MovieLens: Spark-based Big Data Recommendation Analysis

50015627 JIANG Zhuoyang-(full-stack-work 25% Contribution) ,
50016061 GAN Wenxi-(full-stack-work 25% Contribution) ,
50015976 ZHANG Ruiming-(full-stack-work 25% Contribution),
50015859 ZHENG Hanwen-(full-stack-work 25% Contribution)

1 OBJECTIVE

1.1 Task Introduction

This project requires us to process and analyze data from the well-known MovieLens dataset to familiarize ourselves with the SPARK framework. It mainly consists of two parts: first, getting acquainted with SPARK's distributed properties, primarily reflected in the application of its underlying data structure RDD; second, performing common data analysis using SPARK, with a focus on machine learning algorithms related to recommendation systems. The data analysis tools primarily revolve around the MLLib or ML framework within the Spark ecosystem.

1.2 Condition Analysis

This task primarily sets two conditions: using MovieLens as the data source and employing recommendation system predictions as the main methodology. Regarding the MovieLens data source, we've observed that this dataset possesses three significant characteristics: "large volume," "structured," and "user-item correlation." The first two traits indicate the dataset's suitability for data analysis in distributed frameworks to enhance processing efficiency. The third trait signifies its suitability for recommendation-related analysis using "user-item correlation." Regarding the main methodology of recommendation systems, based on our understanding, it can primarily be accomplished through two methods: implicit automatic feature construction based on matrix factorization and explicit manual feature construction based on feature engineering.

1.3 Experimental Objectives

Given a foundational understanding of the two conditions specified for this project, we can propose two categories of objectives for the tasks:

First, considering the "big data" condition of this project, we aim to explore "how to enhance data processing efficiency during the task." To investigate this, we conduct the first set of controlled experiments to compare the impact of different system architectures on the data processing efficiency of the same data analysis method, while keeping the recommendation analysis method constant. In this experiment, our primary comparison involves assessing the efficiency of training models running ALS algorithm under "distributed" and "centralized" modes within the SPARK framework to achieve the experimental goal.

Second, considering the "recommendation analysis" task of this project, our aim is to explore "how to improve the model accuracy of predictive analysis within recommendation systems." For this, we construct the second set of controlled experiments to compare the impact of different model input feature construction methods on the final predictive model accuracy while maintaining constant experimental environments and architectures. In this experiment, our primary comparison involves evaluating the Root Mean Squared Error (RMSE) on the same test set between ALS algorithm based on matrix factorization for extracting implicit features and some classic machine learning algorithms based on manually constructed explicit features to achieve the experimental objectives.

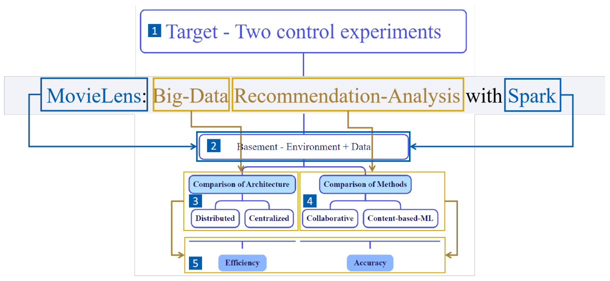


Figure 1: From Objective to Pipeline

2 BASICS: ENVIRONMENT AND DATA

2.1 Environment Selection

Before conducting the experiments, it's necessary to elaborate on the relevant foundational knowledge of the experimental environment to demonstrate the rationality of choosing the SPARK framework and its corresponding ML tools.

2.1.1 *Architecture:*

This experiment selects the SPARK-based distributed data processing framework. Firstly, it offers a simple switch solution for the underlying framework for the first controlled experiment and also encapsulates upper-level machine learning algorithms for the second experiment. This choice is substantiated by some advantages of this framework over Hadoop MapReduce implementation:

- Spark inherits advantages from Hadoop MapReduce.
- Spark enables faster computational processing: In Hadoop, intermediate data is stored in HDFS, whereas in Spark, processes can retain intermediate data.

2.1.2 *Data Structure:*

Initially, this project utilized the fundamental SPARK data structure, RDD, for data exploration and preliminary analysis. However, when conducting the two controlled experiments, the project chose the Dataframe data structure, better suited for structured data handling based on the dataset's "structured" characteristic. Apart from possessing the fundamental strengths of RDD, Dataframe also exhibits adaptability to "structured data" and "high-performance requirements," making it more suitable for complex experimental work.

