DSM010 - Coursework 2: Developing a Recommendation System using Spark and Hadoop

By

Hendrik Matthys van Rooyen

University of London

September 2024

CONTENTS

CHAPTER 1 INTRODUCTION

Recommender systems are vital tools in personalized content delivery. These systems help to predict a user's preferences and recommend items such as books, movies, or products. This project aims to explore different implementations of a recommender system using Spark and Hadoop technologies. The dataset for this project is based on book ratings, consisting of three main tables: ratings, books, and users. The project's goal is to experiment with several approaches, measure their accuracy using RMSE (Root Mean Square Error), and compare collaborative, content-based, and hybrid methods. The primary focus will be on optimizing predictions for user ratings and evaluating the performance of different models.

1.1 PROBLEM STATEMENT

The objective of this research is to develop and compare multiple recommender systems based on the book ratings dataset. The challenge is to implement collaborative filtering, content-based filtering, and hybrid models using Spark and Hadoop frameworks and determine their predictive accuracy by calculating the RMSE for each method. The comparison of RMSE between the predicted rating and the user's actual rating will help identify the most effective method for recommendation.

1.2 DATASET DESCRIPTION

The dataset (Book Recommendation Dataset, Möbius) comprises three tables: ratings, books, and users, representing the interactions between users and books. These tables will serve as the foundation for building the recommendation models.

1. **Ratings Table**:
   - **Columns**: User-ID, ISBN, Book-Rating
   - **Purpose**: Provides the rating a user has assigned to a book.
2. **Books Table**:
   - **Columns**: ISBN, Book-Title, Book-Author, Year-Of-Publication, Publisher
   - **Purpose**: Contains metadata about the books being rated.
3. **Users Table**:
   - **Columns**: User-ID, Location, Age
   - **Purpose**: Provides demographic data for users.

1.3 RESEARCH OBJECTIVES

1. **Baseline RMSE**: Establish a baseline RMSE by implementing a simple model to predict user ratings and measure its performance.
2. **Collaborative Filtering (ALS)**: Develop a model using the Alternating Least Squares (ALS) algorithm to predict user ratings based on the historical rating patterns of similar users.
3. **Content-Based Filtering (KMeans)**: Create a content-based model using clustering methods to group similar books and users, then use these clusters to predict ratings.
4. **Hybrid Models**: Implement hybrid approaches combining ALS with content-based features (book metadata and user demographics) and evaluate the resulting models.

1.4 METHODOLOGY

The project will be divided into the following phases:

1.4.1 DATA PREPROCESSING AND EXPLORATION

The first step is to preprocess the dataset, ensuring the data is clean and structured for further analysis. This will involve handling missing values, filtering outliers, and encoding categorical features where necessary. Each of the three tables—ratings, books, and users—will undergo exploratory analysis to better understand their distributions and correlations.

- **Data Preprocessing Tasks**:
    - Remove null or invalid entries in the ratings, books, and users tables.
    - Standardize user age categories and filter out unusual age values.
    - Ensure the integrity of book ISBNs across the tables.
    - Normalize book titles and publishers for consistent representation.

1.4.2 BASELINE MODEL AND RMSE CALCULATION

To measure the effectiveness of more advanced methods, a simple baseline model will be established. This model will predict a random rating for all books in the dataset and use this prediction for every user-book pair. The RMSE of these predictions will serve as the baseline for comparing more sophisticated models.

- **Baseline RMSE Formula**:

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

Where (y i) is the actual rating, and (y-hat i) is the predicted rating.

1.4.3 COLLABORATIVE FILTERING WITH ALS

Collaborative filtering will be implemented using the Alternating Least Squares (ALS) algorithm. This method will rely solely on the ratings table to predict the interaction between users and books by factoring in the preferences of similar users.

- **ALS Method**:
    - Train the model using the ratings data.
    - Predict ratings for user-book pairs not present in the training set.
    - Compute the RMSE by comparing predicted ratings to actual ratings.
- **Evaluation**: The RMSE for ALS will be calculated and compared to the baseline model.

1.4.4 CONTENT-BASED FILTERING WITH KMEANS CLUSTERING

The content-based model will leverage the metadata from the books and users tables. The goal is to group similar books and users into clusters and predict ratings based on these clusters. KMeans clustering will be used to segment books into 10 clusters and users into 10 clusters.

- **Book Clustering**: Books will be clustered based on features like the title, author, year of publication, and publisher.
- **User Clustering**: Users will be grouped based on their location and age.
- **Rating Prediction**: Once clusters are established, predictions will be based on the average rating within each cluster.
- **Evaluation**: The RMSE for the content-based model will be calculated and compared to both the baseline and ALS models.

1.4.5 HYBRID RECOMMENDER SYSTEMS

The final phase of the project will involve developing hybrid recommender systems that combine collaborative filtering (ALS) with content-based methods (book and user features).

- **Hybrid Model 1: ALS + Books**: Combine the ALS algorithm with features from the books table to improve recommendations.
- **Hybrid Model 2: Books + Users**: Combine the content-based features of both books and users to improve prediction accuracy.
- **Hybrid Model 3: ALS + Books + Users**: Develop a comprehensive hybrid model incorporating collaborative filtering, book metadata, and user demographic features.

