Lab 03: Machine Learning on Spark

May 2025

Table of Contents

1 Team's Contribution

StudentID	Member	Task	Contribution
22120210	Lê Võ Nhật Minh	Exercise 3.2	Low-Level Operations Implementation
		Report	Detailed algorithms & approaching for exercise 3.2.3
22120227	Nguyễn Hữu Nghĩa	Exercise 3.1	High-Level and MLlib RDD-Based Implementation
		Report	Detailed algorithms & approaching for exercise 3.1.1 and 3.1.2
22120262	Nguyễn Lê Tấn Phát	Exercise 3.2	High-Level and MLlib RDD-Based Implementation
		Report	Detailed algorithms & approaching for exercise 3.2.1 and 3.2.2
22120405	Nguyễn Thanh Tuấn	Exercise 3.1	Low-Level Operations Implementation
		Report	Detailed algorithms & approaching for exercise 3.1.3

2 Classification with Logistic Regression

2.1 Structured API Implementation

Objective

This section focuses on building a Logistic Regression model using Spark Structured API (`spark.ml`). The workflow includes:

- Appropriate data preprocessing.
- Combining features into a single vector.
- Normalizing data if necessary.
- Training the model with `LogisticRegression`.
- Evaluating results using metrics: accuracy, AUC, precision, recall.
- Splitting data into training and testing sets for objective assessment.

Implementation Process and Explanation

1. Reading and Inspecting the Data The dataset `creditcard.csv` is loaded, and the distribution of the Class label is checked (0 = legitimate, 1 = fraudulent). The data is highly imbalanced, with over 99% of transactions being legitimate and less than 0.2% fraudulent. Therefore, metrics like PR-AUC are more meaningful than accuracy alone.

2. Data Preprocessing

- Duplicate rows are checked and removed.
- No missing values are found in this dataset.
- The `Time` and `Amount` variables are normalized since their distributions differ from PCA features (V1–V28).

3. Feature Vector Construction All input features (Time, Amount, V1 to V28) are combined into a single `features` vector for use in machine learning models.

4. Train/Test Splitting The dataset is split into 80% training and 20% testing, using a fixed seed to ensure reproducibility.

5. Model Training A `LogisticRegression` model is trained using the `features` column. Using the Structured API allows chaining all steps into a `Pipeline`, simplifying maintenance and ensuring consistency.

6. Prediction and Evaluation The model is applied to the test set. The following evaluation metrics are calculated (example values shown):

Metric	Value	Comment
Accuracy	0.9992	High due to imbalance, not an accurate measure.
ROC-AUC	0.9850	Good separation between classes.
PR-AUC	0.7498	More meaningful for imbalanced data; fairly good result.
Precision	0.8800	88% of predicted frauds were actual frauds.
Recall	0.6346	Detected 63% of actual fraud cases.

Coefficients of the logistic regression model are briefly analyzed: positive coefficients increase fraud likelihood, while negative coefficients decrease it.

Analysis and Remarks

- **Model quality:** ROC-AUC and PR-AUC indicate that the model is effective despite data imbalance.
- **Efficiency:** The use of Structured API with a pipeline streamlines the process and suits large-scale datasets.

2.2 MLlib RDD-Based Implementation

Objective

This part implements logistic regression using Spark MLlib with RDD-based functions rather than DataFrames. This low-level approach offers a deeper understanding of Spark's parallel processing with RDDs and its distributed ML capabilities.

Implementation Process and Explanation

1. Reading and Preprocessing Data The original data is read from a CSV file into an RDD of text lines.

- The header is removed.
- Each line is parsed into a list of floats.
- Each entry becomes a `LabeledPoint`, where:
  - `Label` = Class column
  - `Features` = vector of Time, Amount, V1–V28

2. Train/Test Split The dataset is split into 80% training and 20% testing to ensure an objective model evaluation. This ratio is commonly used in machine learning to provide enough data for learning while reserving a portion for unbiased testing.

3. Model Selection and Comparison

- Two algorithms are tested: SGD and LBFGS.
- Different configurations are tried to optimize performance.
- Evaluation metrics include PR-AUC and F1-score.

