Assignment 2 APACHE-SPARK

Yanting Chen

1 Part A

1.1 Count the odd and even numbers using the file 'integer.txt' and download it from Quercus. Show your code and output.

- Code

```python
# Import necessary libraries
from pyspark.sql import SparkSession

# Create Spark session
spark = SparkSession.builder.appName("OddEvenCount").getOrCreate()

# Read the integer.txt file
df = spark.read.text("/FileStore/tables/integer.txt")

# Show the content of the dataframe
df.show()

# Convert the dataframe to RDD and extract the integers
numbers_rdd = df.rdd.map(lambda row: int(row[0]))

# Function to determine if a number is odd or even
def odd_even(num):
    if num % 2 == 0:
        return ("Even", 1)
    else:
        return ("Odd", 1)

# Map the numbers to odd/even and reduce by key to count them
count_rdd = numbers_rdd.map(odd_even).reduceByKey(lambda a, b: a + b)

# Collect the result
result = count_rdd.collect()

# Print the result
for key, count in result:
    print(f"{key}: {count}")
```

- Output
  Odd: 496 Even: 514

- Screen shot

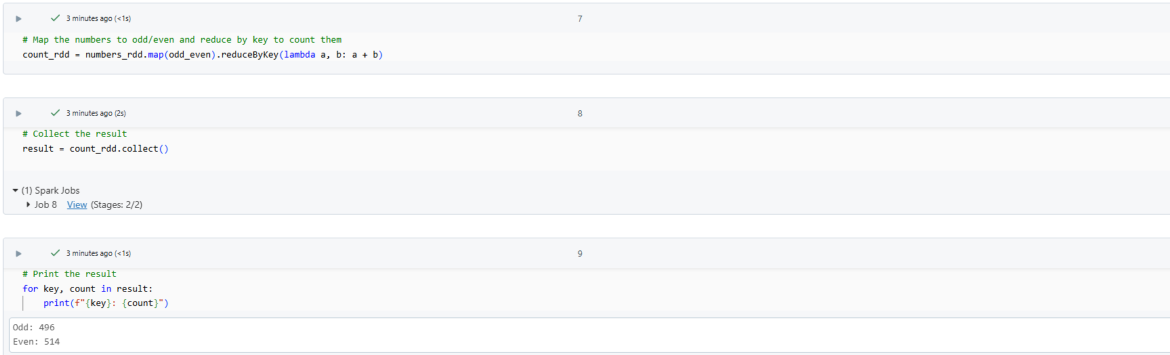


Figure 1: result

1.2 Calculate the salary sum per department using the file 'salary.txt' and download it from Quercus. Show the department name and salary sum. Show your code and output.

- Code

```
# Import necessary libraries
from pyspark.sql import SparkSession
from pyspark.sql.functions import split, col, sum as spark_sum

# Create Spark session
spark = SparkSession.builder.appName("SalarySumPerDepartment").getOrCreate()

# Read the salary.txt file
df = spark.read.text("/FileStore/tables/salary.txt")

# Split the lines into department and salary
split_col = split(df['value'], ' ')
df = df.withColumn('department', split_col.getItem(0))
df = df.withColumn('salary', split_col.getItem(1).cast('float'))

# Group by department and calculate the sum of salaries
result = df.groupBy('department').agg(spark_sum('salary').alias('salary_sum'))

# Show the result
display(result)
```

- Output

Department	**Salary Sum**
Sales	3488491.0
Developer	3221394.0
Research	3328284.0
Marketing	3158450.0
QA	3360624.0

Table 1: Summary of Salaries by Department

- Screen shot

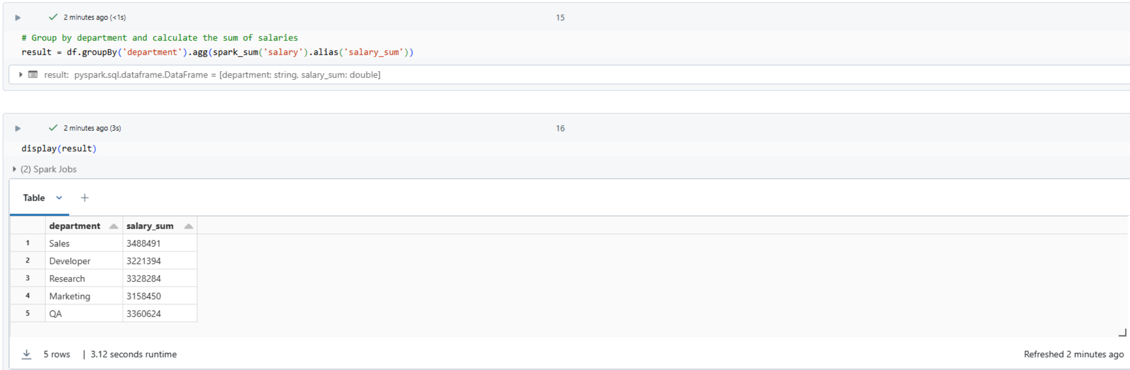


Figure 2: result 2

1.3 Implement MapReduce using PySpark on file 'shakespeare.txt' and download it from the Quercus. Show how many times these particular words appear in the document: Shakespeare, When, Lord, Library, GUTENBERG, WILLIAM, COLLEGE and WORLD. (Count exact words only)

