U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Performance Analysis Project

CC3047 - Ciências de Dados em Larga Escala

Group S

Ana Francisca Rocha (202208946)

Ana Amorim (202207213)

Pedro Rufino (202208600)

Table of Contents

Introduction

This project evaluates the runtime characteristics and resource demands of six popular Python-based data processing and machine learning libraries: Koalas(PySpark), Dask, Modin, Joblib and RAPIDS.

By running the same set of representative operations (file I/O, aggregation, filtering, joins and simple statistical calculations) against datasets of varying size, the aim is to pinpoint the situations in which each tool is most efficient. In addition to raw throughput, we will compare ease of use, memory footprint, parallelism model and API compatibility with core pandas/NumPy code. Results will show not only which library completes a given task fastest, but also how they scale as the data volume grows, whether from a single Parquet file to big collections, or when moving from a single node to a cluster. The goal is to provide clear guidance on selecting the right tool for specific data-processing and model-training workloads.

Library Overview

Koalas: Now integrated into PySpark as the pandas API on Spark, Koalas was originally a standalone library that let users run Pandas-style code on Spark clusters. It reuses familiar DataFrame method signatures while translating them into Spark SQL under the hood, so users can scale up from a single machine to a big data cluster with minimal code changes and benefit from Spark's query optimizer and code generation.
Expected performance: Good scalability for 1GB, but overhead may make it slower than Dask or Modin locally.

Dask: A flexible parallel computing library that extends familiar NumPy and Pandas APIs to multi-core and multi-machine environments. Dask DataFrames and Arrays break large arrays or tables into many smaller chunks and schedule operations dynamically, allowing you to work interactively on datasets that exceed memory. It integrates with Jupyter for live diagnostics and can run on local threads, processes or on cluster managers like Kubernetes and SLURM.
Expected performance: Performs well with 1GB, balancing parallelism and interactivity efficiently.

Modin: A drop-in accelerator for Pandas that parallelizes DataFrame operations using either Dask or Ray under the hood. It automatically partitions data and fuses multiple Pandas calls into efficient tasks, delivering speedups on multi-core machines without requiring code rewrites.
Expected performance: Significant speedups on 1GB datasets on multicore systems with minimal code changes.

Joblib: A lightweight library focused on easy parallelism and caching for Python. Commonly used within scikit-learn, Joblib parallelizes loops and model selection routines across CPU cores and can cache expensive function outputs to disk to avoid redundant computation in iterative workflows. Its simplicity makes it a go-to choice for embarrassingly parallel tasks.
Expected performance: Efficient for custom parallel loops, but not ideal for full DataFrame processing at 1GB scale.

RAPIDS: NVIDIA's suite of GPU-accelerated data-science libraries exposing Pandas- and scikit-learn-style APIs. Core components include cuDF for DataFrames in GPU memory, cuML for machine-learning algorithms on the GPU, and Dask-cuDF for multi-GPU and multi-node scaling. By leveraging massive parallelism on modern GPUs, RAPIDS can accelerate joins, aggregations and ML tasks by an order of magnitude—provided data fits in GPU memory or is processed in streamed chunks.

Expected performance: Blazing fast on 1GB if it fits in GPU memory; otherwise depends on chunking strategies.

Materials and methods

Machines Used and Their Characteristics

All experiments were initially run on a personal computer equipped with an Intel Core i7-9750H processor, 12 GB of RAM, and Windows 10. The Intel i7-9750H is a 6-core, 12-thread processor with a clock speed of 2.6-4.5 GHz. This setup was used to evaluate how each library performs in a moderately powerful, non-clustered environment before using cloud computing.