Each hybrid model will be evaluated by calculating its RMSE and comparing it to the previously implemented models.

1.5 EXPECTED OUTCOMES

The expected results from this project are as follows:

1. **Baseline RMSE**: A simple model will establish a benchmark against which more advanced methods will be compared.
2. **Collaborative Filtering (ALS)**: This method is expected to outperform the baseline by leveraging user-to-user similarities in the dataset.
3. **Content-Based Filtering (KMeans)**: Clustering books and users based on their features may improve recommendations, especially for new or infrequently rated books.
4. **Hybrid Models**: The combination of ALS with content-based features is anticipated to produce the best results, as it integrates both user behavior and additional information about books and users.

1.6 TOOLS AND TECHNOLOGIES

- **Apache Spark**: Spark will be used for implementing the collaborative filtering (ALS) and clustering (KMeans) algorithms. Its in-memory processing capability makes it well-suited for handling large datasets.
- **Hadoop**: Hadoop's distributed storage system (HDFS) will provide the necessary infrastructure for data storage and processing.
- **PySpark**: PySpark will be used as the interface for interacting with Spark, allowing for the efficient execution of machine learning algorithms.
- **Python Libraries**: Additional Python libraries such as Pandas and Matplotlib will be employed for data preprocessing, analysis, and visualization.

CHAPTER 2 THE PROJECT

2.1 PROJECT STRUCTURE

1. **Data Loading and Preprocessing**:
   - The DataLoader class is responsible for loading the required datasets from CSV files into PySpark DataFrames. It handles books, users, and ratings data, and performs necessary preprocessing like column selection and renaming.
2. **Model Training**:
   - **Baseline Model**: A simple baseline model implemented in BaselineTrainer randomly assigns ratings to books for evaluation purposes. This model serves as a benchmark to compare more advanced models.
   - **ALS Model**: The ALSTrainer class trains a collaborative filtering model using the Alternating Least Squares (ALS) algorithm. This model predicts ratings based on user-item interactions, relying on a user-item matrix.
   - **Content-Based Model**: The ContentBasedTrainer class clusters users and books based on features like age, location (for users), and title, author, and publisher (for books). KMeans clustering is used to group similar users and books, which can then be used to make recommendations.
   - **Hybrid Model**: The HybridTrainer class integrates predictions from both the ALS and content-based models. It averages the predictions from both models to create a final recommendation score for each user-book pair.
3. **Model Evaluation**:
   - All models are evaluated using the Root Mean Squared Error (RMSE), a common metric in regression tasks, to assess their accuracy.
     The test() methods in each model class generate predictions and compare them to the actual ratings to compute RMSE. This evaluation allows the comparison of different models, including the baseline, ALS, content-based, and hybrid models.
4. **Concurrency in Model Training**:
   - The script run_trainers_concurrently.py utilizes Python's threading to train and evaluate the models concurrently. This improves efficiency by leveraging multiple threads to run the baseline, ALS, and content-based models in parallel. Once the models are trained, the script also runs the evaluation phase concurrently.

2.2 DEPLOYMENT WORKFLOW

The deployment process consists of several scripts to upload data and code, and run the Spark jobs on a distributed system (Hadoop and Spark cluster).

1. **upload_data.bat**:
   - This script automates the process of uploading the datasets to an HDFS (Hadoop Distributed File System) via an SSH connection. It first uploads the

local data to the remote SSH server and then moves the data from the SSH server to the HDFS. The script ensures that any existing data is removed from both the SSH server and HDFS to avoid conflicts before uploading the new data.

2. **upload_spark.bat**:
   - This script uploads the Spark job files (Python scripts) to the SSH server. It deletes any existing Spark scripts on the server and replaces them with the latest versions from the local machine. This ensures that the most up-to-date code is deployed on the server before execution.
3. **run_spark.bat**:
   - This script initiates the Spark job on the remote server via SSH. It runs the spark_runner.py script, which contains the logic to execute the recommendation system using the trained models. The script ensures that the Spark job is run on the distributed Spark cluster.

2.3 HOW IT FUNCTIONS

1. **Data Upload**:
   - The project begins with data upload via the upload_data.bat script. This script moves the datasets to the SSH server and then into the HDFS where Spark can access them.
2. **Code Upload**:
   - The next step involves uploading the Spark scripts to the SSH server using upload_spark.bat. This ensures that the remote environment has the most up-to-date Python scripts required for running the recommendation system.
3. **Running the Recommendation System**:
   - Once the data and code are uploaded, the run_spark.bat script is executed. It runs the recommendation system on the Spark cluster, utilizing the ALS, content-based, and hybrid models. Each model is trained and evaluated, with results displayed after completion.
4. **Model Training and Evaluation**:
   - The system concurrently trains the ALS, content-based, and baseline models using Python threading. After training, it evaluates the models by predicting user ratings and comparing them to the actual ratings using RMSE. The hybrid model combines predictions from the ALS and content-based models, averaging them to provide more robust recommendations.