- Code

```
# Import necessary libraries
from pyspark.sql import SparkSession

# Create Spark session
spark = SparkSession.builder.appName("WordCount").getOrCreate()

# Read the shakespeare.txt file
df = spark.read.text("/FileStore/tables/shakespeare_1.txt")

# Show the content of the dataframe
display(df)

# Convert the dataframe to RDD
lines_rdd = df.rdd.map(lambda row: row[0])

# List of words to count
words_to_count = ["Shakespeare", "When", "Lord", "Library", "GUTENBERG", "WILLIAM",
    "COLLEGE", "WORLD"]

# Function to count specified words in a line
def word_count(line, words):
    line_words = line.split()
    word_counts = []
    for word in words:
        count = line_words.count(word)
        if count > 0:
            word_counts.append((word, count))
    return word_counts

# FlatMap the lines RDD to count occurrences of the specified words
word_counts_rdd = lines_rdd.flatMap(lambda line: word_count(line, words_to_count))

```

```python
# Reduce by key to sum the counts for each word
word_counts = word_counts_rdd.reduceByKey(lambda a, b: a + b)

# Collect the result
result = word_counts.collect()

# Print the result
for word, count in result:
    print(f"{word}: {count}")
```

- Output
  Shakespeare: 22
  GUTENBERG: 99
  Library: 2
  WILLIAM: 115
  WORLD: 98
  COLLEGE: 98
  When: 393
  Lord: 341

- Screen shot

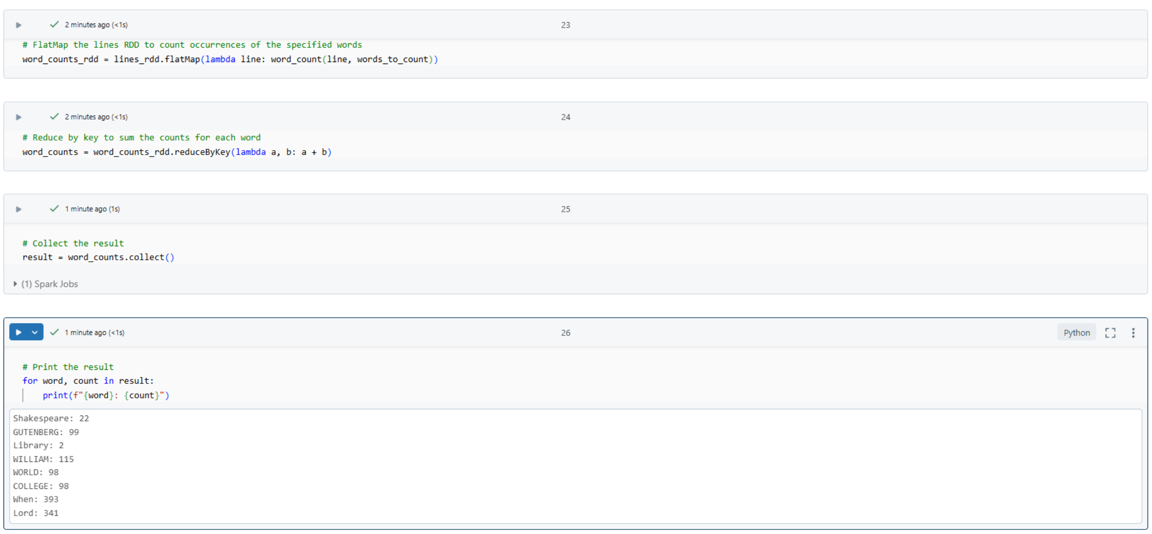


Figure 3: result 3

1.4 Calculate the top 15 and bottom 15 words using the file 'shakespeare.txt' and download it from Quercus. Show 15 words with the most count and 15 words with the least count. You can limit by 15 in ascending and descending order of count. Show your code and output.