Results with SGD:

Config	PR-AUC	F1	Comment
50 iters, step 10	0.7180	0.0524	Very low F1, unstable.
50 iters, step 15	0.7194	0.0692	Slightly better, but still poor.
50 iters, step 20	0.6932	0.0624	Worse PR-AUC and F1.
100 iters, step 10	0.7150	0.1200	Reasonable tradeoff.
100 iters, step 15	0.7173	0.1487	High F1, close to optimal.
100 iters, step 20	0.7200	0.1509	Highest F1.
200 iters, step 10	0.7200	0.0802	F1 dropped.
200 iters, step 15	0.7231	0.0830	Slight PR-AUC gain, still low F1.
200 iters, step 20	0.7227	0.0828	No clear improvement.

Observation: PR-AUC is stable but F1 is low → poor performance. Increasing iterations didn't help.

Results with LBFGS:

Iterations	PR-AUC	F1
50	0.4004	0.7816
100	0.4004	0.7816
200	0.4004	0.7816

Observation: Lower PR-AUC but much higher F1-score → LBFGS detects fraud better and is more stable.

Comparison Summary:

Algorithm	Best Config	PR-AUC	F1
SGD	100 iters, step 20	0.7200	0.1509
LBFGS	50 iters	0.4004	0.7816

Conclusion: LBFGS is selected due to significantly higher F1-score despite lower PR-AUC.

4. Prediction and Evaluation Model predictions are evaluated using accuracy, precision, recall, ROC-AUC, and PR-AUC.

Metric	Value	Comment
Accuracy	0.9993	High due to imbalance.
Precision	0.7727	77% of fraud predictions are correct.
Recall	0.7907	79% of real fraud cases detected.
ROC-AUC	0.9074	Good class separation.
PR-AUC	0.4003	Moderate; most relevant for imbalance.

Comparison with Structured API Performance is comparable. RDD-based methods require more manual steps, while Structured API simplifies the process via Pipelines. RDD offers more control, suitable for unstructured or custom workflows.

Summary

Advantages of RDD-Based Approach:

- Greater control and transparency.

- Flexibility in custom workflows.

Disadvantages:

- More complex and error-prone.
- Lacks built-in tools (e.g., Pipelines, CrossValidator).

Conclusion: RDD-based MLlib achieves similar performance to Structured API but requires more manual implementation. It's suited for custom, low-level control but not ideal for standard machine learning workflows.

2.3 Low Level Operations

Objective

This part manually implements logistic regression using fundamental RDD transformations instead of MLlib's high-level APIs. This low-level approach provides hands-on experience with Spark's distributed data handling and helps build a deeper understanding of how gradient descent and model training work at the core level.

Implementation Process and Explanation

1. Reading and formatting data Data is read from `creditcard.csv` using `SparkContext` and stored in RDD. RDD undergoes several transformations to format.

- Eliminate the header line.
- Split each line into a list of data.
- Remove the time column, as it is used to record the time transactions are carried out. It doesn't have any effect on the model at all.
- Convert all the data to float type.
- Separate features and the label. Each line is a tuple with the first item being the feature list and the second item being a single label.
- Extend each feature list to have an extra value `1.0` at the end of the list for the coefficient computation.
  - Feature: `V1-V28` columns
  - Label: `class` column

2. Preprocessing data Features are processed using Z-Score Normalization. The class implements this algorithm can be seen in `classification/preprocessor.py`. The process includes:

- Split RDD into train data set and test data set with a proportion of 80:20.
- Apply the Z-Score Normalization to the train data set.
  - Calculate mean and standard deviation for each column feature in parallel processing fashion. Specifically, it does this by aggregating the sum, sum of squares, and count of rows in a distributed way, then using those to calculate per-feature statistics.
  - For every value, use the mean and standard deviation of its corresponding feature to normalize it.
- Add another column into the original RDD indicating the normalized feature list. We intentionally do this to replicate the way high-level MLlib does.

