AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF COMPUTING, AMARAVATI

Department of Computer Science and Engineering

MINI - PROJECT REPORT ON

PRODUCT RECOMMENDATION SYSTEM USING COLLABRATIVE FILTERING WITH PYSPARK

Submitted

By

A. CHARITH, B. NATRAJ KOUSHIK REDDY, V. VARUN SAI, D. SRIDHAR ADITHYA

AV.EN.U4AIE22001, AV.EN.U4AIE22004, AV.EN.U4AIE22046, AV.EN.U4AIE22049

B. Tech (AIE) SIXTH SEMESTER

During the academic year

2024- 2025

Under the Guidance of

Dr. V LAKSHMI CHETANA

Assistant Professor (Sl. Gr.)

Department of Computer Science and Engineering

Amrita School of Computing

Amrita Vishwa Vidyapeetham, Amaravati Campus

Contents

1. Introduction
2. Problem Statement
3. Objectives
4. Literature Review
5. Methodology
    - Overview of the approach used
    - Model or system architecture (Draw architecture diagram)
6. Implementation
    - Screenshots or code snippets (if applicable)
7. Results and Discussion
    - Dataset Description
    - Experimental Setup
    - Evaluation Metrics used
    - Performance evaluation with tables/graphs
    - Comparison with existing methods
8. Challenges and Learnings
9. Conclusion and Future Scope
10. References

ABSTRACT

In the ever-changing arena of e-commerce, personalized recommendation systems enhance user experience through product suggestion. This project aims to build a highly scalable and efficient product recommendation system through a wide array of collaborative filtering techniques in conjunction with PySpark and Python libraries. The dataset employed in this study deals with user-product interactions in the beauty product domain and includes user IDs, product IDs, and associated ratings.

The preprocessing pipeline contains steps for loading, cleaning, and analyzing the dataset for null value treatments, duplicate row removal, and descriptive statistics generation. Visualizations such as pie charts and heat maps assist in uncovering rating distributions and some feature correlations. Feature engineering steps, such as aggregating average ratings and ratings count per user and per product, enhance the modeling dataset.

Different recommendation strategies are being studied. User-based and item-based collaborative filtering models using cosine similarity over sparse user-item matrices, top-N recommendations were made by masking products already rated. Matrix factorization algorithms based on the latent factor modeling approach, including Alternating Least Squares (ALS) with PySpark's MLlib and Singular Value Decomposition (SVD) with SciPy, were employed. Moreover, K-Nearest Neighbor (KNN) using different distance measures including cosine and Pearson correlation were considered for memory-based recommendation tasks in the testing phase. Hybrid approaches integrating collaborative and content-based features were explored to resolve cold-start problems and improve performance. There was also a brief exploration of newly emerging deep learning-based recommendation approaches using neural embeddings.

The models employ error-based metrics like Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) normalized using MinMaxScaler to evaluate performance. With an acceptable predictive performance and high scalability due to the distributed processing of PySpark, this project, therefore, builds a sound framework for the implementation of classical with hybrid and scalable solutions suitable for product recommendations in data markets with a high volume of user-product interactions.

Keywords: Collaborative Filtering, **Product Recommendation System**, **PySpark**, **Matrix Factorization, User-Item Interaction, Personalized Recommendations.**

1.INTRODUCTION

Because of tremendous data that is digitalized and changes in a growing market, the demand of recommendation systems became valuable assets to be valued in today's modern online platform. From e-commerce sites such as Amazon to online streaming services like Netflix, these platforms used recommendation systems to suggest products, movies, music, and other items based on users' past behavior and preferences. These systems are useful in discovering new items for users, enhancing user engagement while at the same time increasing businesses' revenues through the increased user satisfaction and retention that come with such an experience.

Recommendation Systems

Specification in a recommendation system refers to a software tool or an algorithm meant to predict users' preferences and accordingly suggest items to them; these systems would be divided into three: content-based filtering, collaborative filtering, and hybrid models.

Content-based filtering recommends items that are similar to those liked by a user in the past.

This specific project revolves around collaborative filtering which identifies trends in the user-item interaction for an item to be suggested according to the preferences of similar items.

Hybrid models are models that combine both content and collaborative approaches for better performance.

Collaborative Filtering

Collaborative filtering is the method of making automatic predictions about the interests of a user by collecting data about the preferences of many different users. This is based on the logic that if individuals would have postulated similar things in the past, they are likely to agree in the present or future. The main methods pertaining to Collaborative filtering emerge from the two:

User-based filtering: Generates recommendations based on the items that similar users liked by a particular user.

Item-based filtering: Suggests items that are similar to those the user previously interacted with.

These types of memory-based methods work perfectly for little datasets; however, they become costly and have decreased accurate performance while implemented on sparse or massive datasets. Thus, advanced types of model-based methodologies, including matrix factorization techniques, are applied significantly for scalable and predictive improvements.

Alternating Least Squares (ALS)

Alternating Least Squares (ALS) is a matrix factorization technique usually applied to collaborative filtering. In ALS, the user-item interaction matrix is factorized in two lower-dimension user factors and item factors, but their multiplication approximates the original interaction matrix. It minimizes the root grade square error between predicted and actual interactions, alternating between fixing user factors and item factors during optimization.

ALS is particularly effective for implicit feedback data, such as clicks, purchases, or views, where no explicit rating is provided. It uses a confidence-based approach to model the uncertainty of user preferences, which enhances its performance on real-world datasets.

2. Problem Statement

Personalized product recommendations are becoming an essential element in improving user experience, augmenting customer retention, and increasing sales in today's digital market. The e-commerce platform continues to grow ever bigger and more complicated, wherein the user interaction volume and item listing have massively increased. These very reasons make existing recommendation strategies impractical for manual intervention and traditional models inefficient when applied to large-scale real-world data.

More conventional recommendation systems based on content filtering or passerby collaborative filtering suffer from their own problems: data sparsity, cold-start problems, and inadequate scalability. Each of these issues is intensified by the presence of millions of users and products. More than that, they simply do not adapt to changing user preferences and cannot learn from underlying patterns present in the historical behavior.

Collaborative Filtering, particularly model-based techniques like Matrix Factorization using Alternating Least Squares (ALS), promise to reveal hidden relationships between users and products through learning from past interactions. However, the deployment of such algorithms at scale calls for sophisticated infrastructure and high-performance computing capabilities.

Apache Spark, through its Python API, allows for the distributed processing of large data sets, making it a natural candidate for scalable recommendation engine design. Nevertheless, a comprehensive evaluation of collaborative filtering algorithms in their Big Data setup, focusing on ALS methods, user-based, and item-based approaches, remains largely unaddressed in practical implementations.

This project aims to fill this void by designing a scalable recommendation system on PySpark with a distinct focus on generating personalized product recommendations for users. The system implements collaborative filtering methods to estimate hidden preferences in users and facilitate recommending relevant items, thereby improving the complete user experience in such large-scaled platforms. The system is designed to pursue equilibrium between scalability, responsiveness, and personalization, making it good for real-time applications in e-commerce, media streaming, and retail platforms.

3. Objectives

The primary goal of this research is to develop a **Big Data-driven machine learning framework** for accurate and scalable **Product Recommendation** using **PySpark**. The study focuses on leveraging distributed computing and Filtering Based to handle large-scale energy datasets efficiently. The key objectives of this research are:

By developing scalable online recommendation systems utilizing distributed computing in PySpark, consider efficient processing of large-scale user-item interaction databases for real-time recommendations.

Implement collaborative filtering techniques such as ALS
Model-based collaborative filtering shown as the Application of Alternating Least Squares (ALS) for latent discovery of user product pair preferences as in accurate recommendations.

Explore and compare different recommendation algorithms
Research on U-based, I-based, and matrix factorization algorithms within the big data domain space.

Resolve problems of sparsity and cold starts
Work on the following strategies implicit feedback plus user/item metadata since it

improves the quality of the recommendation for the first time founds on new users or new products.

Desiging a personalized recommendation pipeline

You can build a recommender pipeline that recommends products, adjusted according to user behavior over time changes.

Optimize recommendation performance for real-life application

Balance scalability, speed, and cost-effectiveness to prepare the system for the many eyes looking into it in real-time applications.

Argue the quality of recommendations using business relevant metrics

Rather than prediction accuracy, success of recommendation will be based on relevance, diversity, novelty, and user satisfaction for practical use.

4.Literature Review

Collaborative filtering and deep learning techniques have bolstered further advancement of product recommendation systems, especially in the e-commerce domain. One solution aims at conduct big data-driven machine learning models for product recommendations based on the Amazon Reviews Dataset. Such solutions will mingle big data technologies with machine learning toward effective recommendation construction. However, no effort was made toward real-time recommendation, emphasizing the loophole of real-time personalized suggestions for users.