2.4 KEY COMPONENTS OF THE RECOMMENDATION SYSTEM

- **ALS (Collaborative Filtering)**: Learns latent factors for users and books, predicting ratings based on historical interactions.
- **Content-Based Filtering**: Clusters users and books based on their features and provides recommendations by finding similarities within clusters.

- **Hybrid Model**: Combines both ALS and content-based predictions to improve recommendation accuracy, addressing the limitations of each individual model.
- **Threading**: Utilizes Python’s threading to run training and evaluation processes in parallel, optimizing performance in a distributed environment.

CHAPTER 3 SUMMARY AND CONCLUSIONS

In this project, the objective was to explore and compare different implementations of a recommender system using Spark and Hadoop technologies, specifically targeting the book ratings dataset. The system aimed to predict user ratings for books and evaluate the performance of these predictions by calculating the Root Mean Square Error (RMSE). I employed several models, including a random baseline model, collaborative filtering using ALS (Alternating Least Squares), content-based filtering through KMeans clustering, and various hybrid approaches combining collaborative and content-based features.

3.1 GOALS AND CHALLENGES

The primary goal of the project was to create a robust recommender system that could accurately predict user preferences for books. The accuracy of these predictions was assessed using the RMSE, which provided a quantitative measure of how well the models performed compared to the actual user ratings. The secondary goal was to compare different techniques to identify which method yielded the most reliable and accurate predictions.

Key challenges faced during the project included the following:

- **Data Complexity**: The book ratings dataset contains three interrelated tables—ratings, books, and users. Preprocessing these data tables required cleaning, transformation, and joining them to form a coherent input for the models.
- **Model Diversity**: Implementing and evaluating multiple models—collaborative filtering, content-based clustering, and hybrid approaches—required ensuring that each model could be effectively trained and tested within the same framework.
- **Parallel Processing**: With a large dataset, efficient computation was critical. The use of Spark and Hadoop technologies allowed for distributed data storage and processing, but integrating and synchronizing various tasks (such as model training and testing) presented additional complexity.

3.2 RESULTS OBTAINED

The following RMSE values were calculated for each model:

- **Random Baseline Model**: RMSE = 5.2621
- **Collaborative Filtering (ALS Model)**: RMSE = 3.9582
- **Content-Based Model (Books + Users)**: RMSE = 4.4009
- **Hybrid Model (ALS + Books)**: RMSE = 4.1523
- **Hybrid Model (ALS + Books + Users)**: RMSE = 4.1381

3.3 INTERPRETATION OF RESULTS

1. **Random Baseline Model**: The baseline model predicted ratings based on the overall average rating in the dataset, regardless of individual user preferences or book features. This resulted in a relatively high RMSE of 5.2621, indicating poor predictive power. As expected, this model serves as a benchmark against which the more complex models can be evaluated.

2. **Collaborative Filtering (ALS)**: The ALS model, which leverages user-to-user and item-to-item similarities, significantly improved the RMSE to 3.9582. This improvement demonstrated that ALS is a strong method for predicting user ratings based on historical interaction data alone.
3. **Content-Based Filtering**: Using KMeans clustering to group books and users based on their features (such as titles, authors, and demographic information), the content-based model achieved an RMSE of 4.4009. While this approach did improve over the baseline, it did not outperform the collaborative filtering model. This suggests that relying solely on metadata about books and users is less effective than using historical rating data when predicting user preferences.
4. **Hybrid Models**: The hybrid models combined collaborative filtering (ALS) with content-based features to improve accuracy. The **ALS + Books** hybrid achieved an RMSE of 4.1523, while the **ALS + Books + Users** hybrid further improved the RMSE to 4.1381. Although the improvement over the ALS model was modest, these results demonstrate that hybrid systems can provide a slight advantage by incorporating additional content-based information.

3.4 DISCUSSION

The results indicate that collaborative filtering using ALS is the most effective approach for this dataset, achieving the lowest RMSE. The success of the ALS model highlights the importance of leveraging user-item interaction data when predicting preferences in recommender systems. The content-based model, while helpful, struggled to outperform ALS, likely due to the limited metadata available for clustering books and users. This suggests that book metadata (such as title, author, and publication year) and user demographics (such as age and location) alone are insufficient for accurate predictions.

The hybrid models, although more complex, only slightly improved upon the ALS results. The integration of content-based information (books and users) did not significantly lower the RMSE, implying that the additional metadata provided limited new insights beyond what the collaborative filtering model had already captured. This may suggest that, for this dataset, user behavior (as captured in the ratings table) is a more reliable predictor than the content attributes of books or the demographics of users.

3.5 CONCLUSION

This project successfully implemented and compared multiple approaches to building a recommender system using Spark and Hadoop technologies. The evaluation of the models using RMSE showed that collaborative filtering (ALS) performed the best, followed by hybrid models that incorporated content-based features. The random baseline model, as expected, performed the worst.

The results indicate that for book recommendation systems, collaborative filtering provides the most accurate predictions. The use of hybrid models offered only marginal improvement, suggesting that the book metadata and user demographics did not contribute significantly to the predictive power of the system in this case. These findings can be applied to other domains with similar datasets, helping to guide the choice of algorithms in recommender system development.