3. Training the model Train the normalized train dataset using Logistic Regression. The class implements this algorithm can be seen in `classification/trainer.py`. The process includes:

- Extract the feature list and the label from the given RDD based on the configured indices. A new RDD with necessary data is created.
- Initialize a weight vector of all zeros.
- Compute `N`, the number of total samples.
- Create a simple list containing a number of processed data in one iteration.
- Cache RDD.
- Train the model for each iteration:
  - The gradient of the loss with respect to w for a single data point is:

$$\frac{\partial L}{\partial w} = (\hat{y}_i - y_i)x_i$$

    * $x_i \in \mathbb{R}^n$: the input feature vector.
    * $y_i \in \{0, 1\}$: the true binary label.
    * $\hat{y}_i \in (0, 1)$: the predicted probability.
    * $w \in \mathbb{R}^n$: the weight vector of the model.
  - For an iteration of `batch_size` samples, the update is the average of all computed gradients.
- Update the weights.
- Print progress every `checkpoint_interval` iterations.
- Stop early if the maximum number of iterations is reached or the change in weights is less than `tolerance`.
- Clean-up.
- Return the model.

Prediction and evaluation After getting the model, the test dataset is used to make prediction. A new column is added into RDD. This new RDD is then used for evaluation. The evaluation can be seen in `classification/evaluator.py`. It includes: accuracy, precision, recall and f1_score.

Results

Metric	Value	Comment
Accuracy	0.9988	High due to imbalance.
Precision	0.8636	86% of fraud predictions are correct.
Recall	0.3689	37% of real fraud cases detected.
F1 Score	0.5170	Moderate overall performance.
Time	85.76s	Acceptable training time.

Reason of choices

- **model: Logistic Regression**: Easy to implement.
- **learning_rate = 0.001**: Because of the skewed dataset with limited number of label `1`, small learning rate is more suitable to fit.

- **batch_size = 2048**: Using Spark on distributed file system, small batch size may not utilize the whole power of distributed system. A large enough batch size will later be divided further to worker nodes. This can increase efficiency.
- **num_iterations = 200**: After observations, the dataset is not so large and the model is simple. Furthermore, it takes time to train the model. Hence, 200 is balance enough between time and efficiency.

Weakness The model result may change remarkably in every run due to randomness in training section.

Prediction comparison with Structured API and MLlib RDD-based The performance of the low-level RDD implementation of logistic regression is compared with the Structured API and MLlib RDD-based implementations to assess their relative strengths and weaknesses. The comparison focuses on evaluation metrics, computational efficiency, and implementation complexity.

Implementation	Accuracy	Precision	Recall	F1 Score
Structured API	0.9992	0.8800	0.6346	0.7373
MLlib RDD (LBFGS)	0.9993	0.7727	0.7907	0.7816
Low-Level RDD	0.9988	0.8636	0.3689	0.5170

- **Structured API:** Achieves high precision (88%) and moderate recall (63.46%), with a strong F1 score (0.7373). The use of Pipelines simplifies preprocessing and training, making it efficient and scalable for large datasets. PR-AUC (0.7498) and ROC-AUC (0.985) indicate robust performance on imbalanced data.
- **MLlib RDD (LBFGS):** Offers the highest recall (79.07%) and F1 score (0.7816), making it effective at detecting fraud cases despite a lower PR-AUC (0.4003). The RDD-based approach provides flexibility but requires manual feature engineering and preprocessing.
- **Low-Level RDD:** Shows high precision (86.36%) but significantly lower recall (36.89%) and F1 score (0.5170), indicating poor detection of fraud cases. The manual implementation of gradient descent and normalization is computationally intensive (85.76s training time) and less optimized for imbalanced data.

The Structured API balances performance and ease of use, making it ideal for standard machine learning workflows. MLlib RDD with LBFGS excels in recall and F1 score, suitable for scenarios prioritizing fraud detection over precision. The low-level RDD approach underperforms due to limited recall, likely caused by the simple gradient descent implementation and sensitivity to the imbalanced dataset. However, it provides valuable insight into Spark's distributed computing mechanics.

Summary

Advantages of Low-Level RDD:

- **Flexibility:** Allows complete control over preprocessing (e.g., Z-Score normalization), model training (e.g., custom gradient descent), and evaluation, enabling tailored implementations for non-standard workflows.
- **Transparency:** Exposes every step of the computation, making it easier to debug and understand the mechanics of logistic regression. regression in a distributed environment.

