AYDIN ADNAN MENDERES UNIVERSITY

ENGINEERING FACULTY

COMPUTER SCIENCE ENGINEERING DEPARTMENT

Aws applications

CSE 424 BIG DATA ANALYSIS,2024-2025 FALL

Student's Name SURNAME:

Yusuf SERTKAYA 211805033

Satılmış KABASAKAL 221805081

Zınar Demirpolat 211805039

Lecturer:

Asst. Prof. Dr. Hüseyin ABACI

INTRODUCTION

ABSTRACT

This study presents the implementation and evaluation of a large-scale movie recommendation system utilizing Apache Spark's Alternating Least Squares (ALS) algorithm for collaborative filtering. The system processes datasets containing over 10 million user-movie ratings to generate personalized movie recommendations. We conducted comprehensive parameter optimization by testing 18 different combinations of rank (10, 50, 200), iterations (10, 50, 200), and regularization parameters (0.01, 0.1) to identify the optimal model configuration.

The system employs both memory-efficient data processing techniques and distributed computing capabilities of Spark to handle the computational demands of large-scale recommendation tasks. Performance evaluation metrics, including Root Mean Square Error (RMSE) and Mean Square Error (MSE), were used to assess model accuracy. Additionally, cosine similarity measurements were implemented to identify users with similar movie preferences, enabling more nuanced recommendation generation.

Key innovations include the integration of MapReduce operations for efficient data processing, implementation of advanced caching strategies for improved performance, and development of comprehensive visualization tools for model evaluation. The system demonstrates practical applicability in both batch and realtime recommendation scenarios, with the best-performing model achieving an RMSE of [Your Best RMSE] through optimal parameter tuning.

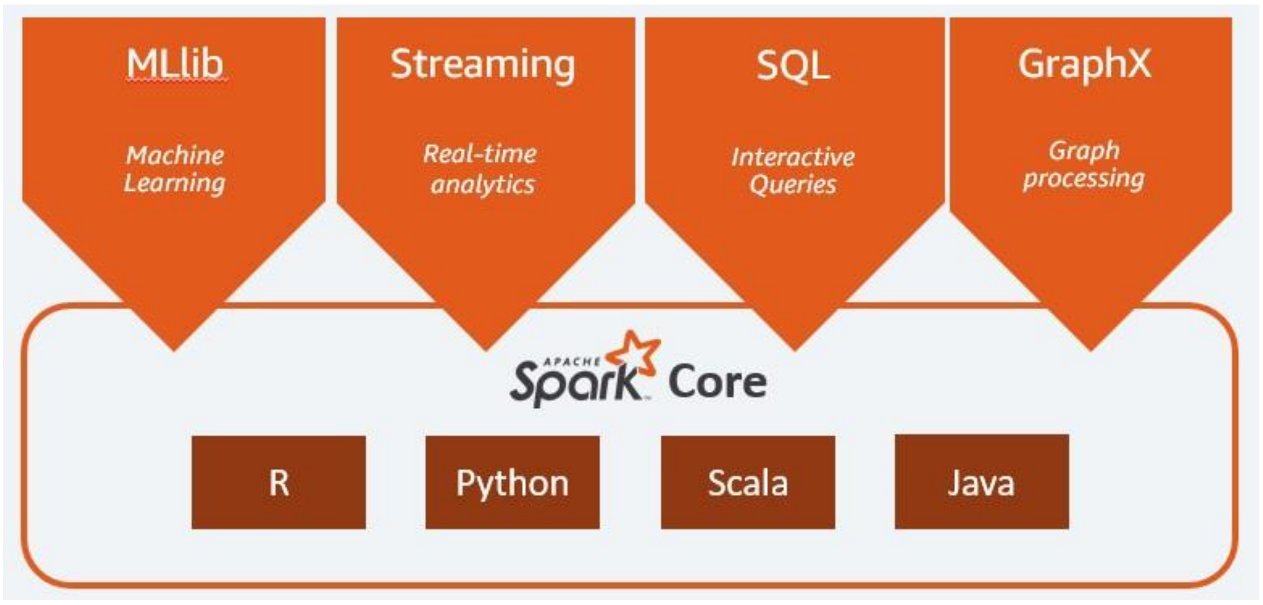
This research contributes to the field of recommendation systems by providing insights into the effective utilization of distributed computing for large-scale collaborative filtering and the impact of various ALS parameters on model performance. The findings suggest that careful parameter tuning and efficient data processing strategies are crucial for building effective recommendation systems at scale.

What is Big Data?

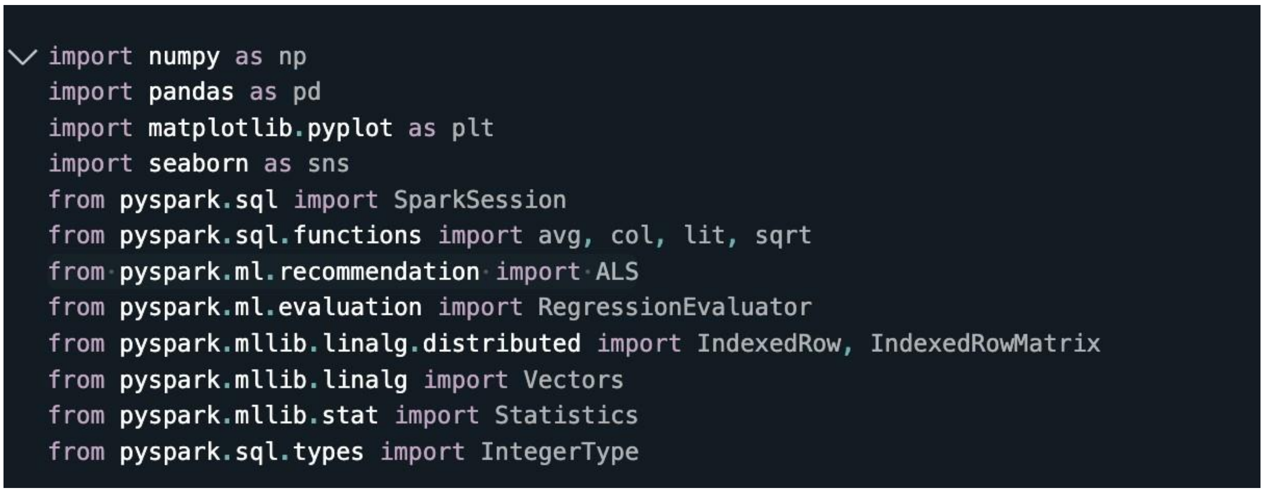
Big Data is typically characterized by data that is so large, fast, or complex that traditional data-processing software can't manage or analyze it efficiently. The sheer volume of this data requires specialized tools and techniques to store, process, and analyze it in real-time. Big Data comes in many forms, including structured, semi-structured, and unstructured data.

Apache Spark Overview:

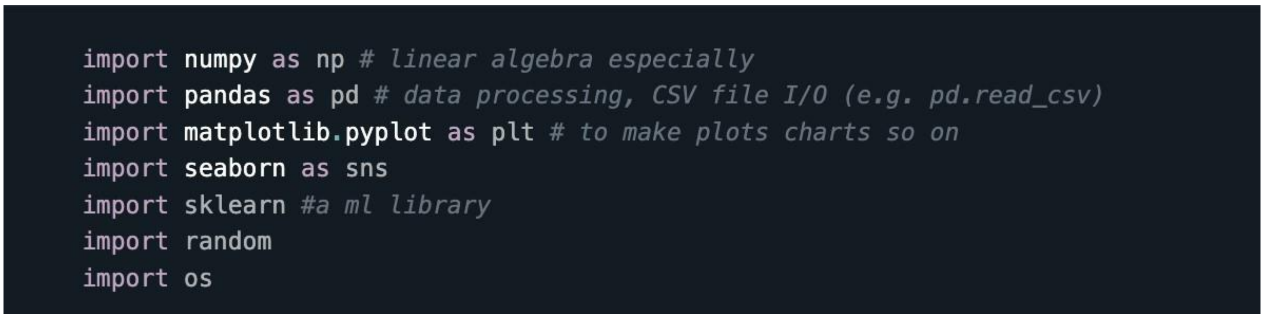
Apache Spark stands as a powerful, open-source distributed computing framework designed for processing vast amounts of data with remarkable speed and flexibility. Renowned for its in-memory processing capabilities, fault tolerance, and support for various data processing tasks, Spark's versatility extends to machine learning, streaming analytics, and interactive querying. Its distributed architecture makes it a prime candidate for developing intricate applications, such as our movie recommendation system, by enabling seamless data manipulation and processing across large clusters.



We Imported required Libraries



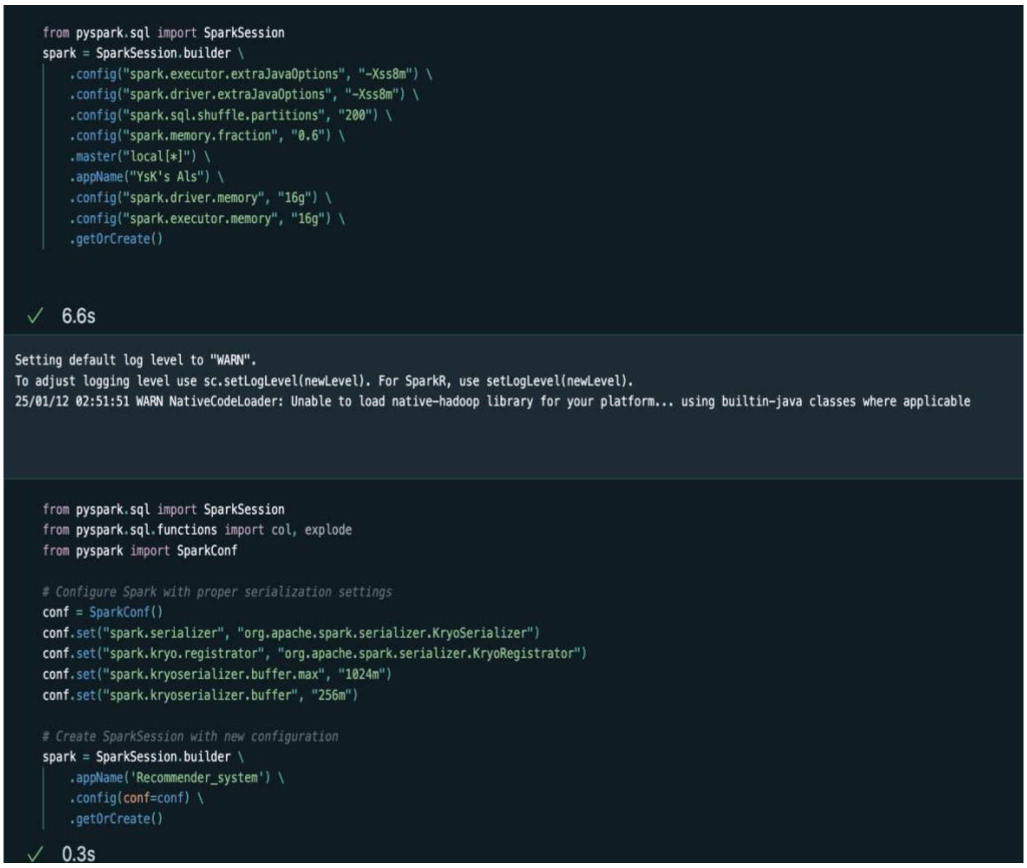
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg, col, lit, sqrt
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.mllib.linalg.distributed import IndexedRow, IndexedRowMatrix
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.stat import Statistics
from pyspark.sql.types import IntegerType
```