In the future, further refinements to the hybrid models, such as more sophisticated feature engineering or deeper integration of content-based and collaborative filtering methods, could potentially yield greater improvements in accuracy. Additionally, experimenting with other clustering algorithms or incorporating more granular book metadata (e.g., genres or themes) might enhance the effectiveness of content-based filtering approaches.

CHAPTER 4 BIBLIOGRAPHY

ALS — PySpark 3.5.2 documentation (no date). Available at: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.mllib.recommendation.ALS.html (Accessed: 10 September 2024).

Book Recommendation Dataset (no date). Available at: https://www.kaggle.com/datasets/arashnic/book-recommendation-dataset (Accessed: 29 August 2024).

Casalegno, F. (2022) *Recommender Systems — A Complete Guide to Machine Learning Models*, *Medium*. Available at: https://towardsdatascience.com/recommender-systems-a-complete-guide-to-machine-learning-models-96d3f94ea748 (Accessed: 28 August 2024).

Desai, U. (2024) 'Recommendation Systems Explained: Understanding the Basic to Advance', *Medium*, 4 January. Available at: https://utsavdesai26.medium.com/recommendation-systems-explained-understanding-the-basic-to-advance-43a5fce77c47 (Accessed: 28 August 2024).

pyspark.sql.SparkSession — PySpark 3.5.2 documentation (no date). Available at: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.SparkSession.html (Accessed: 31 August 2024).

RegressionEvaluator — PySpark 3.5.2 documentation (no date). Available at: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.evaluation.RegressionEvaluator.html (Accessed: 10 September 2024).

CHAPTER 5 ADDENDUM A: CODE

upload_data.bat

```
@echo off
set SSH_PATH=Coursework2
set HDFS_PATH=Coursework2
set hadoop=/opt/hadoop/current/bin/hadoop

:: Delete "data" folder on SSH server if they exist
echo "Removing existing folders on SSH server if they exist..."
ssh -i id_rsa hrooy001@lena.doc.gold.ac.uk "rm -rf %SSH_PATH%/data"

:: Upload "data" folder to SSH server
echo "Uploading data folder to SSH server..."
scp -i id_rsa -r data hrooy001@lena.doc.gold.ac.uk:%SSH_PATH%/

:: Check if HDFS directory exists, if not create it
echo "Checking if HDFS directory exists..."
ssh -i id_rsa hrooy001@lena.doc.gold.ac.uk %hadoop% dfs -mkdir %HDFS_PATH%
ssh -i id_rsa hrooy001@lena.doc.gold.ac.uk %hadoop% dfs -mkdir %HDFS_PATH%/data
ssh -i id_rsa hrooy001@lena.doc.gold.ac.uk %hadoop% dfs -mkdir %HDFS_PATH%/output

:: Delete "data" folder from HDFS if it exists
echo "Removing existing data folder from HDFS if it exists..."
ssh -i id_rsa hrooy001@lena.doc.gold.ac.uk %hadoop% "dfs -rm -r -f %HDFS_PATH%/data"

:: Upload "data" folder from SSH to HDFS
echo "Uploading data folder from SSH server to HDFS..."
ssh -i id_rsa hrooy001@lena.doc.gold.ac.uk %hadoop% "dfs -put %SSH_PATH%/data %HDFS_PATH%/"

echo "Operation completed!"
pause
```

upload_spark.bat

```
@echo off
set SSH_PATH=Coursework2
set HDFS_PATH=Coursework2
set hadoop=/opt/hadoop/current/bin/hadoop

:: Delete "spark" folder on SSH server if they exist
echo "Removing existing folders on SSH server if they exist..."
ssh -i id_rsa hrooy001@lena.doc.gold.ac.uk "rm -rf %SSH_PATH%/spark"

:: Upload "spark" folder to SSH server
echo "Uploading spark folder to SSH server..."
scp -i id_rsa -r spark hrooy001@lena.doc.gold.ac.uk:%SSH_PATH%/

echo "Operation completed!"
pause
```

run_spark.bat

```
@echo off
set SSH_PATH=Coursework2
set spark=/opt/spark/current/bin/spark-submit

echo "Running Spark job..."
ssh -i id_rsa hrooy001@lena.doc.gold.ac.uk %spark% %SSH_PATH%/spark/spark_runner.py

echo "Operation completed!"
pause

