VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF SCIENCE

INTRODUCTION TO BIG DATA ANALYSIS

Lecturer

PhD. Lê Ngọc Thành | lnthanh@fit.hcmus.edu.vn

Lab Instructors

Nguyễn Trần Duy Minh | ntdminh@fit.hcmus.edu.vn
Huỳnh Lâm Hải Đăng | hlhdang@fit.hcmus.edu.vn

Lab 03: Machine Learning on Spark

1. Introduction to Machine Learning on Big Data with Apache Spark

As datasets continue to grow in volume and complexity, leveraging distributed computing frameworks like Apache Spark becomes essential. Modern machine learning on big data relies on Spark's scalable architecture and optimized execution engine (Catalyst and Tungsten) to handle large-scale model training and prediction efficiently. In this lab, you will explore three implementation approaches:

- **Structured API Implementation:** Use the high-level DataFrame-based API (`spark.ml`) to rapidly build and evaluate models.
- **MLlib RDD-Based Implementation:** Leverage Spark's MLlib built-in functionalities on RDD data (`spark.mllib`) to perform machine learning tasks, deepening your understanding of Spark's handling of RDDs.
- **Low-Level Operations Implementation:** Manually decompose ML algorithms into fundamental RDD operations (without relying on MLlib), enhancing your parallelism mindset and grasp of distributed computation.

1.1. Requirements

After completing this labwork, you should be able to:

- Design and implement ML pipelines using the high-level Structured API.
- Work with Spark's MLlib on RDD data to utilize built-in machine learning functions.
- Decompose ML algorithms into low-level RDD transformations to gain deeper understanding about the way such algorithms are parallelized.
- Evaluate and compare the performance and complexity of each implementation approach.

1.2. Machine Learning under parallelized settings

While the Structured API (`spark.ml`) provides a user-friendly, high-level interface, this lab also requires you to work with:

- **MLlib RDD-Based Implementation:** This approach uses Spark's built-in MLlib functions on RDDs, offering deeper insights into the paradigms of distributed ML.
- **Low-Level Operations Implementation:** You will also implement machine learning algorithms using fundamental RDD transformations, helping you understand how to decompose complex ML tasks into scalable operations such as map, reduce, filter, .etc. This deeper dive strengthens your overall big data processing mindset.

2. Example of Linear Regression

In this example, you will build a simple linear regression model to predict Boston house prices using three different approaches. Each level demonstrates how to leverage Spark's evolving APIs for machine learning tasks.

2.1. Structured API Implementation Example

Using the high-level DataFrame-based API from `spark.ml`, follow these steps:

- **Data Preparation:**

```
data = spark.read.csv("boston_house_price.csv", header=True, \
inferSchema=True)
```

- **Pre-processing:**

```
from pyspark.ml.feature import VectorAssembler
# Use all columns except MEDV as features
input_cols = [col for col in data.columns if col != "MEDV"]
# Assemble feature columns into a single vector
assembler = VectorAssembler(inputCols=input_cols, \
outputCol="features")
assembled_data = assembler.transform(data).select("features", "MEDV")
```

- **Train-test split:**

```
# Split the dataset into training and testing sets
train, test = assembled_data.randomSplit([0.8, 0.2], seed=42)
```

- **Train the Linear Regression model using MLlib:**

```
from pyspark.ml.regression import LinearRegression
# Initialize the Linear Regression estimator
lr = LinearRegression(featuresCol="features", labelCol="MEDV")
model = lr.fit(train)
```

- **Evaluate the obtained model:** using the summary object

```
summary = model.summary
print("Coefficients:", model.coefficients)
print("Intercept:", model.intercept)
```

```
print("RMSE:", summary.rootMeanSquaredError)
print("R²:", summary.r2)
```

- **Sample expected output as follow:**

```
Coefficients: [0.45, 1.23, -0.89, ...]  # One coefficient per feature
Intercept: 2.34
RMSE: 4.56
R²: 0.78
```

2.2. MLlib RDD-Based Implementation Example

This approach leverages Spark MLlib's built-in linear regression functions on RDD data.

- **Load the file to a RDD data:** Load the original CSV file as a text file

```
lines = sc.textFile("boston_house_price.csv")
header = lines.first()
data = lines.filter(lambda line: line != header)
```

- **Parsing the data file:** Parse each line by splitting on commas and converting elements to float

```
parsed = data.map(lambda line: [float(x) for x in line.split(",")])
```

- **Pre-processing:** Convert parsed data into an RDD of LabeledPoint objects `LabeledPoint(label, features)`, with features as a dense vector

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.linalg import Vectors
rdd_data = parsed.map(lambda cols: LabeledPoint(cols[-1], \
Vectors.dense(cols[:-1])))
```

- **Training:** train a linear regression model using Stochastic Gradient Descent

```
from pyspark.mllib.regression import LinearRegressionWithSGD
Model_rdd = LinearRegressionWithSGD.train(rdd_data, \
iterations=100, step=0.0001)
```

- **Evaluate the obtained model:** by computing Mean Squared Error

```
valuesAndPreds = rdd_data.map(\
lambda lp: (lp.label, model_rdd.predict(lp.features)))
```