- Code

```python
# Import necessary libraries
from pyspark.sql.functions import explode, split, col
from pyspark.sql.types import StringType

# Create Spark session
spark = SparkSession.builder.appName("TopBottomWordCount").getOrCreate()
```

```python

# Show the content of the dataframe
display(df)

# Split lines into words
words_df = df.select(explode(split(col("value"), "\\s+")).alias("word"))

# Remove any empty strings resulting from multiple spaces
words_df = words_df.filter(words_df.word != "")

# Count the occurrences of each word
word_counts_df = words_df.groupBy("word").count()

# Sort the words by count in descending order for the top 15
top_15_words = word_counts_df.orderBy(col("count").desc()).limit(15)

# Sort the words by count in ascending order for the bottom 15
bottom_15_words = word_counts_df.orderBy(col("count").asc()).limit(15)

# Show the top 15 words
print("Top 15 words:")
top_15_words.show()

# Show the bottom 15 words
print("Bottom 15 words:")
bottom_15_words.show()
```

- Output

Top 15 Words		**Bottom 15 Words**	
Word	**Count**	**Word**	**Count**
the	11397	soundness	1
and	8777	spoke;	1
I	8556	Paris?	1
of	7873	AWAY	1
to	7421	occidental	1
a	5672	pluck,	1
my	4913	commanders	1
in	4600	lust.	1
you	4060	'Demand	1
And	3547	commits.	1
that	3522	inner	1
is	3481	gav'st,	1
his	3226	DERCETAS,	1
with	3175	online	1
not	3129	Acquaint	1

Table 2: Top 15 and Bottom 15 Words

- Screen shot

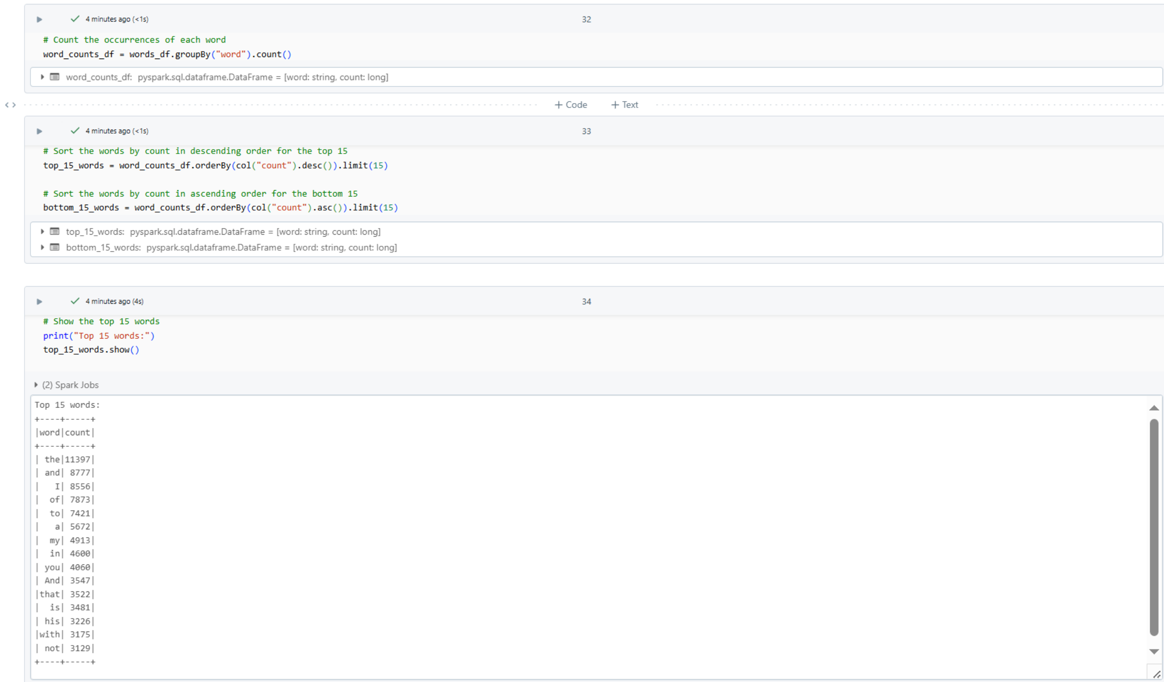


Figure 4: top 15

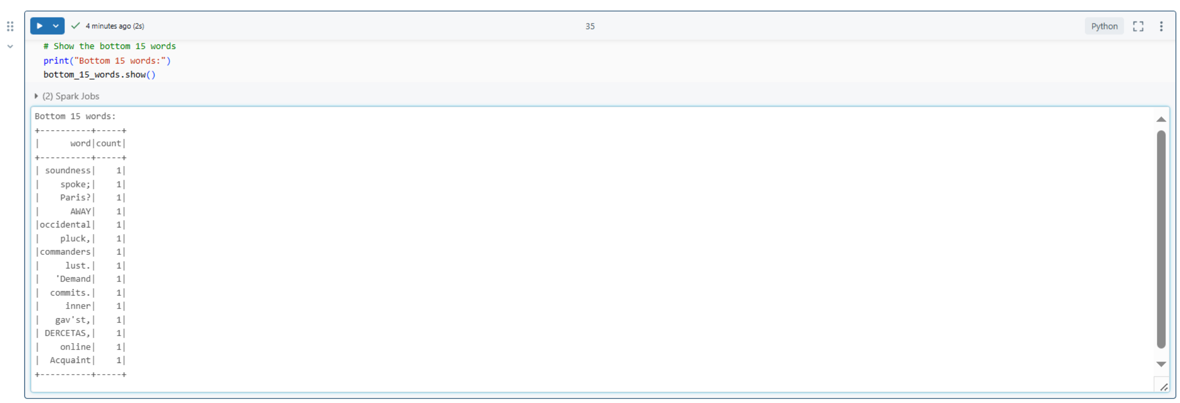


Figure 5: bottom 15

2 Part B

2.1 Describe your data. Calculate the top 12 movies with the highest ratings and the top 12 users who provided the highest ratings. Show your code and output.