```

spark\spark_runner.py

```python
# Import threading and necessary PySpark modules
import threading
from pyspark.sql import SparkSession
from data_loader import DataLoader
from als_trainer import ALSTrainer
from baseline_trainer import BaselineTrainer
from content_based_trainer import ContentBasedTrainer
from hybrid_trainer import HybridTrainer

def run_trainers_concurrently(data_dir):
    """
    Runs the training and testing processes for various recommendation models concurrently using threads.

    Args:
        data_dir (str): Directory where the dataset files are located (HDFS path).
    """

    # Create Spark session
    spark = SparkSession.builder.appName("RecommendationSystem↩
").master("yarn").getOrCreate()
    spark.sparkContext.setLogLevel("ERROR")  # Set log level to minimize Spark logs

    # Load data using the DataLoader class
    data_loader = DataLoader(spark, data_dir)
    ratings, books, users = data_loader.load_data()

    # Initialize the trainers for different models
    baseline_trainer = BaselineTrainer(spark, ratings)
    als_trainer = ALSTrainer(spark, ratings)
    content_based_trainer = ContentBasedTrainer(spark, books, ratings, users)
    hybrid_trainer = HybridTrainer(spark, ratings, books, users, als_trainer,
content_based_trainer)

    # Create threads for training the models
    baseline_train_thread = threading.Thread(target=baseline_trainer.train)
    als_train_thread = threading.Thread(target=als_trainer.train)
    content_based_train_thread = threading.Thread(target=content_based_trainer.train)

    # Start the training threads
    baseline_train_thread.start()
    als_train_thread.start()
    content_based_train_thread.start()

    # Wait for all training threads to complete
    baseline_train_thread.join()
    als_train_thread.join()
    content_based_train_thread.join()

    # Create threads for testing the models
    baseline_test_thread = threading.Thread(target=baseline_trainer.test)
    als_test_thread = threading.Thread(target=als_trainer.test)
    content_based_test_thread = threading.Thread(target=content_based_trainer.test)
    hybrid_test_thread = threading.Thread(target=hybrid_trainer.test)

    # Start the testing threads
    baseline_test_thread.start()
    als_test_thread.start()
    content_based_test_thread.start()
    hybrid_test_thread.start()

    # Wait for all testing threads to complete
    baseline_test_thread.join()
    als_test_thread.join()
    content_based_test_thread.join()
    hybrid_test_thread.join()


# Main execution entry point
if __name__ == "__main__":
    # Specify the HDFS data directory
    data_dir = "hdfs:///user/hrooy001/Coursework2/data"

    # Run the trainers concurrently
    run_trainers_concurrently(data_dir)

```

spark\data_loader.py

```python
# Import necessary PySpark modules
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

class DataLoader:
    """
    A class to handle loading and processing of books, ratings, and users datasets
    from CSV files using PySpark DataFrames.

    Attributes:
        spark (SparkSession): Spark session used to interact with the datasets.
        data_dir (str): Path to the directory containing the CSV files.
    """

    def __init__(self, spark, data_dir):
        """
        Initializes the DataLoader class with a Spark session and the data directory.

        Args:
            spark (SparkSession): Spark session for interacting with the data.
            data_dir (str): Path to the directory where CSV files are stored.
        """
        self.spark = spark
        self.data_dir = data_dir

    def load_data(self):
        """
        Loads and processes the Books, Ratings, and Users datasets from CSV files.
        Returns PySpark DataFrames for each dataset, with relevant columns selected
        and renamed for ease of use.

        Returns:
            tuple: A tuple containing DataFrames for ratings, books, and users.
        """

        # Load the CSV files as DataFrames
        books_df = self.spark.read.csv(f"{self.data_dir}/Books.csv", header=True, inferSchema=True)
        ratings_df = self.spark.read.csv(f"{self.data_dir}/Ratings.csv", header=True, inferSchema=True)
        users_df = self.spark.read.csv(f"{self.data_dir}/Users.csv", header=True, inferSchema=True)

        # Select and rename relevant columns from the Ratings dataset
        ratings = ratings_df.select(
            col("User-ID").alias("userId"),
            col("ISBN").alias("isbn"),
            col("Book-Rating").alias("rating")
        )

        # Select and rename relevant columns from the Books dataset
        books = books_df.select(
            col("ISBN").alias("isbn"),
            col("Book-Title").alias("title"),
            col("Book-Author").alias("author"),
            col("Year-Of-Publication").alias("year"),
            col("Publisher").alias("publisher")
        )

        # Select and rename relevant columns from the Users dataset
        users = users_df.select(
            col("User-ID").alias("userId"),
            col("Location").alias("location"),
            col("Age").alias("age")
        )

        # Return the processed DataFrames for ratings, books, and users
        return ratings, books, users

```

spark\baseline_trainer.py

```python
# Import necessary PySpark modules and functions
from pyspark.sql.functions import udf
from pyspark.sql.types import DoubleType
from pyspark.ml.feature import StringIndexer
from pyspark.ml.evaluation import RegressionEvaluator
import random

class BaselineTrainer:
    """
    A class to train and evaluate a random baseline model for book rating predictions.

    Attributes:
        spark (SparkSession): Spark session used to interact with the datasets.
        ratings (DataFrame): The ratings dataset used for training and testing.
    """

    def __init__(self, spark, ratings):
        """
        Initializes the BaselineTrainer class with a Spark session and the ratings dataset.

        Args:
            spark (SparkSession): Spark session for interacting with the data.
            ratings (DataFrame): The ratings dataset to be used in model training.
        """
        self.spark = spark
        self.ratings = ratings

    def train(self):
        """
        Trains a random baseline model by assigning random ratings to the test dataset.
        The 'isbn' column is indexed and split into training and test sets.
        """

        # Convert 'isbn' column to a numeric index for easier processing
        isbn_indexer = StringIndexer(inputCol="isbn", outputCol="isbnIndex")
        ratings_indexed = isbn_indexer.fit(self.ratings).transform(self.ratings)

        # Split the data into training and test sets (80% training, 20% testing)
        _, test = ratings_indexed.randomSplit([0.8, 0.2])

        # Define a UDF (User Defined Function) to generate random ratings between 0 and 10
        random_rating_udf = udf(lambda: random.uniform(0, 10), DoubleType())

        # Apply random predictions to the test set using the UDF
        self.random_predictions = test.withColumn("prediction", random_rating_udf())

    def test(self):
        """
        Evaluates the random baseline model using Root Mean Squared Error (RMSE).

        Returns:
            float: The RMSE value for the random baseline model.
        """

        # Use RMSE as the evaluation metric
        evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")

        # Calculate and print the RMSE for the random predictions
        random_rmse = evaluator.evaluate(self.random_predictions)
        print(f"RMSE for Random Baseline: {random_rmse}")

        return random_rmse