2.1.3 *Machine Learning Tools:*

To better utilize the Dataframe data structure under the Spark framework, the project mainly utilized the ML framework rather than MLLib. This choice was made not only because the ML framework has a higher priority in official maintenance but also due to its underlying compatibility with Dataframe.

2.2 Environment Configuration

Following the explanation of the foundational environment selection, we proceed with configuring the specific environment:

First, PySpark 3.0.0 was chosen as the programming language for the experimental architecture due to major updates in the MLlib package, making it more user-friendly compared to versions 2.x.

Then, Cloud Dataproc was chosen as the experimental environment because PySpark is pre-installed in it. Moreover, it simplifies the process of spinning up a Spark cluster, adding worker nodes after cluster instantiation, and supports autoscaling policies for automatic addition of worker nodes.

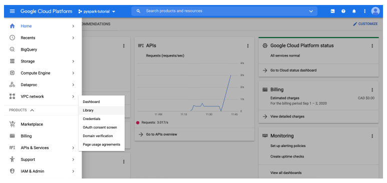
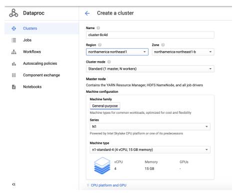
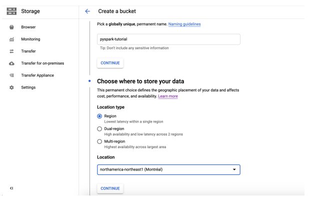


Figure 2: Cloud Dataproc Setup



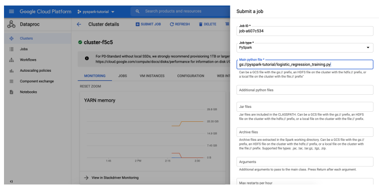
Name: cluster-8c4d
Region: northamerica-northeast1
Zone: northamerica-northeast1-b
Cluster mode: Standard (1 master, N workers)
Master node
Contains the YARN Resource Manager, HDFS NameNode, and all job drivers
Machine configuration
Machine family: General-purpose
Machine types for common workloads, optimized for cost and flexibility
Series: N1
Powered by Intel Skylake CPU platform or one of its predecessors
Machine type: n1-standard-4 (4 vCPU, 15 GB memory)
vCPU: 4 Memory: 15 GB GPUs: -

Figure 3: Create A Dataproc Cluster



Pick a **globally unique**, permanent name.
pyspark-tutorial
Tip: Don't include any sensitive information
Choose where to store your data
This permanent choice defines the geographic placement of your data and affects cost, performance, and availability.
Location type
[✓ Region] Lowest latency within a single region
Dual-region High availability and low latency across 2 regions
Multi-region Highest availability across largest area
Location: northamerica-northeast1 (Montréal)

Figure 4: Cloud Storage Setup



For PD-Standard without local SSDs, we strongly recommend provisioning 1TB or larger https://cloud.google.com/compute/docs/disks/performance for information on disk I/
Job ID *: job-a607c534
Job type *: PySpark
Main python file *: gs://pyspark-tutorial/logistic_regression_training.py
Can be a GCS file with the gs:// prefix, an HDFS file on the cluster with the hdfs:// prefix, or a local file on the cluster with the file:// prefix"
Additional python files
Jar files
Jar files are included in the CLASSPATH. Can be a GCS file with the gs:// prefix, an HDFS file on the cluster with the hdfs:// prefix, or a local file on the cluster with the file:// prefix.
Archive files
Archive files are extracted in the Spark working directory. Can be a GCS file with the gs:// prefix, an HDFS file on the cluster with the hdfs:// prefix, or a local file on the cluster with the file:// prefix. Supported file types: .jar, .tar, .tar.gz, .tgz, .zip.
Arguments
Additional arguments to pass to the main class. Press Return after each argument.
Max restarts per hour

Figure 5: Submit a Spark Job

2.3 Data Import and System Adaptation

Next, data importation was performed for the distributed architecture, and some basic recommendation tasks were undertaken to adapt to the SPARK system architecture's usage.