- Code

```python
# Import necessary libraries
from pyspark.sql import SparkSession
```

```
from pyspark.sql.functions import desc, avg

# Initialize Spark session
spark = SparkSession.builder.appName("RecommenderSystem").getOrCreate()

# Load the data
data = spark.read.csv("/FileStore/tables/movies.csv", header=True, inferSchema=True)

# Display the data
display(data)

# Describe the data
display(data.describe())

# Calculate the average rating for each movie
avg_movie_ratings = data.groupBy("movieId").agg(avg("rating").alias("avg_rating"))

# Get the top 12 movies with the highest average ratings
top_12_movies = avg_movie_ratings.orderBy(desc("avg_rating")).limit(12)

# Show the top 12 movies
display(top_12_movies)


# Calculate the average rating provided by each user
avg_user_ratings = data.groupBy("userId").agg(avg("rating").alias("avg_rating"))

# Get the top 12 users who provided the highest average ratings
top_12_users = avg_user_ratings.orderBy(desc("avg_rating")).limit(12)

# Show the top 12 users
display(top_12_users)
```

- Output

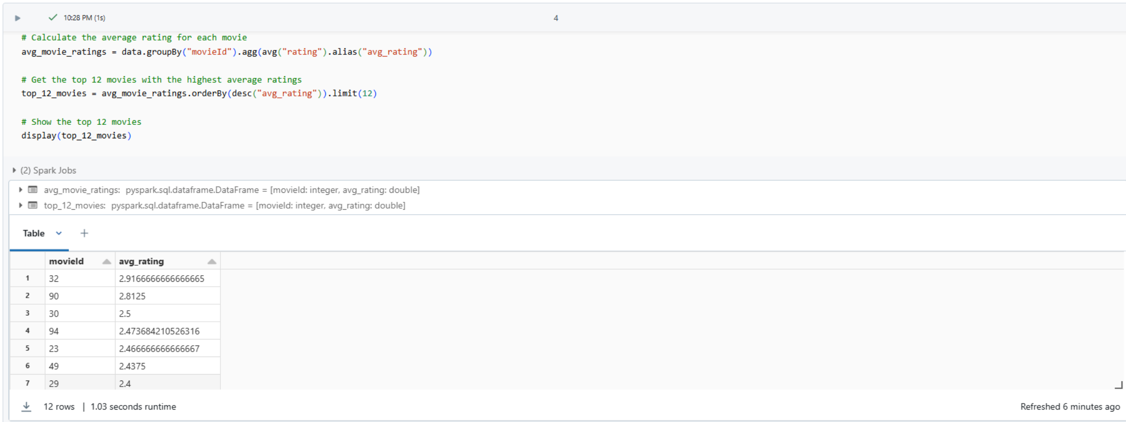
movieId	avg_rating
32	2.916667
90	2.812500
30	2.500000
94	2.473684
23	2.466667
49	2.437500
29	2.400000
18	2.400000
52	2.357143
53	2.250000
62	2.250000
92	2.214286

Table 3: Average Ratings for Movies

userId	avg_rating
11	2.285714
26	2.204082
22	2.160714
23	2.134615
2	2.065217
17	1.956522
8	1.897959
24	1.884615
12	1.854545
3	1.833333
29	1.826087
28	1.820000

Table 4: Average Ratings for Users

- Screen shot



10:28 PM (1s)

```
# Calculate the average rating for each movie
avg_movie_ratings = data.groupBy("movieId").agg(avg("rating").alias("avg_rating"))

# Get the top 12 movies with the highest average ratings
top_12_movies = avg_movie_ratings.orderBy(desc("avg_rating")).limit(12)

# Show the top 12 movies
display(top_12_movies)
```

(2) Spark Jobs

avg_movie_ratings: pyspark.sql.dataframe.DataFrame = [movieId: integer, avg_rating: double]

top_12_movies: pyspark.sql.dataframe.DataFrame = [movieId: integer, avg_rating: double]