Deep learning mechanisms like neural attention and representation learning have been brought in to bolster interaction in recommendation systems. A study employed a private e-commerce dataset in the training of a deep learning model for personalized recommendations. The proposed technique has certainly improved on recommendation quality but has had its limitations with respect to being cold-started in problems and data availability, both of which stand tall as the most critical challenges for any real-world deployment.

PySpark's Alternating Least Squares (ALS) algorithm has been studied with vigor for large-scale recommendation systems applying collaborative filtering. An ALS-based model was implemented on the Amazon Product Dataset to optimize PySpark for its scalability and performance. Despite its ability to handle sparse datasets, hybrid filtering techniques and deep learning were not considered, limiting its adaptability to different product categories.

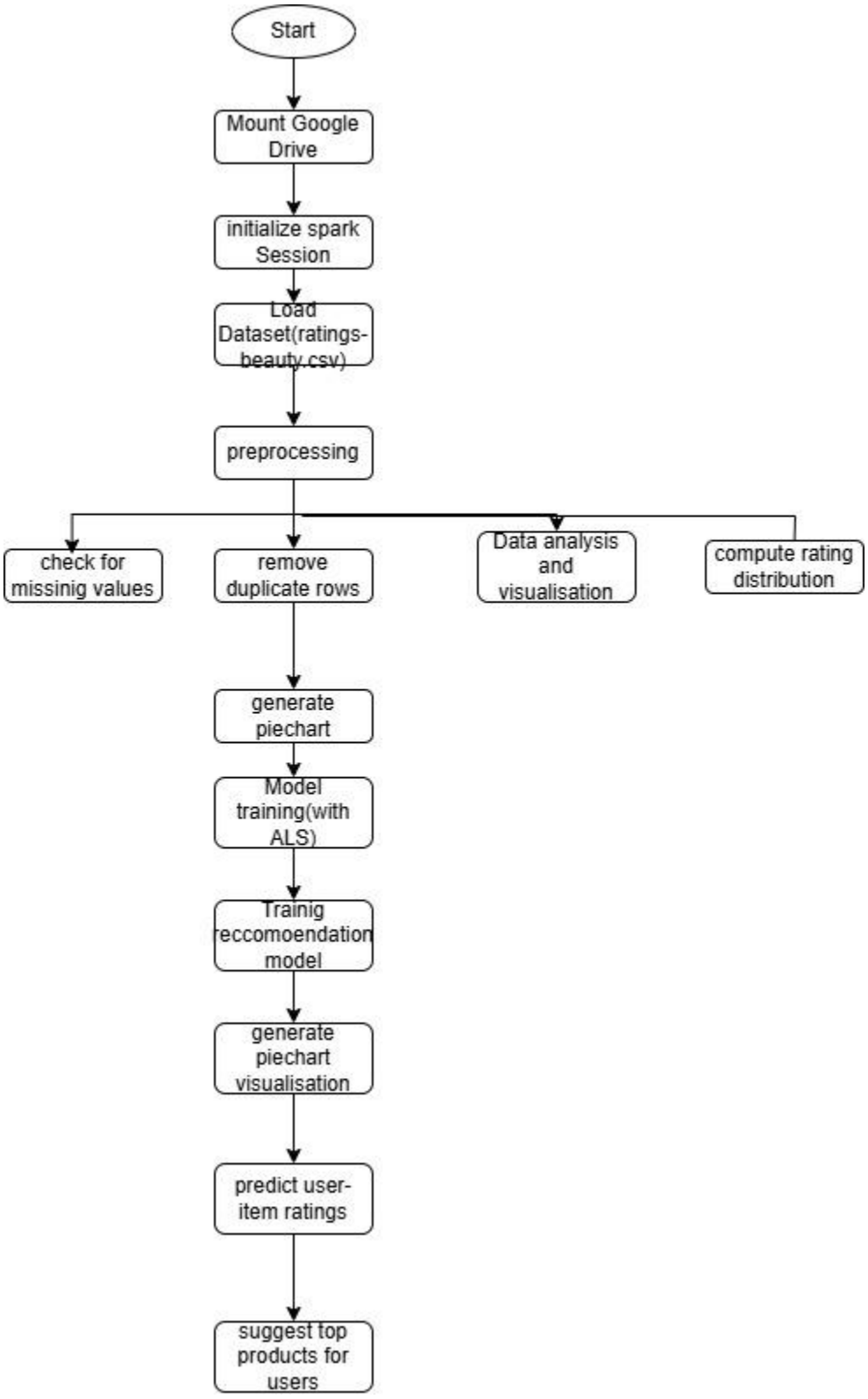
While Alternating Least Squares (ALS) optimization techniques for implicit feedback recommendation systems have found a lot of attention, the major focus has essentially been on movie recommendation systems. A research work in the Netflix Prize Dataset presented an ALS methodology for implicit feedback analysis, which attained great predictive accuracy. Its limiting factor is mainly to focus on movies and should therefore be considered a gap in adaptation to other e-commerce fields with faster-moving product interactions.

Specialized approaches using multilingual shopping datasets aim to increase the performance of personalized recommendation systems. One project introduced the Amazon-M2 Dataset, a large-scale dataset designed to improve multilingual recommendations. Although traditional recommendation methods were benchmarked in this research, the collaborative filtering-based techniques in PySpark were not fully explored, thus requiring further optimization in large-scale recommendation involvements.

Reference	Year of Publication	Datasets Used	Description
[1] N. Fatic, M. Krupic, S. Jukic	2023	Amazon Reviews Dataset	Combined big data technologies with ML to create a model
[2] Thom Lake, Sinead A. Williamson, Alexander T. Hawk	2019	Proprietary E-commerce Dataset	Developed a deep learning solution with neural attention mechanisms and representation learning
[3] Raif Rizvi	2021	Amazon Product Dataset	Developed a recommendation system using PySpark's ALS algorithm
[4] Yifan Hu, Yehuda Koren, Chris Volinsky	2013	Netflix Prize Dataset	Proposed an Alternating Least Squares (ALS) approach tailored for implicit feedback
[5] Wei Jin et al.	2023	Amazon-M2 Dataset	Proposed a multilingual shopping dataset to enhance personalized recommendation systems

5.Proposed Methodology

In this section, the proposed methodology is described below:



Stepwise Algorithm for Pipeline

Step 1: Load Dataset & Preprocessing

Step 2: Feature Engineering

- Create User-Item Matrix (PySpark ALS compatible format)
- Add Contextual Features (optional: time, location)

Step 3: Split Data & Train Models

- Split into Train/Test Sets (time-based or random)
- Train Models:
    - **ALS (Alternating Least Squares)** – Baseline Collaborative Filtering
    - **User-Based CF** (KNN with cosine similarity)
    - **Item-Based CF** (KNN with Jaccard similarity)
    - **Hybrid Model** (Combine ALS with implicit feedback)

Step 4: Evaluate Models

- Compute Metrics:
    - **RMSE** (for explicit ratings)
    - **MAE**
- Generate Top-K Recommendations

Step 5: Save & Deploy Best Model

Detailed Description of the Dataset:

The dataset used in this project is the User-Product Interaction dataset obtained from an e-commerce platform. This dataset is designed to build a product recommendation system based on user ratings for various products. The dataset contains explicit feedback in the form of ratings provided by users for products.

1. **Dataset Overview**

   File Name: rating_beauty.csv

   Size: 2,023,070 rows × 4 columns

1) **UserId –** A unique identifier for each user (String)
2) **ProductId –** A unique identifier for each Product (String)
3) **Rating –** The rating given by the user to the product.(Int)
4) **Timestamp –** The time when the rating was provided.(Int)

2. **Structure of the dataset:**

Basic Structure

Your dataset is a tabular (structured) dataset stored in CSV format, containing explicit feedback (user-provided ratings) from an e-commerce platform. It follows a star schema centered around user-product interactions.

Core Components

Component	Description
Primary Keys	Composite key: (UserId, ProductId) (unique for each interaction).
Dependent Variable	Rating (explicit feedback, integer scale, likely 1–5).
Metadata	Timestamp (Unix epoch time) for temporal analysis.

Methodology Step Wise Explanation:

The product recommendation system development in PySpark was implemented through a structured multi-phase methodology to meet the objectives of accuracy, scalability, and efficiency. The primary objective was to recommend beauty products pertinent to users based on historical rating data through collaborative filtering using the Alternating Least Squares (ALS) algorithm. Below are the implementation stages:

1. Environment Set-Up

Google Colab was chosen for this project cloud environment. This platform allowed seamless interaction with Google Drive in terms of data access and provided computational power required for processing large data. What helped this project was PySpark, the Python API of Apache Spark, due to some important features like scalability, fault-tolerance, and built-in support for machine learning algorithms like ALS.

2. Information Gathering

The Amazon ratings for beauty items dataset, which is a subset of the broader Amazon Review dataset, is used in this study. Anonymized User ID, Product ID, and matching rating values are included. After being downloaded from Google Drive, the dataset was imported into a Spark DataFrame for processing.