2.3.1 *Data Stratification:*

According to the description provided in the README file of the MovieLens dataset, we categorize the six data tables into three primary tiers, each serving a different purpose.

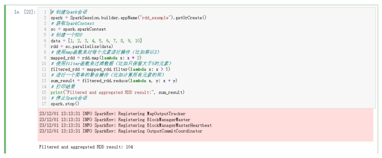
The first tier comprises the fundamental data in rating.csv. This table contains crucial indices of UserID and MovieID representing "users" and "items," respectively, along with the rating label. The former two elements serve as the foundational input for organizing information, while the ratings serve as the output basis for this task.

The second tier consists of the main information data found in movies.csv. This table holds the most comprehensive data, thereby providing the highest-quality feature information available.

Finally, the last tier comprises four additional data tables serving as supplementary information. These tables either exhibit a higher degree of missing information or possess weaker subjective relevance. The tag.csv can be utilized as a central source for generating supplemental feature information. In cases where efficiency in data processing is a priority and computational resources are limited, it might be appropriate to selectively discard information from these supplementary tables.

2.3.2 *Data Import and Summary:*

Due to efficiency requirements and computational resource limitations, focus was primarily on importing foundational data and main information data. RDD (and Dataframe) data structures were directly created using Spark-provided methods. (RDD Operation Example shows below)

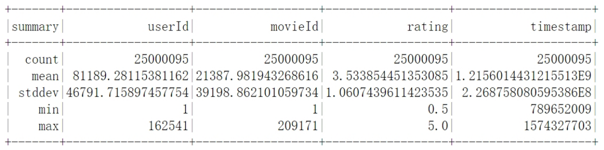


```
In [22]:
# 创建Spark会话
spark = SparkSession.builder.appName("rdd_example").getOrCreate()
# 获取SparkContext
sc = spark.sparkContext
# 创建一个RDD
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
rdd = sc.parallelize(data)
# 使用map函数来对每个元素进行操作（比如乘以2）
mapped_rdd = rdd.map(lambda x: x * 2)
# 使用filter函数来过滤数据（比如只保留大于5的元素）
filtered_rdd = mapped_rdd.filter(lambda x: x > 5)
# 进行一个简单的聚合操作（比如计算所有元素的和）
sum_result = filtered_rdd.reduce(lambda x, y: x + y)
# 打印结果
print("Filtered and aggregated RDD result:", sum_result)
# 停止Spark会话
spark.stop()
```

```
23/12/01 13:13:31 INFO SparkEnv: Registering MapOutputTracker
23/12/01 13:13:31 INFO SparkEnv: Registering BlockManagerMaster
23/12/01 13:13:31 INFO SparkEnv: Registering BlockManagerMasterHeartbeat
23/12/01 13:13:31 INFO SparkEnv: Registering OutputCommitCoordinator

Filtered and aggregated RDD result: 104
```

Figure 6: RDD Operation

Summary statistics for the data imported from rating.csv are explicitly depicted as in Figure 8.

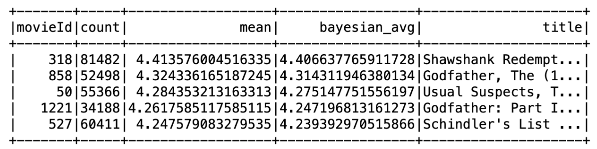


```
+-------+------------------+------------------+------------------+--------------------+
|summary|            userId|           movieId|            rating|           timestamp|
+-------+------------------+------------------+------------------+--------------------+
|  count|          25000095|          25000095|          25000095|            25000095|
|   mean| 81189.28115381162|21387.981943268616| 3.533854451353085|1.2156014431215513E9|
| stddev|46791.715897457754|39198.862101059734|1.0607439611423535| 2.268758080595386E8|
|    min|                 1|                 1|               0.5|           789652009|
|    max|            162541|            209171|               5.0|          1574327703|
+-------+------------------+------------------+------------------+--------------------+
```

