

MovieLens Big Data Analysis for A Movie Recommendation System

Angel Ren, Bhavika Prasannakumar, Chinmaya Gayathri, Liya Li, Sai Srivathsav Aripirala

Department of Applied Data Science, San Jose State University

DATA 228: Big Data Technologies

Dr. Guannan Liu, Ph.D

December 6, 2023

Abstract

The exponential growth of digital content has led to an explosion of data, particularly in the field of movie recommendations. Leveraging the power of Pyspark and other sophisticated big data analysis techniques, this project aims to craft a dynamic movie recommendation system tailored for personalized user preferences by tackling the complexities of big data analysis and conducting comprehensive exploratory data analysis (EDA) to extract valuable insights from the rich MovieLens dataset from GroupLens which is a research lab in the Department of Computer Science and Engineering at the University of Minnesota.

The dataset comprises metadata for over 45,000 movies included in the Full MovieLens Dataset, all released on or before July 2017, along with over 26 million ratings data. It offers a comprehensive view of the film industry, including features like cast and crew information, plot keywords, budget, revenue, release dates, supported languages, production companies, countries of origin, and voting statistics.

To achieve the objectives outlined in the problem statement, we effectively addressed the following methods: for data processing and analysis, we managed and analyzed large-scale movie-related data using PySpark to gain insights into user preferences, movie ratings, and trends. This included data manipulation techniques such as filtering and aggregation. For visualization of insights, we created intuitive data visualizations, such as statistical tables, barplots, and word clouds, to present complex analytical results. This enabled stakeholders to make informed decisions based on the patterns and trends discovered. In the advanced phase, we improved user satisfaction by providing accurate popular movie predictions and relevant movie recommendations. As a result of the thorough EDA, we employed two recommendation

algorithms, namely Alternating Least Squares (ALS) and Singular Value Decomposition (SVD), to offer customized options based on user-provided information.

1. Introduction

This project, "MovieLens Big Data Analysis for A Movie Recommendation System" elaborates on the challenges and objectives in the evolving field of movie recommendations. It highlights the rapid growth of digital content, emphasizing the increased complexity and volume of data in this domain. The overall project target is to enhance user satisfaction by offering personalized suggestions based on shared movie characteristics and user preferences.

The project leverages advanced big data analysis techniques, particularly PySpark, to navigate these complexities. The primary goal is to develop a dynamic, personalized movie recommendation system that adapts to user preferences and evolving trends in viewer behavior. This involves a thorough exploratory data analysis of the comprehensive MovieLens dataset, which encompasses a wide array of movie metadata and ratings data, to glean valuable insights for accurate and relevant movie recommendations. The challenge lies in extracting valuable insights from this extensive dataset, encompassing a wide range of film industry features, and translating these insights into accurate and relevant movie recommendations. Thus, this project also involves the creation of intuitive data visualizations to present complex analytical results to help stakeholders to make informed decisions based on identified patterns and trends. The overall project target is to enhance user satisfaction by offering personalized suggestions based on shared movie characteristics and user preferences.

2. Data Process and EDA

2.1 Data Collection

The original movie data source is the MovieLens dataset for movies ratings and other information collected by the GroupLens research lab in the Department of Computer Science and Engineering at the University of Minnesota. The dataset we used is publicly available on Kaggle and it is ensemble with the GroupLens data and movie details, credits and keywords collected from the TMDB Open API.

Table 1: Original Dataset Basic Information

Data File	File Description	Dimension
movies_metadata.csv	Information on 45,000+ movies including release dates, budget, revenue, production countries, language, and etc.	(24, 45572)
ratings.csv	All ratings from 270k users on all movies.	(4, 26024289)
ratings_small.csv	Subset of ratings_csv for 100k+ ratings from 700 users on 9,000 movies.	(4, 100004)
keywords.csv	Movie plot keywords in a stringified JSON form.	(2, 46419)
links.csv	All movies' TMDB and IMDB IDs.	(3, 45843)
links_small.csv	Subset of links_csv for 9,000+ movies.	(3, 9125)

credits.csv	Cast and crew information on all movies in a stringified JSON form.	(3, 45476)
-------------	---	------------

2.2 Data Introduction and Materials Used

This project utilizes this pre-collected batch of data files including the following csv files. The _small datasets were also used as quick accesses to experiment with PySpark before jumping into the scalable analysis. There are about 26 millions of ratings from 270 thousands of users on a scale of 1-5.

For the programming language, we used Python and especially the PySpark for scalable programming. The seaborn library is mainly used for visualizations and in terms of PySpark, we used some of the pyspark.sql and pyspark.ml libraries and functions.

Figure 1: Python Libraries Used in This Project

```

import os
import sys
import random
from glob import glob

import pandas as pd
from pandas import DataFrame
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
import matplotlib
%matplotlib inline
from mpl_toolkits.mplot3d import Axes3D
import statistics as stats
from tabulate import tabulate
import seaborn as sns
from wordcloud import WordCloud, STOPWORDS

import warnings
warnings.filterwarnings('ignore')
warnings.filterwarnings("ignore", category=UserWarning, module="pyspark")

!pip3 install pyspark
from pyspark import SparkConf
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import ArrayType, StructType, StructField, StringType
from pyspark.sql.functions import explode, col, from_json, year, concat_ws
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
import ast

```

Requirement already satisfied: pyspark in /opt/anaconda3/lib/python3.9/site-packages (3.5.0)
Requirement already satisfied: py4j==0.10.9.7 in /opt/anaconda3/lib/python3.9/site-packages (from pyspark) (0.10.9.7)

For the hardwares, the team has used both Windows and macOS systems when collaborating for coding and documentations. For the softwares, GoogleDrive is the main location for data and documentation storage as it allows simultaneous teamwork. Aside from that, we used the paid GoogleColab Pro+ plan to help increase the GPU and RAM limits to accelerate processing and modeling speeds.