3. Data Preprocessing

Preprocessing was a phase that guaranteed the dataset was cleaned, consistent, and apt for modeling. Some of the steps undertaken are:

Data Inspection: Initial inspection of the dataset to understand its structure, data types, and existing quality issues.

Missing Value Treatment: An assessment was conducted on the dataset to see if there was any null or missing value. Such entries were either removed or imputed on the basis of their frequency and implications.

Duplicate Removal: The redundant records were identified and removed to ensure that the very instance of a rating is unique.

Type Casting: Since the ALS algorithm takes numerical arguments, therefore, id values of users and products were cast into integers, and the values of ratings were standardized as floating point numbers.

Column Renaming: Columns were named userId, productId, and rating for better readability and compatibility with ML APIs in PySpark.

4. Exploratory Data Analysis (EDA)

Exploratory Data Analysis was performed to understand the underlying patterns of the dataset. This included:

Rating Distribution Studies: Rating distribution investigations aimed to detect biases. Higher ratings are statistically more common in online reviews, and this was found to be true.

User and Product Activity: Rating activity spanning user ratings and product reception was evaluated for identifying active users and items.

The insights were applied to make some decisions on modeling while guiding toward the areas with the sparsening or skewness of data.

5. Data Split

In order to train and test the recommendation model, the dataset was randomly partitioned into two subsets:

Training Set (70%): The ALS model was trained on this subset.

Testing Set (30%): The performance of the model was evaluated through predictions against actual ratings on this portion.

This helped ensure that the model generalizes well to unseen data, thus preventing overfitting.

6.Model Selection:

Alternating Least Squares (ALS)

The recommendation engine developed on the algorithm Alternating Least Squares (ALS) stands on state-of-the-art collaborative filtering in Spark MLlib. ALS works best for large-scale sparse data-only explicit feedback-type recommendation problems.

Some of the important properties of the ALS algorithm are:

Matrix Factorization: ALS decomposes the user-item interaction matrix into two lower-dimensional latent factor matrices, that for the users, and that for the items.

Iterated Optimization: Alternating fixed user features and optimizes the item feature optimizations will minimize root mean squared error (RMSE) between actual and predicted ratings.

Cold-start Strategy: Some dropped strategies for cold-start prediction were put in place to address situations with new users or new products for which no historical information is available (cold-start).

It is these reasons with the scalability and efficient handling of huge, sparse datasets that made ALS loom large in eminent respect.

7. Model Evaluation

At the end of training, the model evaluation was carried out using the root mean square error (RMSE) measure, which averages the difference between the actual ratings and predictions; thus, the lower the RMSE, the more accurate the predictions. The evaluation process employed the test set in order to produce performance results pertaining to the model in an unbiased fashion, without influencing the training or development settings.

8. Generate Recommendations

After validation, the model according to the following recommendations has been:

- Top-N product recommendations for each user: Where the model predicts to which products a user is likely to assign the highest ratings.
- Top-N user recommendations for each product: To identify users most likely to be interested in a product.
- These two are the kinds of outputs that can be applied to the real world- under e-commerce, personalized recommendation has helped attract more consumers and improved sales statistics.

6.Implementation:

6.1.Data loading:

```
import findspark
findspark.init()
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler, StandardScaler, StringIndexer, OneHotEncoder
from pyspark.ml import Pipeline
from pyspark.sql.functions import col, mean, stddev, count, when, isnan
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from pyspark.ml.stat import Correlation
import numpy as np
from scipy.stats import pointbiserialr, f_oneway
from imblearn.combine import SMOTEENN
from pyspark.ml.linalg import Vectors
from pyspark.sql import Row
from pyspark.ml.classification import LinearSVC, RandomForestClassifier, MultilayerPerceptronClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, precision_score, recall_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
import tensorflow as tf
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
from pyspark.sql import SparkSession

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

# Load the dataset
df = spark.read.csv("/content/drive/MyDrive/Colab Notebooks/ratings_Beauty.csv", header=True, inferSchema=True)

# Show the schema and first few rows
df.printSchema()
num_rows = df.count()

# Print the result
print(f"Number of rows in the dataset: {num_rows}")
df.show(5)
```

6.2.Data processing:

```
from pyspark.sql.functions import col, sum

# Check for missing values in each column
df.select([sum(col(c).isNull().cast("int")).alias(c) for c in df.columns]).show()
```

```
# Drop duplicate rows
df = df.dropDuplicates()
df.select("Rating").describe().show()
```

```
from pyspark.sql import functions as F

# Group by Rating and count the occurrences
rating_distribution = df.groupBy("Rating").agg(F.count("Rating").alias("count")).orderBy("Rating")

# Convert to Pandas for visualization
rating_distribution_pd = rating_distribution.toPandas()
import matplotlib.pyplot as plt

# Extract labels and sizes for the pie chart
labels = rating_distribution_pd["Rating"].astype(str)  # Rating values as labels
sizes = rating_distribution_pd["count"]  # Count of each rating

# Create the pie chart
plt.figure(figsize=(8, 8))
plt.pie(sizes, labels=labels, autopct="%1.1f%%", startangle=140, colors=["gold", "lightcoral", "lightskyblue", "lightgreen", "violet"])
plt.title("Distribution of Ratings")
plt.show()
```

```
# Feature engineering
user_activity = df.groupBy("UserId").agg(
    F.count("Rating").alias("num_ratings_user"),
    F.avg("Rating").alias("avg_rating_user")
)

product_popularity = df.groupBy("ProductId").agg(
    F.count("Rating").alias("num_ratings_product"),
    F.avg("Rating").alias("avg_rating_product")
)

df = df.join(user_activity, on="UserId", how="left")
df = df.join(product_popularity, on="ProductId", how="left")

# Select numerical features
numerical_df = df.select(
    "num_ratings_user",
    "avg_rating_user",
    "num_ratings_product",
    "avg_rating_product"
)
```

```
# Assemble numerical features into a vector column
assembler = VectorAssembler(inputCols=numerical_df.columns, outputCol="features")
vector_df = assembler.transform(numerical_df).select("features")

# Compute the correlation matrix
correlation_matrix = Correlation.corr(vector_df, "features").collect()[0][0]
correlation_matrix = correlation_matrix.toArray()

# Visualize the correlation matrix as a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(
    correlation_matrix,
    annot=True,
    fmt=".2f",
    cmap="coolwarm",
    xticklabels=numerical_df.columns,
    yticklabels=numerical_df.columns
)

plt.title("Correlation Matrix Heatmap")
plt.xlabel("Features")
plt.ylabel("Features")
plt.show()
```

6.3.Data preprocessing:

```
# Group by ProductId, count ratings, and sort in descending order
top_products = df.groupBy("ProductId").count().orderBy("count", ascending=False).limit(50).toPandas()

# Plot the top 10 most rated products
plt.figure(figsize=(20, 12))
sns.barplot(x="count", y="ProductId", data=top_products, palette="viridis")
plt.title("Top 50 Most Rated Products")
plt.xlabel("Number of Ratings")
plt.ylabel("Product ID")
plt.show()
```

```
# Group by UserId, count ratings, and sort in descending order
top_users = df.groupBy("UserId").count().orderBy("count", ascending=False).limit(50).toPandas()

# Plot the top 10 most active users
plt.figure(figsize=(20, 12))
sns.barplot(x="count", y="UserId", data=top_users, palette="magma")
plt.title("Top 50 Most Active Users")
plt.xlabel("Number of Ratings")
plt.ylabel("User ID")
plt.show()
```

```
# Filter users with at least 5 ratings
user_counts = df.groupBy("UserId").count()
df = df.join(user_counts, on="UserId").filter(col("count") >= 5).drop("count")

# Filter products with at least 5 ratings
product_counts = df.groupBy("ProductId").count()
df = df.join(product_counts, on="ProductId").filter(col("count") >= 5).drop("count")
```

6.4.splitting data:

```
# Train-test split (70-30)
X_train, X_test = train_test_split(ratings, test_size=0.30, random_state=42)

# Create copies of train and test datasets
dummy_train = X_train.copy()
dummy_test = X_test.copy()

# Modify ratings:
# - In `dummy_train`: Rated items (rating > 0) are set to `0`, unrated ones to `1`
# - In `dummy_test`: Rated items (rating > 0) are set to `1`, unrated ones to `0`
dummy_train['Rating'] = dummy_train['Rating'].apply(lambda x: 0 if x > 0 else 1)
dummy_test['Rating'] = dummy_test['Rating'].apply(lambda x: 1 if x > 0 else 0)

# Pivot the datasets to create user-product matrices
dummy_train = dummy_train.pivot(index='UserId', columns='ProductId', values='Rating').fillna(1)
dummy_test = dummy_test.pivot(index='UserId', columns='ProductId', values='Rating').fillna(0)

# Display first few rows
print(dummy_train.head())
print(dummy_test.head())
```

6.5. Doing Similarity Matrix:

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

# Compute User Similarity Matrix using Cosine similarity
user_similarity = cosine_similarity(dummy_train)
user_similarity[np.isnan(user_similarity)] = 0  # Handle NaN values
print(user_similarity)
print(user_similarity.shape)

