COURSE: ***Big Data - CTS43135***

Lab Instruction #6:

Building a Movie Recommendation System Using PySpark and Spark MLlib

Lab Objectives:

- Understand how to process and analyze large-scale data using PySpark and Spark MLlib.
- Explore and manipulate structured datasets with Spark SQL and DataFrames.
- Implement collaborative filtering for recommendation systems using the ALS algorithm.

Prerequisites

- Basic knowledge of Python, SQL and Machine Learning is recommended.
- This lab runs on Python 3.6+ with Apache Spark 3.x.
- A working environment like Jupyter Notebook or Google Colab is recommended for easier execution.

Activity 1: Preparing the MovieLens Dataset

*The **MovieLens** dataset is a widely used benchmark dataset for building and evaluating recommender systems. It is provided by **GroupLens Research** at the University of Minnesota. The dataset contains user ratings for movies, along with user demographic information and movie metadata.*

In this lab, we will use the **MovieLens 1M** dataset, which consists of **1,000,000 ratings** from **6,000 users** on **4,000 movies**. The data is stored in three main files:

1. **`ratings.dat`** – Contains user ratings, structured as `(UserID, MovieID, Rating, Timestamp)`.
2. **`movies.dat`** – Contains movie metadata with `(MovieID, Title, Genres)`.
3. **`users.dat`** – Includes user information such as `(UserID, Gender, Age, Occupation, Zip-code)`.

1. **Download this dataset.** The following command is used to download the MovieLens 1M dataset from the official GroupLens website using wget:

```
!wget -O ml-1m.zip https://files.grouplens.org/datasets/movielens/ml-1m.zip
```

2. **Extract the ZIP file.** Extract the ZIP file using the following command:

```
import zipfile

with zipfile.ZipFile("ml-1m.zip", "r") as zip_ref:
    zip_ref.extractall("ml-1m")  # Extracts the contents into a folder named
"ml-1m"
```

3. **Defining Schemas for MovieLens Dataset in PySpark.** Before processing the dataset, import the necessary PySpark modules:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
from pyspark.sql.types import StructType, StructField, IntegerType, DoubleType,
```

```
StringType
```

Initialize Spark Session

```
spark = SparkSession.builder \
    .appName("MovieLensRecommendation") \
    .getOrCreate()
```

Define Schemas.

```
ratings_schema = StructType([
    StructField("userId", IntegerType(), True),
    StructField("movieId", IntegerType(), True),
    StructField("rating", DoubleType(), True),
    StructField("timestamp", IntegerType(), True)
])
```

Task 1: create a schema for the `movies.data` file from the MovieLens dataset

4. **Load the MovieLens Dataset.** Load Ratings Data from the `ratings.dat`

```
ratings_df = spark.read.csv("ratings.dat", sep="::",
schema=ratings_schema,header=False)
```

Remove **the timestamp column, since we don't need it:**

```
ratings_df = ratings_df.drop("timestamp")
```

5. **Cache the DataFrames.** Since these datasets will be used multiple times, caching improves performance.

```
ratings_df.cache()
movies_df.cache()
```

6. **Count the Number of Ratings and Movies.**

```
ratings_count = ratings_df.count()
movies_count = movies_df.count()
print(f"There are {ratings_count} ratings and {movies_count} movies in the
```

```
dataset.")
```

7. **Display Sample Data.** To verify that the data is loaded correctly:

```
print("Ratings Data Sample:")
ratings_df.show(3)  # Show first 3 rows of ratings data

print("Movies Data Sample:")
movies_df.show(3, truncate=False)  # Show first 3 rows of movies data (without
truncating movie titles)
```

Activity 2: Computing Average Ratings

One way to recommend movies is to always recommend the movies with the highest average rating. In this part, we will use Spark to find the name, number of ratings, and the **average rating of the 20 movies with the highest average rating** and **at least 500 reviews**. We want to filter movies with at least 500 reviews to ensure reliability in the ratings.

1. **Import Required Libraries.** Before performing any transformations, import the necessary PySpark functions:

```
from pyspark.sql import functions as F
```

2. **Compute the Average Ratings Per Movie. Using `ratings_df`, compute the average rating and the number of ratings per movie.**