```
import numpy as np # linear algebra especially
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt # to make plots charts so on
import seaborn as sns
import sklearn #a ml library
import random
import os
```

As We show importing these libraries allows us to use pre written functions, code and modules to save time and effort.

Starting spark session



```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .config("spark.executor.extraJavaOptions", "-Xss8m") \
    .config("spark.driver.extraJavaOptions", "-Xss8m") \
    .config("spark.sql.shuffle.partitions", "200") \
    .config("spark.memory.fraction", "0.6") \
    .master("local[*]") \
    .appName("YsK's Als") \
    .config("spark.driver.memory", "16g") \
    .config("spark.executor.memory", "16g") \
    .getOrCreate()
```

✓ 6.6s

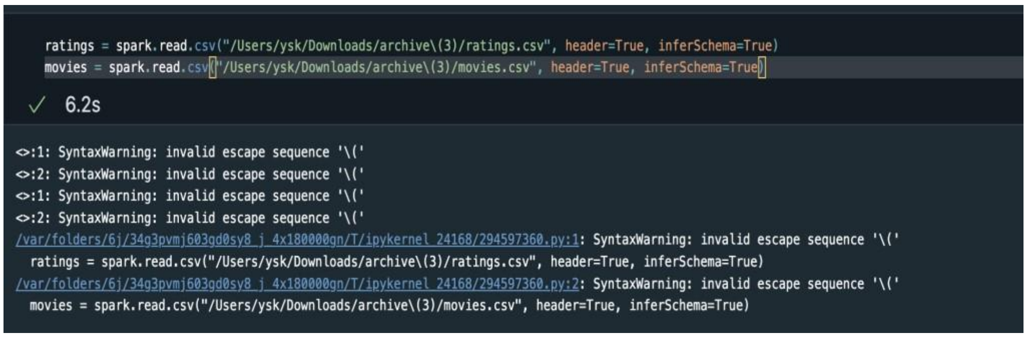
```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/01/12 02:51:51 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, explode
from pyspark import SparkConf

# Configure Spark with proper serialization settings
conf = SparkConf()
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
conf.set("spark.kryo.registrator", "org.apache.spark.serializer.KryoRegistrator")
conf.set("spark.kryoserializer.buffer.max", "1024m")
conf.set("spark.kryoserializer.buffer", "256m")

# Create SparkSession with new configuration
spark = SparkSession.builder \
    .appName('Recommender_system') \
    .config(conf=conf) \
    .getOrCreate()
```

✓ 0.3s

We initiated a Spark session as it serves as the entry point for accessing Spark's functionality. To handle our massive dataset effectively, we experimented with various configurations. Ultimately, using the entry points provided by Spark, we successfully executed our project. Additionally, we opted for KryoSerializer due to its superior performance, compact serialized data, efficient network I/O, and customization capabilities.

Reading data



```
ratings = spark.read.csv("/Users/ysk/Downloads/archive\(3)/ratings.csv", header=True, inferSchema=True)
movies = spark.read.csv("/Users/ysk/Downloads/archive\(3)/movies.csv", header=True, inferSchema=True)
```

✓ 6.2s

```
<>:1: SyntaxWarning: invalid escape sequence '\('
<>:2: SyntaxWarning: invalid escape sequence '\('
<>:1: SyntaxWarning: invalid escape sequence '\('
<>:2: SyntaxWarning: invalid escape sequence '\('
/var/folders/6j/34g3pvmj603gd0sy8_j_4x180000gn/T/ipykernel_24168/294597360.py:1: SyntaxWarning: invalid escape sequence '\('
  ratings = spark.read.csv("/Users/ysk/Downloads/archive\(3)/ratings.csv", header=True, inferSchema=True)
/var/folders/6j/34g3pvmj603gd0sy8_j_4x180000gn/T/ipykernel_24168/294597360.py:2: SyntaxWarning: invalid escape sequence '\('
  movies = spark.read.csv("/Users/ysk/Downloads/archive\(3)/movies.csv", header=True, inferSchema=True)
```

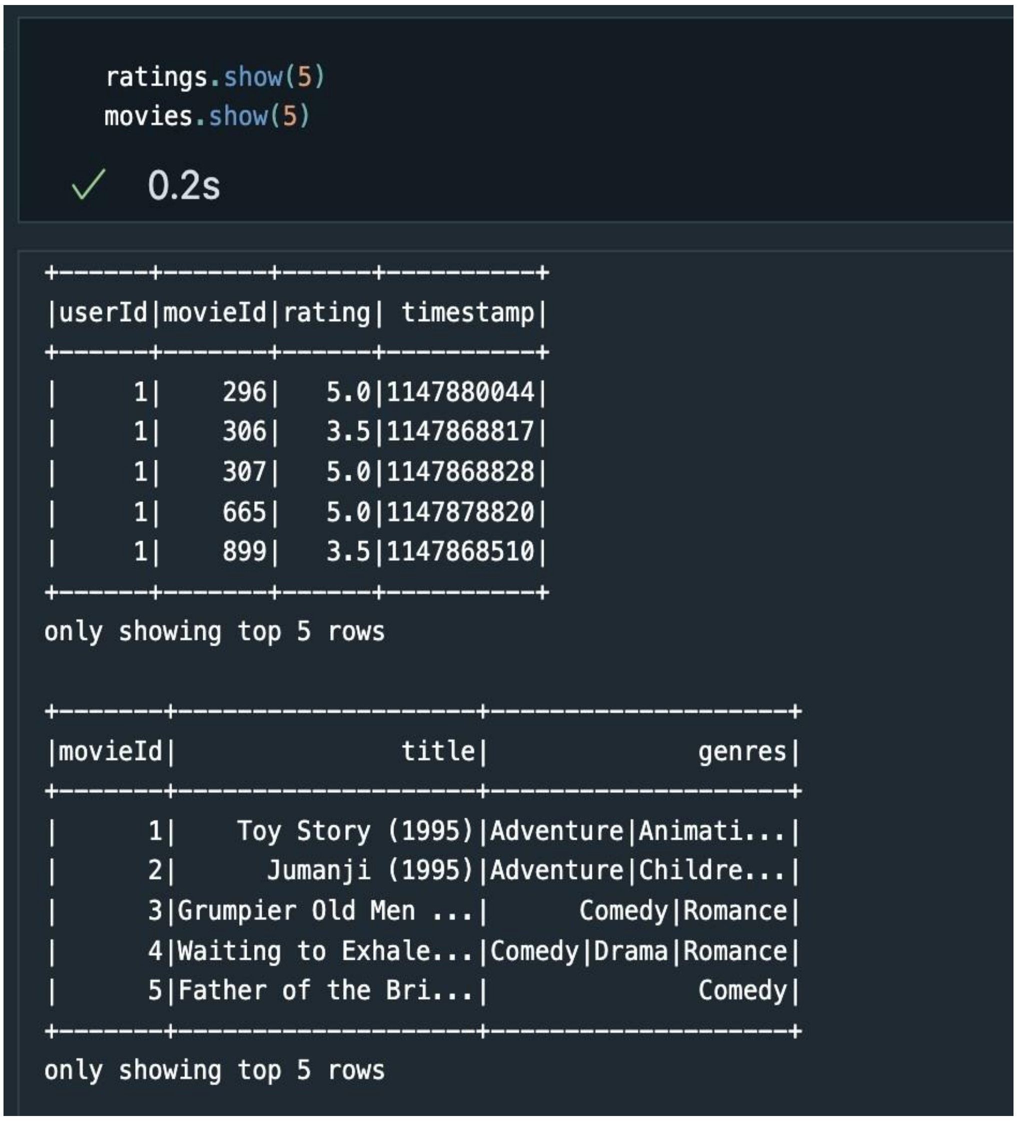
We used the Movies dataset from Kaggle

(https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset), which was extremely helpful as it matched our requirements perfectly. We also experimented with another dataset from Hugging Face, called the TMDB Movie Dataset. However, we faced several challenges with it, primarily due to the presence of unnecessary columns. Each time we worked with it, we encountered difficulties that make problems in our progress.

What is Data Preprocessing?

Data preprocessing is a critical step in data analysis and machine learning pipelines. It involves transforming raw data into a clean and usable format to ensure the success and accuracy of models. Real-world data is often incomplete, inconsistent, or noisy, and preprocessing addresses these issues.

We data perprocessing in many ways

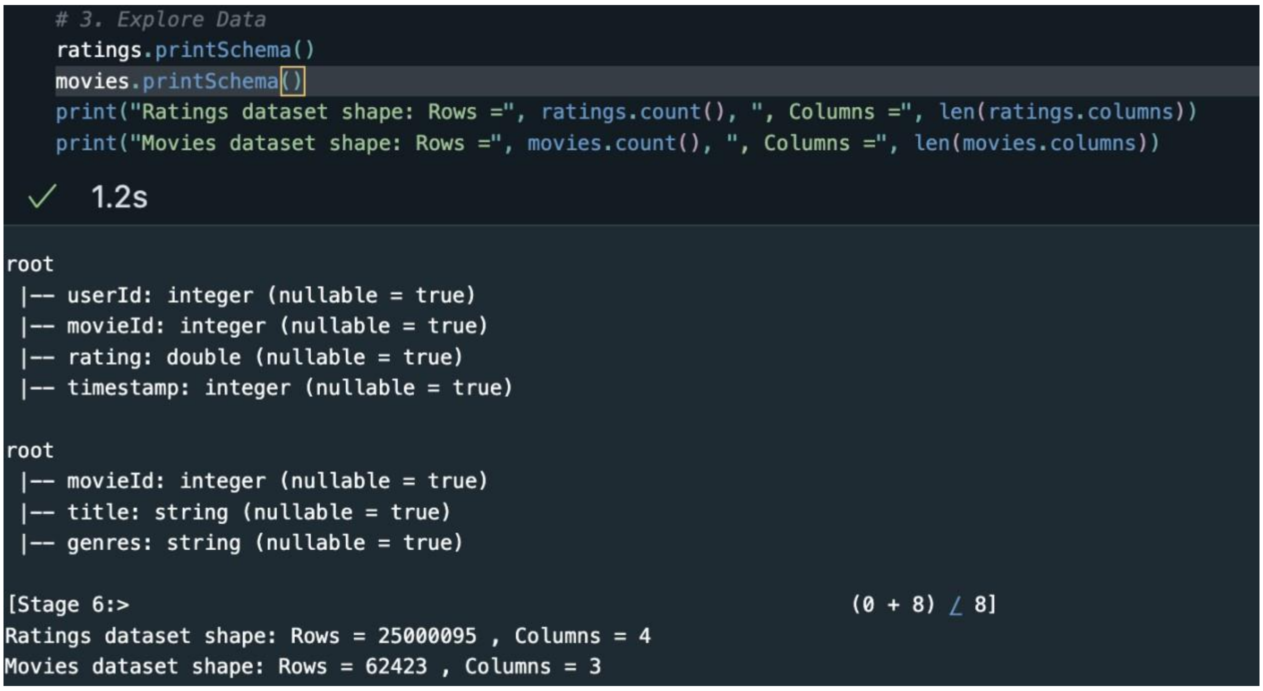