Disadvantages of Low-Level RDD:

- **Poor Performance**.

- **Complexity and Error-Prone:** Manual implementation of preprocessing, training, and evaluation increases coding effort and the risk of errors, especially in distributed settings.
- **Inefficiency:** Less optimized than high-level APIs, which leverage built-in optimizations like Pipelines and advanced optimizers.

Conclusion The low-level RDD implementation of logistic regression provides strengths in its flexibility, transparency, and educational potential, making it an excellent tool for understanding how logistic regression operates in a distributed environment. However, its significant drawbacks—poor performance, high implementation complexity, and lack of built-in optimizations—render it impractical for real-world fraud detection compared to Spark's Structured API or MLlib RDD-based approaches.

3 Regression with Decision Trees

This task aims to build decision trees on big data to predict the duration of each trip in the test set.

3.1 Explore data

The training dataset consists of 1,458,644 taxi trips in New York City, including information such as pickup/dropoff timestamps, coordinates, passenger count, and trip duration (`trip_duration`). After loading the data into a Spark DataFrame and inspecting the schema, we confirmed that none of the columns contained missing values.

The main data analysis steps include:

- **Descriptive statistics:**
  - `trip_duration`: average trip duration is approximately 959 seconds (16 minutes), but the distribution is heavily right-skewed, with a maximum value over 3.5 million seconds (40 days), which is clearly an outlier. 75% of trips have durations under 1,075 seconds (18 minutes).
  - `passenger_count`: ranges from 0 to 9; the median is 1. Trips with 0 passengers raise questions about data validity.
  - `pickup_longitude`, `pickup_latitude`, `dropoff_longitude`, `dropoff_latitude`: many records contain coordinates outside the expected range for New York City (longitude $<$ -121.93 or latitude $>$ 51.88), while typical NYC coordinates are around longitude $\approx$ -74, latitude $\approx$ 40.7.
- **Feature engineering:** We computed the distance between pickup and dropoff points using the `geopy` library (`great_circle` function), and stored the result in a new column `distance(m)`. While the average trip distance was reasonable, we detected several cases with zero or anomalous distances.
- **Speed calculation:** Using `distance(m)` and `trip_duration`, we calculated trip speed (km/h). Some trips had extremely high or low speeds, suggesting data recording errors.
- **Data visualization (for more details, please refer to `DataExplore.ipynb`):**
  - Distribution of `log10(trip_duration)`: using a histogram (log-scale on the x-axis, square-root scale on the y-axis), we observed most trips clustered around `log10(trip_duration)` $\approx$ 15 (about 900 seconds).
  - Trips with `trip_duration` $>$ 1 day: upon inspection, most of these trips had negligible travel distance, indicating possible data issues.
  - Trips under 5 minutes: we examined their speed and distance, finding some trips with abnormally high speeds.

  – Trips with `distance(m)` = 0: several instances also had very long durations (over 1 hour), suggesting they should be removed or flagged.

For more details, please refer to `DataExplore.ipynb`

3.2 Structured API Implementation

In this section, we demonstrate the use of PySpark's Structured API to perform regression using a Decision Tree model. The process involves data pre-processing, model training with parameter tuning, tree structure analysis, and evaluation with regression metrics.

Data Pre-processing: We first clean the data out of outlier using log transformation and MAD on `trip_duration` and `speed_kmh`. Because we know from our Data Exploration, those features is the most problematic.

We first assembled the dataset's numeric input columns into a single feature vector using `VectorAssembler`. This transformation is required because PySpark's machine learning models expect a single vector column as input features. The target label for regression is the `trip_duration` column.

Model Training and Parameter Tuning: We utilized `DecisionTreeRegressor` to train a regression model. Key hyperparameters were carefully selected to control the model's complexity:

- `maxDepth = 10`: Limits the depth of the tree to prevent overfitting.
- `minInstancesPerNode = 20`: Ensures each split contains sufficient training examples.
- `impurity = "variance"`: Uses variance reduction as the impurity measure for regression.