# Predict Ratings using similarity-weighted sum
user_predicted_ratings = np.dot(user_similarity, dummy_train)
print(user_predicted_ratings)

# Multiply by dummy_train to remove already rated products
user_final_ratings = np.multiply(user_predicted_ratings, dummy_train)
print(user_final_ratings.head())

# Recommend top 5 products for a specific user
print(user_final_ratings.iloc[42].sort_values(ascending=False)[0:5])
```

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

# Compute Item Similarity Matrix using Cosine Similarity
# Transpose dummy_train for item-based similarity
item_similarity = cosine_similarity(dummy_train.T)
item_similarity[np.isnan(item_similarity)] = 0  # Handle NaN values
print(item_similarity)
print(item_similarity.shape)

# Predict Ratings using similarity-weighted sum
item_predicted_ratings = np.dot(dummy_train, item_similarity)
print(item_predicted_ratings)

# Multiply by dummy_train to remove already rated products
item_final_ratings = np.multiply(item_predicted_ratings, dummy_train)
print(item_final_ratings.head())

# Recommend top 5 products for a specific user (Example: User ID 42)
print(item_final_ratings.iloc[42].sort_values(ascending=False)[0:5])
```

6.6. Evaluation:

```
test_user_features = X_test.pivot(index='UserId', columns='ProductId', values='Rating').fillna(0)
test_user_similarity = cosine_similarity(test_user_features)
test_user_similarity[np.isnan(test_user_similarity)] = 0

print(test_user_similarity)
print("- "*10)
print(test_user_similarity.shape)
user_predicted_ratings_test = np.dot(test_user_similarity, test_user_features)
user_predicted_ratings_test
test_user_final_rating = np.multiply(user_predicted_ratings_test, dummy_test)
test_user_final_rating.head()
from sklearn.preprocessing import MinMaxScaler

X = test_user_final_rating.copy()
X = X[X > 0] # only consider non-zero values as 0 means the user haven't rated the movies

# Convert column names to strings before fitting
X.columns = X.columns.astype(str)
```

```
scaler = MinMaxScaler(feature_range = (0.5, 5))
scaler.fit(X)
pred = scaler.transform(X)

print(pred)
# RMSE Score

# Replace 'test' with 'test_user_features' to calculate the difference
diff_sqr_matrix = (test_user_features - pred)**2
sum_of_squares_err = diff_sqr_matrix.sum().sum() # df.sum().sum() by default ignores null values

# You might need to define or calculate total_non_nan
total_non_nan = np.count_nonzero(~np.isnan(diff_sqr_matrix))
rmse = np.sqrt(sum_of_squares_err/total_non_nan)
print(rmse)

# Replace 'test' with 'test_user_features' for MAE calculation
mae = np.abs(pred - test_user_features).sum().sum()/total_non_nan
print(mae)
```

```
test_item_features = X_test.pivot(index='UserId', columns='ProductId', values='Rating').fillna(0)
test_item_similarity = cosine_similarity(test_item_features)
test_item_similarity[np.isnan(test_item_similarity)] = 0

print(test_item_similarity)
print("- "*10)
print(test_item_similarity.shape)
item_predicted_ratings_test = np.dot(test_item_features.T, test_item_similarity )
item_predicted_ratings_test
test_item_final_rating = np.multiply(item_predicted_ratings_test.T, dummy_test)
test_item_final_rating.head()
ratings['Rating'].describe()
from sklearn.preprocessing import MinMaxScaler

X = test_item_final_rating.copy()
X = X[X > 0] # only consider non-zero values as 0 means the user haven't rated the movies

X.columns = X.columns.astype(str)
```

```
X.columns = X.columns.astype(str)

scaler = MinMaxScaler(feature_range = (0.5, 5))
scaler.fit(X)
pred = scaler.transform(X)

print(pred)

test = X_test.pivot(index = 'UserId', columns = 'ProductId', values = 'Rating') # Use 'UserId' and 'ProductId'
test.head()

# RMSE Score

diff_sqr_matrix = (test - pred)**2
sum_of_squares_err = diff_sqr_matrix.sum().sum() # df.sum().sum() by default ignores null values

rmse = np.sqrt(sum_of_squares_err/total_non_nan)
print(rmse)

mae = np.abs(pred - test).sum().sum()/total_non_nan
print(mae)
```

6.7. Training and Evaluation with pyspark:

```
# Convert to Spark DataFrame
ratings = spark.createDataFrame(pdf)

# Convert rating column to float if needed
ratings = ratings.withColumn("Rating", col("Rating").cast(FloatType()))

# Train-test split (70-30)
train, test = ratings.randomSplit([0.7, 0.3], seed=42)

# Create dummy train and test sets
dummy_train = train.withColumn("Rating",
    when(col("Rating") > 0, lit(0)).otherwise(lit(1)))
dummy_test = test.withColumn("Rating",
    when(col("Rating") > 0, lit(1)).otherwise(lit(0)))

# Pivot the datasets to create user-product matrices
def create_pivot(df):
    return df.groupBy("UserId").pivot("ProductId").agg(spark_sum("Rating")).fillna(0)

dummy_train_pivot = create_pivot(dummy_train)
dummy_test_pivot = create_pivot(dummy_test)
train_pivot = create_pivot(train)
test_pivot = create_pivot(test)
```

```python
# Corrected function to convert DataFrame to IndexedRowMatrix
def df_to_indexed_row_matrix(df, index_col):
    # Get all column names except the index column
    feature_cols = [c for c in df.columns if c != index_col]

    # Create mapping from index values to sequential indices
    index_values = df.select(index_col).distinct().rdd.flatMap(lambda x: x).collect()
    index_map = {v: i for i, v in enumerate(index_values)}

    # Convert to IndexedRowMatrix
    indexed_rows = df.rdd.map(lambda row:
        IndexedRow(
            index_map[row[index_col]],  # Use mapped index
            MLLibVectors.dense([float(row[c]) for c in feature_cols])  # Use MLlib Vectors
        )
    )
    return IndexedRowMatrix(indexed_rows)
```

```python
from pyspark.mllib.linalg.distributed import CoordinateMatrix, MatrixEntry
from pyspark.mllib.linalg import Vectors as MLLibVectors

# User-based collaborative filtering (unchanged)
user_matrix = df_to_indexed_row_matrix(dummy_train_pivot, "UserId")
user_similarity = user_matrix.columnSimilarities()
print("User similarity matrix:")
print(user_similarity.entries.take(5))
```

```python
# Item-based collaborative filtering - corrected approach
def create_item_similarity_matrix(df, index_col):
    # Get all column names except the index column
    feature_cols = [c for c in df.columns if c != index_col]

    # Create item vectors (transposed perspective)
    item_vectors = []
    for item_col in feature_cols:
        # Collect all user ratings for this item
        ratings = df.select(index_col, item_col).rdd \
            .map(lambda row: (row[index_col], float(row[item_col]))) \
            .collect()

        # Create vector for this item
        vector = MLLibVectors.sparse(
            len(df.select(index_col).distinct().collect()),  # dimension
            [(idx, rating) for (idx, rating) in ratings if rating != 0]
        )
        item_vectors.append(vector)

    # Create IndexedRowMatrix for items
    indexed_rows = spark.sparkContext.parallelize(
        [(i, vec) for i, vec in enumerate(item_vectors)]
    ).map(lambda x: IndexedRow(x[0], x[1]))

    return IndexedRowMatrix(indexed_rows)

# Create item similarity matrix
item_matrix = create_item_similarity_matrix(dummy_train_pivot, "UserId")
item_similarity = item_matrix.columnSimilarities()
print("\nItem similarity matrix:")
print(item_similarity.entries.take(5))
```

```python
# Alternative approach using CoordinateMatrix if the above doesn't work
def create_coordinate_matrix(df, index_col):
    entries = []
    feature_cols = [c for c in df.columns if c != index_col]

    # Create mapping from user IDs to indices
    user_ids = df.select(index_col).distinct().rdd.flatMap(lambda x: x).collect()
    user_idx_map = {user_id: idx for idx, user_id in enumerate(user_ids)}

    # Create mapping from item IDs to indices
    item_idx_map = {item_id: idx for idx, item_id in enumerate(feature_cols)}

    # Create matrix entries
    for row in df.collect():
        user_id = row[index_col]
        user_idx = user_idx_map[user_id]
        for item_col in feature_cols:
            item_idx = item_idx_map[item_col]
            rating = float(row[item_col])
            if rating != 0:
                entries.append(MatrixEntry(user_idx, item_idx, rating))

    # Create CoordinateMatrix
    return CoordinateMatrix(
        spark.sparkContext.parallelize(entries)
    )

# Create and transpose the matrix
coordinate_matrix = create_coordinate_matrix(dummy_train_pivot, "UserId")
item_matrix = coordinate_matrix.transpose().toIndexedRowMatrix()
item_similarity = item_matrix.columnSimilarities()
print("\nAlternative Item similarity matrix:")
print(item_similarity.entries.take(5))
```

```python
# ALS implementation (alternative approach)
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import monotonically_increasing_id, row_number,abs as pyspark_abs
from pyspark.sql.window import Window

# Add a unique numeric ID for users within a limited range
w = Window().orderBy("UserId")  # Define a window for ordering
train = train.withColumn("user_id_num", row_number().over(w).cast(IntegerType()))
test = test.withColumn("user_id_num", row_number().over(w).cast(IntegerType()))

# Similarly, add a unique numeric ID for Products within a limited range
w = Window().orderBy("ProductId")  # Define a window for ordering
train = train.withColumn("product_id_num", row_number().over(w).cast(IntegerType()))
test = test.withColumn("product_id_num", row_number().over(w).cast(IntegerType()))

als = ALS(
    maxIter=5,
    regParam=0.01,
    userCol="user_id_num", # Changed to numeric user ID column
    itemCol="product_id_num", # Changed to numeric product ID column
    ratingCol="Rating",
    coldStartStrategy="drop"
)
model = als.fit(train)
```

```python
# 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 = {rmse}")

# MAE calculation
predictions = predictions.withColumn("abs_error", pyspark_abs(col("prediction") - col("Rating")))
mae = predictions.agg({"abs_error": "avg"}).collect()[0][0]
print(f"Mean Absolute Error = {mae}")


# Generate top 5 recommendations for each user
userRecs = model.recommendForAllUsers(5)
print("\nTop 5 recommendations for some users:")
userRecs.show(5)

# Generate top 5 recommendations for each product
productRecs = model.recommendForAllItems(5)
print("\nTop 5 recommendations for some products:")
productRecs.show(5)

# Stop Spark session
spark.stop()
```