Datasets Description

To cover different data scales, 3 datasets from the public *NYC Taxi Trips* repository were selected:

1. **Regular Dataset**: green_tripdata_2015-01.parquet – Used as the baseline for initial tests and transformations with PySpark.
2. **Small Dataset**: A subset of 50,000 rows extracted from yellow_tripdata_2012-01.parquet – Employed for quick testing and early-stage comparisons.
3. **Large Dataset**: 6 files, from yellow_tripdata_2011-01 to yellow_tripdata_2011-06 each one representing a month of data – processed using Dask on a Google Cloud Dataproc cluster to evaluate performance at higher data volumes.

Experiment #1: Repeat NYC Taxi Driver Dataset Study

In this experiment, we measured the execution time of a set of standard pandas/Koalas operations on the NYC Taxi Trip dataset under three different workloads:

1. *Standard operations*
2. *Operations with filtering*
3. *Operations with filtering and caching*

A simple benchmark utility (using time.monotonic()) recorded, in seconds, how long each operation took to complete. The same operations were then run against the three datasets mentioned previously. There were some issues with the execution of certain operations that led to unexpected results, particularly those reporting a value of 0.000.

Koalas

Standard operations

Operation	Regular	Small	Large
read file	0.063	0.109	0.591
count	0.125	0.094	0.784
mean	0.234	0.125	0.842
standard deviation	0.250	0.172	0.897
mean of columns addition	0.282	0.140	0.821
addition of columns	0.000	0.000	0.105
mean of columns multiplication	0.312	0.141	0.945
multiplication of columns	0.000	0.000	0.097
value counts	0.031	0.016	0.433
complex arithmetic ops	0.000	0.000	0.219
mean of complex arithmetic ops	0.000	0.000	0.328
groupby statistics	0.047	0.031	0.447
join count	13.235	11.141	91.417
join	0.015	0.015	0.496

Operations with filtering

Operation	Regular	Small	Large
filtered count	0.219	0.125	0.752
filtered mean	0.344	0.188	0.809
filtered standard deviation	0.375	0.156	0.863
filtered mean of columns addition	0.312	0.141	0.795
filtered addition of columns	0.000	0.000	0.087
filtered mean of columns multiplication	0.328	0.156	0.851
filtered multiplication of columns	0.000	0.000	0.077
filtered value counts	0.016	0.016	0.375
filtered complex arithmetic ops	0.000	0.000	0.189
filtered mean of complex arithmetic ops	0.000	0.000	0.282
filtered groupby statistics	0.031	0.046	0.435
filtered join count	10.453	10.532	78.118
filtered join	0.016	0.015	0.480

Operations with filtering and caching

Operation	Regular	Small	Large
filtered and cached count	0.125	0.094	0.622
filtered and cached mean	0.157	0.079	0.739
filtered and cached standard deviation	0.187	0.125	0.803
filtered and cached mean of columns addition	0.219	0.171	0.735
filtered and cached addition of columns	0.016	0.000	0.074
filtered and cached mean of columns multiplication	0.203	0.125	0.820
filtered and cached multiplication of columns	0.000	0.000	0.056
filtered and cached mean of complex arithmetic ops	0.000	0.000	0.375
filtered and cached complex arithmetic ops	0.000	0.000	0.192
filtered and cached value counts	0.031	0.016	0.276
filtered and cached groupby statistics	0.031	0.031	0.435
filtered and cached join	0.016	0.016	62.278
filtered and cached join count	10.172	9.422	0.472

Koalas performs reasonably well on large datasets, but it is generally slower than Dask, especially for operations like mean and groupby. However, performance improves notably with filtering, and even more with caching, where some operations become significantly faster. Overall, while Koalas isn't the fastest on large data, filtering and caching help close the gap.