Figure 7: Summary statistics

2.3.3 *Bayesian-based Simple Recommendations:*

To familiarize with the system's operation, a Bayesian method was employed for recommending the top five movies across the entire rating dataset. The final results are presented as in Figure 9.



```
+-------+-----+------------------+-----------------+--------------------+
|movieId|count|              mean|     bayesian_avg|               title|
+-------+-----+------------------+-----------------+--------------------+
|    318|81482| 4.413576004516335|4.406637765911728|Shawshank Redempt...|
|    858|52498| 4.324336165187245|4.314311946380134|Godfather, The (1...|
|     50|55366| 4.284353213163313|4.275147751556197|Usual Suspects, T...|
|   1221|34188|4.261758511758511 5|4.247196813161273|Godfather: Part I...|
|    527|60411| 4.247579083279535|4.239392970515866|Schindler's List ...|
+-------+-----+------------------+-----------------+--------------------+
```

Figure 8: Top 5 Recommendations

2.4 Data Preprocessing and EDA

Following basic data preprocessing, some visual observations were made through Exploratory Data Analysis (EDA). The main results are as follows:

2.4.1 *Ratings:*

RatingThe average and median of user ratings primarily fall between 3 and 5, indicating that users tend to submit higher ratings. The standard deviation of user ratings is mainly concentrated between 0.5 and 1.5, suggesting minimal variation in ratings across different movies.

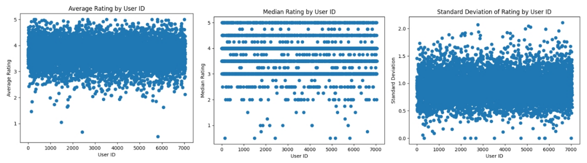


Figure 9: EDA for rating

Rating distribution: Apart from the frequency of ratings,it seems to lack significant meaning or relevance.

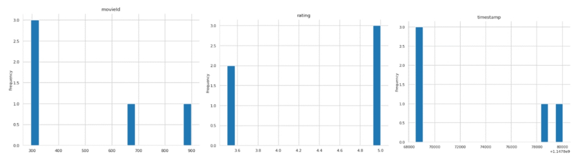


Figure 10: EDA for rating distribution

Movie ID/timestamp vs. rating: When combining common knowledge with the visual representation, it appears to have no apparent correlation.

Time Series: The first graph illustrates the relationship between movie IDs and ratings over time. Based on common knowledge and the visual representation, there seems to be a certain relationship depicted in this graph, while the other three graphs lack distinctive features.

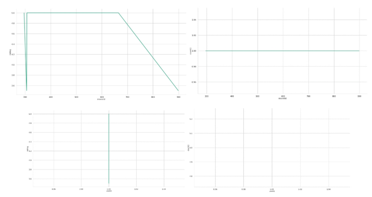


Figure 11: EDA for Time Series of rating

Values: The first graph indicates no correlation between ratings and time. The second graph suggests that higher movie IDs correspond to higher ratings, which seems illogical based on common knowledge, but further exploration is required.

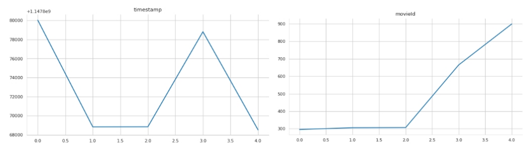


Figure 12: EDA for values

2.4.2 *Movies*.

The frequency of movie IDs is all 1, indicating no correlation and rendering them useless.

In figure 12 Left: No significance, as the frequency of titles is also 1.

In figure 12 Right: The genre categories within the dataset haven't been fully categorized based on features, hence the poor effectiveness of EDA. There are cases of genre overlap without merging, requiring further processing.

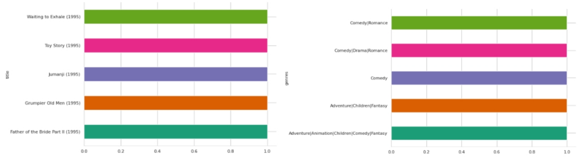


Figure 13: EDA for Movies' title and genre

Then, the presentation of the distribution of ratings by different users doesn't seem to have any correlation.

In figure 13 Left: Demonstrates the overall distribution of movie ratings, primarily concentrated between 3 and 5. In figure 13 Right: Shows the density of ratings given by each user to movies.