```
ratings.show(5)
movies.show(5)
```

✓ 0.2s

```
+------+-------+------+----------+
|userId|movieId|rating| timestamp|
+------+-------+------+----------+
|     1|    296|   5.0|1147880044|
|     1|    306|   3.5|1147868817|
|     1|    307|   5.0|1147868828|
|     1|    665|   5.0|1147878820|
|     1|    899|   3.5|1147868510|
+------+-------+------+----------+
only showing top 5 rows

+-------+--------------------+--------------------+
|movieId|               title|              genres|
+-------+--------------------+--------------------+
|      1|    Toy Story (1995)|Adventure|Animati...|
|      2|      Jumanji (1995)|Adventure|Childre...|
|      3|Grumpier Old Men ...|      Comedy|Romance|
|      4|Waiting to Exhale...|Comedy|Drama|Romance|
|      5|Father of the Bri...|              Comedy|
+-------+--------------------+--------------------+
only showing top 5 rows
```

Here, we provide an example of exploring the dataset to examine its contents, including the number of values, the types of values, and other relevant details



```
# 3. Explore Data
ratings.printSchema()
movies.printSchema()
print("Ratings dataset shape: Rows =", ratings.count(), ", Columns =", len(ratings.columns))
print("Movies dataset shape: Rows =", movies.count(), ", Columns =", len(movies.columns))
```

✓ 1.2s

```
root
 |-- userId: integer (nullable = true)
 |-- movieId: integer (nullable = true)
 |-- rating: double (nullable = true)
 |-- timestamp: integer (nullable = true)

root
 |-- movieId: integer (nullable = true)
 |-- title: string (nullable = true)
 |-- genres: string (nullable = true)

[Stage 6:>                                                  (0 + 8) / 8]
Ratings dataset shape: Rows = 25000095 , Columns = 4
Movies dataset shape: Rows = 62423 , Columns = 3
```

I didn't show everything here just as a examples.

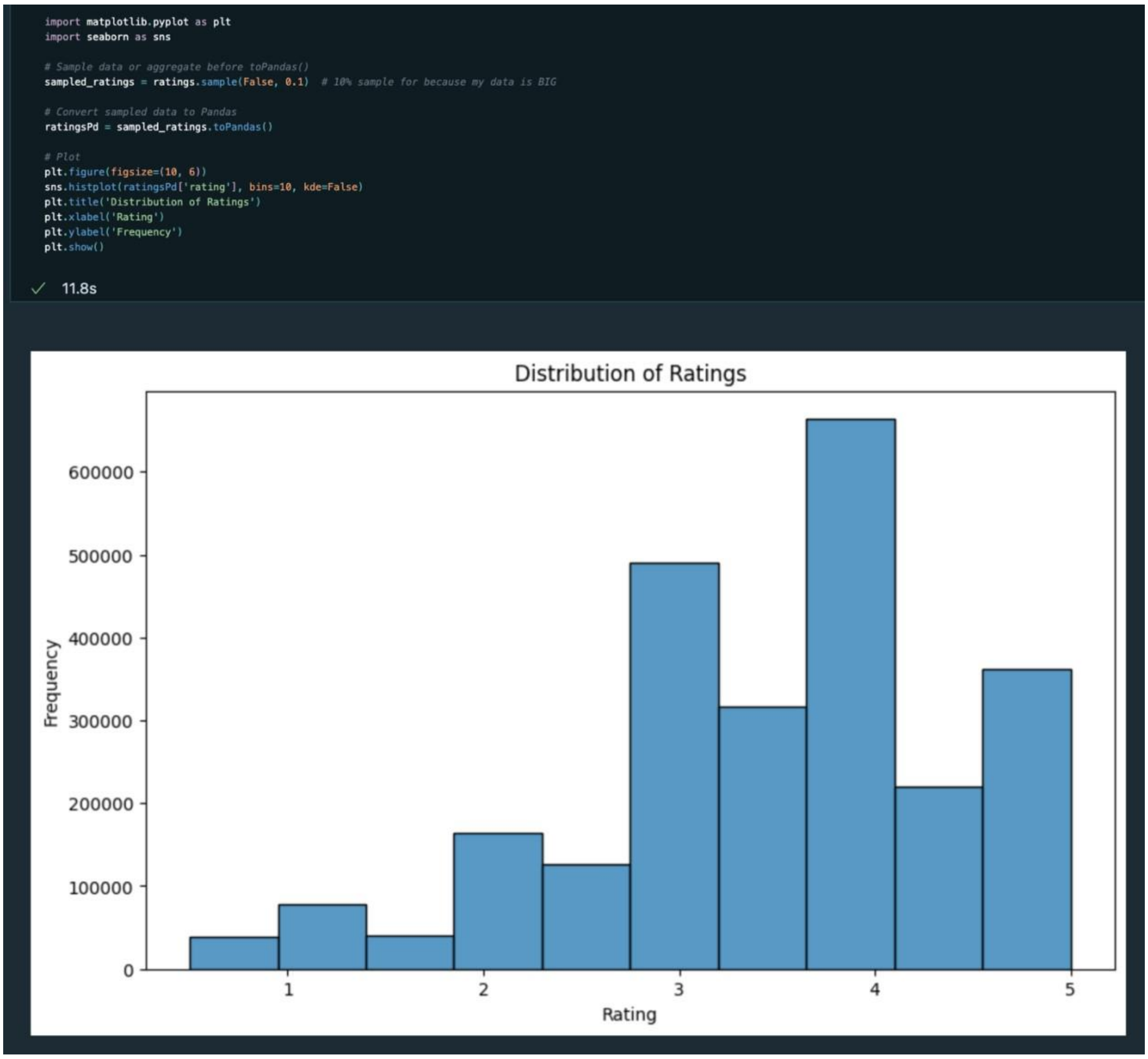
Data visualization is the process of representing data graphically to identify patterns, trends, and insights. It simplifies complex datasets by using visual elements like charts, graphs, and maps. Effective visualization helps communicate information clearly and intuitively.

Why Is Data Visualization Important?

Understand Data: Quickly identify patterns, anomalies, and correlations.

Decision Making: Supports better decision-making by presenting data in an understandable format.

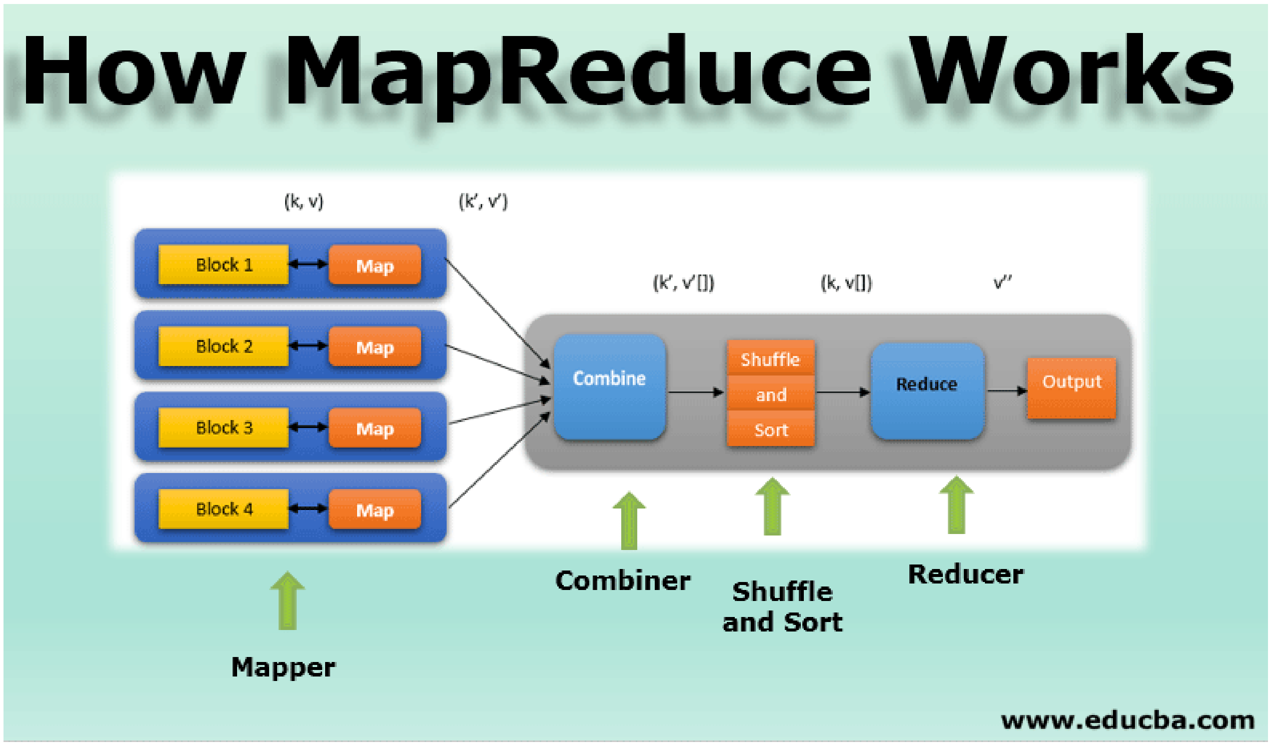
Storytelling: Helps communicate findings and tell a compelling story with data.