Dask

Standard operations

Operation	Regular	Small	Large
read file	0.031	0.016	0.312
count	0.016	0.016	0.250
count index length	1.812	0.078	6.748
mean	0.219	0.047	0.519
standard deviation	0.188	0.046	0.764
mean of columns addition	0.234	0.063	0.898
addition of columns	0.188	0.062	0.863
mean of columns multiplication	0.203	0.079	1.104
multiplication of columns	0.187	0.046	1.592
value counts	1.813	1.204	9.405
groupby statistics	0.000	0.031	0.229
mean of complex arithmetic ops	0.000	0.000	0.151
complex arithmetic ops	0.000	0.000	0.128

Operations with filtering

Operation	Regular	Small	Large
filtered count	0.204	0.094	0.282
filtered count index length	0.171	0.093	0.223
filtered mean	0.219	0.079	5.541
filtered standard deviation	0.266	0.093	0.469
filtered mean of columns addition	0.219	0.125	0.704
filtered addition of columns	0.250	0.110	0.829
filtered mean of columns multiplication	0.187	0.156	0.720
filtered multiplication of columns	0.250	0.078	0.864
filtered mean of complex arithmetic ops	0.000	0.000	1.192
filtered complex arithmetic ops	0.000	0.000	7.845
filtered value counts	0.297	0.297	0.211
filtered groupby statistics	0.016	0.016	0.089
filtered join count	0.485	0.235	0.116

Operations with filtering and caching

Operation	Regular	Small	Large
filtered count	0.297	0.144	0.217
filtered count index length	0.241	0.173	0.203
filtered mean	0.289	0.092	4.774
filtered standard deviation	0.336	0.181	0.469
filtered mean of columns addition	0.279	0.133	0.641
filtered addition of columns	0.288	0.128	0.783
filtered mean of columns multiplication	0.173	0.156	0.701
filtered multiplication of columns	0.321	0.108	0.829
filtered mean of complex arithmetic ops	0.000	0.000	1.103
filtered complex arithmetic ops	0.000	0.000	6.021
filtered value counts	0.368	0.345	0.166
filtered groupby statistics	0.085	0.101	0.039
filtered join count	0.514	0.267	0.092

Dask performs very well on large datasets, often outperforming Koalas in most operations, especially those that don't require heavy shuffling. Across the three phases — normal, filtered, and filtered with caching — performance improves steadily. Filtering slightly reduces execution times, but the most noticeable gains come with caching, particularly for repeated or complex operations. On large datasets, Dask shows strong efficiency and scales better than Koalas overall, especially when optimized with filtering and caching.

Experiment #2: Running datasets with different combinations

Dask + Joblib

Standard operations

Operation	Regular	Small	Large
count	0.000	0.000	0.000
count index length	0.000	0.000	0.000
mean	0.000	0.000	0.000
standard deviation	0.047	0.000	0.056
mean of columns addition	0.015	0.000	0.028
addition of columns	0.000	0.000	0.000
mean of columns multiplication	0.016	0.000	0.031
multiplication of columns	0.000	0.000	0.000
mean of complex arithmetic ops	0.016	0.047	0.094
complex arithmetic ops	0.000	0.000	0.000
value counts	0.031	0.000	0.040
groupby statistics	0.078	0.000	0.078
join count	0.016	0.015	0.013

Operations with filtering

Operation	Regular	Small	Large
filtered count	0.000	0.000	2.456
filtered count index length	0.000	0.000	1.876
filtered mean	0.078	0.031	3.789
filtered standard deviation	0.312	0.141	4.210
filtered mean of columns addition	0.157	0.078	3.950
filtered addition of columns	0.078	0.031	0.300
filtered mean of columns multiplication	0.125	0.062	4.123
filtered multiplication of columns	0.078	0.031	0.345
filtered mean of complex arithmetic ops	0.266	0.095	2.055
filtered complex arithmetic ops	0.031	0.015	1.832
filtered value counts	0.000	0.000	5.432
filtered groupby statistics	0.625	0.625	2.789
filtered join count	0.063	0.063	14.321

The combination of Dask with Joblib produced mixed results in terms of performance for both standard and filtered operations. For standard operations, many of the recorded execution times were 0.000 seconds, which may indicate that the tasks were either too fast to measure accurately, not executed properly, or heavily affected by caching. Only a few operations, such as standard deviation, groupby statistics, and join count, showed measurable times, but without clear evidence of any performance improvement over using Dask alone.