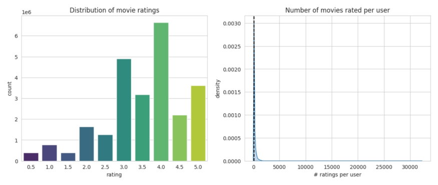


Figure 14: EDA for movie's rating situation

3 CONTRASTING EXPERIMENT ONE: SYSTEM ARCHITECTURE COMPARISON

3.1 Experimental Process:

After environment setup and data import, we commenced the comparison experiment on system architectures. The primary experimental approach was based on the first objective of this project: exploring "how to improve data processing efficiency during the task." The experiment proceeded as follows:

First, we kept the recommendation analysis method constant and chose the "ALS" algorithm, which automatically generates implicit features through matrix factorization, as the sole analytical method. This algorithm eliminates the need for intricate manual feature engineering, serving as the baseline test algorithm for framework processing efficiency.

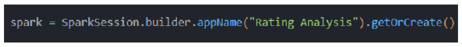
Then, by maintaining consistent ALS hyperparameters, loss functions, and optimizers, and utilizing the same training data, we compared the architectural efficiency through the computationally intensive algorithm training process.

Through this, using both Local and Cluster modes under the SPARK framework, we aimed to explore the efficiency differences between centralized and distributed computing. Given that the baseline algorithm's various selectable objects remained unchanged and the training data was identical, we simply compared the training process times to demonstrate the impact of different system architectures on the data processing efficiency of the same data analysis method.

3.2 Experimental Operations:

In Cloud Dataproc, using highly encapsulated spark sessions, we created two different mode sessions using the master() method:

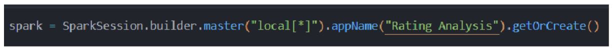
The session created using master(local[*]) is based on the centralized framework implemented in local mode, simulating distributed computing through multithreading but fundamentally operating as a centralized single-machine processing mode.



```
spark = SparkSession.builder.appName("Rating Analysis").getOrCreate()
```

Figure 15: Creat session for local mode

Without calling master(local[*]), directly framing your session in our pre-configured environment would automatically select the cluster mode, utilizing our previously created client cluster to implement the data processing computation in the Spark framework.



```
spark = SparkSession.builder.master("local[*]").appName("Rating Analysis").getOrCreate()
```

Figure 16: Creat session for cluster mode

4 CONTRASTING EXPERIMENT TWO: PREDICTION METHOD COMPARISON:

4.1 Experimental Process:

After contrasting the efficiency of system architectures, we needed to compare commonly used prediction solutions in the recommendation system for the second objective: exploring "how to improve the model accuracy of prediction analysis in the recommendation system." The experiment proceeded as follows:

First, maintaining the experiment's runtime environment and architecture as cluster mode, we created sessions based on distributed computing.

Second, initially constructing the training dataset from the foundational data rating.csv and partitioning it into a 7:3 training/test set split, we ensured consistency for subsequent different methods.

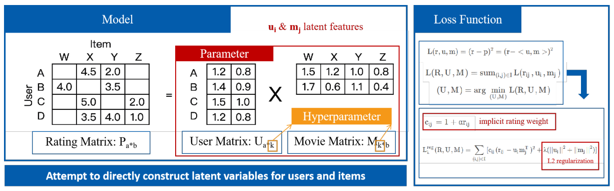
Then, utilizing the pre-packaged ALS algorithm from the pyspark.ML library, we trained a model and obtained an ALS model to test the predicted RMSE on the test set.

Last, given that ALS defaults to treating rating values as continuous data, we configured the subsequent machine learning tasks as regression tasks. Designing features manually and conducting corresponding feature engineering, we generated feature vectors for all training test data inputs. Employing the pyspark.ML library, we constructed a simple decision tree regression model, training the model based on computational resource allowances, and obtained a decision tree regression model for actual testing, deriving the predicted RMSE.

4.2 ALS Method:

4.2.1 *Theoretical Basis:*

ALS is essentially a model training scheme based on matrix factorization. The object of matrix factorization is the "co-occurrence matrix," a two-dimensional matrix constructed with "user-item" as the two-dimensional keys and ratings as values. In this experiment, the "user-movie" co-occurrence matrix is decomposed into user and movie matrices containing model parameters. Each row of the user matrix represents the latent feature vector of that user, and each column of the movie matrix represents the latent feature vector of that movie. The dimension of the feature vector is a manually selected model hyperparameter. The training process of this model optimizes the ALS loss function under L2 regularization using known rating data, thereby training the user and movie matrix parameters to obtain the model.