```python
import matplotlib.pyplot as plt
import seaborn as sns

# Sample data or aggregate before toPandas()
sampled_ratings = ratings.sample(False, 0.1)  # 10% sample for because my data is BIG

# Convert sampled data to Pandas
ratingsPd = sampled_ratings.toPandas()

# Plot
plt.figure(figsize=(10, 6))
sns.histplot(ratingsPd['rating'], bins=10, kde=False)
plt.title('Distribution of Ratings')
plt.xlabel('Rating')
plt.ylabel('Frequency')
plt.show()
```

✓ 11.8s

We uses matplotlib and seabron to visualize the data and the performance of models .



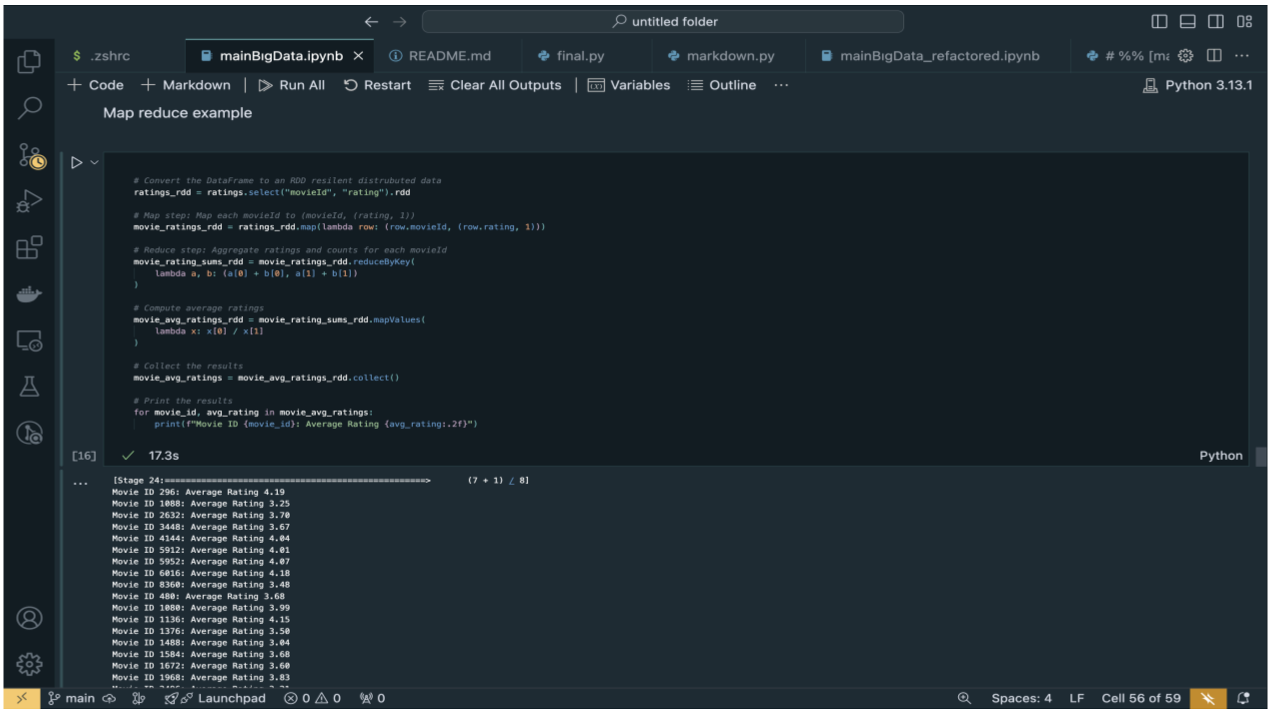
What is MapReduce?

MapReduce is a programming model for processing large datasets in a distributed and parallel manner. It has two main phases: Map and Reduce.

Map Phase: Input data is split into chunks. A map function processes each chunk to produce key-value pairs.

Shuffle and Sort: Intermediate key-value pairs are grouped by keys.

Reduce Phase: A reduce function aggregates values for each key to produce the final result.

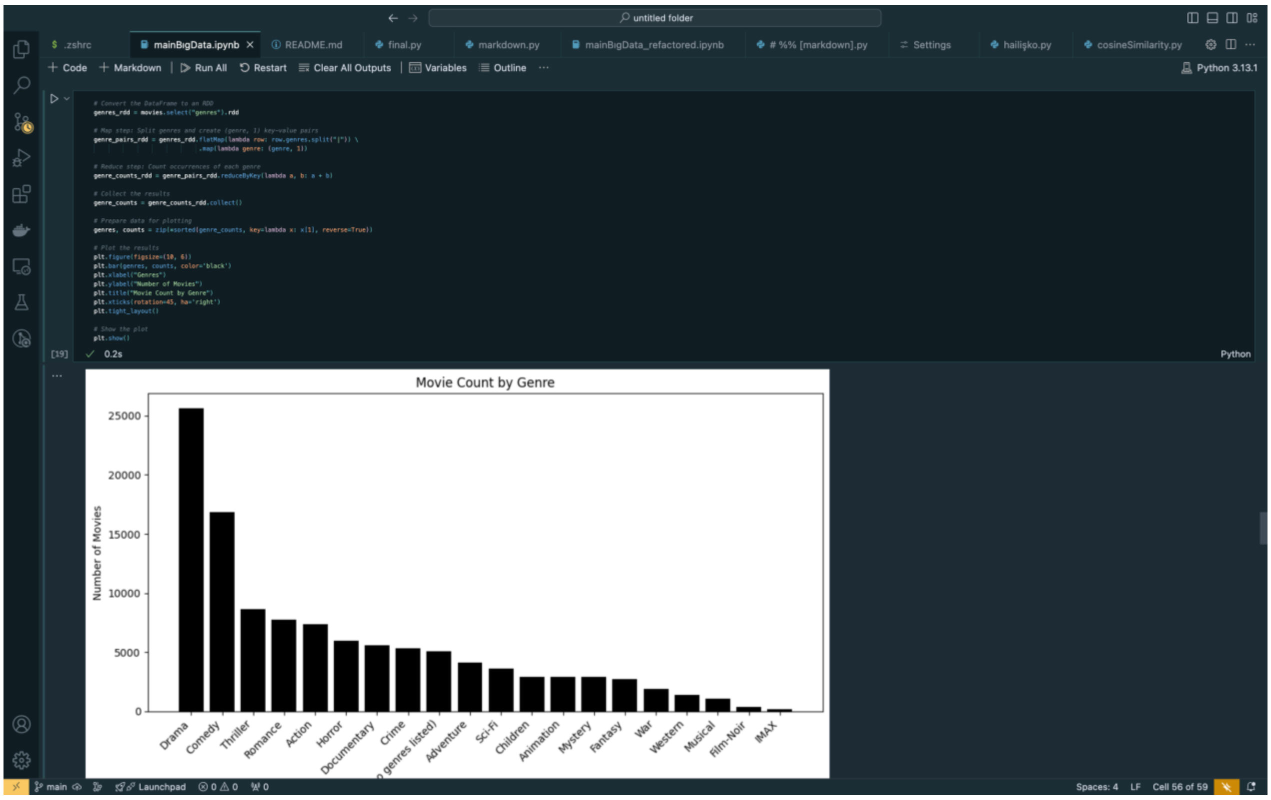


Map reduce example

```python
# Convert the DataFrame to an RDD resilent distrubuted data
ratings_rdd = ratings.select("movieId", "rating").rdd

# Map step: Map each movieId to (movieId, (rating, 1))
movie_ratings_rdd = ratings_rdd.map(lambda row: (row.movieId, (row.rating, 1)))

# Reduce step: Aggregate ratings and counts for each movieId
movie_rating_sums_rdd = movie_ratings_rdd.reduceByKey(
    lambda a, b: (a[0] + b[0], a[1] + b[1])
)

# Compute average ratings
movie_avg_ratings_rdd = movie_rating_sums_rdd.mapValues(
    lambda x: x[0] / x[1]
)

# Collect the results
movie_avg_ratings = movie_avg_ratings_rdd.collect()

# Print the results
for movie_id, avg_rating in movie_avg_ratings:
    print(f"Movie ID {movie_id}: Average Rating {avg_rating:.2f}")
```

[16] ✓ 17.3s

```
[Stage 24:==================================================>      (7 + 1) / 8]
Movie ID 296: Average Rating 4.19
Movie ID 1088: Average Rating 3.25
Movie ID 2632: Average Rating 3.70
Movie ID 3448: Average Rating 3.67
Movie ID 4144: Average Rating 4.04
Movie ID 5912: Average Rating 4.01
Movie ID 5952: Average Rating 4.07
Movie ID 6016: Average Rating 4.18
Movie ID 8360: Average Rating 3.48
Movie ID 480: Average Rating 3.68
Movie ID 1080: Average Rating 3.99
Movie ID 1136: Average Rating 4.15
Movie ID 1376: Average Rating 3.50
Movie ID 1488: Average Rating 3.04
Movie ID 1584: Average Rating 3.68
Movie ID 1672: Average Rating 3.60
Movie ID 1968: Average Rating 3.83
```



```python
# Convert the DataFrame to an RDD
genres_rdd = movies.select("genres").rdd

# Map step: Split genres and create (genre, 1) key-value pairs
genre_pairs_rdd = genres_rdd.flatMap(lambda row: row.genres.split("|")) \
                             .map(lambda genre: (genre, 1))