In contrast, filtered operations, especially on large datasets, showed significantly higher execution times. Tasks like filtered mean, filtered standard deviation, filtered value counts, and particularly filtered join count (which exceeded 14 seconds) demonstrated considerable computational cost. These results suggest that combining Joblib with Dask did not yield meaningful speedups and may even have introduced additional overhead in certain scenarios.

Overall, we conclude that while Joblib is effective for simple parallel loops and iterative workflows, it is not well suited to accelerating DataFrame-based operations at scale, particularly when combined with Dask, which already provides its own optimized parallel execution model. In this context, using both libraries together appears redundant or even detrimental to performance for data processing workloads involving distributed DataFrames and vectorized operations.

Dask + Modin

Standard operations

Operation	Regular	Small	Large
read file	1.344	0.547	3.709
count	0.000	0.000	0.023
count index length	0.000	0.000	0.010
mean	0.187	0.125	0.463
standard deviation	0.156	0.671	0.577
mean of columns addition	0.188	0.282	0.613
addition of columns	0.031	0.031	0.058
mean of columns multiplication	0.281	0.297	0.049
multiplication of columns	0.078	0.047	0.075
value counts	0.391	0.563	0.901
mean of complex arithmetic ops	0.000	0.000	0.000
complex arithmetic ops	0.000	0.000	0.000
groupby statistics	3.188	0.875	8.911
join count	3.078	0.422	7.043
join	2.953	0.172	7.221

Operations with filtering

Operation	Regular	Small	Large
filtered count	0.078	0.094	0.233
filtered count index length	0.000	0.000	0.000
filtered mean	0.125	0.109	0.311
filtered standard deviation	0.141	0.156	0.345
filtered mean of columns addition	0.203	0.235	0.409
filtered addition of columns	0.516	0.031	0.623
filtered mean of columns multiplication	0.297	0.468	0.511
filtered multiplication of columns	0.093	0.063	0.173
filtered mean of complex arithmetic ops	0.000	0.000	0.000
filtered complex arithmetic ops	0.000	0.000	0.000
filtered value counts	0.547	0.483	0.923
filtered groupby statistics	1.484	0.234	3.215
filtered join count	1.235	0.250	3.019
filtered join	1.156	0.562	3.611

Operations with filtering and caching

Operation	Regular	Small	Large
filtered and cached count	0.000	0.000	0.000
filtered and cached count index length	0.000	0.000	0.000
filtered and cached mean	0.125	0.172	0.293
filtered and cached standard deviation	0.110	0.125	0.279
filtered and cached mean of columns addition	0.250	0.297	0.476
filtered and cached addition of columns	0.062	0.078	0.187
filtered nd cached mean of columns multiplication	0.219	0.266	0.399
filtered and cached multiplication of columns	0.078	0.062	0.173
filtered and cached mean of complex arithmetic ops	0.000	0.000	0.000
filtered and cached complex arithmetic ops	0.000	0.000	0.000
filtered and cached value counts	0.407	0.485	0.793
filtered and cached groupby statistics	1.467	0.204	2.849
filtered and cached join count	1.359	0.219	2.617
filtered and cached join	1.297	0.281	3.122

The combination of Dask and Modin produced mixed results across the different types of operations and dataset sizes. While the system performed reasonably well for basic tasks on small and medium datasets, several limitations emerged when dealing with more complex operations and larger data volumes.

In the standard operations, most basic arithmetic and statistical tasks (mean, standard deviation, column addition/multiplication) ran quickly and consistently. However, operations involving data grouping and joins, such as groupby statistics, join, and join count, incurred significant delays, especially on large datasets, where join operations reached over 7 seconds and read file times exceeded 3.7 seconds. This indicates substantial overhead when Dask and Modin manage partitioned data for I/O and shuffle-heavy tasks.