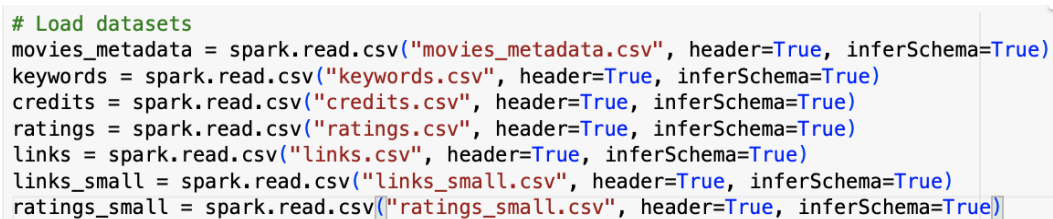
2.3 Spark Session Creation and Data Loading

Using Google Colab, we set the spark configuration to 8G spark executor memory, and then created a spark session called MovieRecommendation using the `SparkSession.builder.appName()` function and the object is named “spark”. Setting the logging level of “spark” to that only error messages will be displayed in the console logs is done by `spark.sparkContext.setLogLevel(“Error”)`. This is to reduce the amount of log output and focus

on important error messages, helping to improve the clarity of logs. When there are internal processing interruptions, this can be set to “Info” or “Debug” to assist development or debugging.

In the data loading section, we directly used the read.csv function to read in data files and named them the same as the file names. Passing “True” to the header indicates that the first row of the csv file contains header information which is important to the later information extraction. Passing “True” to the inferSchema tells Spark to automatically infer the data types of each column.

Figure 2: Dataset Loading PySpark Screenshot

A screenshot of a code editor showing PySpark code for loading datasets. The code is as follows:

```
# Load datasets
movies_metadata = spark.read.csv("movies_metadata.csv", header=True, inferSchema=True)
keywords = spark.read.csv("keywords.csv", header=True, inferSchema=True)
credits = spark.read.csv("credits.csv", header=True, inferSchema=True)
ratings = spark.read.csv("ratings.csv", header=True, inferSchema=True)
links = spark.read.csv("links.csv", header=True, inferSchema=True)
links_small = spark.read.csv("links_small.csv", header=True, inferSchema=True)
ratings_small = spark.read.csv("ratings_small.csv", header=True, inferSchema=True)
```

2.4 Original Data Statistics Preview

We created a function called Display_csv_info_spark() with input of the file path and filename to output the first 5 rows of the dataset and dimension as well as all column names. Below are the outputs for our main datasets movies_metadata and ratings. For more outputs displayed, please refer to the code file.

Figure 3: movies_metadata.csv Information Display Output Screenshot

```
display_csv_info_spark(movies_metadata, "movies_metadata.csv")

File: movies_metadata.csv
First 5 rows:
-----+-----
|adult|belongs_to_collection
-----+-----
|False|{'id': 10194, 'name': 'Toy Story Collection', 'poster_path': '/7G9915LfUQ2lVfwMEhDsn3kT4B.jpg', 'backdrop
|False|NULL
|False|{'id': 119050, 'name': 'Grumpy Old Men Collection', 'poster_path': '/nLvUdqqPgm3F85NMCii9gVFUcet.jpg', 'ba
|False|NULL
|False|{'id': 96871, 'name': 'Father of the Bride Collection', 'poster_path': '/nts4i0mNnq7GNicycMJ9pSAn204.jpg',
-----+-----
only showing top 5 rows

Number of rows: 45572
Number of columns: 24

Column Names:
adult
belongs_to_collection
budget
genres
homepage
id
imdb_id
original_language
original_title
overview
popularity
poster_path
production_companies
production_countries
release_date
revenue
runtime
spoken_languages
status
tagline
title
video
vote_average
vote_count
```

As we can see that the movies_metadata has 45,572 rows and 24 columns including JSON information of belongs_to_collection, original_language, title, release_date, budget and revenue etc. Below is the ratings data which has about 26 millions of ratings in 4 columns including userID, movieID, rating and timestamp columns. Notice that in PySpark, the truncate parameter as a parameter in the show() function decides whether the displayed strings should be truncated or not. Choosing “False”, it shows the full content of the strings and that explains why the movies_metadata output is cut.

Figure 4: ratings.csv Information Display Output Screenshot

```
display_csv_info_spark(ratings, 'ratings.csv')

File: ratings.csv
First 5 rows:
+-----+-----+-----+-----+
|userId|movieId|rating|timestamp|
+-----+-----+-----+-----+
|1      |110    |1.0   |1425941529|
|1      |147    |4.5   |1425942435|
|1      |858    |5.0   |1425941523|
|1      |1221   |5.0   |1425941546|
|1      |1246   |5.0   |1425941556|
+-----+-----+-----+-----+
only showing top 5 rows

Number of rows: 26024289
Number of columns: 4

Column Names:
userId
movieId
rating
timestamp
```

Using the `join()` function, we merged the `movies_metadata`, `links`, `ratings` datasets to form a `merged_data` object. The connection between these two datasets is that the “movieID” column from the `ratings` data matches with the “id” column from the `movies_metadata` data, while in the `links` data the common identifier is “movieID”.