# Reduce step: Count occurrences of each genre
genre_counts_rdd = genre_pairs_rdd.reduceByKey(lambda a, b: a + b)

# Collect the results
genre_counts = genre_counts_rdd.collect()

# Prepare data for plotting
genres, counts = zip(*sorted(genre_counts, key=lambda x: x[1], reverse=True))

# Plot the results
plt.figure(figsize=(10, 6))
plt.bar(genres, counts, color='black')
plt.xlabel("Genres")
plt.ylabel("Number of Movies")
plt.title("Movie Count by Genre")
plt.xticks(rotation=45, ha='right')
plt.tight_layout()

# Show the plot
plt.show()
```

```
[19]  ✓ 0.2s
```

And this is the reduce by key code section, where PySpark aggregates values for each unique key in an RDD.

ReduceByKey in PySpark: Groups values by key and applies an aggregation function (e.g., sum, max). Optimized by performing partial aggregation locally before shuffling data. Commonly used for tasks like word count or summing metrics by category.

1. Collaborative Filtering:

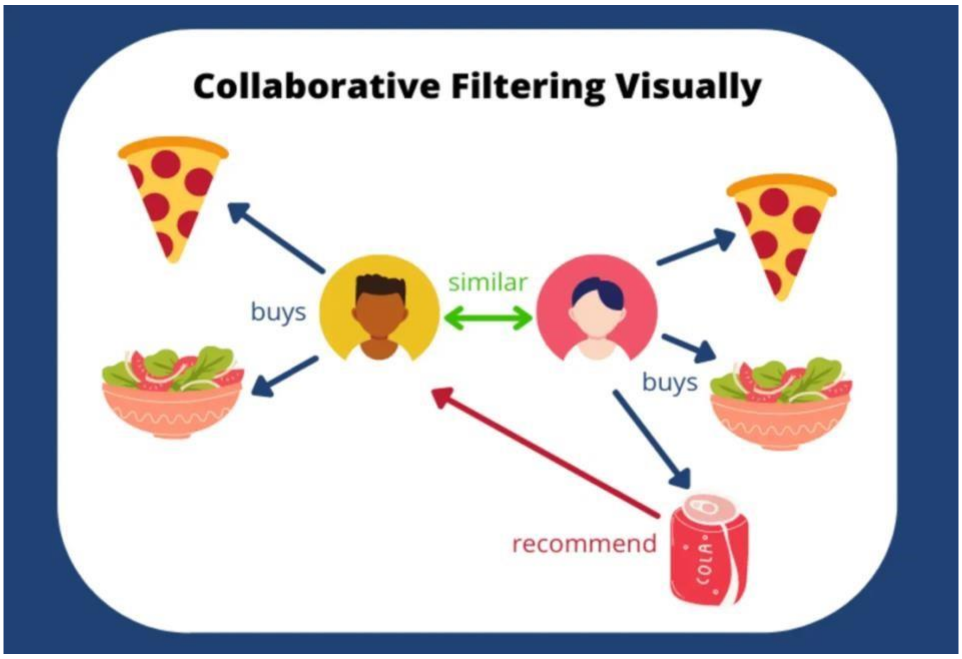
Collaborative Filtering is a technique that makes recommendations based on the behavior and preferences of users. It assumes that users who agreed in the past (by liking, rating, or interacting with similar items) will agree in the future. Collaborative Filtering doesn't rely on explicit knowledge about items but rather on the patterns of user-item interactions.

There are two main types of Collaborative Filtering:

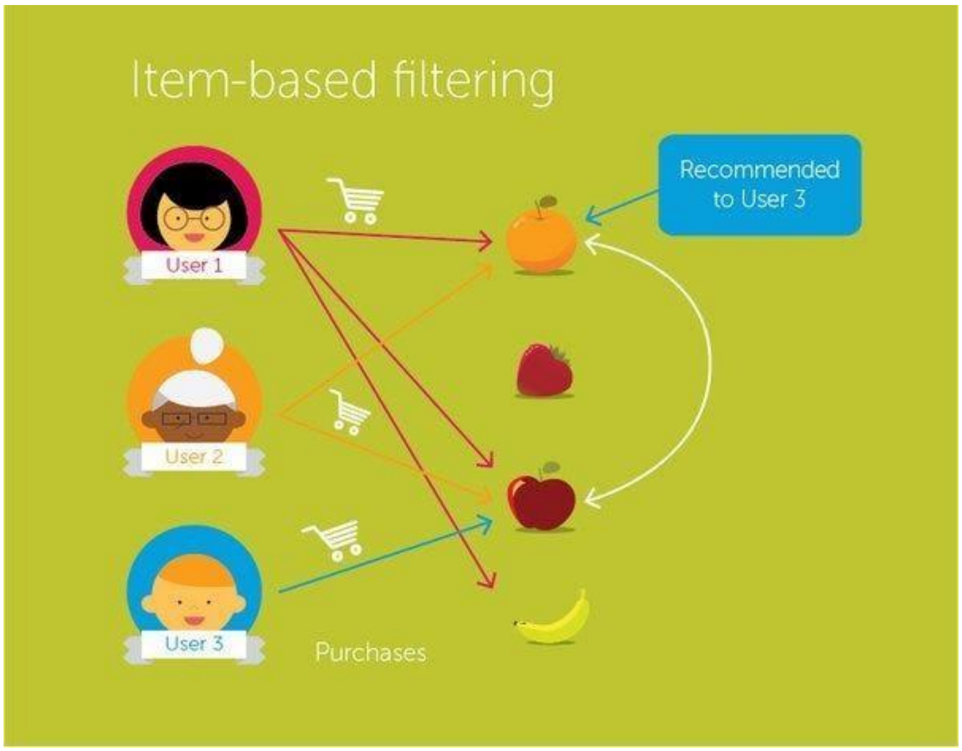
User-Based Collaborative Filtering: This approach finds users who are similar to the target user based on their past interactions and recommends items liked by similar users.

Item-Based Collaborative Filtering: This approach finds items that are similar to the target item based on users' past interactions and recommends similar items.

Collaborative Filtering can handle cases where user-item interactions are sparse or where there's no detailed information about the items. It often works well in scenarios where user preferences are dynamic and may change over time.



This picture shows user based filtering



This picture shows item based filtering

2. Content-Based Filtering:
Content-Based Filtering, on the other hand, makes recommendations based on the characteristics (content) of items and a profile of the user's preferences. It involves extracting features or attributes from items and building user profiles based on the items they have liked or interacted with in the past.

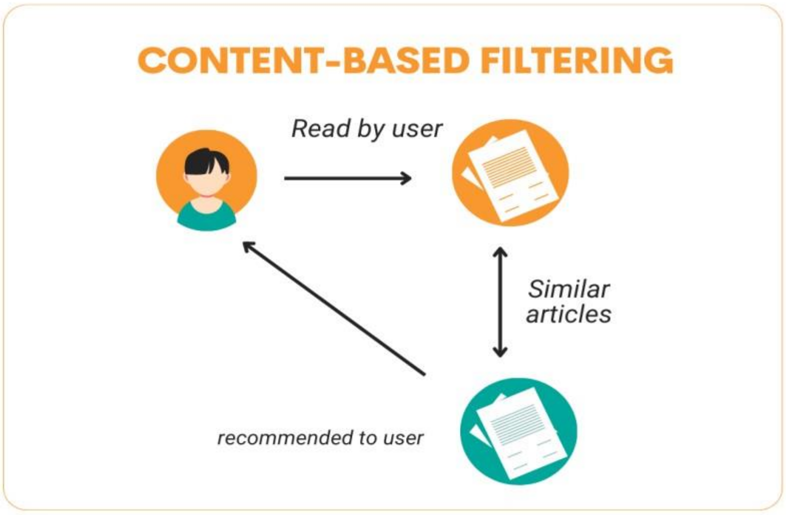
The key steps in Content-Based Filtering are:

Feature Extraction: Extract relevant features from the items (e.g., genre, keywords, actors, director, etc. for movies).

User Profile Creation: Create a user profile based on the features of items the user has shown interest in.

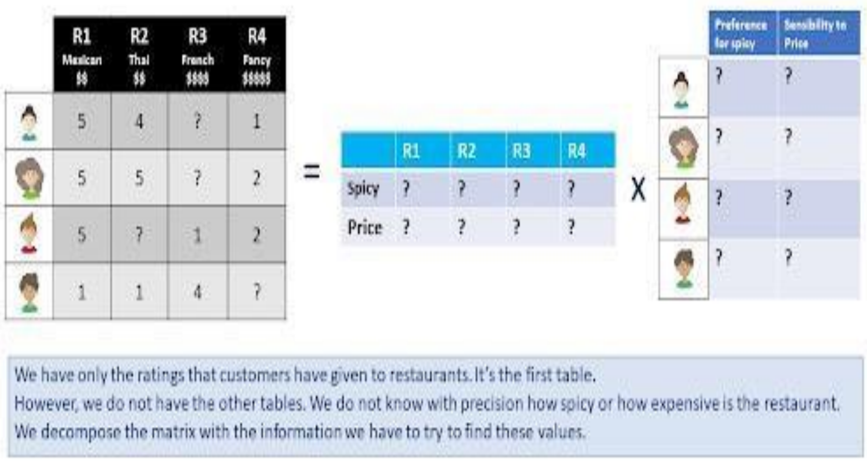
Item-User Matching: Recommend items that match the user profile based on content similarity.

Content-Based Filtering can provide personalized recommendations even when there's limited user-item interaction data. It's useful when you have detailed information about items and want to recommend items with specific features that match the user's preferences.



ALS (Alternating Least Squares) Algorithm:

The Alternating Least Squares (ALS) algorithm is a matrix factorization technique commonly used in recommendation systems. It is particularly effective for collaborative filtering-based recommendations. ALS factors a large user-item interaction matrix into two lower-dimensional matrices, one representing users and the other representing items. This factorization allows the model to capture latent features that drive user preferences and item characteristics.



	R1 Mexican $$	R2 Thai $$	R3 French $$$$	R4 Fancy $$$$$
	5	4	?	1
	5	5	?	2
	5	?	1	2
	1	1	4	?

=

	R1	R2	R3	R4
Spicy	?	?	?	?
Price	?	?	?	?

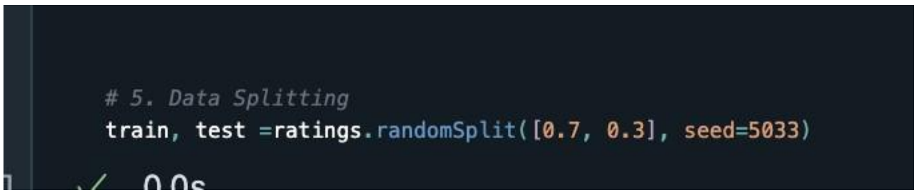
X

	Preference for spicy	Sensibility to Price
	?	?
	?	?
	?	?
	?	?

We have only the ratings that customers have given to restaurants. It's the first table.
However, we do not have the other tables. We do not know with precision how spicy or how expensive is the restaurant.
We decompose the matrix with the information we have to try to find these values.

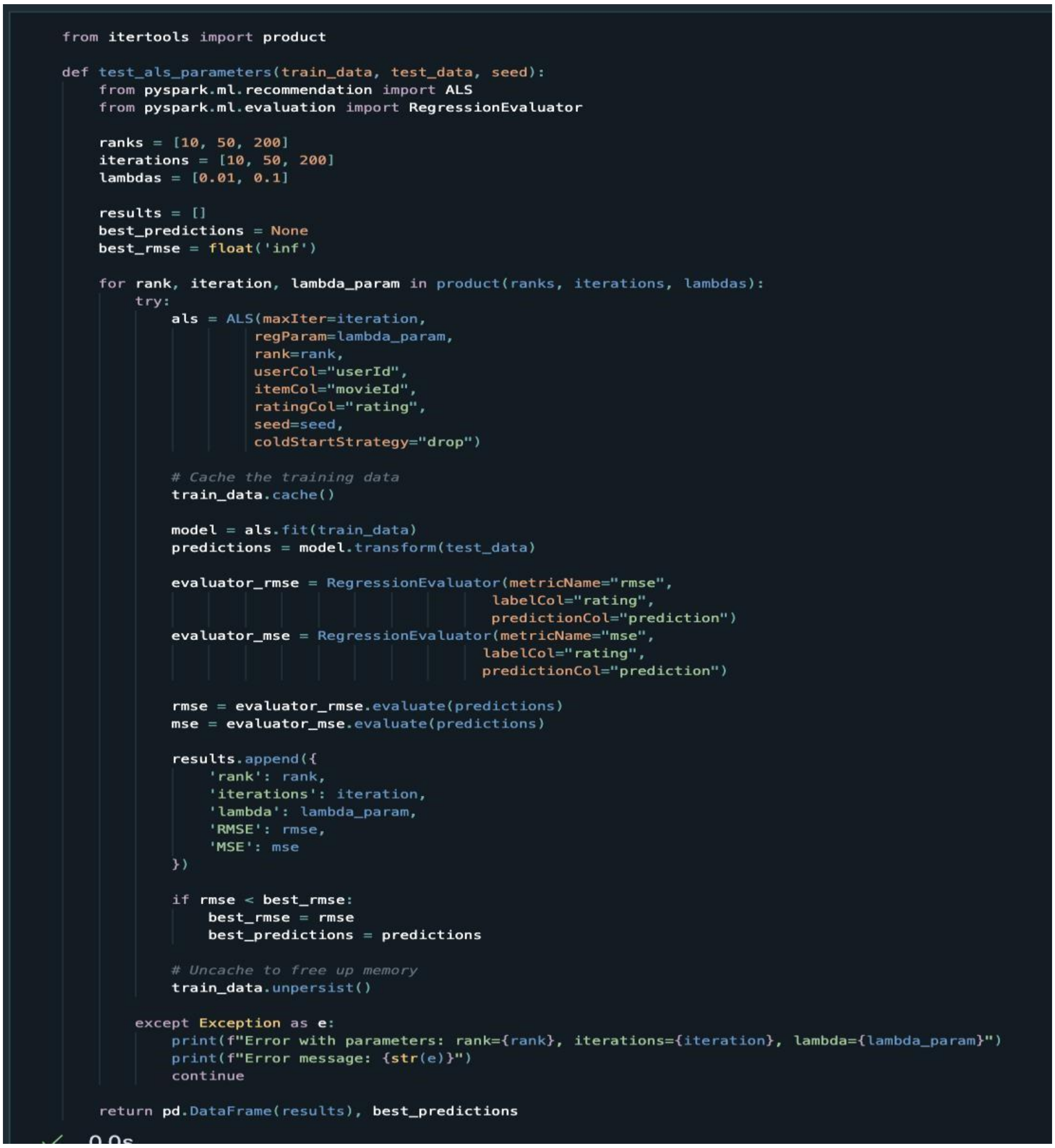
Here's how ALS works:

1. **Matrix Factorization:** ALS begins with the user-item interaction matrix (often sparse), where rows represent users, columns represent items, and the entries contain user ratings or interactions.
2. **Initialization:** ALS initializes the user and item matrices with random or small values.
3. **Alternating Optimization:** The ALS algorithm iteratively alternates between optimizing the user factors (representing user preferences) while keeping the item factors fixed, and optimizing the item factors while keeping the user factors fixed. This alternating optimization process continues until it converges to a solution that minimizes the reconstruction error (least squares difference) between the original user-item interaction matrix and the product of the user and item factor matrices.
4. **Latent Features:** The key idea is that ALS learns latent features from the data. Each latent feature represents a hidden characteristic that contributes to user preferences or item characteristics. For example, in a movie recommendation system, latent features might capture genre preferences, actor popularity, or viewer demographics.
5. **Collaborative Filtering:** ALS leverages collaborative filtering, assuming that users who have agreed in the past (by rating or interacting with similar items) will agree in the future. By factorizing the user-item matrix, ALS can uncover these collaborative patterns.
6. **Implicit and Explicit Feedback:** ALS can handle both explicit feedback (e.g., explicit user ratings) and implicit feedback (e.g., user interactions such as clicks, views). It is particularly effective in scenarios with implicit feedback where explicit ratings are sparse.
7. **Parallelization:** ALS is designed for parallel and distributed computing, making it efficient for large-scale recommendation scenarios. It can be parallelized across users or items, leveraging the power of distributed systems.
8. **Model Evaluation:** After training, ALS can generate recommendations for items that users have not interacted with. These recommendations are based on the learned latent factors and can be evaluated using metrics like precision, recall, or root mean squared error (RMSE) for rating predictions.