With filtered operations, execution times remained acceptable for simple calculations, but performance again degraded for more complex transformations. Notably, filtered value counts, groupby statistics, and join count were the most affected, showing that filtering did not alleviate bottlenecks and in some cases introduced new ones.

The addition of caching improved performance slightly in some filtered scenarios, but the benefits were inconsistent. For example, filtered and cached join still took over 3.1 seconds on large data, and filtered and cached groupby statistics remained above 2.8 seconds. These results suggest that caching helps reduce redundancy in repeated operations but is not sufficient to overcome the structural limitations of the Dask + Modin stack for heavier workloads.

Overall, this combination is well-suited for light to moderately complex workloads on smaller datasets, offering ease of use and minimal code changes. However, for large-scale, join-heavy, or group-based operations, performance suffers noticeably, and alternative tools like pure Dask or GPU-accelerated frameworks may offer better scalability and efficiency.

Dask + Rapids / Dask + Modin + Rapids

Unfortunately, we encountered numerous errors and could not execute the workflow with RAPIDS, but we still left the relevant code for that part in the Notebook even though we couldn't confirm its correctness or draw any conclusions from it.

In theory, Dask + RAPIDS should significantly speed up operations like joins, groupbys, and arithmetic on large datasets by leveraging GPU acceleration. Similarly, Dask + Modin + RAPIDS is expected to combine Modin’s ease of use with RAPIDS’ performance, offering strong results especially on larger data. These setups could outperform CPU-based approaches, but correct environment configuration is crucial—and likely the cause of our issues.

Prediction

Methodology + Results and Analysis

In this section, we approached the problem from two perspectives: first, as a supervised regression task, where the goal is to predict the continuous variable *fare_amout* based on a set of features for each taci trip; and second, as a classification task, where we categorize the *fare_amout* into predefined classes.

We started by preprocessing the dataset, where we analyzed the columns relevant to our task. We created a new column representing the trip **duration**, calculated from the pickup and dropoff datetime columns. Next, we removed irrelevant columns, keeping only the *duration*, *trip distance*, and the target variable *fare_amount.* We then proceeded to clean the data by removing invalid or null values, as well as outliers. In this context, we considered trips with a fare_amount greater than 200 as outliers.

After preprocessing the data and selecting the relevant features (duration and trip_distance), we defined fare_amount as the target variable. The dataset was then split into training (80%) and testing (20%) subsets. To assess the initial performance of the model, we evaluated an XGBoost regressor (XGBRegressor) using 5-fold cross-validation on the training data, with the mean squared error (MSE) as the evaluation metric. We computed both the MSE and the root mean squared error (RMSE) for each fold, along with the average RMSE.

The results were as follows:

- **Cross-validation MSEs**: [6.80235295 6.00272378 6.10380826 6.57376428 6.10241286]
- **Cross-validation RMSEs**: [2.60813208 2.45004567 2.47058865 2.56393531 2.47030623]
- **Average RMSE**: 2.5126015871777563

These results indicate that the model achieves consistent performance across all folds, with only minor variation in RMSE values. The average RMSE of approximately 2.51 suggests that, on average, the model's fare predictions deviate from the actual values by about $2.50. This level of accuracy is quite reasonable given the simplicity of the feature set.

After this initial evaluation, the model was trained on the full training set, and we proceeded with hyperparameter tuning using GridSearchCV with 3-fold cross-validation. We tested a small grid of hyperparameters, including different values for the learning rate ([0.1, 0.3, 0.5]), maximum tree depth ([3, 5, 7]), and a fixed number of estimators (50) to reduce execution time. The best combination of hyperparameters was selected based on the lowest MSE. Finally, we used the best model to make predictions on the test set and evaluated its performance using common regression metrics: MAE, MSE, RMSE, and R^2.