The tree was trained as part of a `Pipeline` which also included feature indexing for automatic categorical feature handling.

Model Interpretation: After training, we extracted the decision tree model from the pipeline to analyze its internal structure:

- Tree Depth: `10`
- Number of Nodes: 1997 (as printed from `tree_model.numNodes`)
- Feature Importances:

```
- vendor_id: 0.00
- passenger_count: 0.00
- distance_m: 0.88
- pickup_hour: 0.10
- pickup_dayofmonth: 0.01
- pickup_month: 0.01
- store_and_fwd_flag_index: 0.00
```

These importances indicate how influential each input feature is in determining the final prediction.

Model Evaluation: The trained model was evaluated on a held-out test dataset using standard regression metrics:

- **RMSE (Root Mean Squared Error)**: Measures average prediction error magnitude.

- **R² (Coefficient of Determination)**: Indicates how well the model explains data variance.

```
RMSE = 4921.696555070559
R² = 0.010458172159521717
```

These results suggest that the decision tree model is barely accurate and captures the underlying patterns of the dataset.

Conclusion: By using PySpark's high-level Structured API, we successfully built and interpreted a decision tree regression model.

For more details, please refer to `Structureed_API.ipynb`

3.3 MLlib RDD-based implementation

In this section, we implement a regression model using the RDD-based API provided by MLlib, PySpark's lower-level machine learning library. The workflow mirrors that of the Structured API version but requires more manual control over data transformation and model construction.

Data Preparation: We first clean the data out of outlier like we did in the Implementation of Structured API.

After that, we converted the dataset into an RDD of `LabeledPoint` objects. This transformation involves mapping each row to a `LabeledPoint(label, features)` structure, where the label corresponds to `trip_duration` and the features are a dense vector of selected numeric input columns (e.g., `distance_m`, `pickup_hour`, etc.).

Model Training: We trained a regression decision tree using `DecisionTree.trainRegressor()` from MLlib. Key parameters were set to reflect those used in the Structured API implementation:

- `categoricalFeaturesInfo = {}`: All features are treated as continuous.
- `impurity = "variance"`: Consistent impurity measure for regression.
- `maxDepth = 10`, `minInstancesPerNode = 20`: Control overfitting and ensure robustness.

Model Evaluation: The trained model was evaluated on the test RDD using regression metrics. Predictions were compared to ground truth labels to calculate Root Mean Squared Error (RMSE) and R-squared (R^2):

```
RMSE = 3171.4570734151303
R² = 0.02449659530880599
```

These results indicate better predictive performance than the Structured API implementation.

Comparison and Insights: Compared to the Structured API, the RDD-based implementation yields a significantly lower RMSE (3171 vs. 4921) and a slightly higher R^2 (0.024 vs. 0.010). This improvement could be due to subtle differences in how the data pipeline is handled internally. For example, the Structured API's use of a Pipeline and feature indexing might introduce transformation artifacts or affect feature scaling, while the RDD approach processes raw features directly.

However, the RDD-based approach also lacks many conveniences such as automatic handling of categorical features, integration with cross-validation tools, and modular pipeline design, which the Structured API provides out-of-the-box.

Conclusion: The RDD-based implementation of decision tree regression achieved better accuracy metrics on the same dataset, albeit with more manual coding and less abstraction. While useful for educational purposes and fine-grained control, the Structured API remains the preferred option for maintainability and integration in modern ML workflows.