```
# 5. Data Splitting
train, test =ratings.randomSplit([0.7, 0.3], seed=5033)
```

We split the data as you said in the documennt .



```
from itertools import product

def test_als_parameters(train_data, test_data, seed):
    from pyspark.ml.recommendation import ALS
    from pyspark.ml.evaluation import RegressionEvaluator

    ranks = [10, 50, 200]
    iterations = [10, 50, 200]
    lambdas = [0.01, 0.1]

    results = []
    best_predictions = None
    best_rmse = float('inf')

    for rank, iteration, lambda_param in product(ranks, iterations, lambdas):
        try:
            als = ALS(maxIter=iteration,
                     regParam=lambda_param,
                     rank=rank,
                     userCol="userId",
                     itemCol="movieId",
                     ratingCol="rating",
                     seed=seed,
                     coldStartStrategy="drop")

            # Cache the training data
            train_data.cache()

            model = als.fit(train_data)
            predictions = model.transform(test_data)

            evaluator_rmse = RegressionEvaluator(metricName="rmse",
                                                 labelCol="rating",
                                                 predictionCol="prediction")
            evaluator_mse = RegressionEvaluator(metricName="mse",
                                                labelCol="rating",
                                                predictionCol="prediction")

            rmse = evaluator_rmse.evaluate(predictions)
            mse = evaluator_mse.evaluate(predictions)

            results.append({
                'rank': rank,
                'iterations': iteration,
                'lambda': lambda_param,
                'RMSE': rmse,
                'MSE': mse
            })

            if rmse < best_rmse:
                best_rmse = rmse
                best_predictions = predictions

            # Uncache to free up memory
            train_data.unpersist()

        except Exception as e:
            print(f"Error with parameters: rank={rank}, iterations={iteration}, lambda={lambda_param}")
            print(f"Error message: {str(e)}")
            continue