2.5 EDA on Merged Data

2.5.1 Missing and Abnormal Data Handling

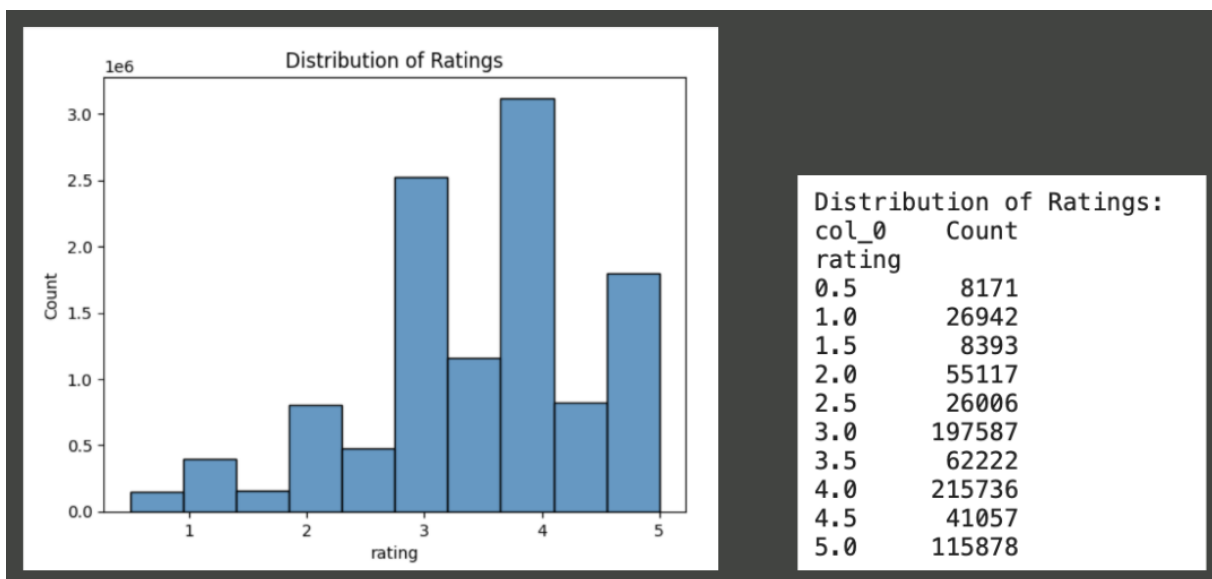
The datasets we used for this project came in pretty clean already but we wanted to improve the quality a bit more. Thus, we applied a few basic preprocessing steps to it. The first step is to handle missing values. We dropped all missing values and resulted in only 757,109 rows of data with the same amount of 30 columns. And we removed features that are not the focus for this project, e.g. “imdb_id” column. Comparing the “title” and “original title” columns, we also discovered that some file original titles are including other languages, which might impact the further EDA. However, the “title”

column is the English version of it, so we decided to drop the “original title” column and only use “title” for the rest of the project scope. For movie revenues that have value 0, that indicates that there’s no known information about them. We replaced all these 0 values with N/A for further missing data dropped. And we also created two columns “year” as the released year number and “return” as the ratio of revenue to budget for potential visualization needs.

2.5.2 What is the ratings distribution?

Selecting only the “rating” column in the merged_data and converting the object using toPandas(), we plotted the histogram using seaborn with 10 bins. Along with the same table generated using pandas crosstab(), we discovered that the most given rating is 4/5 which is a pretty good score. In fact, we can see that there are massive full scores ratings as well, 115878 ratings give 5 points.

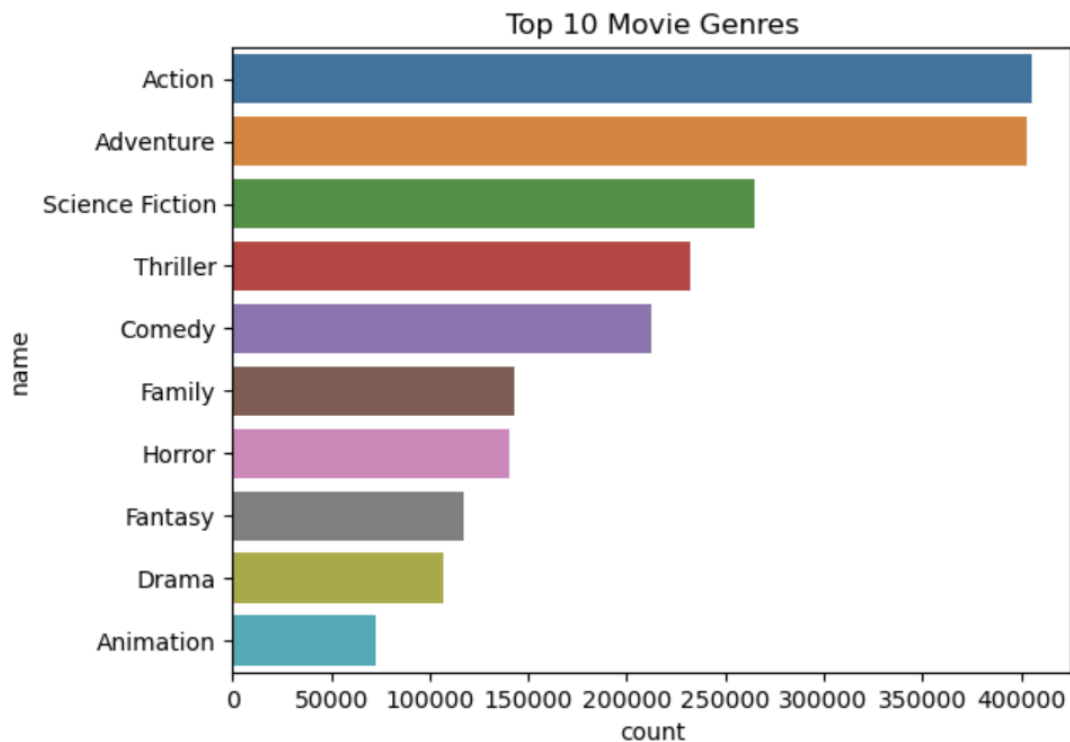
Figure 5: Visualizations of Ratings Distribution