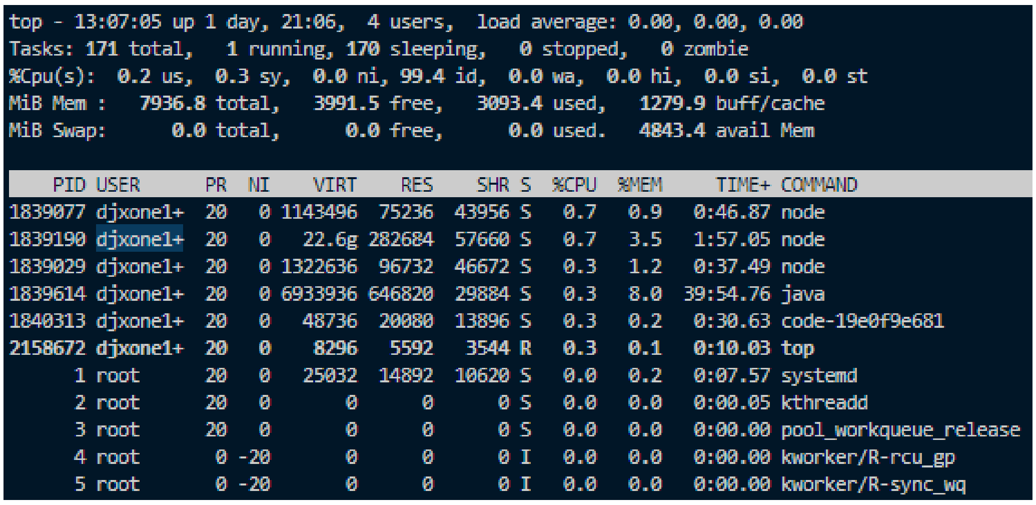
For more details, please refer to `MLlib_RDD_Based.ipynb`

3.4 Low-level Operations

- **Technical constraints:** When using low-level operations, resource utilization and optimization must be handled almost entirely manually. As a result, each machine configuration requires a different approach to resource management. This notebook is run on a machine with 8GB of memory and 8 vCPUs; the specific approach will be explained in more detail in the following sections.
- **Intermediate storage[1]:** is an extremely flexible storage mechanism in Spark. It supports various strategies for storing and reusing RDDs, thereby accelerating computation performance based on user needs. In this notebook, the chosen storage strategy is MEMORY_AND_DISK, which temporarily stores data in RAM and falls back to disk if memory overflows. This approach is necessary because low-level RDD operations such as filter, map, etc., use a lazy evaluation mechanism, meaning computations are not executed until an action is called.

  In many cases, a certain processing step or model training phase needs to be repeated multiple times on the same intermediate data. Without temporary storage, Spark would be forced to recompute the entire lineage each time due to its lazy evaluation model, resulting in unnecessary resource consumption and longer execution times. Using persist() with a safe strategy such as MEMORY_AND_DISK helps avoid redundant recomputations, especially in iterative machine learning algorithms or large-scale data pipelines.

  It is especially important to monitor memory usage when using persist within recursive functions (as in the case of the tree-building function). On Linux systems, the top command can be used to track memory consumption during code execution.



```
top - 13:07:05 up 1 day, 21:06,  4 users,  load average: 0.00, 0.00, 0.00
Tasks: 171 total,   1 running, 170 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.3 sy,  0.0 ni, 99.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   7936.8 total,   3991.5 free,   3093.4 used,   1279.9 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   4843.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
1839077 djxone1+  20   0 1143496  75236  43956 S   0.7   0.9   0:46.87 node
1839190 djxone1+  20   0   22.6g 282684  57660 S   0.7   3.5   1:57.05 node
1839029 djxone1+  20   0 1322636  96732  46672 S   0.3   1.2   0:37.49 node
1839614 djxone1+  20   0 6933936 646820  29884 S   0.3   8.0  39:54.76 java
1840313 djxone1+  20   0   48736  20080  13896 S   0.3   0.2   0:30.63 code-19e0f9e681
2158672 djxone1+  20   0    8296   5592   3544 R   0.3   0.1   0:10.03 top
      1 root      20   0   25032  14892  10620 S   0.0   0.2   0:07.57 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.05 kthreadd
      3 root      20   0       0      0      0 S   0.0   0.0   0:00.00 pool_workqueue_release
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-rcu_gp
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-sync_wq
```

Figure 1: top command window

- **Data Preparation:** Read the file line by line and extract the fields (separated by commas) necessary for training and model building. After processing, each line will be in the format: (list of features, target variable).

- **Model Training:**
  - **Node Structure in Decision Trees**

    The `Node` class represents a single node in a decision tree. Each node stores information about the splitting condition (i.e., which feature and threshold are used), along with its child nodes or a predicted value in the case of a leaf node.