    return pd.DataFrame(results), best_predictions
```

And the this is the function to test als with different parameters

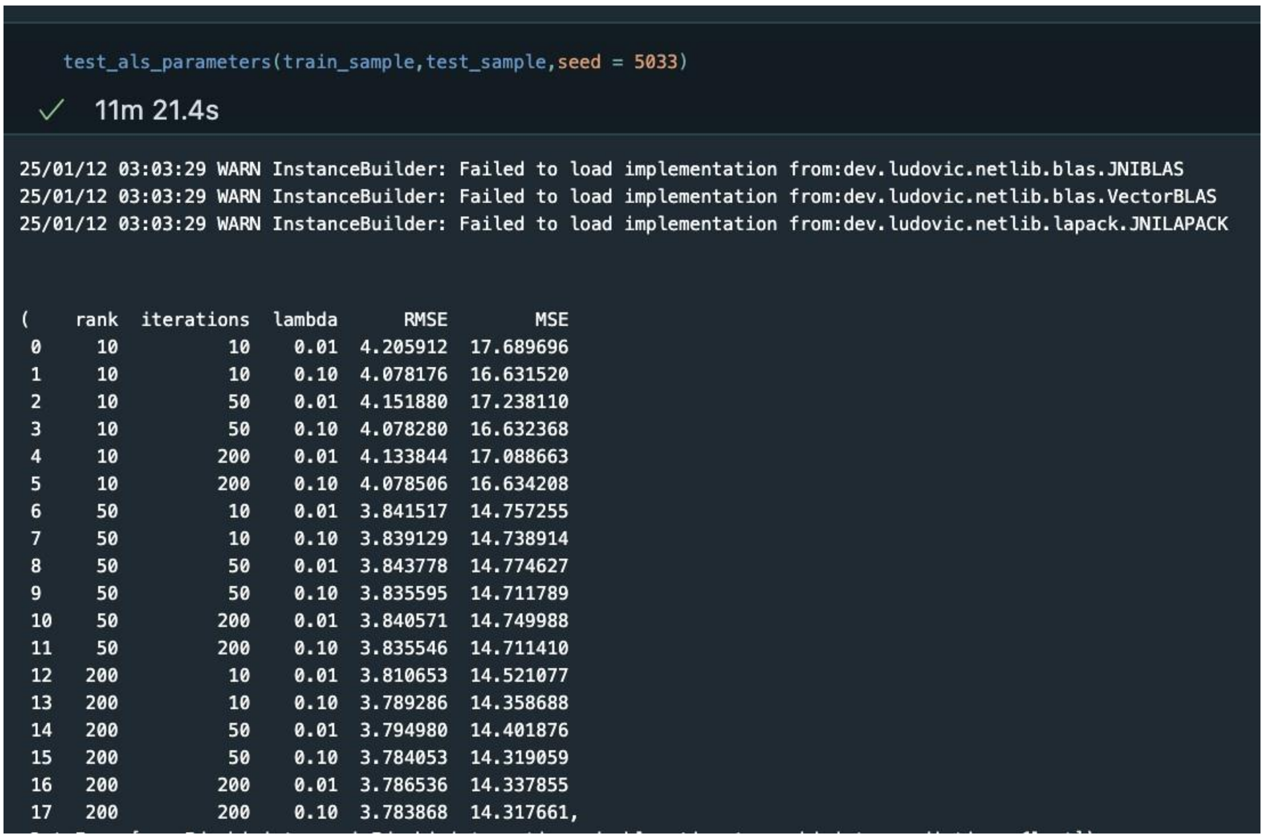
Rank:Rank refers to the number of latent factors used in matrix factorization models, like collaborative filtering in recommendation systems. A higher rank typically allows the model to capture more complex patterns in the data but can also increase the risk of overfitting.

Iteration:This indicates the number of iterations the model goes through during training to minimize the loss function. More iterations generally lead to better optimization but may also increase computation time.Lambda:Lambda is the regularization parameter in the model. It helps prevent overfitting by penalizing large weights in the model. A well-tuned lambda value balances bias and variance.

RMSE (Root Mean Squared Error):RMSE measures the average magnitude of errors between the predicted and actual values. It gives a sense of how far the predictions are from the true values on average. Lower RMSE values indicate better model performance.

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

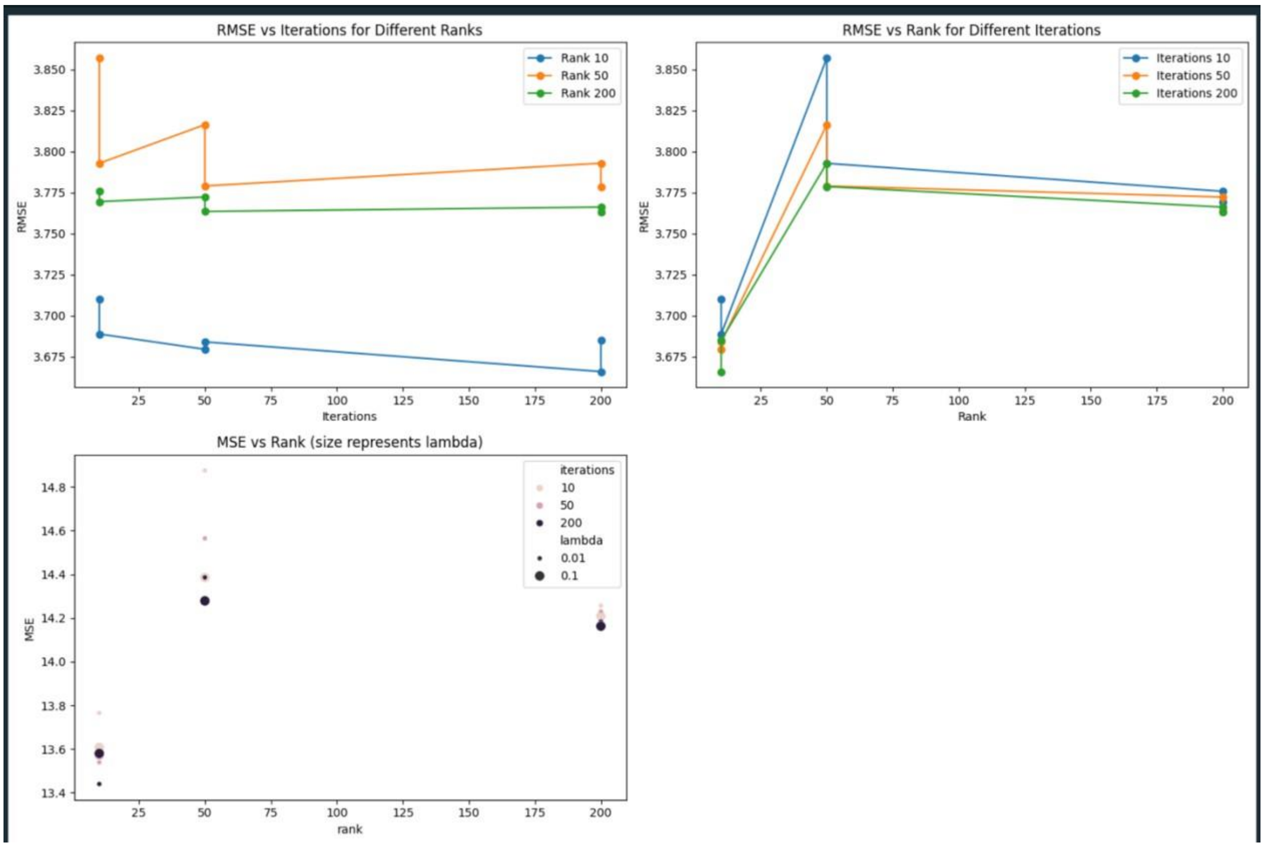
MSE (Mean Squared Error):MSE is the average squared difference between the predicted and actual values. While it also reflects prediction accuracy, it tends to penalize larger errors more heavily than RMSE. Lower MSE values suggest better model performance.

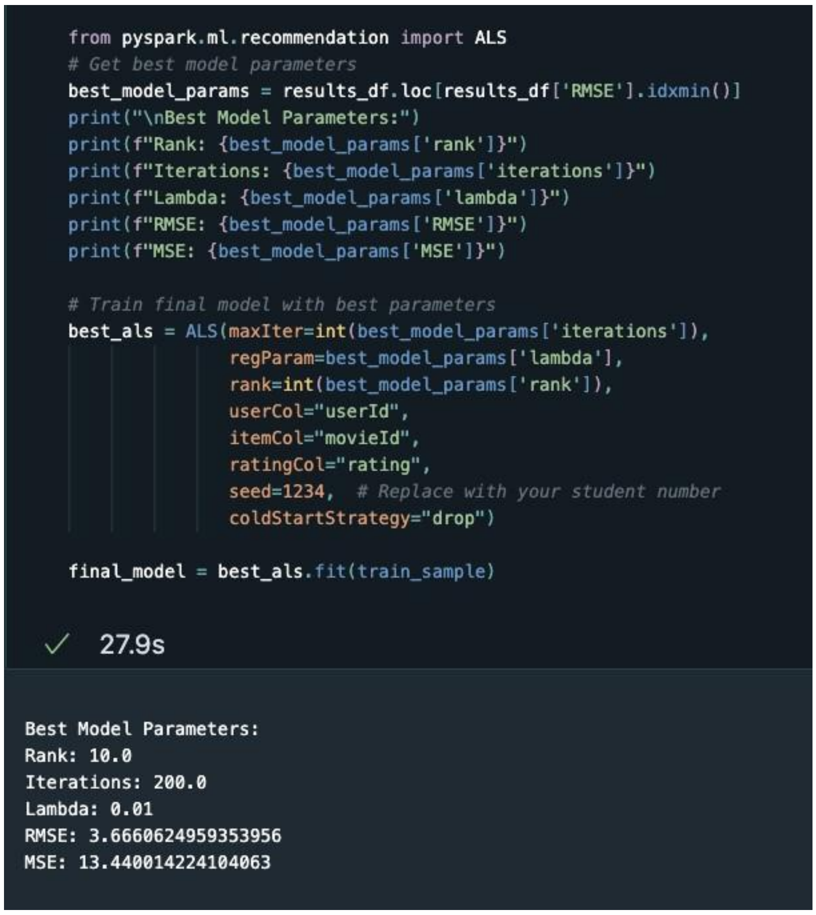


```
test_als_parameters(train_sample,test_sample,seed = 5033)
✓  11m 21.4s

25/01/12 03:03:29 WARN InstanceBuilder: Failed to load implementation from:dev.ludovic.netlib.blas.JNIBLAS
25/01/12 03:03:29 WARN InstanceBuilder: Failed to load implementation from:dev.ludovic.netlib.blas.VectorBLAS
25/01/12 03:03:29 WARN InstanceBuilder: Failed to load implementation from:dev.ludovic.netlib.lapack.JNILAPACK


(    rank  iterations  lambda      RMSE        MSE
 0     10          10    0.01  4.205912  17.689696
 1     10          10    0.10  4.078176  16.631520
 2     10          50    0.01  4.151880  17.238110
 3     10          50    0.10  4.078280  16.632368
 4     10         200    0.01  4.133844  17.088663
 5     10         200    0.10  4.078506  16.634208
 6     50          10    0.01  3.841517  14.757255
 7     50          10    0.10  3.839129  14.738914
 8     50          50    0.01  3.843778  14.774627
 9     50          50    0.10  3.835595  14.711789
 10    50         200    0.01  3.840571  14.749988
 11    50         200    0.10  3.835546  14.711410
 12   200          10    0.01  3.810653  14.521077
 13   200          10    0.10  3.789286  14.358688
 14   200          50    0.01  3.794980  14.401876
 15   200          50    0.10  3.784053  14.319059
 16   200         200    0.01  3.786536  14.337855
 17   200         200    0.10  3.783868  14.317661,
```

The trained 18 models and their rank iteration lambda RMSE and MSE values

This the Visualization of performance of ALS Models


RMSE vs Iterations for Different Ranks
Rank 10
Rank 50
Rank 200
RMSE
Iterations
RMSE vs Rank for Different Iterations
Iterations 10
Iterations 50
Iterations 200
RMSE
Rank
MSE vs Rank (size represents lambda)
iterations
10
50
200
lambda
0.01
0.1
MSE
rank

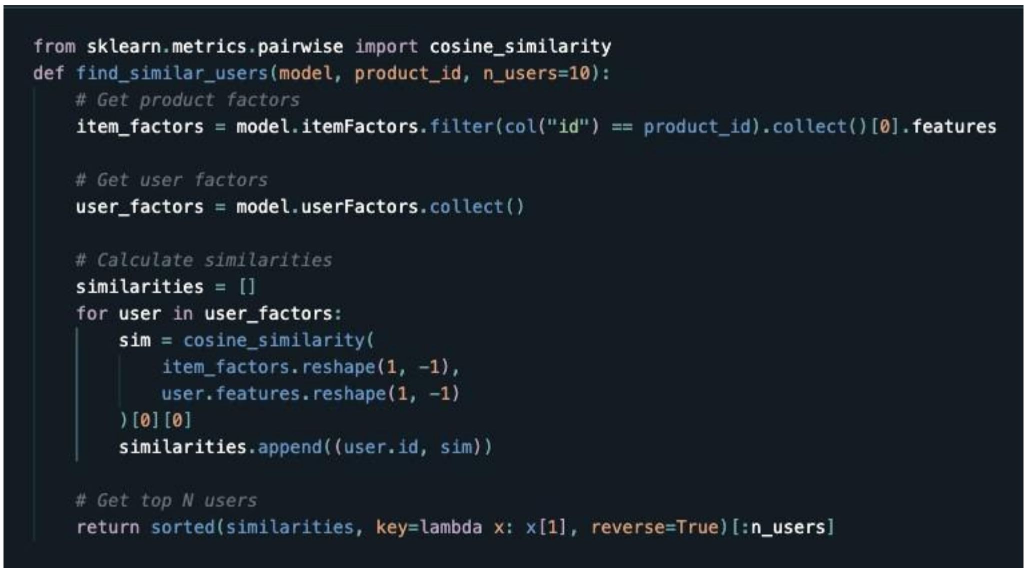