2.5.3 Which are the top 10 movie genres?

Utilizing the `groupBy()`, `count()` and `orderBy()` with `limit()` functions to obtain the top 10 movie genres, we resulted in Action and Adventure movies being the most genres that have received the most ratings to almost 400,000. We compared this output with the incoming visualizations for further conclusions, to answer the question of whether top movie genres get the highest average ratings.

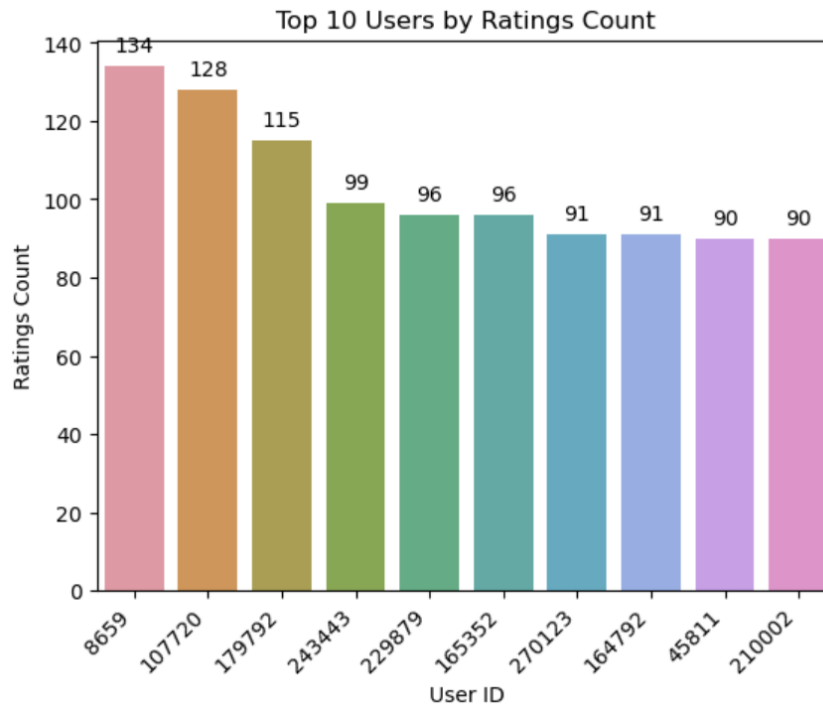
Figure 6: Visualizations of Top 10 Movie Genres



2.5.4 Who are the top 10 users that gave the most ratings?

For this question, we used the same method using the `groupBy()`, `count()` and `orderBy()` with `limit()` functions but when it comes to the seaborn visualization, we changed the parameters a bit. For example, set the seaborn color palette to use “Blues_d”, rotate x-axis ticks to 45 degrees so they don’t interact and add y-values on top of the bars to help reading the results. The discovery out of this barplot is that userID 8659 had contributed to movie ratings for 134 times. And userID 107720 and 179792 also did more than 100 ratings.

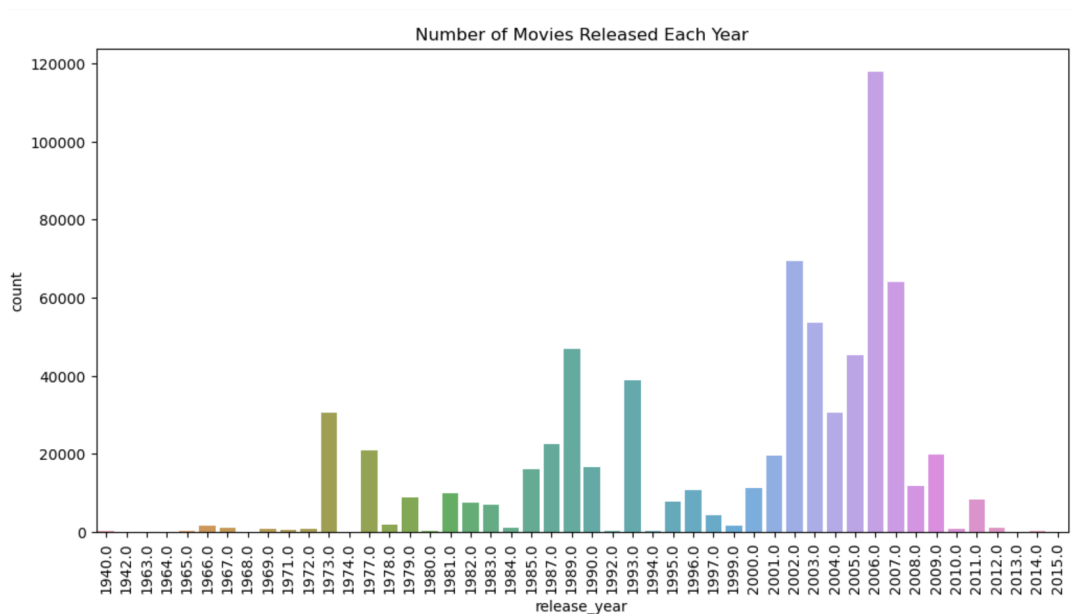
Figure 7: Visualizations of Top 10 Users By Ratings Count



2.5.5 What are the numbers of movie ratings each year?

In order to obtain the numbers of movies released each year, we used the same method using the `groupBy()`, `count()` and `orderBy()` and discovered that approximately after 2002, movie ratings increased more than twice. Especially in 2006, we can see that the most movie ratings were given this year, up to about 120,000. For future improvement, if we analyze only movie ratings and other features in 2006, we will see the further patterns here.