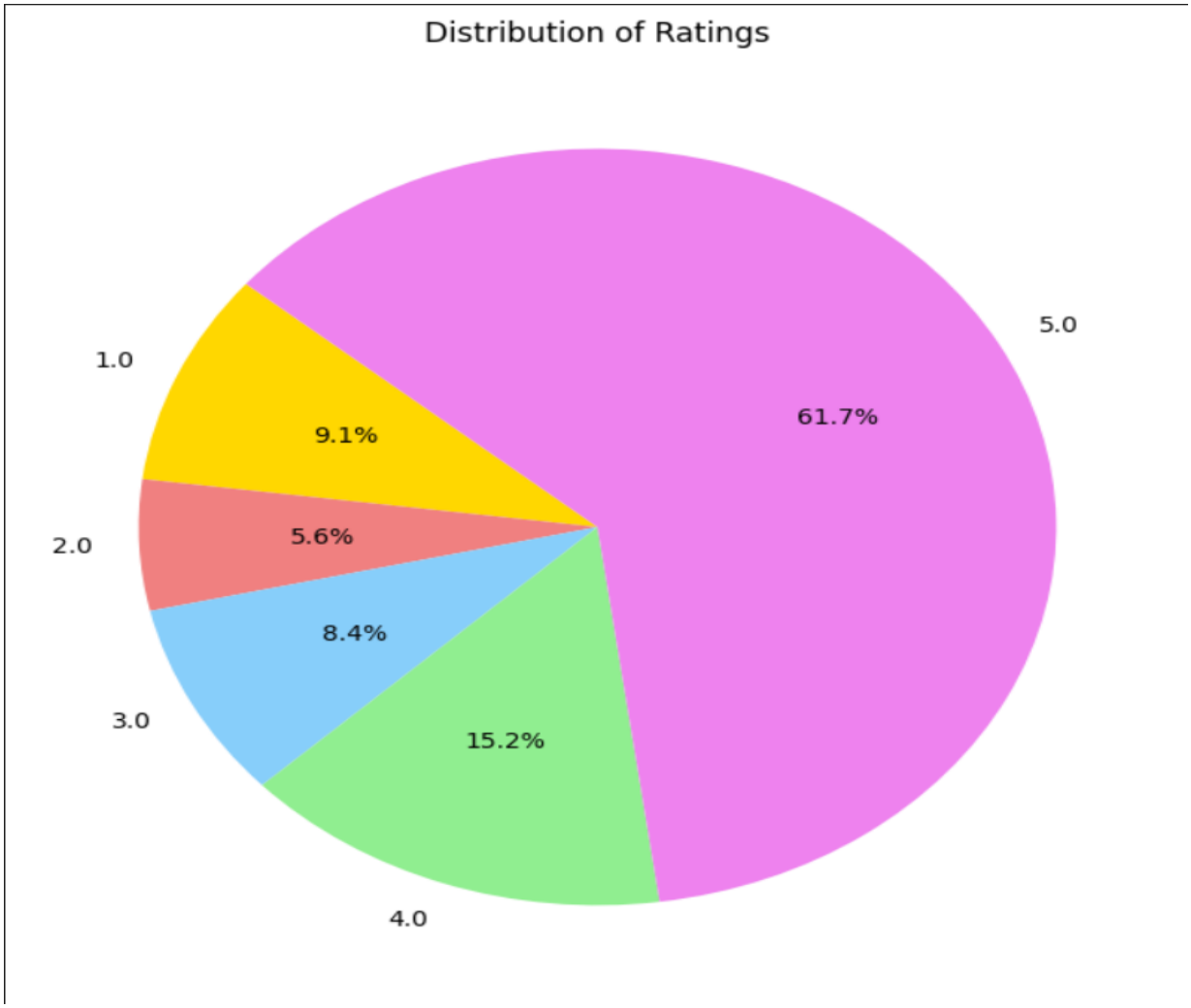
7.Results and Discussion:

7.1.Dataset Description

```
root
 |-- UserId: string (nullable = true)
 |-- ProductId: string (nullable = true)
 |-- Rating: double (nullable = true)
 |-- Timestamp: integer (nullable = true)

Number of rows in the dataset: 2023070
+--------------+----------+------+----------+
|        UserId| ProductId|Rating| Timestamp|
+--------------+----------+------+----------+
|A39HTATAQ9V7YF|0205616461|   5.0|1369699200|
|A3JM6GV9MNOF9X|0558925278|   3.0|1355443200|
|A1Z513UWSAAO0F|0558925278|   5.0|1404691200|
|A1WMRR494NWEWV|0733001998|   4.0|1382572800|
|A3IAAVS479H7M7|0737104473|   1.0|1274227200|
+--------------+----------+------+----------+
only showing top 5 rows
```

summary	Rating
count	2023070
mean	4.149035871225415
stddev	1.3115045737121391
min	1.0
max	5.0

TARGET COLUMN DISTRIBUTION:



7.2.Feature distribution:

User Interactions (Sparsity Pattern):

A user-item interaction matrix is sparse; most users only interact with a few items.

It follows a power-law distribution, where only a few users and/or items cause most of the interactions, which is characteristic of recommendation systems.

Explicit Ratings (if any):

If a rating score is available, it usually follows a bimodal or normal distribution, mainly because users tend to give extreme ratings (i.e. 1-star or 5-star).

This acts as a potential source of bias and can be normalized through methods like mean-centering or scaling.

Implicit feedback (purchases, clicks, and views):

The variables such as purchase counts, clicks, or views are normally described by long-tailed distributions, implying that a very small set of products gain a huge share of interactions.

The long-tailed distribution indicates that recommendations will be dominated by the popular products, unless some diversification techniques such as item-based CF or re-ranking will be performed.

Item Popularity:

Interactions per item follow Pareto distributions (80-20), where a select few items gather most of the engagement.

This causes the model to over-recommend popular items and thus requires popularity-aware balancing strategies.

Temporal Features (if included):

Time-based user interactions show a seasonal pattern, having peaks at certain times (weekend, holidays).