```
mse = valuesAndPreds.map(lambda pl: (pl[0] - pl[1])**2).mean()

print("Mean Squared Error =", mse)
```

- **Sample expected output as follow:**

```
Mean Squared Error = 18.25
```

2.3. Low-level Operations Implementation Example

In this manual approach, you will implement linear regression using fundamental RDD transformations—without using MLlib's built-in functions.

- **Read and parse the data into RDD:**

```
# Load the CSV file as a text file and filter out the header

lines = sc.textFile("boston_house_price.csv")

header = lines.first()

data = lines.filter(lambda line: line != header)

# Parse each line: split by comma and convert each element to float.

# Assume that the last column is the label (MEDV) and all previous columns are features.

parsedData = data.map(\

lambda line: [float(x) for x in line.split(",")])

# Create an RDD where each record is a tuple: (features as list, label)

rdd_data = parsedData.map(lambda cols: (cols[:-1], cols[-1]))
```

- **Implement the Linear Regression model using low-level operations:**

```
# Initialize weights manually (number of features equals the length of the features list)

num_features = rdd_data.first()[0]  # sample features list

initial_weights = [0.0] * len(num_features)

learning_rate = 0.0001

# Define a function to compute prediction as the dot product of weights and feature vector

def predict(features, weights):

    return sum(w * f for w, f in zip(weights, features))
```

- **Compute the prediction as well as gradient of the regressor for a single step:** for each data point, compute `(prediction - label) * feature_vector`, then `sum` across all data points.

```
gradients = rdd_data.map(lambda x: \
[ (predict(x[0], initial_weights) - x[1]) * f for f in x[0] ]) \
.reduce(lambda a, b: [x + y for x, y in zip(a, b)])
```

- **Perform the gradient update for this step:** update weights with a single gradient descent step by `new_weight = old_weight - learning_rate * gradient`

```
updated_weights = [w - learning_rate * grad \
for w, grad in zip(initial_weights, gradients)]
```

- **Check the weights after updated by the above gradient descent step:**

```
print("Updated Weights:", updated_weights)
```

- **Sample expected output as follow:** one updated weight per feature

```
Updated Weights: [0.12, -0.03, 0.07, ...]
```

3. Problem Statements

In this lab, you will tackle two fundamental machine learning problems: **classification** and **regression**. These problems are cornerstones of ML research and practice, and solving them on big data platforms like Spark demonstrates the scalability and practical applicability of distributed learning. By working on these tasks, you will not only learn to build models using high-level APIs but also deepen your understanding of the underlying distributed computation paradigms through hands-on implementations using MLlib's RDD-based functions and manual, low-level operations.

Allowed programming language(s): *Python* and *Scala*. Of course, similar to prior labs, any team having Scala-based implementation will be awarded an additional bonus to the associated problem(s). Please refer to the grading criteria in the last section for more details.

Datasets:

- **Classification:** Credit Card Fraud Detection – Available at: https://www.kaggle.com/mlg-ulb/creditcardfraud
- **Regression:** New York City Taxi Trip Duration – Available at: https://www.kaggle.com/c/nyc-taxi-trip-duration/data

3.1. Classification with Logistic Regression

Classification is a core ML task that involves assigning labels to instances. Logistic regression, in particular, is a fundamental algorithm used for binary classification. In the context of big data, developing a scalable classifier using Spark showcases how large-scale data can be processed in parallel while ensuring robust performance evaluation.

3.1.1. Structured API Implementation (High-Level)

Objective: Use Spark's high-level Structured API (`spark.ml`) to construct a logistic regression model.

- Preprocess your data appropriately (e.g., handle missing values and standardize features if necessary).
- Use a `VectorAssembler` (or equivalent high-level transformer) to combine numeric features into a single vector column.
- Utilize the `LogisticRegression` estimator, then inspect model coefficients, intercept, and evaluation metrics such as accuracy, AUC, precision, and recall.
- Consider splitting your dataset into training and testing sets to assess model performance.

3.1.2. MLlib RDD-Based Implementation

Objective: Leverage MLlib's built-in functions on RDD data to implement logistic regression.

- Convert your preprocessed dataset into an RDD of `LabeledPoint` objects.
- Use MLlib's logistic regression functions (e.g., `LogisticRegressionWithSGD` or `LogisticRegressionWithLBFGS`) to train your model.
- Experiment with parameters such as the number of iterations and step-size (learning rate) to improve convergence.
- After training, compute evaluation metrics—either through built-in utilities or by manually comparing predictions with actual labels.
- Document any observations on how the performance compares to the Structured API approach.

3.1.3. Low-Level Operations

Objective: Manually implement logistic regression using fundamental RDD transformations without relying on MLlib's built-in functions.

- Parse the dataset and create an RDD where each record contains a feature vector (as a list of floats) and the label.
- Write your own functions to compute the dot product between the weight vector and features, apply the sigmoid function to produce probabilities, and derive gradients for each data point.
- Implement a simple gradient descent update in a multi-iteration setting to demonstrate the update mechanism.

- Compare your low-level computed predictions with those from the Structured APIs and MLlib implementations.
- Explain your choices (e.g., learning rate, iteration count) and discuss any challenges encountered in decomposing the algorithm.

3.2. Regression with Decision Trees

Regression tasks involve predicting a continuous response variable. Decision trees offer an interpretable model that can capture non-linear relationships. Using decision trees on big data demonstrates how Spark can efficiently partition data and build scalable, parallel models.

3.2.1. Structured API Implementation (High-Level)

Objective: Use Spark's Structured API (`spark.ml`) to construct a decision tree regressor.

- Preprocess your dataset by assembling numeric features into a vector using high-level transformers.
- Train a `DecisionTreeRegressor` model, adjusting parameters such as maximum tree depth and impurity (e.g., variance) to control model complexity.
- Extract and analyze the tree structure and feature importances to understand model decisions.
- Evaluate the model using metrics such as RMSE and R^2 on a test dataset.

3.2.2. MLlib RDD-Based Implementation

Objective: Implement a decision tree regressor using MLlib's built-in functions on RDD data.

- Convert your dataset into an RDD of `LabeledPoint` objects.
- Train the decision tree model using the MLlib functions available for regression.
- Pay attention to the parameters for tree construction to handle the dataset's scale.
- Compute performance metrics (e.g., RMSE) and compare them with results obtained via the Structured API.
- Provide insights into any differences observed between the two approaches.

3.2.3. Low-Level Operations

Objective: Manually simulate decision tree logic using fundamental RDD operations without relying on MLlib's pre-built estimator.

- For simplicity, consider implementing a traditional fixed-depth decision tree where you may look for popular construction algorithms.
- Provide sample predictions for a few test cases to illustrate your low-level implementation.

- Discuss the challenges of scaling this approach to deeper trees as well as more trees in an ensemble (a forest!).

4. Submission Guidelines

This lab requires a group's submission where the work of your group's members is compressed into a single file and only one representative may submit this file on Moodle. The submission file contains a single folder named <*RepresentativeID*> where student ID of the first member (that your group has registered in earlier form) is used. Its internal structure will be as follow:

```
<RepresentativeID>
├─ docs
│ ├── Report.pdf # Include your approach, detailed explanations, and screenshots of outputs
│ ├── drive_link.txt # Predicted results of all tasks here
│
├─ src
│ ├── Classification
│ │ ├── Structured_API      # Folder code for Structured API Implementation
│ │ ├── MLlib_RDD_Based     # Folder code for MLlib RDD-Based Implementation
│ │ └── Low_Level           # Folder code for Low-Level Operations Implementation
│ │
│ ├── Regression
│ │ ├── Structured_API      # Folder code for Structured API Implementation
│ │ ├── MLlib_RDD_Based     # Folder code for MLlib RDD-Based Implementation
│ │ └── Low_Level           # Folder code for Low-Level Operations Implementation
│
└── README.md               # (Optional) Instructions to run your code
```

The `drive_link.txt` contains a single link to Google Drive folder that is structurally organized as described below. Of course, any edit beyond the determined deadline published on Moodle will invalidate your team's result.

```
<RepresentativeID>
|--- <Classification/Regression>_Structured.csv
|--- <Classification/Regression>_MLlib.csv
|--- <Classification/Regression>_Low_level.csv
```

The grading criteria for both problems in Problem Statements section are summarized in the below table with each problem contributes **5.25 points**.

Accomplished requirements	Points
High-level Structured API	**1.25**
- Load the dataset	0.25
- Pre-process the data (including train-test split)	0.25
- Successfully train and evaluate the model	0.25
- Detailed report	0.25
- Implemented in Scala	0.25
MLlib RDD-based Implementation	**2.0**

- Load the dataset	0.25
- Successfully parse the dataset	0.25
- Pre-process the data (including train-test split)	0.25
- Successfully train the model	0.25
- Successfully evaluate the model	0.25
- Detailed report	0.25
- Your model's performance matches that of MLlib	0.25
- Implemented in Scala	0.25
Low-level Operations	**2.0**
- Load the dataset	0.25
- Successfully parse the dataset	0.25
- Pre-process the data (including train-test split)	0.25
- Successfully train the model	0.25
- Successfully evaluate the model	0.25
- Detailed report, including member's assigned tasks	0.25
- Your model's performance matches that of MLlib	0.25
- Implemented in Scala	0.25
Total points per problem	**5.25**

You must strictly follow the above file structure and compress the whole folder into a ZIP file named <*RepresentativeID*>*.zip*, which is your final file to be submitted to Moodle.

Also note that:

- Ensure your code is well-documented with clear comments.
- Include all necessary files, logs, and screenshots to verify successful execution.
- Each task can be accomplished under complex environments and different programming languages, remember to provide instructions for running each task if this is the case.
- The team's members are responsible for ensuring the integrity of the team's submissions, any cheating or plagiarism detected will result in zero scores for FULL labworks, meaning that if a single member of your team is found to be cheating or plagiarizing in a single lab then all of 4 labs' scores of all members in the team will be zeroes.

Happy Coding and Best of Luck!

The Instructor./.