```

spark\content_based_trainer.py

```python
# Import necessary PySpark modules for clustering, evaluation, and feature engineering
from pyspark.ml.feature import CountVectorizer, StringIndexer, VectorAssembler
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.sql.functions import udf, col, when, max as spark_max, min as spark_min
from pyspark.sql.types import DoubleType
from pyspark.ml.linalg import Vectors
from pyspark.sql import functions as F

class ContentBasedTrainer:
    """
    A class to train and evaluate a content-based recommendation system by clustering users and books,
    and then making predictions based on the distances from cluster centers.

    Attributes:
        spark (SparkSession): Spark session used to interact with the datasets.
        books_df (DataFrame): DataFrame containing book information.
        ratings (DataFrame): DataFrame containing user ratings for books.
        users (DataFrame): DataFrame containing user information.
        seed (int, optional): Seed value for reproducibility. Default is None.
    """

    def __init__(self, spark, books, ratings, users, seed=None):
        """
        Initializes the ContentBasedTrainer class with a Spark session, books, ratings, users datasets, and an optional seed.

        Args:
            spark (SparkSession): Spark session for interacting with the data.
            books (DataFrame): The books dataset used for clustering.
            ratings (DataFrame): The ratings dataset.
            users (DataFrame): The users dataset.
            seed (int, optional): Seed for reproducibility. Default is None.
        """
        self.spark = spark
        self.books_df = books
        self.ratings = ratings
        self.users = users
        self.seed = seed

    def train(self):
        """
        Trains the user and book clusters using KMeans clustering.
        """
        self.train_user_clusters()
        self.train_book_clusters()

    def train_user_clusters(self):
        """
        Clusters users based on age and location using KMeans.
        """

        # Prepare user data and fill missing values
        user_features = self.users.select("userId", "age", "location")
        user_features = user_features.withColumn("age", user_features["age"].cast(DoubleType()))
        user_features = user_features.na.fill({"age": 30, "location": "unknown"})  # Handle missing values

        # Index categorical features (location)
        indexer_location = StringIndexer(inputCol="location", outputCol="locationIndex")
        user_features_indexed = indexer_location.fit(user_features).transform(user_features)

        # Assemble features for clustering
        assembler = VectorAssembler(inputCols=["age", "locationIndex"], outputCol="features")
        users_assembled = assembler.transform(user_features_indexed)

        # Apply KMeans clustering to users
        kmeans_users = KMeans(k=10, seed=self.seed)
        self.model_users = kmeans_users.fit(users_assembled.select("features"))

        # Add cluster assignments to the DataFrame
        self.user_clusters = self.model_users.transform(users_assembled)

        # Calculate distances of users from their cluster centers
        cluster_centers = self.model_users.clusterCenters()

        def calculate_distance(features, prediction):
            cluster_center = cluster_centers[int(prediction)]
            return float(Vectors.squared_distance(features, cluster_center))

        distance_udf = udf(calculate_distance, DoubleType())
        self.user_clusters = self.user_clusters \
            .withColumn("distance_from_center", distance_udf(self.user_clusters["features"], self.user_clusters["prediction"]))

        print('Trained user clusters')

    def train_book_clusters(self):
        """
        Clusters books based on title, author, and publisher using KMeans.
        """

        # Prepare book data and remove rows with missing values in key features
        book_features = self.books_df.select("isbn", "title", "author", "publisher")
        book_features = book_features.na.drop(subset=["title", "author", "publisher"])

        # Index categorical features (title, author, publisher)
        indexer_title = StringIndexer(inputCol="title", outputCol="titleIndex")
        indexer_author = StringIndexer(inputCol="author", outputCol="authorIndex")
        indexer_publisher = StringIndexer(inputCol="publisher", outputCol="publisherIndex")

        # Apply indexers and assemble features
        books_indexed = indexer_title.fit(book_features).transform(book_features)
        books_indexed = indexer_author.fit(books_indexed).transform(books_indexed)
        books_indexed = indexer_publisher.fit(books_indexed).transform(books_indexed)

        # Assemble features for clustering
        assembler = VectorAssembler(inputCols=["titleIndex", "authorIndex", "publisherIndex"], outputCol="features")
        books_assembled = assembler.transform(books_indexed)

        # Apply KMeans clustering to books
        kmeans_books = KMeans(k=10, seed=self.seed)
        self.model_books = kmeans_books.fit(books_assembled.select("features"))

        # Add cluster assignments to the DataFrame
        self.book_clusters = self.model_books.transform(books_assembled)

        # Calculate distances of books from their cluster centers
        cluster_centers = self.model_books.clusterCenters()

        def calculate_distance(features, prediction):
            cluster_center = cluster_centers[int(prediction)]
            return float(Vectors.squared_distance(features, cluster_center))

        distance_udf = udf(calculate_distance, DoubleType())
        self.book_clusters = self.book_clusters \
            .withColumn("distance_from_center", distance_udf(self.book_clusters["features"], self.book_clusters["prediction"]))

        print('Trained book clusters')

    def recommend_books(self, user_id):
        """
        Recommends books for a given user based on the user's cluster and ratings from similar users.

        Args:
            user_id (int): The ID of the user for whom book recommendations are generated.

        Returns:
            DataFrame: A DataFrame containing top 10 recommended books for the user.
        """

        # Find the user's cluster
        user_cluster = self.user_clusters.filter(self.user_clusters.userId == user_id).select("prediction").collect()[0][0]

        # Find highly rated books from users in the same cluster
        similar_user_books = self.book_clusters.join(self.ratings, "isbn").join(self.user_clusters, "userId").filter(self.user_clusters.prediction == user_cluster)

        # Return top N book recommendations from similar users
        recommended_books = similar_user_books.groupBy("isbn").agg({"rating": "avg"}).orderBy("avg(rating)", ascending=False).limit(10)

        return recommended_books

    def test(self):
        """
        Evaluates the content-based recommendation system by calculating the RMSE for predictions.

        Returns:
            float: The RMSE value for the content-based model.
        """

        # Join test ratings with user and book clusters
        test_with_user_clusters = self.ratings.join(self.user_clusters, "userId", how="inner")
        test_with_clusters = test_with_user_clusters.join(
            self.book_clusters.withColumnRenamed("distance_from_center", "book_distance_from_center").withColumnRenamed("prediction", "book_prediction"),
            "isbn",
            how="inner"
        )

        # Calculate min and max distances for normalization
        distance_stats = test_with_clusters.agg(
            spark_max("distance_from_center").alias("max_user_distance"),
            spark_min("distance_from_center").alias("min_user_distance"),
            spark_max("book_distance_from_center").alias("max_book_distance"),
            spark_min("book_distance_from_center").alias("min_book_distance")
        ).first()

        max_user_distance = distance_stats["max_user_distance"]
        min_user_distance = distance_stats["min_user_distance"]
        max_book_distance = distance_stats["max_book_distance"]
        min_book_distance = distance_stats["min_book_distance"]

        # Normalize distances and calculate final prediction
        normalize_distance_expr = (
            (col("distance_from_center") - min_user_distance) / (max_user_distance - min_user_distance) * 10.0 +
            (col("book_distance_from_center") - min_book_distance) / (max_book_distance - min_book_distance) * 10.0
        ) / 2.0

        test_with_clusters = test_with_clusters.withColumn("final_prediction", normalize_distance_expr)

        # Evaluate the model using RMSE
        evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
predictionCol="final_prediction")
        rmse = evaluator.evaluate(test_with_clusters)

        print(f"RMSE for Content-Based Model: {rmse}")
        return rmse