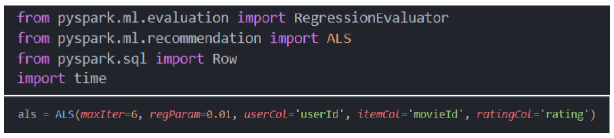


Model
u_i & m_j latent features
Item
User
Parameter
Hyperparameter
Rating Matrix: P_{a*b}
User Matrix: U_{a*k}
Movie Matrix: M_{k*b}
Attempt to directly construct latent variables for users and items
Loss Function
$L(r,u,m) = (r-p)^2 = (r- < u,m >)^2$
$L(R,U,M) = sum_{(i,j)\in I} L(r_{ij}, u_i, m_j)$
$(U,M) = \arg\min_{(U,M)} L(R,U,M)$
$c_{ij} = 1 + \alpha r_{ij}$ implicit rating weight
$L_\lambda^{reg}(R,U,M) = \sum_{(i,j)\in I} [c_{ij}(r_{ij} - u_i m_j^T)^2 + \lambda(||u_i||^2 + ||m_j||^2)]$
L2 regularization

Figure 17: Theory of ALS Algorithm

4.2.2 *Experimental Operations:*

In this experiment, we directly employed the highly encapsulated ALS algorithm from the pyspark.ml library for model training, as shown below:



```
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row
import time
```

```
als = ALS(maxIter=6, regParam=0.01, userCol='userId', itemCol='movieId', ratingCol='rating')
```

Figure 18: Code of ALS Algorithm

4.3 Manually Engineered Machine Learning Methods:

4.3.1 *Pipeline:*

Based on previous experience in machine learning projects, we have established a comprehensive technical path for such tasks: initially constructing the input domain of machine learning models through feature engineering, followed by model selection and training, and finally evaluating and deploying the model.

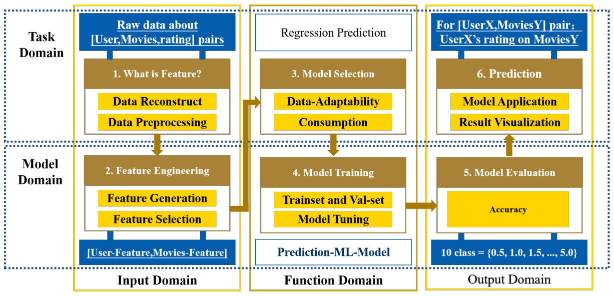
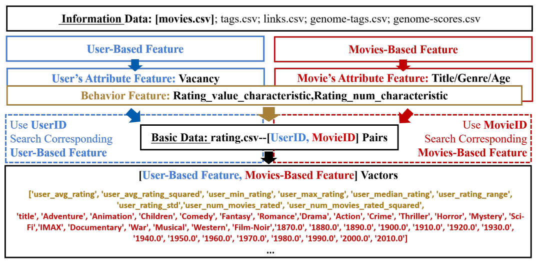


Figure 19: Manually Engineered Machine Learning Methods:

4.3.2 Feature Engineering:

Our feature engineering primarily focuses on the characteristics of the recommendation system. Generally, regarding the "user-item" relationship in the recommendation system, there are two types of features: "attribute" features and "behavioral" features. The former includes "user attribute features" and "item attribute features," while the latter is the "user's behavioral features on items."



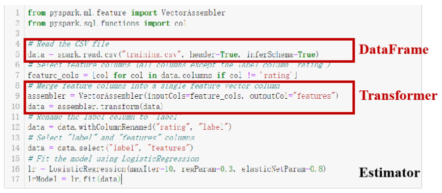
Information Data: [movies.csv]; tags.csv; links.csv; genome-tags.csv; genome-scores.csv
User-Based Feature
Movies-Based Feature
User's Attribute Feature: Vacancy
Movie's Attribute Feature: Title/Genre/Age
Behavior Feature: Rating_value_characteristic,Rating_num_characteristic
Use UserID Search Corresponding User-Based Feature
Basic Data: rating.csv--[UserID, MovieID] Pairs
Use MovieID Search Corresponding Movies-Based Feature
[User-Based Feature, Movies-Based Feature] Vactors
['user_avg_rating', 'user_avg_rating_squared', 'user_min_rating', 'user_max_rating', 'user_median_rating', 'user_rating_range', 'user_rating_std','user_num_movies_rated', 'user_num_movies_rated_squared',
'title', 'Adventure', 'Animation', 'Children', 'Comedy', 'Fantasy', 'Romance','Drama', 'Action', 'Crime', 'Thriller', 'Horror', 'Mystery', 'Sci-Fi','IMAX', 'Documentary', 'War', 'Musical', 'Western', 'Film-Noir','1870.0', '1880.0', '1890.0', '1900.0', '1910.0', '1920.0', '1930.0', '1940.0', '1950.0', '1960.0', '1970.0', '1980.0', '1990.0', '2000.0', '2010.0']
...