```
movie_ids_with_avg_ratings_df = (ratings_df
    .groupBy('movieId')  # Group by movie ID
    .agg(F.count(ratings_df.rating).alias("count"),   # Count the number of
ratings
```

```
        F.avg(ratings_df.rating).alias("average"))    # Compute the average
rating
)
```

- `groupBy('movieId')` → Groups ratings by movie ID.
- `count()` → Counts how many users rated each movie.
- `avg()` → Computes the average rating per movie.

Now, print the results:

```
print("movie_ids_with_avg_ratings_df:")
movie_ids_with_avg_ratings_df.show(3, truncate=False)
```

Expected Output format:

```
+-------+-----+------------------+
|movieId|count|average           |
+-------+-----+------------------+
|1831   |7463 |2.5785207021305103|
|431    |8946 |3.695059244355019 |
|631    |2193 |2.7273141814865483|
+-------+-----+------------------+
```

3. **Add Movie Titles to the DataFrame.** To include movie names, join `movies_df` with the ratings dataset.

```
movie_names_with_avg_ratings_df = movie_ids_with_avg_ratings_df.join(
    movies_df,  # Joining with movies dataset
    movie_ids_with_avg_ratings_df.movieId == movies_df.ID  # Matching on movie ID
)
```

Now, print the results:

```
print("movie_names_with_avg_ratings_df:")
movie_names_with_avg_ratings_df.show(3, truncate=False)
```

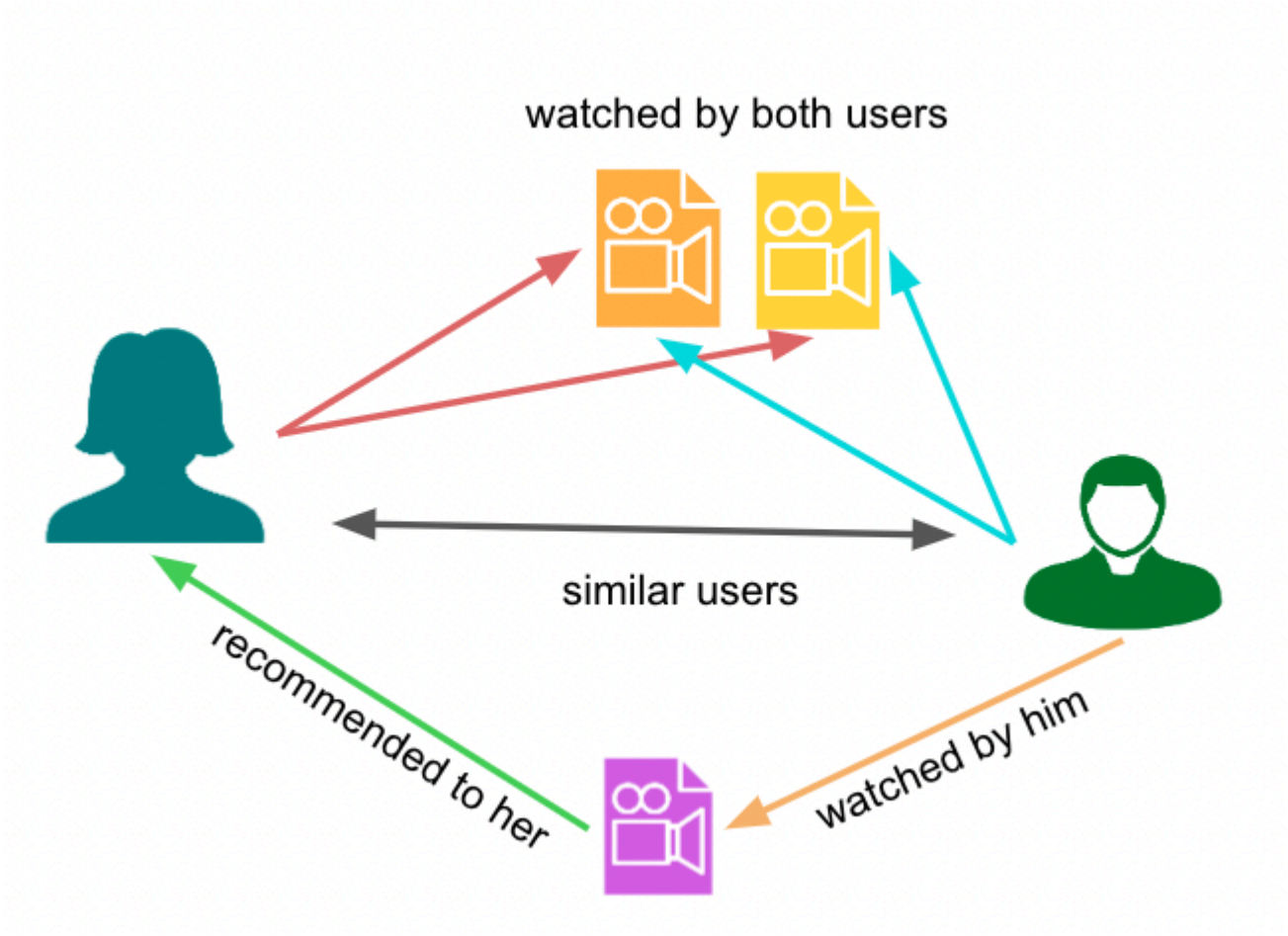
Expected Output format:

```
+-------+-----------------------------+-----+-------+
|average|title                        |count|movieId|
+-------+-----------------------------+-----+-------+
|5.0    |Ella Lola, a la Trilby (1898)|1    |94431  |
|5.0    |Serving Life (2011)          |1    |129034 |
|5.0    |Diplomatic Immunity (2009? ) |1    |107434 |
+-------+-----------------------------+-----+-------+
```

Task 2: Filter the movies dataset to include only movies with at least 500 reviews and sort them by highest average rating.

Activity 3: Collaborative Filtering with ALS

We are going to use a technique called collaborative filtering. Collaborative filtering is a method of making automatic predictions (filtering) about the interests of a user by collecting preferences or taste information from many users (collaborating). The underlying assumption of the collaborative filtering approach is that if a person A has the same opinion as a person B on an issue, A is more likely to have B's opinion on a different issue x than to have the opinion on x of a person chosen randomly.

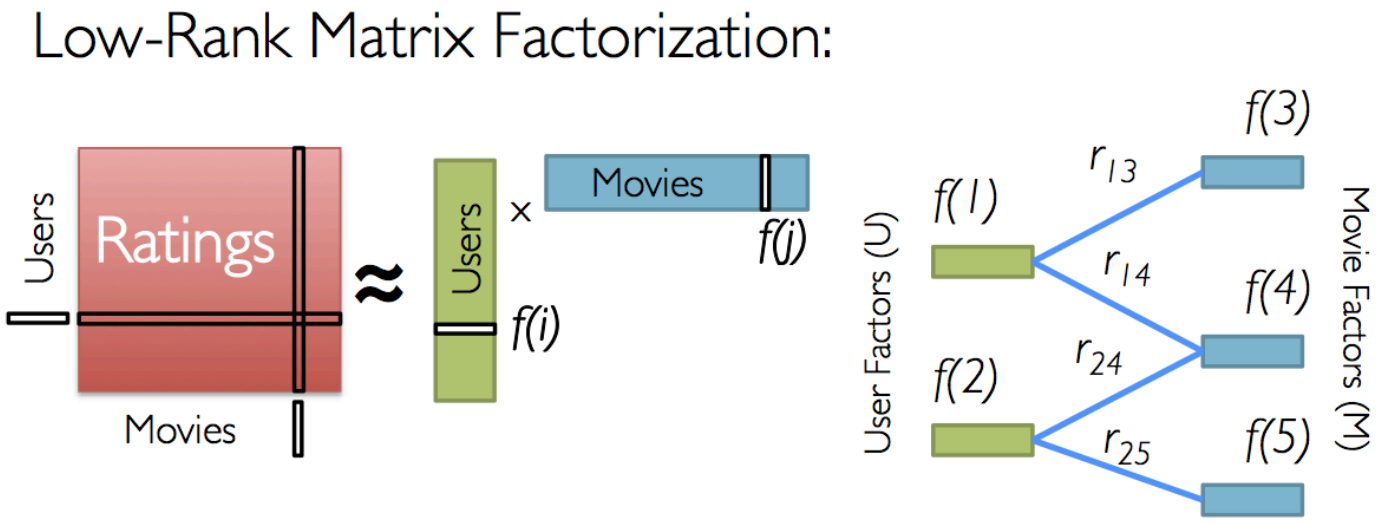


For movie recommendations, we start with a matrix whose entries are movie ratings by users (shown in red in the diagram below). Each column represents a user (shown in green) and each row represents a particular movie (shown in blue).

Since not all users have rated all movies, we do not know all of the entries in this matrix, which is precisely why we need collaborative filtering. For each user, we have ratings for only a subset of the movies. With collaborative filtering, the idea is to approximate the ratings matrix by factorizing it as the product of two matrices: one that describes properties of each user (shown in green), and one that describes properties of each movie (shown in blue).

Lab Instruction #6: Building a Movie Recommendation System Using PySpark and Spark MLlib

---



Iterate:

$$f[i] = \arg\min_{w \in \mathbb{R}^d} \sum_{j \in \mathrm{Nbrs}(i)} \left(r_{ij} - w^T f[j]\right)^2 + \lambda ||w||_2^2$$

We want to select these two matrices such that the error for the users/movie pairs where we know the correct ratings is minimized. The Alternating Least Squares algorithm does this by first randomly filling the users matrix with values and then optimizing the value of the movies such that the error is minimized. Then, it holds the movies matrix constant and optimizes the value of the user's matrix. This alternation between which matrix to optimize is the reason for the "alternating" in the name.

This optimization is what's being shown on the right in the image above. Given a fixed set of user factors (i.e., values in the users matrix), we use the known ratings to find the best values for the movie factors using the optimization written at the bottom of the figure. Then we "alternate" and pick the best user factors given fixed movie factors.

1. **Creating a Training Set.** We need to break up the ratings_df dataset into three pieces:
   - A training set (DataFrame), which we will use to train models
   - A validation set (DataFrame), which we will use to choose the best model
   - A test set (DataFrame), which we will use for our experiments

To randomly split the dataset into the multiple groups, we can use the pySpark randomSplit() transformation. randomSplit() takes a set of splits and a seed and returns multiple DataFrames.