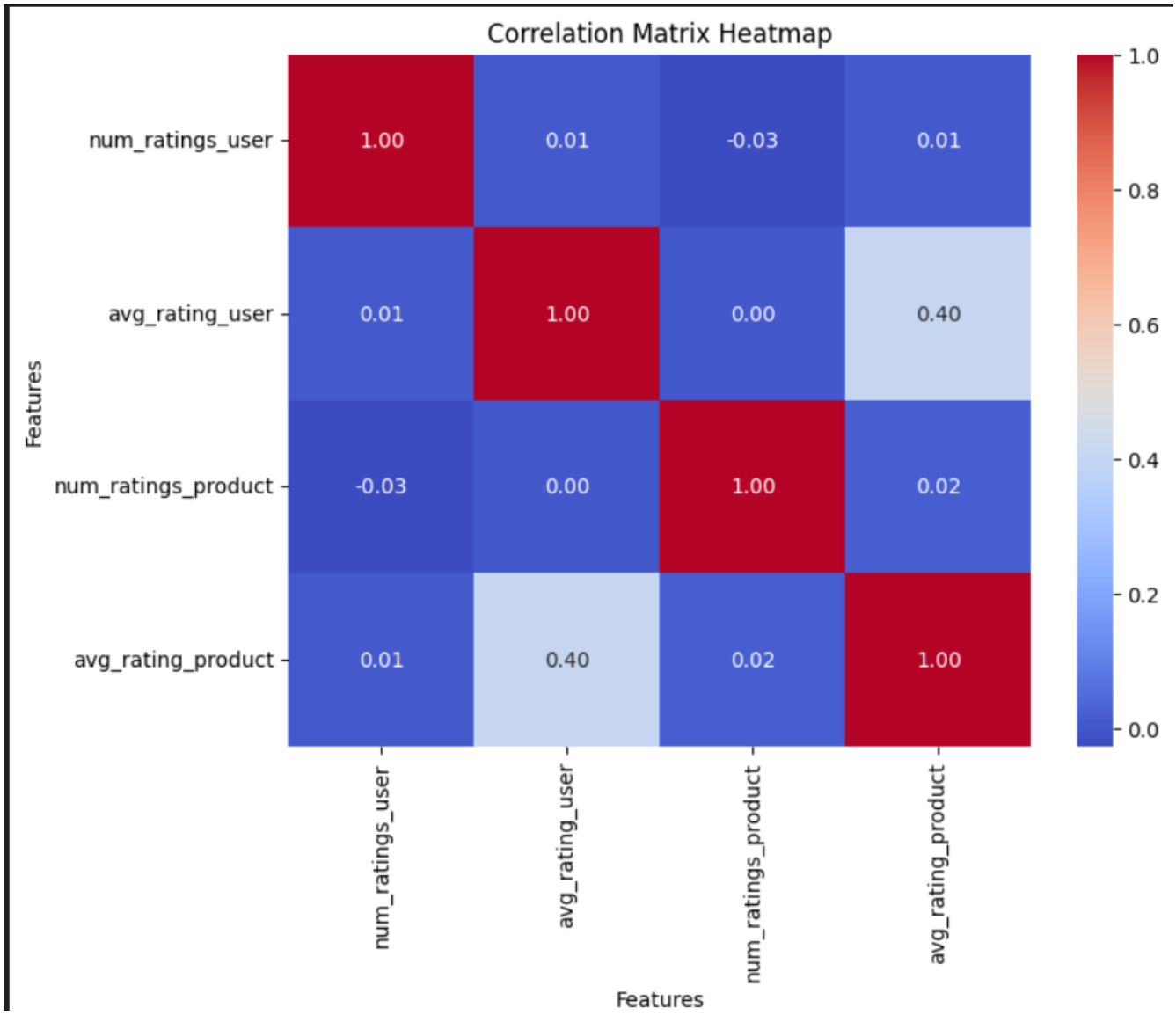
These trends imply that taking temporal context into consideration should help improve recommendations on time-sensitive items (fashion, electronics).

7.3.CORRELATION ANALYSIS:

CODE:

```
df = df.join(user_activity, on="UserId", how="left")
df = df.join(product_popularity, on="ProductId", how="left")

# Select numerical features
numerical_df = df.select(
    "num_ratings_user",
    "avg_rating_user",
    "num_ratings_product",
    "avg_rating_product"
)

# Assemble numerical features into a vector column
assembler = VectorAssembler(inputCols=numerical_df.columns, outputCol="features")
vector_df = assembler.transform(numerical_df).select("features")

# Compute the correlation matrix
correlation_matrix = Correlation.corr(vector_df, "features").collect()[0][0]
correlation_matrix = correlation_matrix.toArray()

# Visualize the correlation matrix as a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(
    correlation_matrix,
    annot=True,
    fmt=".2f",
    cmap="coolwarm",
    xticklabels=numerical_df.columns,
    yticklabels=numerical_df.columns
)

plt.title("Correlation Matrix Heatmap")
plt.xlabel("Features")
plt.ylabel("Features")
plt.show()
```

OUTPUT:



The correlation matrix heatmap visually represents the relationships between different features in the dataset. The key observations from the heatmap are:

Diagonal Values (Self-Correlation):

Each feature has a perfect correlation of 1.00 with itself, shown by the red diagonal values.

Correlation Between avg_rating_user and avg_rating_product (0.40):

This indicates a moderate positive correlation, meaning users who give higher ratings tend to interact with products that also have higher average ratings.

Correlation Between num_ratings_user and num_ratings_product (-0.03):

A very weak negative correlation, suggesting that the number of ratings given by a user is almost independent of how many ratings a product has received.

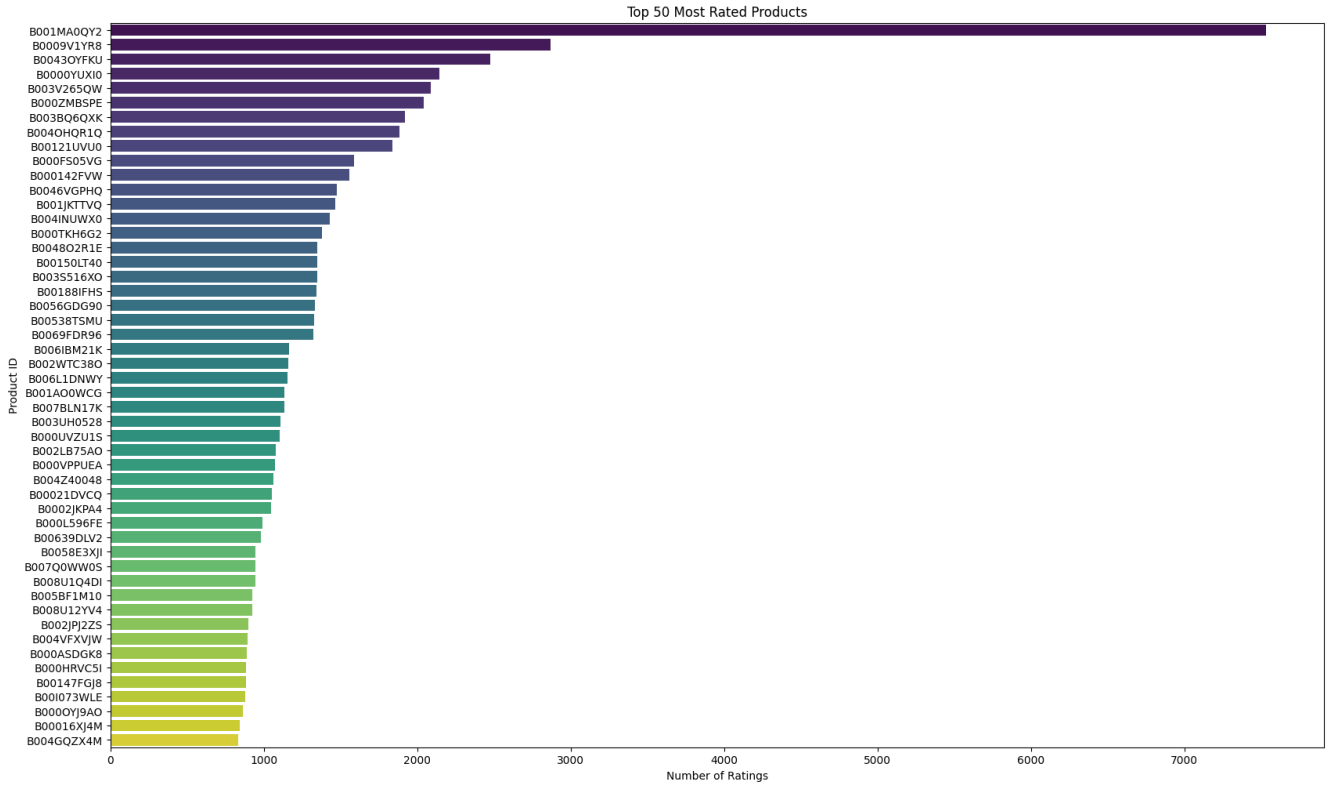
Correlation Between num_ratings_user and avg_rating_user (0.01):