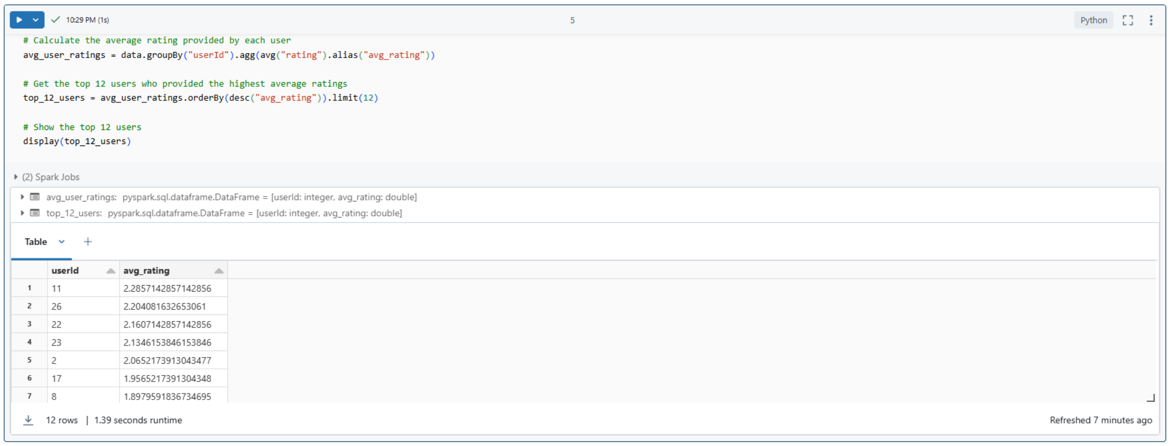
Table

	movieId	avg_rating
1	32	2.9166666666666665
2	90	2.8125
3	30	2.5
4	94	2.473684210526316
5	23	2.466666666666667
6	49	2.4375
7	29	2.4

12 rows | 1.03 seconds runtime

Refreshed 6 minutes ago

Figure 6: Top 12 movies with the highest ratings



10:29 PM (1s)

Python

```
# Calculate the average rating provided by each user
avg_user_ratings = data.groupBy("userId").agg(avg("rating").alias("avg_rating"))

# Get the top 12 users who provided the highest average ratings
top_12_users = avg_user_ratings.orderBy(desc("avg_rating")).limit(12)

# Show the top 12 users
display(top_12_users)
```

(2) Spark Jobs

avg_user_ratings: pyspark.sql.dataframe.DataFrame = [userId: integer, avg_rating: double]

top_12_users: pyspark.sql.dataframe.DataFrame = [userId: integer, avg_rating: double]

Table

	userId	avg_rating
1	11	2.2857142857142856
2	26	2.204081632653061
3	22	2.1607142857142856
4	23	2.1346153846153846
5	2	2.0652173913043477
6	17	1.9565217391304348
7	8	1.8979591836734695

12 rows | 1.39 seconds runtime

Refreshed 7 minutes ago

Figure 7: Top 12 users who provided the highest ratings

2.2 Split the dataset into train and test. Try 2 different combinations for e.g. (60/40, 70/30, 75/25 and 80/20). (Train your model and use collaborative filtering approach on 70 percent of your data and test with the other 30 percent and so on). Show your code and output.

- Code

```
from pyspark.sql import SparkSession
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS


# Function to train ALS model and evaluate performance
def train_and_evaluate(data, train_ratio, test_ratio):
    # Split the data into training and test sets
    (training, test) = data.randomSplit([train_ratio, test_ratio])

    # Build the recommendation model using ALS on the training data
    als = ALS(maxIter=10, regParam=0.1, userCol="userId", itemCol="movieId",
        ratingCol="rating", coldStartStrategy="drop")
```

```python

    # Train the model
    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(f"Root-mean-square error for {train_ratio*100}/{test_ratio*100} split =
        {rmse}")

    return rmse

# Perform training and evaluation for different splits
rmse_70_30 = train_and_evaluate(data, 0.7, 0.3)
rmse_80_20 = train_and_evaluate(data, 0.8, 0.2)
```

- Output

  Root-mean-square error for 70.0/30.0 split = 1.0436185387207275
  Root-mean-square error for 80.0/20.0 split = 0.9892476560245133

- Screen shot

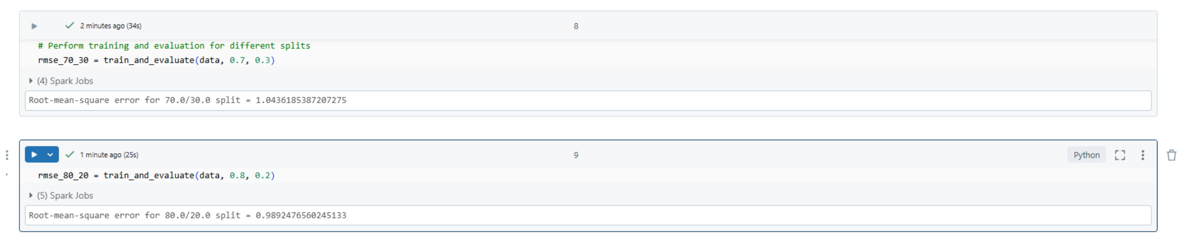


Figure 8: output

2.3 Explain MSE, RMSE and MAE. Compare and evaluate both of your models with evaluation metrics (RMSE or MAE), show your code and print your results. Describe which one works better and why

- Explain MSE, RMSE and MAE

  Mean Squared Error (MSE)

  Definition: Mean Squared Error is the average of the squares of the errors, where the error is the difference between the predicted value and the actual value.