```

spark\als_trainer.py

```python
# Import necessary PySpark modules for recommendation systems and evaluation
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.feature import StringIndexer

class ALSTrainer:
    """
    A class to train and evaluate an ALS (Alternating Least Squares) model for collaborative filtering.

    Attributes:
        spark (SparkSession): Spark session used to interact with the datasets.
        ratings (DataFrame): The ratings dataset used for training and testing.
        seed (int, optional): Seed value for reproducibility of results. Default is None.
    """

    def __init__(self, spark, ratings, seed=None):
        """
        Initializes the ALSTrainer class with a Spark session, ratings dataset, and an optional seed.

        Args:
            spark (SparkSession): Spark session for interacting with the data.
            ratings (DataFrame): The ratings dataset to be used for model training.
            seed (int, optional): Seed for reproducibility. Default is None.
        """
        self.spark = spark
        self.ratings = ratings
        self.seed = seed

    def train(self):
        """
        Trains the ALS model on the ratings dataset. The 'isbn' column is first indexed,
        and the data is split into training and test sets.
        """

        # Convert 'isbn' column to a numeric index for the ALS model
        isbn_indexer = StringIndexer(inputCol="isbn", outputCol="isbnIndex")
        ratings_indexed = isbn_indexer.fit(self.ratings).transform(self.ratings)

        # Split the data into training and test sets with an optional seed for reproducibility
        self.training, self.test_set = ratings_indexed.randomSplit([0.8, 0.2], seed=self.seed)

        # Train the ALS model with nonnegative ratings, specifying the seed for reproducibility
        als = ALS(
            userCol="userId",
            itemCol="isbnIndex",
            ratingCol="rating",
            coldStartStrategy="drop",  # Handle cold start issue by dropping NaN predictions
            nonnegative=True,  # Ensure no negative ratings
            maxIter=10,  # Number of iterations
            regParam=0.1,  # Regularization parameter to prevent overfitting
            rank=20,  # Number of latent factors
            seed=self.seed  # Seed for reproducibility
        )

        # Fit the ALS model on the training set
        self.model = als.fit(self.training)

    def test(self):
        """
        Evaluates the ALS model on the test set using Root Mean Squared Error (RMSE).

        Returns:
            float: The RMSE value for the ALS model.
        """

        # Generate predictions on the test set using the trained ALS model
        predictions = self.model.transform(self.test_set)

        # Use RMSE as the evaluation metric for model performance
        evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")

        # Calculate and print the RMSE value
        rmse = evaluator.evaluate(predictions)
        print(f"RMSE for ALS Model: {rmse}")

        return rmse