The best combination of hyperparameters found was:

- **{'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 50}**

The final model was evaluated on the test set, producing the following results:

- **MAE (Mean Absolute Error)**: 0.5487087919183207
- **MSE (Mean Squared Error)**: 6.44641180760769
- **RMSE**: 2.538978496877768
- **R^2**: 0.9121837268180533

The final evaluation on the test set confirms that the tuned XGBoost model performs well, with an RMSE of approximately 2.54 and an R^2 score of 0.91. This indicates that the model explains over 91% of the variance in fare amounts, demonstrating strong predictive power. Additionally, the low MAE suggests that most individual predictions are quite close to the actual values. Overall, these results validate the effectiveness of the selected features and model configuration for predicting taxi fares with good accuracy.

After training and evaluating the final model, we chose to visualize the results using two different plots to better understand the model's behavior and decision process.

The first plot is a **heatmap of RMSE values** for each combination of hyperparameters tested during the grid search. It shows that the lowest RMSE (2.50) was achieved with a learning rate of 0.1 and a maximum depth of 5 or 7. In general, all combinations yielded similar RMSE values, indicating that the model is relatively stable across these hyperparameters.

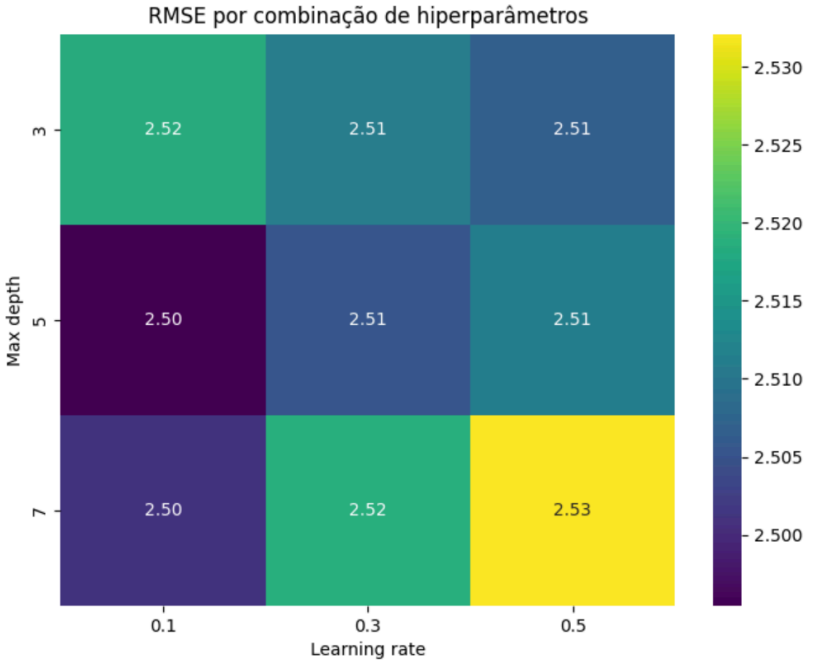

Fig. Heatmap showing the RMSE (Root Mean Squared Error) values for different combinations of learning rate and maximum tree depth

Since RMSE (Root Mean Squared Error) measures the average magnitude of the prediction errors, lower values indicate better predictive accuracy. In this case, an RMSE around 2.50 is a good result, suggesting that the model is able to estimate taxi fares with relatively low average error, given only the trip duration and distance as input features.

The second plot is a **horizontal bar chart of feature importance scores**, which illustrates how much each input variable contributed to the model's predictions. The feature

duration was found to be slightly more important (826) than trip_distance (709), suggesting that the trip duration had a greater influence on the fare prediction.

