 = time.time()
    print(f'[INFO] Computation time for preprocessing the data: {t1-t0}s')
    return rapids_data


def training(X, y):#X = rapids_data.drop('fare_amt', axis = 1), y = rapids_data['fare_amt']
    # Hold out of the data
    X_train = X[0:train_ind]
    X_test = X[train_ind:]
    y_train = y[0:train_ind]
    y_test = y[train_ind:]

    #Convertion to a DMatrix
    X_train_DM = xgb.DMatrix(X_train.values, label=y_train.values)

    # Parameters and Model training
    param = {'objective': 'reg:squarederror'}

    t0 = time.time()

    bst = xgb.train(param, X_train_DM)

    t1 = time.time()

    print(f'[INFO] Computation time for training the model: {t1-t0}s')
    return bst, X_train, y_train, X_test, y_test


from datetime import datetime
def save_model(model, model_name):
    io.save_obj(model, model_name + datetime.now().strftime("%H%M%S"))


import cuml.metrics.regression as cuml_metrics
import sklearn.metrics as skl_metrics
# Prediction and test

def pred_test(model, X_test, y_test):
    #Convertion to a DMatrix
    X_test_DM = xgb.DMatrix(X_test.values, label = y_test.values)

    # Predictions
    prediction = model.predict(X_test_DM)

    # Evaluation metrics computation
    max_error = skl_metrics.max_error(y_test.to_array(), prediction)
    variance = cuml_metrics.r2_score(y_test.to_array(), prediction)
    mean_absolute_error = cuml_metrics.mean_absolute_error(np.float64(y_test.to_array()), np.float64(prediction))
    mean_squared_error = cuml_metrics.mean_squared_error(np.float64(y_test.to_array()), np.float64(prediction))
    root_mean_squared = np.sqrt(cuml_metrics.mean_squared_error(np.float64(y_test.to_array()), np.float64(prediction)))

    print('Max Error:', max_error)
```

```python
    print('R^2:', variance)
    print('MAE:', mean_absolute_error)
    print('MSE:', mean_squared_error)
    print('RMSE:', root_mean_squared)

    # Metrics
    gdf_metrics = cudf.DataFrame({'Max Error' : max_error,
                                  'Explained Variance': variance,
                                  'Mean absolute error': mean_absolute_error,
                                  'Mean squared error': mean_squared_error,
                                  'Root mean Squared error': root_mean_squared})

    io.save_obj(gdf_metrics, '../outputs/Metrics/Rapids_model_metrics'+ datetime.now().
    strftime("%H%M%S"))
    return gdf_metrics


def pipeline_rapids():
    t0 = time.time()
    # Read data
    df = import_data("../../data/2009-01")

    df = preprocessing(df)
    # Training (serial function, but XGBR training is parallel)
    model, X_train, y_train, X_test, y_test = training(df.drop('fare_amt', axis = 1), df['
    fare_amt'])

    # Save the model (serial)
    save_model(model, '../outputs/Models/Rapids_model')

    # Predict and model metrics
    df_metrics = pred_test(model, X_test, y_test)

    t1 = time.time()
    print(f'[INFO] Computation time to perform the pipeline: {t1-t0}s')
    return df_metrics

pipeline_rapids()

%load_ext line_profiler

%lprun -f pipeline_rapids pipeline_rapids()


import yappi

yappi.clear_stats()

yappi.start()
import_data("../../data/2009-01")
yappi.stop()

stats_import_data = yappi.get_func_stats(filter_callback=lambda x: x.ttot>2.5)#.print_all()
stats_import_data.save('../outputs/Yappi/ystat.import_data' + datetime.now().strftime("%H%M%S"
    ) + '.prof', type = "ystat")

#stats.save('../outputs/Dask/callgrind.preproc.' + datetime.now().isoformat(), 'callgrind')


import yappi

df = import_data("../../data/2009-01")

yappi.clear_stats()

yappi.start()
df = preprocessing(df)
yappi.stop()

stats_preprocessing = yappi.get_func_stats(filter_callback=lambda x: x.ttot>2.5)#.print_all()
stats_preprocessing.save('../outputs/Yappi/ystat.preprocessing' + datetime.now().strftime("%H%
    M%S") + '.prof', type = "ystat")

#stats.save('../outputs/Dask/callgrind.preproc.' + datetime.now().isoformat(), 'callgrind')
```

```python
import yappi

yappi.clear_stats()

yappi.start()
model, X_train, y_train, X_test, y_test = training(df.drop('fare_amt', axis = 1), df['fare_amt'])
yappi.stop()

stats_training = yappi.get_func_stats(filter_callback=lambda x: x.ttot>2.5)#.print_all()
stats_training.save('../outputs/Yappi/ystat.training'+ datetime.now().strftime("%H%M%S") + '.prof', type = "ystat")

#stats.save('../outputs/Dask/callgrind.preproc.' + datetime.now().isoformat(), 'callgrind')


import yappi

yappi.clear_stats()

yappi.start()
save_model(model, 'Rapids_model_yappi')
yappi.stop()

stats_save_model = yappi.get_func_stats(filter_callback=lambda x: x.ttot>0.1)#.print_all()
stats_save_model.save('../outputs/Yappi/ystat.save_model' + datetime.now().strftime("%H%M%S") + '.prof', type = "ystat")

#stats.save('../outputs/Dask/callgrind.preproc.' + datetime.now().isoformat(), 'callgrind')


import yappi

yappi.clear_stats()

yappi.start()
pred_test(model, X_test, y_test)
yappi.stop()

stats_prediction = yappi.get_func_stats(filter_callback=lambda x: x.ttot>0.5)#.print_all()
stats_prediction.save('../outputs/Yappi/ystat.prediction' + datetime.now().strftime("%H%M%S") + '.prof', type = "ystat")

#stats.save('../outputs/Dask/callgrind.preproc.' + datetime.now().isoformat(), 'callgrind')


import yappi

yappi.clear_stats()

yappi.start()
pipeline_rapids()
yappi.stop()

stats_pipeline = yappi.get_func_stats(filter_callback=lambda x: x.ttot>10)#.print_all()
stats_pipeline.save('../outputs/Yappi/ystat.pipeline' + datetime.now().strftime("%H%M%S") + '.prof', type = "ystat")

#stats.save('../outputs/Dask/callgrind.preproc.' + datetime.now().isoformat(), 'callgrind')
```