  Formula:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

  Where:

  - n is the number of observations.
  - y_i is the actual value.
  - $\hat{y}_i$ is the predicted value.

Characteristics:

- MSE gives a higher weight to larger errors due to the squaring term, which can be useful if large errors are particularly undesirable.
- It is always non-negative, and values closer to 0 indicate a better fit.

Root Mean Squared Error (RMSE)

Definition: Root Mean Squared Error is the square root of the average of the squares of the errors. It is essentially the square root of MSE.

Formula:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Where:

- The terms are the same as in the MSE formula.

Characteristics:

- RMSE is in the same units as the target variable, making it more interpretable than MSE.
- Like MSE, it is sensitive to large errors due to the squaring of each term.

Mean Absolute Error (MAE)

Definition: Mean Absolute Error is the average of the absolute differences between the predicted values and the actual values.

Formula:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Where:

- The terms are the same as in the MSE formula, except we take the absolute value of the error instead of squaring it.

Characteristics:

- MAE is more robust to outliers than MSE and RMSE because it does not square the errors.
- It provides a linear score which means all individual differences are weighted equally in the average.

Comparison and Usage

- **MSE** is useful when you want to heavily penalize larger errors and is commonly used in mathematical and statistical contexts where this characteristic is desired.
- **RMSE** is often preferred in contexts where the interpretability of error units is important, as it provides a measure in the same units as the target variable.
- **MAE** is useful when you need a metric that is less sensitive to outliers and gives an equal weight to all errors.

- Code

  I will use both RMSE and MAE to evaluate our models for the two different splits: 70/30 and 80/20.

```python
# Function to train ALS model and evaluate performance using RMSE and MAE
def train_and_evaluate(data, train_ratio, test_ratio):
    # Split the data into training and test sets
    (training, test) = data.randomSplit([train_ratio, test_ratio])

    # Build the recommendation model using ALS on the training data
    als = ALS(maxIter=10, regParam=0.1, userCol="userId", itemCol="movieId",
        ratingCol="rating", coldStartStrategy="drop")

    # Train the model
    model = als.fit(training)

    # Evaluate the model by computing the RMSE and MAE on the test data
    predictions = model.transform(test)

    evaluator_rmse = RegressionEvaluator(metricName="rmse", labelCol="rating",
        predictionCol="prediction")
    rmse = evaluator_rmse.evaluate(predictions)

    evaluator_mae = RegressionEvaluator(metricName="mae", labelCol="rating",
        predictionCol="prediction")
    mae = evaluator_mae.evaluate(predictions)

    print(f"Evaluation metrics for {train_ratio*100}/{test_ratio*100} split:")
    print(f"Root-mean-square error (RMSE) = {rmse}")
    print(f"Mean absolute error (MAE) = {mae}")

    return rmse, mae

# Perform training and evaluation for different splits
rmse_70_30, mae_70_30 = train_and_evaluate(data, 0.7, 0.3)
rmse_80_20, mae_80_20 = train_and_evaluate(data, 0.8, 0.2)

# Compare results
print("\nComparison of RMSE and MAE for different splits:")
print(f"70/30 split - RMSE: {rmse_70_30}, MAE: {mae_70_30}")
print(f"80/20 split - RMSE: {rmse_80_20}, MAE: {mae_80_20}")
```

- Output

Evaluation metrics for 70.0/30.0 split:
Root-mean-square error (RMSE) = 1.1907133944556987
Mean absolute error (MAE) = 0.7950203727834441
Evaluation metrics for 80.0/20.0 split:
Root-mean-square error (RMSE) = 0.9469574329220144
Mean absolute error (MAE) = 0.6663839472865416

Comparison of RMSE and MAE for different splits:
70/30 split - RMSE: 1.1907133944556987, MAE: 0.7950203727834441
80/20 split - RMSE: 0.9469574329220144, MAE: 0.6663839472865416