Figure 8: Visualizations of Yearly Distribution of Movie Ratings

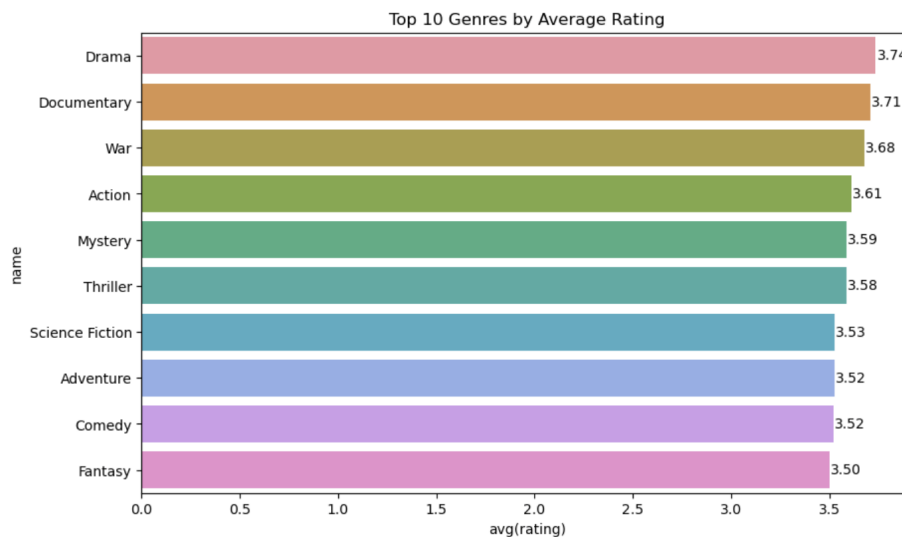


* Vizt typo on title: “Number of Movie Ratings Each Year”

2.5.6 What are the top 10 genres by average rating?

Using the similar PySpark programming methods to prepare the data for plotting, we created this barplot to show the top 10 genres by average ratings. The winner is Drama movies with 3.74 average ratings score, which is similar to the second place as Documentary movies. It's interesting to see that Drama movies do not have the most ratings counts but are actually rated with higher average scores.

Figure 9: Visualizations of Top 10 Genres by Average Rating



2.5.7 What are the top 10 highest rated movies?

With another aggregation created to obtain the top 10 highest rated movies using average ratings, we plotted the horizontal bar chart. We can see that the top 10 movies include Hannibal Rising, Ice Age: The Meltdown, and etc, are all known famous movies. Notice that some of these movie titles fall onto the Action and Adventure genres and we can easily see the potential relationships.

Figure 10: Visualizations of Top 10 Highest Rated Movies

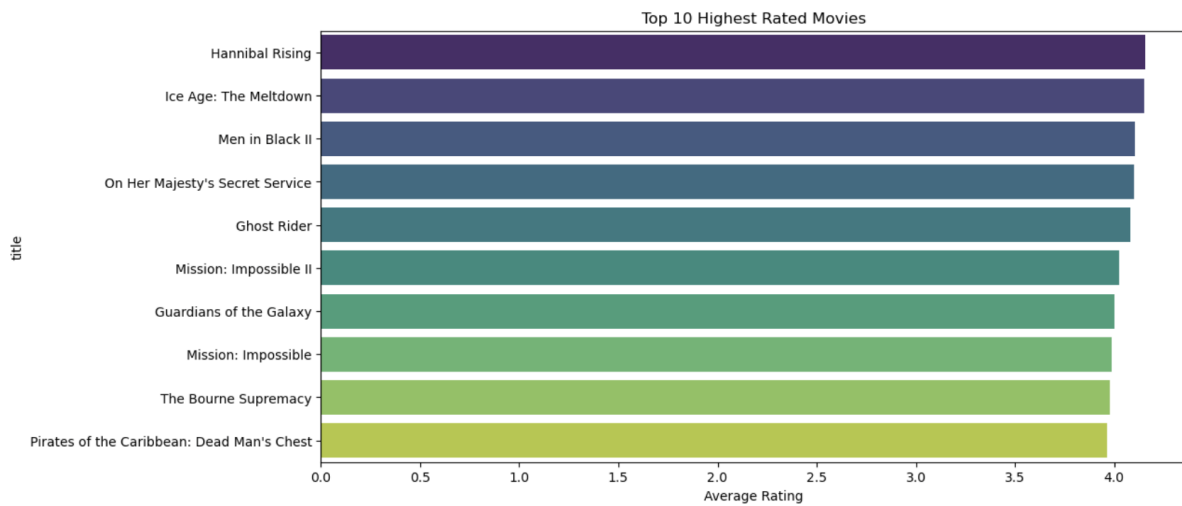


Figure 11: Visualizations of Movie Titles WordCloud

[illegible]

2.5.9 What are the highest grossing franchises?

To obtain the highest grossing franchises list, we filtered out rows that doesn't have collection information using the filter() function, and used a get_collection_name() function along with the pyspark.sql importing functions F to help create the pivot table of collection information, title, grossing count and mean, sum, min and max grossings. From this table, we can see that the movie Avatar from the Avatar Collection has the highest grossing and the revenue is almost twice as the lower-ranked movies on this list. It's also interesting to discover that two Avengers Collection movies are on this list. It's not always the collection of the highest grossing movie that shows up more, not even the movies from the same collection earning a lot.