```
# Convert DataFrame to RDD for MLlib compatibility
ratings_rdd = ratings_df.rdd.map(lambda row: Rating(row.userId, row.movieId,
row.rating))

# Randomly split the RDD into training (60%), validation (20%), and test (20%)
sets
seed = 1800009193
(training_rdd, validation_rdd, test_rdd) = ratings_rdd.randomSplit([0.6, 0.2,
0.2], seed=seed)

# Cache the RDDs for performance
training_rdd.cache()
validation_rdd.cache()
test_rdd.cache()

# Print dataset sizes
print(f'Training: {training_rdd.count()}, Validation: {validation_rdd.count()},
Test: {test_rdd.count()}\n')

# Show sample data from each dataset
print("Training Sample:")
print(training_rdd.take(3))  # Take 3 samples from training set

print("Validation Sample:")
print(validation_rdd.take(3))  # Take 3 samples from validation set

print("Test Sample:")
print(test_rdd.take(3))  # Take 3 samples from test set
```

---

4. **Alternating Least Squares.** In this section, you will implement a function to train an ALS (Alternating Least Squares) model using Spark MLlib. Let's first write a function to train an ALS model and evaluate its performance.

   Import Required Libraries. Before we start, ensure you have imported all necessary libraries:

```
from pyspark.mllib.recommendation import ALS, Rating
from math import sqrt
```

   - `ALS` → The Alternating Least Squares model for collaborative filtering.
   - `Rating` → Represents a `(userId, movieId, rating)` tuple.
   - `sqrt` → Used for computing RMSE.

   Train an ALS model on the training data. To train a recommendation model, we use the `ALS.train()`

```
model = ALS.train(training_rdd, rank, iterations=iterations, lambda_=reg_param)
```

   - `training_rdd` → The training dataset in RDD format.
   - `rank` → Number of latent factors (complexity of the model).
   - `iterations` → Number of optimization iterations.
   - `lambda_` → The regularization parameter (to prevent overfitting).

   Predicts ratings on validation data. Once the model is trained, we predict movie ratings for the validation dataset:

```
# Convert predictions to the format ((userId, movieId), rating)
predictions_rdd = predictions_rdd.map(lambda r: ((r.user, r.product), r.rating))

# Convert actual ratings to the same format ((userId, movieId), rating)
```

```
actual_ratings_rdd = validation_rdd.map(lambda x: ((x[0], x[1]), x[2]))