- Screen shot

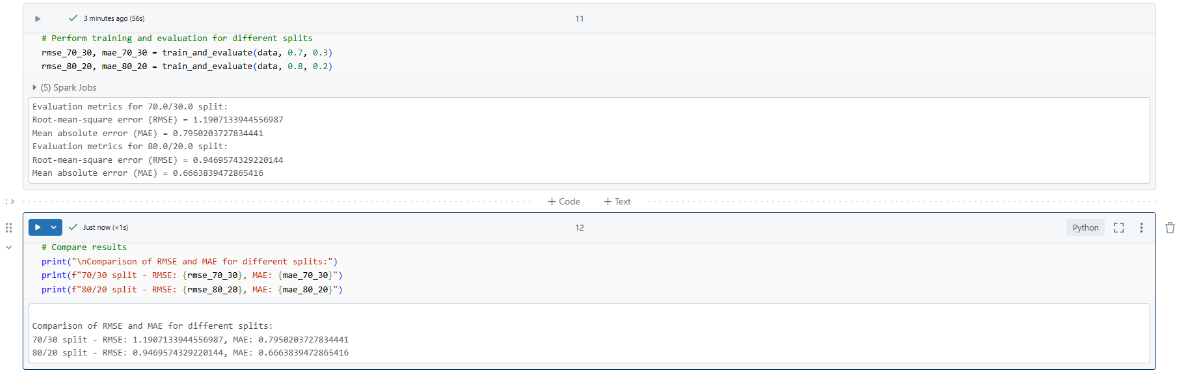


Figure 9: output

- Describe which one works better and why

MAE provides a more straightforward interpretation of the average error, showing that the average prediction error is around 0.7950 for the 70/30 split and 0.6664 for the 80/20 split.

RMSE indicates the error magnitude, showing larger values due to squaring errors. For datasets with significant outliers, RMSE would give a clearer indication of the model's sensitivity to these outliers.

In this case, both metrics indicate that the model trained with the 80/20 split performs better, but MAE might be more informative for understanding the average prediction error directly. If RMSE is significantly higher than MAE, it indicates the presence of outliers since RMSE penalizes larger errors more due to squaring the differences. In the given results, RMSE values are higher than MAE values, indicating that there might be some outliers in the data.

2.4 Now tune the parameters of your algorithm to get the best set of parameters. Explain different parameters of the algorithm which you have used for tuning your algorithm. Evaluate all your models again. Show your code with the best values and output.

- Explanation of Parameters Used for Tuning

1. rank

Determines the number of latent factors in the model. A higher number can capture more complex relationships but may also increase the risk of overfitting.

2. maxIter

Number of iterations the algorithm runs to optimize the latent factors. More iterations can improve convergence but also increase computation time.

3. regParam

Controls the regularization of the model to prevent overfitting. Higher values imply stronger regularization.

- Code
  I will use cross-validation to find the best combination of these parameters.

```python

from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# Split the data into training and test sets (80/20 split)
(training, test) = data.randomSplit([0.8, 0.2])

# Build the recommendation model using ALS on the training data
als = ALS(userCol="userId", itemCol="movieId", ratingCol="rating",
    coldStartStrategy="drop")

# Define the evaluator
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
    predictionCol="prediction")

# Define the parameter grid for tuning
paramGrid = ParamGridBuilder() \
    .addGrid(als.rank, [10, 50, 100]) \
    .addGrid(als.maxIter, [10, 15, 20]) \
    .addGrid(als.regParam, [0.01, 0.1, 1.0]) \
    .build()

# Create a CrossValidator
crossval = CrossValidator(estimator=als,
                          estimatorParamMaps=paramGrid,
                          evaluator=evaluator,
                          numFolds=3)

# Run cross-validation, and choose the best set of parameters
cvModel = crossval.fit(training)

# Make predictions on the test data
predictions = cvModel.transform(test)

# Evaluate the model
rmse = evaluator.evaluate(predictions)
print(f"Best Model Root-mean-square error (RMSE) = {rmse}")

# Show the best parameters
best_model = cvModel.bestModel
print(f"Best rank: {best_model._java_obj.parent().getRank()}")
print(f"Best maxIter: {best_model._java_obj.parent().getMaxIter()}")
print(f"Best regParam: {best_model._java_obj.parent().getRegParam()}")
```

- Output

  Best Model Root-mean-square error (RMSE) = 0.8919648948409354
  Best rank: 10
  Best maxIter: 20
  Best regParam: 0.1