Figure 12: Visualizations of Highest Grossing Franchises

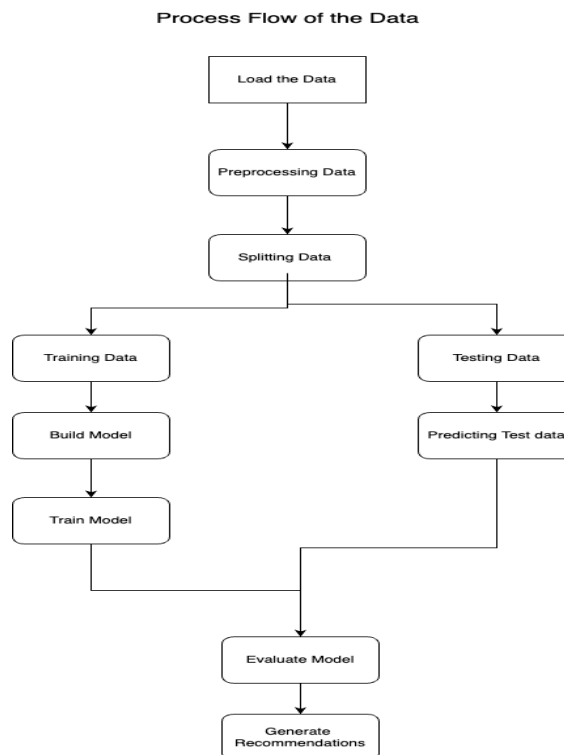
belongs_to_collection	title	mean	sum	count	min	max
Avatar Collection	Avatar	2.787965087E9	2.787965087E9	1	2787965087	2787965087
Star Wars Collection	Star Wars: The Force Awakens	2.068223624E9	2.068223624E9	1	2068223624	2068223624
The Avengers Collection	The Avengers	1.51955791E9	1.51955791E9	1	1519557910	1519557910
Jurassic Park Collection	Jurassic World	1.51352881E9	1.51352881E9	1	1513528810	1513528810
The Fast and the Furious Collection	Furious 7	1.50624936E9	1.50624936E9	1	1506249360	1506249360
The Avengers Collection	Avengers: Age of Ultron	1.405403694E9	1.405403694E9	1	1405403694	1405403694
Harry Potter Collection	Harry Potter and the Deathly Hallows: Part 2	1.342E9	1.342E9	1	1342000000	1342000000
Frozen Collection	Frozen	1.274219009E9	1.274219009E9	1	1274219009	1274219009
The Fast and the Furious Collection	The Fate of the Furious	1.238764765E9	1.238764765E9	1	1238764765	1238764765
Iron Man Collection	Iron Man 3	1.215439994E9	1.215439994E9	1	1215439994	1215439994

only showing top 10 rows

3. Proposed method

Following the below flowchart, we delivered the two proposed methods. In order to provide movie recommendations, we employed two methodologies: the Alternating Least Squares algorithm (ALS) with PySpark, and the Singular Value Decomposition algorithm (SVD) with Python, trying to compare two distinct working environments at the same time. As we learned in class, PySpark serves as a distributed system capable of handling large-scale data processing across clusters, while Python operates as a standalone environment, suited for smaller, single-machine data analysis tasks. Our comparative study will provide a comparison to assist us in understanding the operational differences, as well as the potential advantages and disadvantages of each environment in the context of collaborative filtering for recommendation systems.

Figure 13. FlowChart of the Data Flow



- Load the Data: A CSV file called "ratings_small.csv" is used to import the ratings data into a Spark DataFrame called ratings_small. We assume that the data has columns with names like "userId," "movieId," and "rating."
- Splitting the Data: The randomSplit method divides the dataset into training and testing sets. Twenty percent of the data is set aside for testing (test set), while the remaining eighty percent is set aside for training (training set).
- Merge the Data: merged_data DataFrame to appropriate data types (userId to int, movieId to int, rating to float).
- Construct the ALS Model: The ALS class from Spark's MLlib is used to generate the Alternating Least Squares (ALS) recommendation model. Important variables consist of:
 1. maxIter: Maximum iteration count (5 in this example).
 2. regParam: The regularization parameter (in this case, 0.1) to prevent overfitting.
 3. The names of the columns in the DataFrame that show user, item, and rating information userId, movieId, rating.
 4. coldStartStrategy: Managing plan for problems arising from cold starts (users or objects not observed during training). Since it's set to "drop," no new users or objects will be added while the prediction is being made.
- Train the Model: On the training set, the fit approach is used to train the ALS model.
- Prediction: The model generates predictions on the test data.
- Evaluate the Model: The Root Mean Squared Error (RMSE) is the evaluation statistic that is used to assess the trained model on the test set. The discrepancy between expected and actual ratings is measured by RMSE.
- Generate the Recommendations: For a certain user (in this case, user with ID 8659),

recommendations are created. Lastly, it uses the `movies_metadata` DataFrame and the movie IDs from the recommendations to display information about the suggested films, including title, genres, release date, vote average, and vote count. Keep in mind that `movies_metadata` need to be a DataFrame with information on movies.