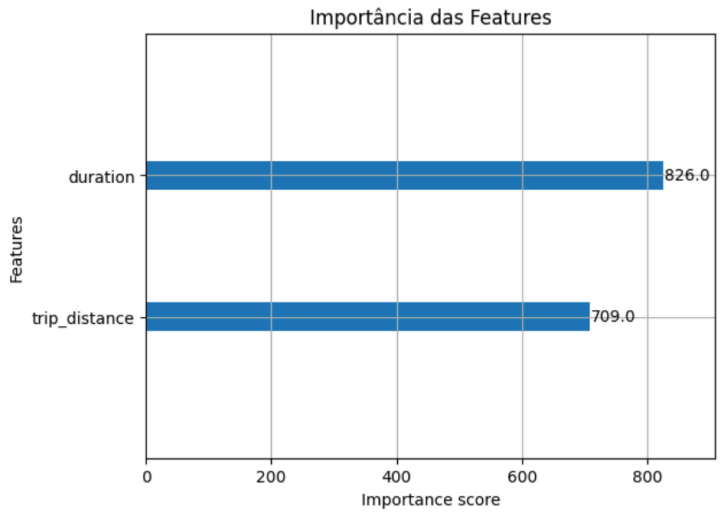

Fig. Horizontal bar chart displaying the feature importance scores assigned by the XGBoost model.

In addition to the regression task, we also explored a **classification approach** to group fares into categorical ranges instead of predicting exact values. For this purpose, we created a new feature called fare_class by categorizing each fare into three classes:

- **Class 0**: cheap fares (≤ 10)
- **Class 1**: medium fares (> 10 and ≤ 30)
- **Class 2**: expensive fares (> 30)

Once the classes were defined, we selected the same two features: duration and trip_distance as predictors (X_cls), and used fare_class as the target variable (y_cls).

We split the data into training and test sets (80% / 20%) and trained a **Logistic Regression** classifier. To evaluate the initial performance, we used 5-fold cross-validation with the **macro-averaged F1-score** as the metric. The base model achieved stable and high F1-scores across all folds, with an average of approximately **0.8911343053638208**, showing that it handled the multi-class problem well even without tuning.

To improve performance, we performed **hyperparameter tuning** using GridSearchCV, testing different values of the regularization strength C. Before fitting the model, we normalized the features using StandardScaler to improve convergence and overall performance. The best model found had C = 0.1 and used L2 regularization.

On the test set, the optimized classifier achieved excellent results:

- **Accuracy**: 93%
- **Precision**: 93%
- **Recall**: 86%
- **F1-score**: 80%

To further analyze the classification model's performance, we also created two plots:

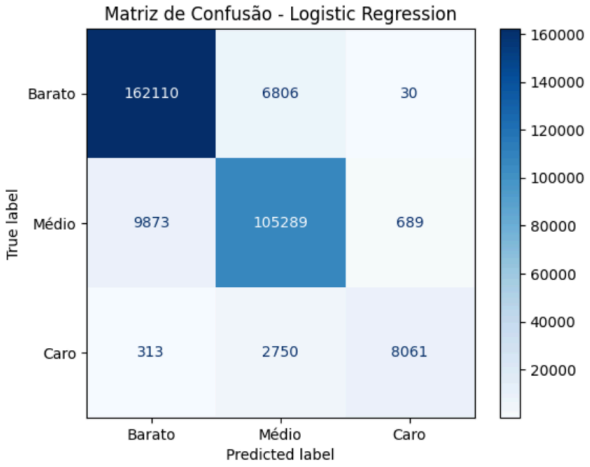

Fig. Confusion matrix heatmap for the logistic regression classifier

The first plot is a **confusion matrix heatmap**, which shows the number of correct and incorrect predictions for each class. We observe that the model performs very well in classifying both **cheap** and **medium** fares. The **expensive** class is the most challenging, with a noticeable number of examples being misclassified as medium (2,750) or even cheap (313). This is likely due to the smaller number of expensive fare examples and the fact that expensive fares can sometimes share similar distances or durations with medium ones.