Figure 20: Feature engineering

4.3.3 Model Training:

After dividing the training dataset into training and test sets, we tested the linear regression model's predictive performance on sklearn and found it to be unsatisfactory. This was primarily due to the model's simplicity, resulting in underfitting.

Subsequently, we opted for a more complex decision tree regression model, training the decision tree model in pyspark.ml. Attention was mainly paid to DataFrame settings and the use of the transform method. The DataFrame is a fundamental data structure that stores objects in a Spark MLlib-specific Vector type, while the Transformer (transform() method) primarily manipulates data in DataFrame, used for feature transformation and applying trained models, converting feature datasets into data with predicted results.



```python
from pyspark.ml.feature import VectorAssembler
from pyspark.sql.functions import col

# Read the CSV file
data = spark.read.csv("training.csv", header=True, inferSchema=True)
# Select feature columns (all columns except the label column 'rating')
feature_cols = [col for col in data.columns if col != 'rating']
# Merge feature columns into a single feature vector column
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
data = assembler.transform(data)
# Rename the label column to 'label'
data = data.withColumnRenamed("rating", "label")
# Select "label" and "features" columns
data = data.select("label", "features")
# Fit the model using LogisticRegression
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
lrModel = lr.fit(data)
```

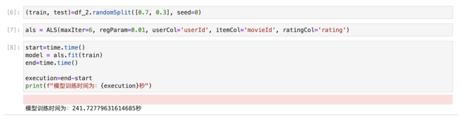
Figure 21: Important methods when training model

5 RESULTS: EFFICIENCY AND ACCURACY

5.1 Experiment 1 Results: Efficiency Comparison

By measuring the runtime in both modes, the following results were obtained:

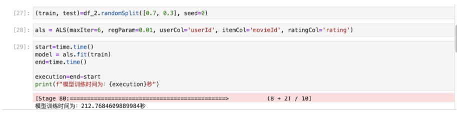
In the local mode, the runtime for the ALS model training process was 241.7278 seconds.



```
[6]: (train, test)=df_2.randomSplit([0.7, 0.3], seed=0)

[7]: als = ALS(maxIter=6, regParam=0.01, userCol='userId', itemCol='movieId', ratingCol='rating')

[8]: start=time.time()
     model = als.fit(train)
     end=time.time()

     execution=end-start
     print(f"模型训练时间为：{execution}秒")

     模型训练时间为：241.72779631614685秒
```

Figure 22: Local mode result

In the cluster mode, the runtime for the ALS model training process was 212.7685 seconds.



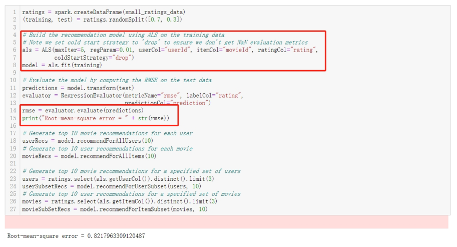
```
[27]: (train, test)=df_2.randomSplit([0.7, 0.3], seed=0)

[28]: als = ALS(maxIter=6, regParam=0.01, userCol='userId', itemCol='movieId', ratingCol='rating')

[29]: start=time.time()
      model = als.fit(train)
      end=time.time()

      execution=end-start
      print(f"模型训练时间为：{execution}秒")

      [Stage 80:==========================================>          (8 + 2) / 10]
      模型训练时间为：212.7684609889984秒
```

Figure 23: Cluster mode result

In summary, within a more complex computational process, the efficiency difference between distributed and centralized computing is notable. Under the same conditions of environment, task, method, and data, distributed computing exhibits higher efficiency compared to centralized computing.