3.1 Alternating Least Squares with PySpark

The Alternating Least Squares (ALS) algorithm is a methodology employed in collaborative filtering to predict ratings and create personalized recommendations. This algorithm is a built-in part of MLlib, which is Apache Spark's specialized machine learning library. ALS starts from initially factorizing the user-item interaction matrix into more manageable latent factor matrices that encapsulate user preferences and item characteristics. It then goes through an iterative process to optimize these factors with the objective of minimizing the reconstruction error from the original matrix while incorporating regularization techniques to prevent overfitting. For our dataset, we first randomly split the data into training (80%) and testing (20%) sets. Then, we removed missing values from both the training and testing sets to ensure data equality. Following that, an ALS model is instantiated with hyperparameters such as the maximum number of iterations(`maxIter`), regularization parameter(`regParam`), and column names corresponding to users, items (movies), and ratings. We also specified a strategy to handle cold starts by dropping any unseen user-item pairs during predictions. The model is then trained on the cleaned training dataset and subsequently used to predict ratings on the test set.

Figure 14. Sampling and Testing the Model of Altering Least Square(ALS)

```
ratings_small = spark.read.csv("ratings_small.csv", header=True, inferSchema=True)

# Split the dataset into training and testing sets
(training, test) = ratings_small.randomSplit([0.8, 0.2])

# Check for and handle missing values
training = training.na.drop()
test = test.na.drop()

# Build the recommendation model using ALS
als = ALS(maxIter=20, regParam=0.1, userCol="userId", itemCol="movieId", ratingCol="rating", coldStartStrategy="drop")
model = als.fit(training)

# Evaluate the model on the test set
predictions = model.transform(test)
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")
rmse = evaluator.evaluate(predictions)
print(f"Root Mean Squared Error (RMSE) on the test data = {rmse}")

# Generate movie recommendations for a given user
user_id = 1
user_unrated_movies = test.filter(col("userId") != user_id).select("movieId", "userId")

# Generate recommendations for unrated movies
recommendations = model.transform(user_unrated_movies)
recommendations.show()
```

Figure 15. Altering Least Square Model

```
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.sql import functions as F

# Convert necessary columns to appropriate data types
merged_data = merged_data.withColumn("userId", merged_data["userId"].cast("int"))
merged_data = merged_data.withColumn("movieId", merged_data["movieId"].cast("int"))
merged_data = merged_data.withColumn("rating", merged_data["rating"].cast("float"))

# Create ALS model
als = ALS(maxIter=5, regParam=0.01, userCol="userId", itemCol="movieId", ratingCol="rating", coldStartStrategy="drop")

# Split the data into training and testing sets
(training, test) = merged_data.randomSplit([0.8, 0.2])

# Fit the model to the training data
model = als.fit(training)

# Generate predictions on the test data
predictions = model.transform(test)

# Evaluate the model using RMSE
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")
rmse = evaluator.evaluate(predictions)
print(f"Root Mean Squared Error (RMSE) = {rmse}")

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

# Show the recommendations for a specific user (replace USER_ID with the actual user ID)
user_id = 8659
user_recommendations = userRecs.filter(F.col("userId") == user_id).select("recommendations.movieId").collect()[0][0]

# Display the recommended movies
recommended_movies = movies_metadata.filter(F.col("id").isin(user_recommendations))
recommended_movies.select("title", "genres", "release_date", "vote_average", "vote_count").show(truncate=False)
```

3.2 Singular Value Decomposition with Python

SVD, short for Singular Value Decomposition, is also a collaborative filtering algorithm that can capture the intricacies of user preferences and offer recommendations. It employs a

matrix factorization method that decomposes a matrix into three separate matrices, capturing the core characteristics of the original data. In the context of recommendation systems, SVD serves the purpose of predicting missing values in the user-item interaction matrix, thus enabling the recommendation of products or content. It detects hidden factors that account for observed ratings and helps in generating recommendations based on the patterns found in the decomposed matrices. In our use case, we applied SVD to predict movie ratings and to provide corresponding movie recommendations. Initially, we loaded the movie metadata and user ratings into Pandas DataFrame, then merged them to align movie titles with user ratings. Next, we created a user-item matrix, filling in missing values with zeros to represent unrated movies. This matrix was then divided into training and testing sets. Finally, we applied truncated SVD to the training data to identify latent factors, which allowed us to generate predictions for the test data.

Figure 16. Singular Value Decomposition

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.decomposition import TruncatedSVD

# Load the movies metadata
movies = pd.read_csv("movies_metadata.csv", dtype={'id': 'str'})

# Load the ratings dataset
ratings = pd.read_csv("ratings_small.csv", dtype={'movieId': 'str'})

# Merge the ratings and movies datasets
df = pd.merge(ratings, movies[['id', 'original_title']], left_on='movieId', right_on='id', how='left')

# Create a user-item matrix
user_item_matrix = df.pivot_table(index='userId', columns='original_title', values='rating')

# Fill missing values with 0
user_item_matrix = user_item_matrix.fillna(0)

# Split the user-item matrix into training and test sets
train_user_item_matrix, test_user_item_matrix = train_test_split(user_item_matrix, test_size=0.2, random_state=42)

# Filter the test set to include only items present in the training set
common_movies = train_user_item_matrix.columns.intersection(test_user_item_matrix.columns)
test_user_item_matrix_filtered = test_user_item_matrix[common_movies]

# Predictions using SVD
svd = TruncatedSVD(n_components=50, random_state=42)
train_svd_matrix = svd.fit_transform(train_user_item_matrix)
test_svd_matrix = svd.transform(test_user_item_matrix_filtered)

# Reconstruct predicted ratings
predicted_ratings = svd.inverse_transform(test_svd_matrix)
```

Figure 17. Prediction for A Chosen User