# Join actual ratings with predictions
joined_rdd = actual_ratings_rdd.join(predictions_rdd)
```

- **Actual ratings format:** `((userId, movieId), actual_rating)`
- **Predicted ratings format:** `((userId, movieId), predicted_rating)`
- **Joined format:** `((userId, movieId), (actual_rating, predicted_rating))`

Computes RMSE as an evaluation metric. We compute the Root Mean Squared Error (RMSE), which measures the accuracy of our predictions:

```
# Compute RMSE
mse = joined_rdd.map(lambda x: (x[1][0] - x[1][1]) ** 2).mean()
rmse = sqrt(mse)
```

- `MSE` **(Mean Squared Error)** → `(actual - predicted)²` for each rating.
- `RMSE` → Square root of the mean of all squared errors.

5. **Implementing Everything in a Single Function.** Now, we put everything together in one function:

```
def train_als_model(training_rdd, validation_rdd, rank, iterations=5,
reg_param=0.1):
    """
    Trains an ALS model and evaluates its performance using RMSE.

    Parameters:
    training_rdd (RDD): Training dataset in RDD format
    validation_rdd (RDD): Validation dataset in RDD format
    rank (int): Number of latent factors for ALS
    iterations (int): Number of iterations for ALS optimization
    reg_param (float): Regularization parameter for ALS

    Returns:
    model: Trained ALS model
    rmse (float): Computed RMSE value
```

```
    """
    # Train ALS model
    model = ALS.train(training_rdd, rank, iterations=iterations,
lambda_=reg_param)

    # Generate predictions for validation set
    predictions_rdd = model.predict(validation_rdd.map(lambda x: (x[0], x[1])))

    # Format predictions as ((userId, movieId), rating)
    predictions_rdd = predictions_rdd.map(lambda r: ((r.user, r.product),
r.rating))
    actual_ratings_rdd = validation_rdd.map(lambda x: ((x[0], x[1]), x[2]))

    # Join actual ratings with predictions
    joined_rdd = actual_ratings_rdd.join(predictions_rdd)

    # Compute RMSE
    mse = joined_rdd.map(lambda x: (x[1][0] - x[1][1]) ** 2).mean()
    rmse = sqrt(mse)

    return model, rmse
```

6. **Train the Model with Sample Parameters.** Now, let's train the ALS model with a sample rank = 8 and evaluate its RMSE:

```
# Define model parameters
rank = 8
iterations = 5
reg_param = 0.1

# Train the model and get RMSE
als_model, rmse = train_als_model(training_rdd, validation_rdd, rank, iterations,
reg_param)

print(f"Trained ALS Model with Rank {rank}, RMSE: {rmse}")
```

Task 3: Tune ALS hyperparameters by testing different ranks (`4, 8, 12, 16, 20`) and regularization parameters (`0.01, 0.1, 0.5, 1.0`) to find the best model with the lowest RMSE.

Credits

Lab Assignment: Spark MLlib – Book Recommendation

Objective

- Load and process the Book-Crossing dataset using PySpark.
- Perform data cleaning and transformation to structure the dataset for recommendations.
- Use Spark MLlib's ALS (Alternating Least Squares) to build a book recommendation system.
- Tune hyperparameters to optimize the recommendation model.
- Evaluate model performance using Root Mean Squared Error (RMSE).

Instructions

Download this dataset: Book-Crossing Dataset.

This dataset contains user ratings for books, which will be used to build a recommendation system using Spark MLlib. Your goal is to process the dataset using Spark and apply **ALS (or similar) collaborative filtering** to build a book recommendation system.

- Load and preprocess the dataset, ensuring valid user ratings.
- Filter out books with very few ratings to improve model performance.
- Train an ALS model using PySpark MLlib to generate book recommendations.
- Evaluate the model using Root Mean Squared Error (RMSE).
- Tune hyperparameters (`rank`, `lambda_`, `iterations`) to optimize the recommendation model.
- Generate and display the top 5 book recommendations for a given user.

Submission

- **Submission deadline:** 2 weeks from the assignment date.
- **Submission Format:** Upload the Executed Notebook (or similar) to LMS (lms.siu.edu.vn).

Suggested Resources

- https://spark.apache.org/docs/latest/ml-collaborative-filtering.html
- https://spark.apache.org/sql/