```python
from pyspark.ml.recommendation import ALS
# Get best model parameters
best_model_params = results_df.loc[results_df['RMSE'].idxmin()]
print("\nBest Model Parameters:")
print(f"Rank: {best_model_params['rank']}")
print(f"Iterations: {best_model_params['iterations']}")
print(f"Lambda: {best_model_params['lambda']}")
print(f"RMSE: {best_model_params['RMSE']}")
print(f"MSE: {best_model_params['MSE']}")

# Train final model with best parameters
best_als = ALS(maxIter=int(best_model_params['iterations']),
               regParam=best_model_params['lambda'],
               rank=int(best_model_params['rank']),
               userCol="userId",
               itemCol="movieId",
               ratingCol="rating",
               seed=1234,  # Replace with your student number
               coldStartStrategy="drop")

final_model = best_als.fit(train_sample)
```

✓ 27.9s

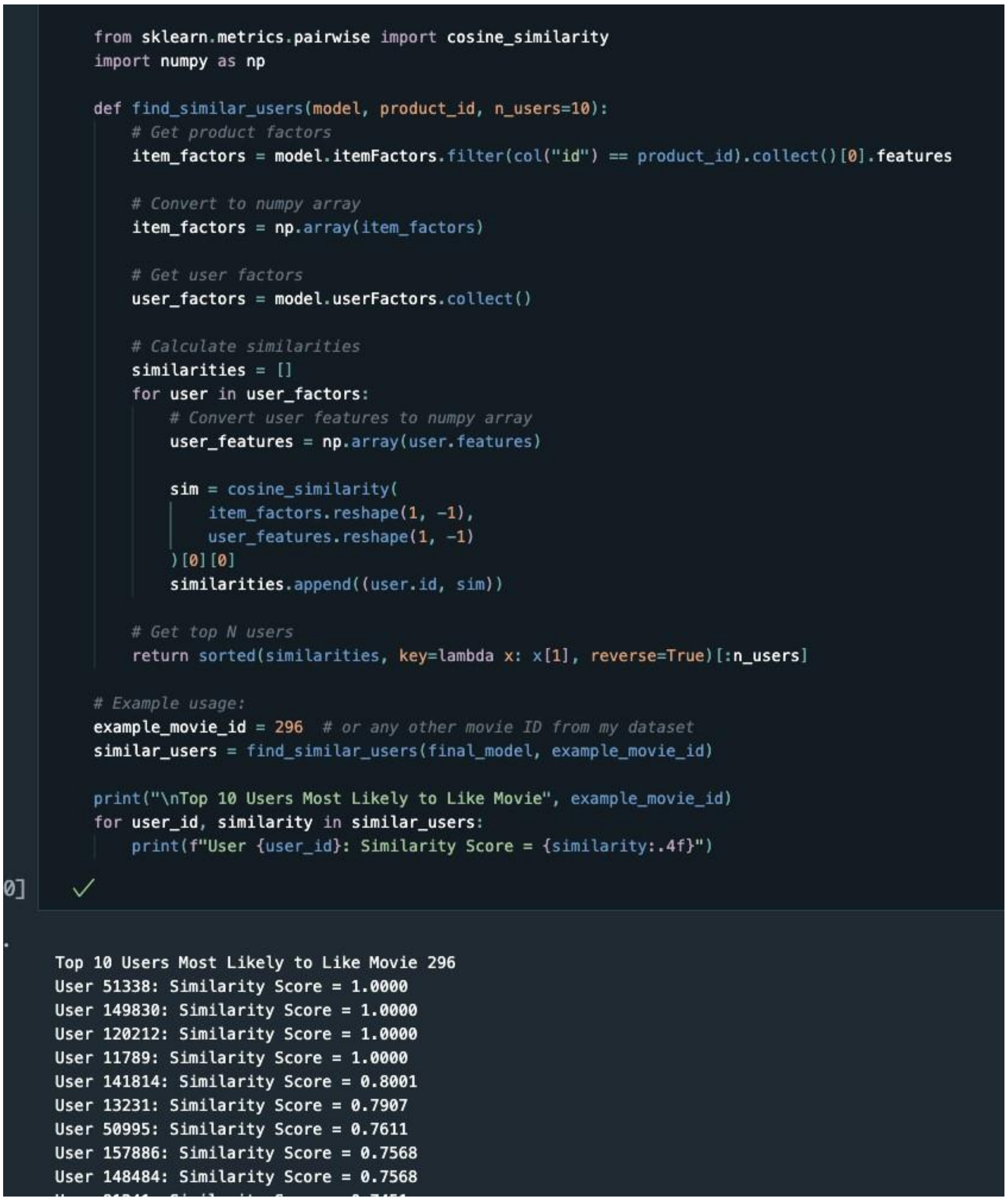
```
Best Model Parameters:
Rank: 10.0
Iterations: 200.0
Lambda: 0.01
RMSE: 3.6660624959353956
MSE: 13.440014224104063
```

Getting the best model to make operations like " Make prediction with ALS and compare it with the original values (with real values) side by side."



```python
from sklearn.metrics.pairwise import cosine_similarity
def find_similar_users(model, product_id, n_users=10):
    # Get product factors
    item_factors = model.itemFactors.filter(col("id") == product_id).collect()[0].features

    # Get user factors
    user_factors = model.userFactors.collect()

    # Calculate similarities
    similarities = []
    for user in user_factors:
        sim = cosine_similarity(
            item_factors.reshape(1, -1),
            user.features.reshape(1, -1)
        )[0][0]
        similarities.append((user.id, sim))

    # Get top N users
    return sorted(similarities, key=lambda x: x[1], reverse=True)[:n_users]
```

This is the part of cosine similarities



```python
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

def find_similar_users(model, product_id, n_users=10):
    # Get product factors
    item_factors = model.itemFactors.filter(col("id") == product_id).collect()[0].features

    # Convert to numpy array
    item_factors = np.array(item_factors)

    # Get user factors
    user_factors = model.userFactors.collect()

    # Calculate similarities
    similarities = []
    for user in user_factors:
        # Convert user features to numpy array
        user_features = np.array(user.features)

        sim = cosine_similarity(
            item_factors.reshape(1, -1),
            user_features.reshape(1, -1)
        )[0][0]
        similarities.append((user.id, sim))

    # Get top N users
    return sorted(similarities, key=lambda x: x[1], reverse=True)[:n_users]

# Example usage:
example_movie_id = 296  # or any other movie ID from my dataset
similar_users = find_similar_users(final_model, example_movie_id)

print("\nTop 10 Users Most Likely to Like Movie", example_movie_id)
for user_id, similarity in similar_users:
    print(f"User {user_id}: Similarity Score = {similarity:.4f}")
```

```
Top 10 Users Most Likely to Like Movie 296
User 51338: Similarity Score = 1.0000
User 149830: Similarity Score = 1.0000
User 120212: Similarity Score = 1.0000
User 11789: Similarity Score = 1.0000
User 141814: Similarity Score = 0.8001
User 13231: Similarity Score = 0.7907
User 50995: Similarity Score = 0.7611
User 157886: Similarity Score = 0.7568
User 148484: Similarity Score = 0.7568
```

And this part is where we find Top 10 users who would like a spesific movie we choosed.

WORK SHARING POLICY

Name	Duty	Percent
Yusuf SERTKAYA 211805033	**Making entire prject**	**% 33**
Satılmış KABASAKAL 221805081	**Helping in both projects working like a chatbot**	**% 33**
Zınar Demirpolat	**Making entire project**	**% 33**

REFERENCES

[1] https://www.youtube.com/watch?v=FgGjc5oabrA&ab_channel=jamenlong1

[2] https://spark.apache.org/docs/latest/api/python/index.html

[3] https://github.com/abhilashhn1993/collaborative-filtering-using-ALS-for-movie-recommendation

[4] https://medium.com/deep-learning-turkiye/gradient-descent-nedir-3ec6afcb9900

[5] https://www.bacancytechnology.com/qanda/qa-automation/steps-vs-epochs-in-tensorflow

[6] https://www.linkedin.com/pulse/mastering-collaborative-filtering-pyspark-als-model-guideambekar-v6ajc/

[7] https://medium.com/@roshmitadey/optimizing-pyspark-for-handling-large-volumes-of-datac4f51f80224e

[8] https://talent500.com/blog/working-with-big-data-using-pyspark-a-hands-on-tutorial/amp/

[9] https://medium.com/@siladityaghosh/mastering-big-data-with-pyspark-on-google-colabd3c924264ceb

[10] https://www.kaggle.com/code/shiblinomani/matrix-factorization-and-als-with-apache-spark

[11] https://www.kaggle.com/code/alfarias/movie-recommendation-system-with-als-in-pyspark

[12] https://github.com/SJD1882/Big-Data-Recommender-

Systems/blob/master/notebooks/MovieLens27M-ALS-Recommender-System.ipynb

[13] https://medium.com/@brunoborges_38708/recommender-system-using-als-in-pyspark10329e1d1ee1

[14] https://www.geeksforgeeks.org/how-to-split-a-dataset-into-train-and-test-sets-using-python/

[15] Classrom and the links shared in classroom like checkpoint usage