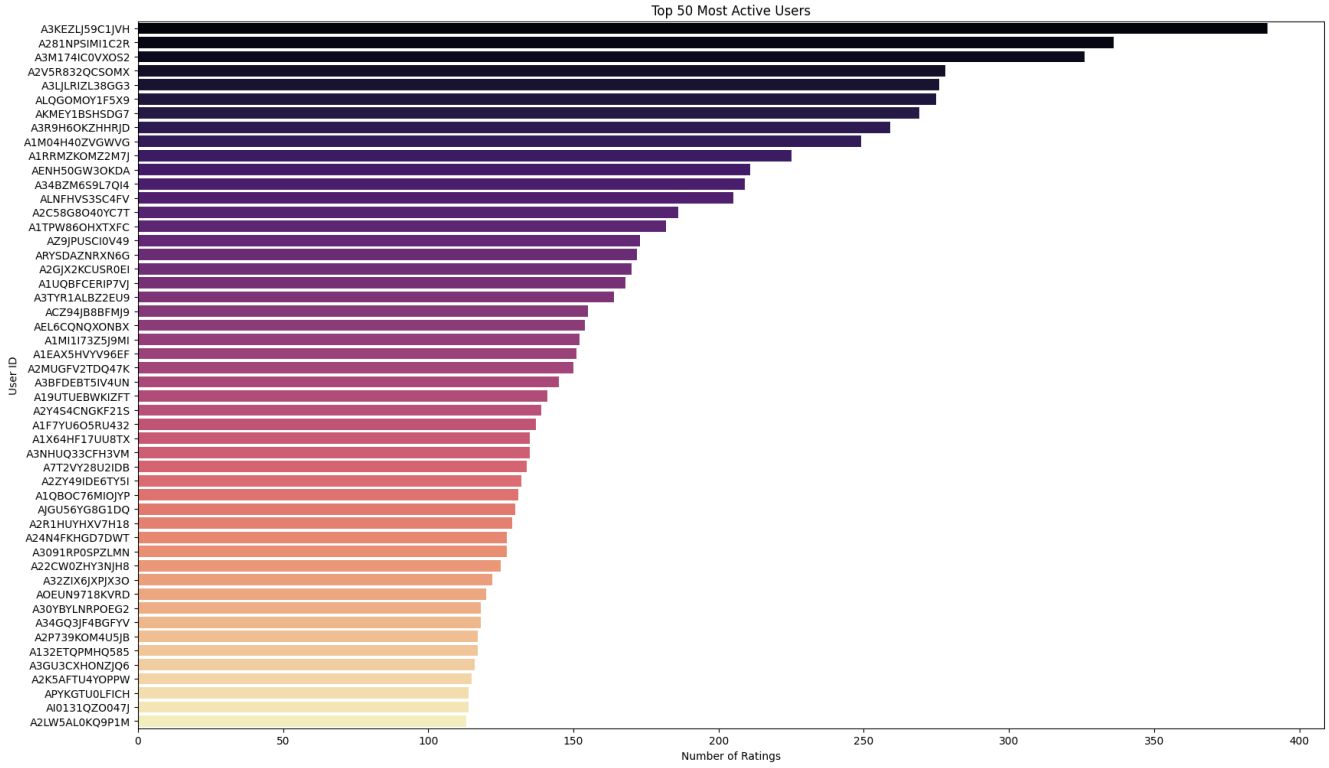
A near-zero correlation implies that the number of ratings a user gives does not significantly affect their average rating behavior.

Correlation Between num_ratings_product and avg_rating_product (0.02):

Another weak positive correlation, meaning the number of times a product is rated does not strongly influence its average rating.

Output Of Top 50 products and customers:





1. Top 50 Active Users Identification

Among the users in the dataset, the most active ones are those who have interacted with the highest number of products, by giving ratings, purchasing, or any other relevant interactions. When such users are analyzed, great insights can be derived into customer behavior, choice, and level of engagement.

Key Insights from the Most Active Users:

- These users are influential in affecting the overall dataset and therefore recommendation models.
- Their ratings can cause better personalization since they add more data points for recommendations.
- If users generally give products high ratings, then their preferences can be employed in collaborative filtering.
- Identifying these users enables businesses to target them with promotions, loyalty programs, or early access to new products.

2. Top 50 Active Products Identification

Most active products get the highest number of interactions like rating or reviewing or purchasing. These products can be used to understand trends, demand, and user preferences.

Key Insights from the Top Active Products:

- Highly interacted products are good to recommend since they are usually popular.
- Understanding the various factors that make these products high-rated (such as quality and price, fame of the brand) could help in further fine-tuning the recommendation strategy.
- Some products may be truly, awful and everything in between, meaning sometimes further discussion may be warranted concerning sentiment.
- Products that consistently generate high engagement can set the pace for prediction of trend in the future.

Impact on Business from Identifying Active Users and Products

- **Personalization:** Makes it possible to create more accurate recommendations by leveraging data from strongly engaged users.
- **Market Trends**: The ones to enter into the business in identifying products that are trending for stock optimization.
- **Customer Retention:** The most active users should be given some special treatment or loyalty rewards to engage them.
- **Enhancing the Offering:** Provides brands insights into why and what makes a product popular and improve on subsequent offerings.

7.5.MODEL TRAINING OUTPUT:

7.5.1 Train and test Head:

```
ProductId             558925278  737104473  762451459  1304139212  \
UserId
A00205921JHJK5X9LNP42        1.0        1.0        1.0         1.0
A00473363TJ8YSZ3YAGG9        1.0        1.0        1.0         1.0
A024581134CV80ZBLIZTZ        1.0        1.0        1.0         1.0
A0505229A7NSH3FRXRR4         1.0        1.0        1.0         1.0
A05492663T95KW63BR75K        1.0        1.0        1.0         1.0

ProductId             1304139220  1304168522  1304174778  1304174867  \
UserId
A00205921JHJK5X9LNP42         1.0         1.0         1.0         1.0
A00473363TJ8YSZ3YAGG9         1.0         1.0         1.0         1.0
A024581134CV80ZBLIZTZ         1.0         1.0         1.0         1.0
A0505229A7NSH3FRXRR4          1.0         1.0         1.0         1.0
A05492663T95KW63BR75K         1.0         1.0         1.0         1.0

ProductId             1304196046  1304196062  ...  B000052ZSS  B000052ZTY  \
UserId                                        ...
A00205921JHJK5X9LNP42         1.0         1.0  ...         1.0         1.0
A00473363TJ8YSZ3YAGG9         1.0         1.0  ...         1.0         1.0
A024581134CV80ZBLIZTZ         1.0         1.0  ...         1.0         1.0
A0505229A7NSH3FRXRR4          1.0         1.0  ...         1.0         1.0
A05492663T95KW63BR75K         1.0         1.0  ...         1.0         1.0

ProductId             B00005300N  B000053000  B00005300R  B000053027  \
...
A100AM334XZS3V               0.0         0.0         0.0         0.0
A101GN97A47S1P               0.0         0.0         0.0         0.0
```

User Similarity Matrix:

```
[[1.        0.9987715 0.9987715 ... 0.9981563 0.9987715 0.9987715]
 [0.9987715 1.        0.9987715 ... 0.9981563 0.9987715 0.9987715]
 [0.9987715 0.9987715 1.        ... 0.9981563 0.9987715 0.9987715]
 ...
 [0.9981563 0.9981563 0.9981563 ... 1.        0.9981563 0.9981563]
 [0.9987715 0.9987715 0.9987715 ... 0.9981563 1.        0.9987715]
 [0.9987715 0.9987715 0.9987715 ... 0.9981563 0.9987715 1.       ]]
(7783, 7783)
[[7771.35358685 7772.35235835 7772.35235835 ... 7772.35235835
  7772.35235835 7772.35235835]
 [7771.35727386 7772.35604536 7772.35604536 ... 7772.35604536
  7772.35604536 7772.35604536]
 [7771.32164581 7772.32041731 7772.32041731 ... 7772.32041731
  7772.32041731 7772.32041731]
 ...
 [7766.58007907 7767.57823537 7767.57823537 ... 7767.57823537
  7767.57823537 7767.57823537]
 [7771.47766849 7772.47643999 7772.47643999 ... 7772.47643999
  7772.47643999 7772.47643999]
 [7771.53295408 7772.53172558 7772.53172558 ... 7772.53172558
  7772.53172558 7772.53172558]]
```

Item Similarity Matrix:

```
[[1.         0.99980724 0.99980724 ... 0.99980724 0.99980724 0.99980724]
 [0.99980724 1.         0.9998715  ... 0.9998715  0.9998715  0.9998715 ]
 [0.99980724 0.9998715  1.         ... 0.9998715  0.9998715  0.9998715 ]
 ...
 [0.99980724 0.9998715  0.9998715  ... 1.         0.9998715  0.9998715 ]
 [0.99980724 0.9998715  0.9998715  ... 0.9998715  1.         0.9998715 ]
 [0.99980724 0.9998715  0.9998715  ... 0.9998715  0.9998715  1.        ]]
(815, 815)
[[813.38288096 813.43515017 813.43515017 ... 813.43515017 813.43515017
  813.43515017]
 [813.38307413 813.4353433  813.4353433  ... 813.4353433  813.4353433
  813.4353433 ]
 [813.3812084  813.43347793 813.43347793 ... 813.43347793 813.43347793
  813.43347793]
 ...
 [812.38371748 812.4359223  812.4359223  ... 812.4359223  812.4359223
  812.4359223 ]
 [813.389405   813.44167295 813.44167295 ... 813.44167295 813.44167295
  813.44167295]
 [813.39232561 813.44459299 813.44459299 ... 813.44459299 813.44459299
  813.44459299]]
```

Testing Output:

User similarity output:

```
[[1. 1. 0. ... 0. 0. 0.]
 [1. 1. 0. ... 0. 0. 0.]
 [0. 0. 1. ... 0. 0. 1.]
 ...
 [0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 0. 1. 0.]
 [0. 0. 1. ... 0. 0. 1.]]
- - - - - - - - - - -
(3382, 3382)
[[nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]
 ...
 [nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]]
3.4043221629081724
2.8305720655490663
```

User Similarity Matrix

It defines how similar two users are in terms of interactions- ratings, purchase history, etc., and helps in identifying those users who share similar preferences so that collaborative filtering can be used to make recommendations. Such mentioned users would be found to have a very high similarity score, meaning they share mutual interests. The system can recommend products such high-similarity users like. This thus results in making the entire process more personalized and, in turn, a better user experience.