5.2 Experiment 2 Results: Accuracy Comparison

Comparing the RMSE obtained by both methods on the same test set, trained on the same training set: With automatic matrix decomposition for generating implicit features using well-tuned hyperparameters, the RMSE for the ALS model tested on the test set was 0.8218.

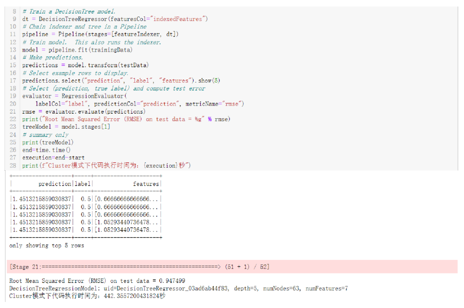


```python
ratings = spark.createDataFrame(small_ratings_data)
(training, test) = ratings.randomSplit([0.7, 0.3])

# Build the recommendation model using ALS on the training data
# Note we set cold start strategy to 'drop' to ensure we don't get NaN evaluation metrics
als = ALS(maxIter=5, regParam=0.01, userCol="userId", itemCol="movieId", ratingCol="rating",
          coldStartStrategy="drop")
model = als.fit(training)

# Evaluate the model by computing the RMSE on the test data
predictions = model.transform(test)
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
                                predictionCol="prediction")
rmse = evaluator.evaluate(predictions)
print("Root-mean-square error = " + str(rmse))

# Generate top 10 movie recommendations for each user
userRecs = model.recommendForAllUsers(10)
# Generate top 10 user recommendations for each movie
movieRecs = model.recommendForAllItems(10)

# Generate top 10 movie recommendations for a specified set of users
users = ratings.select(als.getUserCol()).distinct().limit(3)
userSubsetRecs = model.recommendForUserSubset(users, 10)
# Generate top 10 user recommendations for a specified set of movies
movies = ratings.select(als.getItemCol()).distinct().limit(3)
movieSubSetRecs = model.recommendForItemSubset(movies, 10)
```

```
Root-mean-square error = 0.8217963309120487
```

Figure 24: ALS result

After manual feature engineering to construct input feature vectors with similarly well-tuned hyperparameters, the RMSE for the decision tree regression model trained on the same test set was 0.9474.



```python
# Train a DecisionTree model.
dt = DecisionTreeRegressor(featuresCol="indexedFeatures")
# Chain indexer and tree in a Pipeline
pipeline = Pipeline(stages=[featureIndexer, dt])
# Train model.  This also runs the indexer.
model = pipeline.fit(trainingData)
# Make predictions.
predictions = model.transform(testData)
# Select example rows to display.
predictions.select("prediction", "label", "features").show(5)
# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
treeModel = model.stages[1]
# summary only
print(treeModel)
end=time.time()
execution=end-start
print(f"Cluster模式下代码执行时间为：{execution}秒")
```

```
+------------------+-----+--------------------+
|        prediction|label|            features|
+------------------+-----+--------------------+
|1.4513215859030837|  0.5|[0.66666666666666...|
|1.4513215859030837|  0.5|[0.66666666666666...|
|1.4513215859030837|  0.5|[0.66666666666666...|
|1.4513215859030837|  0.5|[1.05293440736478...|
|1.4513215859030837|  0.5|[1.05293440736478...|
+------------------+-----+--------------------+
only showing top 5 rows

[Stage 21:====================================================> (51 + 1) / 52]

Root Mean Squared Error (RMSE) on test data = 0.947499
DecisionTreeRegressionModel: uid=DecisionTreeRegressor_03ad6ab44f83, depth=5, numNodes=63, numFeatures=7
Cluster模式下代码执行时间为：442.3557200431824秒
```

Figure 25: Decision Tree result

In summary, under similar tuning levels and data foundations, the ALS model's RMSE was more than 0.1 lower than the classical decision tree model, indicating that to some extent, the feature quality generated by ALS automatically exceeds the level achieved by manual feature engineering constrained by computational resources.

Therefore, we attempted to provide potential reasons: Due to resource constraints, the feature engineering conducted might not have been comprehensive enough to capture crucial data characteristics. ALS-generated latent features might encompass implicit behaviors and attributes (such as clicks, browsing history, etc.) that are difficult to leverage through conventional feature engineering.