    * **feature_index**: The index of the feature used for splitting at this node.
    * **threshold**: The split threshold — samples with values less than the threshold go to the left child, others go to the right.
    * **left, right**: References to the left and right child nodes.
    * **value**: The predicted output value if the node is a leaf.

    This structure forms the basis of a binary decision tree and supports recursive construction and prediction.
  - **Adaptive Quantile Computation**

    The adaptive quantile function determines a set of candidate split thresholds for a given feature, based on the size of the dataset.

    * The number of quantiles is determined dynamically using either the square root (`sqrt`) or the base-2 logarithm (`log`) of the dataset size.
    * All distinct values of the selected feature are extracted, sorted, and then sampled evenly to construct representative thresholds.
    * This technique limits the number of candidate splits, reducing computation and improving scalability when training decision trees over large datasets.

    The strategy is especially effective in Apache Spark, where minimizing data shuffling and full dataset scans significantly boosts performance.
  - **Finding the Best Split:** `find_best_split_rdd`

    This function identifies the best feature and threshold to split the current dataset based on minimizing the Mean Squared Error (MSE). It operates as follows:

    * For each feature, a set of candidate thresholds is computed using adaptive quantiles.
    * For every candidate threshold:
      · The dataset is virtually split into two groups: those less than or equal to the threshold (left) and those greater (right).
      · Aggregate[2] statistics (sum, squared sum, and count) for both groups are computed using Spark's `aggregate` function to minimize data scans.
      · The MSE for each side is calculated, and a weighted average gives the total MSE of the split.
    * The split with the lowest MSE across all features is selected as the best.

    This method is efficient in distributed environments because it uses Spark transformations that preserve lazy evaluation and reduce memory overhead.
  - **Recursive Tree Construction:** `build_tree_rdd`

    This function builds a regression decision tree recursively using RDD operations. It includes several key stages:

    * The input RDD is persisted using `MEMORY_AND_DISK` to optimize repeated access.
    * A stopping condition checks whether the current node has reached maximum depth or contains fewer samples than a threshold.
    * If stopping conditions are not met, the function attempts to find the best split using `find_best_split_rdd`.
    * The RDD is split into two branches using `filter` based on the chosen threshold.
    * The function is called recursively on each branch to construct left and right subtrees.
    * After building both branches, the original RDD is unpersisted to release memory.

  This method leverages Spark's parallelism while keeping the model construction logic entirely in low-level RDD operations, which allows fine control over memory and computation.

- **Model Evaluation:** Since the model optimization and construction were performed entirely manually, the decision tree was built with a shallow depth (`max_depth = 3`) and a relatively large split threshold (`min_sample = 1% of the dataset`) to ensure results could be obtained within a limited time frame (under one hour).

  Regarding the model's performance:

  - The **R^2 score of 0.0006** suggests that the model **barely learns any meaningful patterns** from the data.
  - The **RMSE of 4608.93** indicates a **high prediction error**, especially considering that the `trip_duration` variable mostly ranges in the hundreds.

  The RMSE and R^2 evaluation functions were implemented using the `aggregate` operator to **minimize the number of passes over the data**, thereby improving computational efficiency.

References

[1] Apache Spark. *RDD Programming Guide.* Accessed: 2025-05-10. 2025. URL: `https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence`.

[2] Apache Spark. *RDD Programming Guide - Transformations.* Accessed: 2025-05-10. 2025. URL: `https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations`.

[3] Apache Spark. *MLlib - Decision Tree Regression.* Accessed: 2025-05-7. 2025. URL: `https://spark.apache.org/docs/latest/ml-classification-regression.html#decision-tree-regression`.

[4] Headsortails (Kaggle). *NYC Taxi EDA: Update - The Fast The Curious.* Accessed: 2025-05-9. 2023. URL: `https://www.kaggle.com/code/headsortails/nyc-taxi-eda-update-the-fast-the-curious/report#extreme-trip-durations`.

[5] Apache Spark. *MLlib - Decision Tree.* Accessed: 2025-05-7. 2025. URL: `https://spark.apache.org/docs/latest/mllib-decision-tree.html`.