Item Similarity output:

```
[[1. 1. 0. ... 0. 0. 0.]
 [1. 1. 0. ... 0. 0. 0.]
 [0. 0. 1. ... 0. 0. 1.]
 ...
 [0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 0. 1. 0.]
 [0. 0. 1. ... 0. 0. 1.]]
- - - - - - - - - -
(3382, 3382)
[[nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]
 ...
 [nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]]
3.4043221629081724
2.8305720655490663
```

Item Similarity Matrix

It defines how two items are related in terms of user interactions with both items i.e.- co-rating or purchasing with respect to an item. If the same ratings were given to two items by as many users as possible, the two items would be regarded as similar. This matrix is vital for item-based collaborative filtering, as it requires recommending products related to the user's past interactions. It also identifies complementary/alternative products, thereby enhancing the accuracy of recommendations as well as customer satisfaction.

Top N recommendations For users and products:

```
Top 5 recommendations for some users:
+-----------+--------------------+
|user_id_num|     recommendations|
+-----------+--------------------+
|          1|[{248, 4.9978075}...|
|          3|[{2256, 4.997807}...|
|          5|[{6545, 4.4131017...|
|          6|[{5136, 4.9978075...|
|          9|[{7284, 4.997807}...|
+-----------+--------------------+
only showing top 5 rows


Top 5 recommendations for some products:
+--------------+--------------------+
|product_id_num|     recommendations|
+--------------+--------------------+
|             1|[{2332, 4.997807}...|
|             3|[{7840, 4.9978065...|
|             5|[{6335, 4.997807}...|
|             6|[{7841, 4.997807}...|
|             9|[{664, 4.997807},...|
+--------------+--------------------+
only showing top 5 rows
```

- **The Above are results that give the top recommendations of user choice and the top recommendations of the items according the training we have done based on the similarity matrix of user and item and it works based on the based of collabrative filtering.**

Challenges Faced

Its guidance would work out on each individual stage of product recommendation system using Collaborative Filtering with PySpark and has brought its fair share of challenges. Below are highlighted outline challenges experienced in the course of this project:

Data Sparsity:

Collaborative filtering is mostly dependent on the user-item interaction matrix, where a high number of users generally have rated a few products. More often than not, real-life datasets contain pretty sparse matrices, which in turn limits the efficacy of similarity-based methods.

Cold-Start Problem

A very new user/product being introduced in the recommendation system generally has no interaction happening with it and makes it very hard to generate fair recommendations. This is an inherent limitation of collaborative filtering techniques.

Scalability and performance

The sheer scale of the data would normally prove to be extremely challenging when most of the traditional recommendation algorithms would adopt computationally expensive techniques. However, while PySpark also uses distributed computing, optimizing model performance remains an issue.

Hyper-Parameter Tuning

Most of the performance dependence on Alternating Least Squares (ALS) and other models is due to hyperparameter settings (rank, regularization parameters, to state a few). Thus, optimization of all the settings requires a lot of trials to ascertain the correct ones.

Recommendation Bias

A recommendation system generally begins developing biases towards well-known items, causing long tails to be neglected. This means a decrease in diversity in recommendations.

Evaluation Complexity

The recommendation models generally fail to have specific classification problems when it comes to evaluation metrics like Precision@K, Recall@K, and Mean Average Precision (MAP), which requires detailed attention.

Data imbalance

A few users and/or products account for the majority of interactions. While all these facts must always be captured, this imbalance must also be dealt with to ensure fairness with recommendations.

Challenges with Real-Time Deployment

A recommendation system is supposed to be deployed as an API for real-time user interactions, which would necessitate low-latency predictions and retrieval by filtering.

Key Learnings from the Project

A number of important takeaways and lessons were learned as a result of handling the various challenges above:

Effective Data Preprocessing is Critical:

It can significantly improve the performance of recommendation models by processing missing values, filtering sparse interactions, and normalizing data.

ALS is Powerful but Requires Optimization:

PySpark's ALS is a very good algorithm for collaborative filtering, but it is challenging to optimize it for good results with respect to rank, iterations, and regularization factor hyperparameters.

Hybrid Techniques Enhance Recommendations :

The combined effects or benefits of user-based and item-based collaborative filtering with content-based measures transform into the cold short problems in recommendation diversity.

Parallel Processing with PySpark Enhanced Scalability:

Using PySpark to leverage distributed computing capabilities in parallel processes allowed for efficient processing of large amounts of recommendation data to enable real-time recommendations.

9.CONCLUSION AND FUTURE SCOPE:

Conclusion:

PySpark-based Product Recommendation System was employed to analyze user-product interactions and provide recommendations upon it. The loading of data for ratings_Beauty.csv was performed at the beginning of the project, followed by pre-processing operations such as missing value handling and duplicate record removal in order to obtain data quality. Exploratory data analysis was performed to recognize the structure of the data based on schema analysis and rating distribution. A pie chart diagram was created to illustrate user ratings of products, and it indicated user activity. Data were preprocessed and utilized to train the model wherein the Alternating Least Squares (ALS) algorithm, an effective collaborative filtering method, was used. The model was trained to learn about user preferences from their historical ratings and tested by making predictions of ratings for users.

After training, the model generated customized product recommendations through ranking the top fit products for each user. System performance was measured by using Root Mean Squared Error (RMSE), an aspect that verified predictive rating precision. Lower RMSE indicated better performance, verifying the capability of the model to make proper recommendations. Findings demonstrated that the system had the capability to select proper product recommendations effectively, enhancing the experience of the user in an e-commerce setting. Visualizing rating distributions also assisted in interpreting user preferences, and potential future enhancements might involve hyperparameter tuning, such as including more product metadata, and experimenting with various recommendation algorithms for precision.

Future Scope:

Future Enhancement Prospects for any Product Recommendatory System based on Collaborative Filtering using PySpark

Real-Time Personalization- Enhancing the recommendation engine toward real-time dynamic suggestion generation based on users' latest behavior to uplift engagement and conversion rates.

Hybrid Filtering Models- Mixing-based filtering with collaborative filtering to provide better suggestions to new users for new items concerning the problems of cold-start.

Graph Recommendations- Employing Graph Neural Networks (GNNs) that model the complex relationships existing between users and items to capture deeper interactions rather than only classical matrix factorization ones.

Computing Contextual Recommendations- Incorporating time-sensitive factors, location-based factors, and device-based factors for personalizing recommendations by user contexts (e.g., time of day, browsing patterns).

Reinforced Recommendation- A recommendation system based upon reinforcement learning ensures that recommendations are adjusted on-the-fly based on accumulating user feedback and long-term user engagement.

Scale-up-Optimizing Distributed Computing Methods in PySpark for Effective Handling of Very Large Datasets in Cloud-Based Environments such as AWS or GCP. Explainable AI in Recommendations- Development of interpretable recommendation models that are framed in such a way that they give users insights into why a certain product is recommended, thus building their trust and ease of use.

Recommendation Bias and Fairness: Algorithmic bias should be corrected to ensure diversity and fairness in recommendations and not over-recommending popular items.

Integration with AR/VR Shopping: Augmenting the recommendation engine with AR and VR capabilities to create an engaging shopping experience and allow for immersive recommendations.

Cross Platform Synchronization: Development of a multi-device recommendation system that offers seamless experience across web, mobile, and smart assistant interfaces.

References:

1. Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, Large-scale parallel collaborative filtering for the Netflix prize, Berlin, Heidelberg, Springer-Verlag, In AAIM '08, pages 337– 348, 2008.
2. Sp Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, Technical Report UCB/EECS-2011-82, EECS Department, University of California, Berkeley, 2011
3. Michael Armbrust, et al, "Spark SQL: Relational Data Processing in Spark", in Proceedings of Association for Computing Machinery, Inc. ACM 978-1-4503-2758, Pages 1383-1394, May 27, 2015
4. M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. Pages 987-994 In SIGMOD, 2009.
5. Z.-D. Zhao and M.-S. Shang, "User-based collaborative-filtering recommendation algorithms on hadoop," in Knowledge Discovery and Data Mining, 2010. WKDD'10. Third International Conference on. IEEE,2010, pp. 478–481.
6. S. Golder and B. A. Huberman. The structure of collaborative tagging systems. Journal of Information Science, vol. 32 no. 2 198-208, April, 2006