```

spark\hybrid_trainer.py

```python
# Import necessary PySpark modules for recommendation systems, clustering, and evaluation
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.sql.functions import col
from pyspark.sql import functions as F
from als_trainer import ALSTrainer
from content_based_trainer import ContentBasedTrainer

class HybridTrainer:
    """
    A class to train and evaluate a hybrid recommendation model combining collaborative filtering
    (using ALS) and content-based filtering (using KMeans clustering).

    Attributes:
        spark (SparkSession): Spark session used to interact with the datasets.
        ratings (DataFrame): The ratings dataset used for training and testing.
        books (DataFrame): The books dataset.
        users (DataFrame): The users dataset.
        als_trainer (ALSTrainer): The ALS trainer for collaborative filtering.
        content_trainer (ContentBasedTrainer): The content-based trainer for clustering.
        seed (int, optional): Seed value for reproducibility. Default is None.
    """

    def __init__(self, spark, ratings, books, users, als_trainer=None, content_trainer=None, seed=None):
        """
        Initializes the HybridTrainer class with Spark session, ratings, books, users, ALS trainer,
        content-based trainer, and optional seed for reproducibility.

        Args:
            spark (SparkSession): Spark session for interacting with the data.
            ratings (DataFrame): The ratings dataset.
            books (DataFrame): The books dataset.
            users (DataFrame): The users dataset.
            als_trainer (ALSTrainer, optional): An ALS trainer. Default is None.
            content_trainer (ContentBasedTrainer, optional): A content-based trainer. Default is None.
            seed (int, optional): Seed for reproducibility. Default is None.
        """
        self.spark = spark
        self.ratings = ratings
        self.books = books
        self.users = users
        self.seed = seed
        self.als_trainer = als_trainer
        self.content_trainer = content_trainer

    def train(self):
        """
        Trains both the ALS model (collaborative filtering) and the content-based model (KMeans clustering).
        """
        # Train ALS model for collaborative filtering
        self.als_trainer = ALSTrainer(self.spark, self.ratings, self.seed)
        self.als_trainer.train()

        # Train content-based model using KMeans clustering
        self.content_trainer = ContentBasedTrainer(self.spark, self.books, self.ratings, self.users, self.seed)
        self.content_trainer.train()

    def test(self):
        """
        Tests the hybrid recommendation system by combining ALS and content-based predictions and evaluating the model's performance.

        Returns:
            float: The RMSE value for the hybrid model.
        """

        # Generate ALS-based predictions
        als_predictions = self.als_trainer.model.transform(self.als_trainer.test_set)

        # Filter out the ratings test set
        ratings_test = self.ratings.filter(self.ratings["rating"].isNotNull())

        # Join the test set with user and book clusters from the content-based model
        test_with_user_clusters = ratings_test.join(self.content_trainer.user_clusters, "userId", how="inner")
        test_with_clusters = test_with_user_clusters.join(
            self.content_trainer.book_clusters.withColumnRenamed("distance_from_center", "book_distance_from_center")
                .withColumnRenamed("prediction", "book_prediction"),
            "isbn",
            how="inner"
        )

        # Join ALS predictions with the test set containing clusters from the content-based model
        combined_predictions = als_predictions.join(
            test_with_clusters,
            on=["userId", "isbn"],
            how="inner"
        ).select(
            "isbn",
            "userId",
            als_predictions["prediction"].alias("als_pred"),
            test_with_clusters["book_prediction"].alias("content_pred"),
            test_with_clusters["rating"]
        )

        # Get the min and max ratings for scaling predictions
        rating_stats = self.als_trainer.test_set.agg(
            F.max("rating").alias("max_rating"),
            F.min("rating").alias("min_rating")
        ).collect()[0]
        max_rating = rating_stats["max_rating"]
        min_rating = rating_stats["min_rating"]

        # Combine ALS and content-based predictions, and scale them for final predictions
        combined_predictions = combined_predictions.withColumn(
            "hybrid_prediction",
            (combined_predictions["als_pred"] + combined_predictions["content_pred"]) / 2
        ).withColumn(
            "hybrid_prediction_scaled",
            ((F.col("hybrid_prediction") - min_rating) / (max_rating - min_rating)) * 10
        )

        # Evaluate the hybrid model using RMSE
        evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
predictionCol="hybrid_prediction_scaled")
        hybrid_rmse = evaluator.evaluate(combined_predictions)

        print(f"RMSE for Hybrid Model: {hybrid_rmse}")
        return hybrid_rmse

```