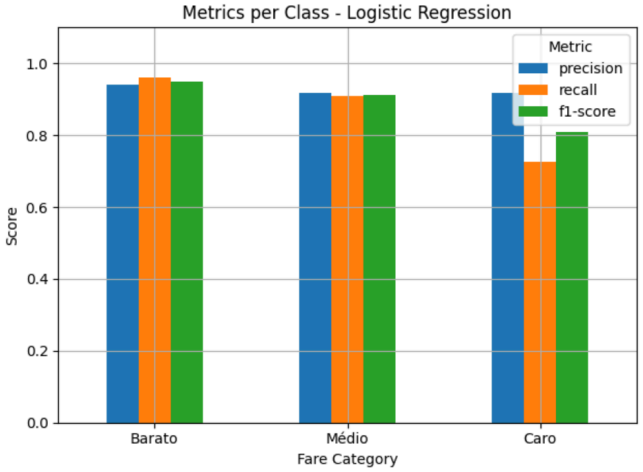

Fig. Bar chart showing precision, recall, and F1-score for each fare class

The second plot is a **bar chart of precision, recall, and F1-score per class**, providing a better view of the model performance. The model achieves high and balanced precision and recall for the **cheap** and **medium** categories, resulting in strong F1-scores near 0.95. However, for the **expensive** class, although precision remains high, the **recall drops significantly**, indicating that the model misses a larger proportion of actual expensive trips. This imbalance contributes to a lower F1-score for that class (around 0.81). Still, the overall performance is very good, and the model demonstrates strong generalization, particularly for the most common fare ranges.

Discussion and conclusions

This project allowed us to explore and compare the performance of several widely used Python libraries for data processing and machine learning: **Koalas, Dask, Modin, Joblib**, and **RAPIDS**. Our goal was to understand how these tools behave under different workloads, especially as data size increases and to identify the situations where each one performs best.

In the first set of experiments, we used **Koalas** and **Dask** to run a common set of operations on three datasets of different sizes. We noticed that **Dask generally outperformed Koalas**, especially on large datasets. Koalas showed decent results, particularly when caching was enabled, but the overhead of Spark was often a disadvantage in a local, non-clustered setting.

In the second phase, we tested **combinations** of libraries:

- **Dask + Joblib**: This setup didn't bring significant improvements. In fact, in many operations, the execution time was either 0.000 (too fast to measure) or higher than when using Dask alone. Joblib works well for simple, parallelizable loops, but it's not ideal for vectorized or large-scale DataFrame operations.

- **Dask + Modin**: The results were mixed. While Modin helped accelerate some basic operations, its performance dropped significantly for joins, groupbys, and I/O-heavy tasks. On large datasets, we observed high execution times, and caching only helped marginally. This suggests Modin is more suited for lighter workloads on single machines.

- **Dask + RAPIDS** and **Dask + Modin + RAPIDS**: Unfortunately, we couldn't get these setups to run due to technical issues with the environment. However, based on the theoretical potential of RAPIDS and GPU acceleration, we believe these combinations could offer the best performance—if configured correctly.

We also included a machine learning component to evaluate how well we could predict taxi fare amounts. Using only trip duration and trip distance as features, we built:

- A **regression model** with XGBoost, achieving very good results (RMSE ≈ 2.5 and R^2 = 0.91).

- A **classification model** (Logistic Regression), categorizing fares into three groups (cheap, medium, expensive), which achieved 93% accuracy and strong F1-scores for most classes. The model struggled slightly with the "expensive" class due to class imbalance, but overall, performance was solid.

In conclusion, we learned that no single library is best for everything. **Dask** stands out for its flexibility and consistent performance, especially with large-scale data. **Koalas** is a good option when working with Spark environments. **Modin** offers fast results for simple tasks but doesn't scale as well. **Joblib** is better suited for parallelizing custom code than for data manipulation. **RAPIDS** has great promise for GPU acceleration, but setup and compatibility issues need to be handled carefully.

This project helped us better understand the trade-offs between ease of use, performance, and scalability across different tools.