- Screen shot

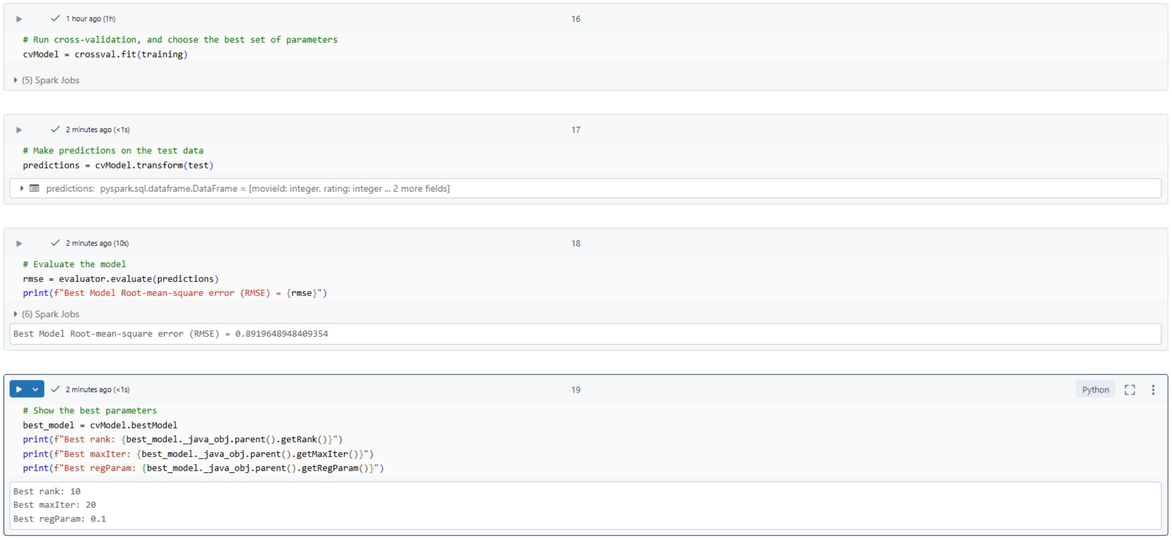


Figure 10: output

2.5 Calculate the top 12 movie recommendations for user ID 10 and user ID 12. Show your code and output.

- Code
  I will use the trained ALS model to generate recommendations

```python

# Get the top 12 movie recommendations for user ID 10 and user ID 12
user_10_recs = best_model.recommendForUserSubset(training.filter(training.userId ==
    10), 12)
user_12_recs = best_model.recommendForUserSubset(training.filter(training.userId ==
    12), 12)

# Show recommendations for user ID 10
print("Top 12 movie recommendations for user ID 10:")
user_10_recs.show(truncate=False)

# Show recommendations for user ID 12
print("Top 12 movie recommendations for user ID 12:")
user_12_recs.show(truncate=False)
```

- Output

userId	recommendations
10	{92, 3.396891}, {40, 2.9589322}, {49, 2.869875}, {2, 2.8113012}, {81, 2.6793509}, {89, 2.5736306} {25, 2.5259755}, {91, 2.4805465}, {26, 2.4473178}, {62, 2.3785038}, {82, 2.3199089}, {4, 2.1039045}

Table 5: Top 12 Movie Recommendations for User ID 10

userId	recommendations
12	{46, 4.6264577}, {64, 4.198725}, {27, 4.187562}, {35, 3.9660196}, {55, 3.8417332}, {65, 3.796472} {48, 3.6769905}, {50, 3.5198574}, {49, 3.4262657}, {16, 3.3998501}, {94, 3.383744}, {90, 3.317638}

Table 6: Top 12 Movie Recommendations for User ID 12

- Screen shot

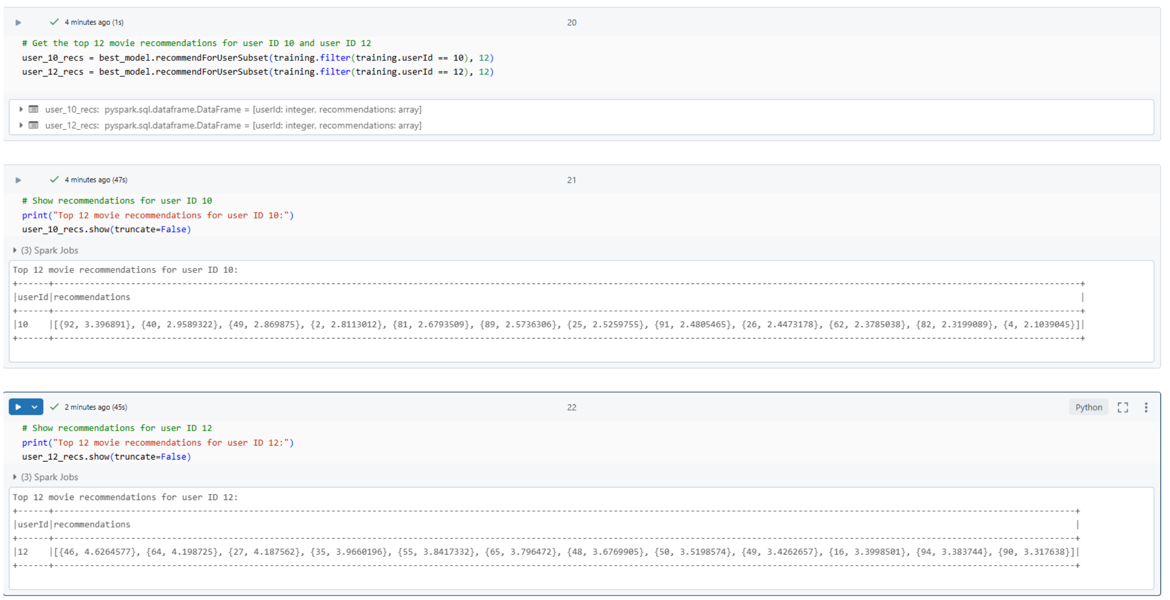


Figure 11: output