```
import numpy as np

# Choose a user for recommendation
available_user_indices = test_user_item_matrix_filtered.index
user_id_for_recommendation = np.random.choice(available_user_indices)

# Get the predicted ratings for the chosen user
user_predicted_ratings = predicted_ratings[available_user_indices.get_loc(user_id_for_recommendation)]

# Get the indices of the top N recommended movies
N = 10 # You can choose any number of recommendations
top_movie_indices = user_predicted_ratings.argsort()[::-1]

# Get the corresponding movie titles
top_movie_titles = user_item_matrix.columns[top_movie_indices]

# Display the top recommended movies
print(f"Top {N} Recommended Movies for User {user_id_for_recommendation}:\n")
for i, movie_title in enumerate(top_movie_titles, 1):
    print(f"{i}. {movie_title}")
```

4. Result

The models' performance is evaluated using the Root Mean Squared Error (RMSE) metric, which measures the average magnitude of prediction errors. In both ALS and SVD models, RMSE values fall within the range of 0 to 2, which is considered a decent result as an evaluation metric. The ALS model, implemented with PySpark, displays an RMSE of approximately 1.465, while the SVD model, employing a Python framework, achieves a significantly lower RMSE of approximately 0.4478. These outcomes suggest that, given the dataset and experimental conditions, SVD with Python outperforms PySpark's ALS in minimizing prediction errors and providing superior personalized movie recommendations. The superior performance of SVD may be attributed to the inherent characteristics of the algorithms while our project primarily focuses on experimenting with various environments to relate to materials we had from the lecture.

Table 2: Model Evaluation Metrics Results

Model	Root Mean Squared Error (RMSE)
ALS with Pyspark	1.4650165950599872
SVD with Python	0.44781990220337586

Figure 18: Result of Altering Least Square

title	genres
Planet of the Apes	[{'id': 878, 'name': 'Science Fiction'}, {'id': 12, 'name': 'Adventure'}, {'id': 18, 'name': 'Drama'}, {'id': 28, 'name': 'Action'}]
They ain't afraid of no ghost.	[{'id': 35, 'name': 'Comedy'}, {'id': 14, 'name': 'Fantasy'}]
From Russia with Love	[{'id': 28, 'name': 'Action'}, {'id': 53, 'name': 'Thriller'}, {'id': 12, 'name': 'Adventure'}]
For Your Eyes Only	[{'id': 12, 'name': 'Adventure'}, {'id': 28, 'name': 'Action'}, {'id': 53, 'name': 'Thriller'}]
AVP: Alien vs. Predator	[{'id': 12, 'name': 'Adventure'}, {'id': 878, 'name': 'Science Fiction'}, {'id': 28, 'name': 'Action'}]
Meet the Fockers	[{'id': 35, 'name': 'Comedy'}, {'id': 10749, 'name': 'Romance'}]
The Legend of Zorro	[{'id': 28, 'name': 'Action'}, {'id': 12, 'name': 'Adventure'}]
Spider-Man 3	[{'id': 14, 'name': 'Fantasy'}, {'id': 28, 'name': 'Action'}, {'id': 12, 'name': 'Adventure'}]
The Gamers: Dorkness Rising	[{'id': 12, 'name': 'Adventure'}, {'id': 35, 'name': 'Comedy'}, {'id': 14, 'name': 'Fantasy'}]
Insidious: Chapter 2	[{'id': 27, 'name': 'Horror'}, {'id': 53, 'name': 'Thriller'}]

The PySpark ALS model achieved a commendable RMSE of 1.44, showcasing its effectiveness in providing reliable movie recommendations, despite a slightly higher scale on the RMSE. And, the Python SVD model demonstrated impressive precision with a lower RMSE of 0.45, indicating its capability to deliver accurate and fine-tuned movie recommendations, highlighting its strength on the RMSE scale.

Figure 19: Result of Singular Value Decomposition with Python

Top 10 Recommended Movies for User 481:

1. Terminator 3: Rise of the Machines

2. Sous le Sable

3. Scarface

4. Dawn of the Dead

5. License to Wed

6. Sleepless in Seattle

7. 5 Card Stud

8. Shriek If You Know What I Did Last Friday the Thirteenth

9. The Prisoner of Zenda

10. The Talented Mr. Ripley

Calculate and print Root Mean Squared Error (RMSE)

rmse = np.sqrt(mse)

print(f'Root Mean Squared Error: {rmse}')

Root Mean Squared Error: 0.44781990220337586

5. Discussion

For the two methods we chose, each of them has its own set of pros and cons. ALS is inherently parallelizable, making it a good fit for distributed computing frameworks like Apache Spark. However, ALS faces challenges with new users or items that have no ratings (the cold start problem), although strategies like dropping these cases can mitigate the issue. Furthermore, the performance of ALS is highly sensitive to the choice of hyperparameters, necessitating careful tuning accordingly. On the other hand, while SVD excels at reducing the number of features in a dataset while retaining critical information, it is notorious for its computational cost especially when dealing with large matrices. This can make it less suitable for extremely large datasets or those requiring real-time processing. Furthermore, SVD is known for its high memory usage due to the necessity of storing three matrices instead of just one, which can impose constraints when handling very large datasets.

To enhance recommendation accuracy, incorporating Natural Language Processing (NLP) techniques to analyze the textual components of movies, like 'overview' and 'keywords', could be highly beneficial. These NLP methods can extract semantic information, enriching the recommendation model's understanding of content and context. This semantic component could significantly improve the prediction of movie ratings and the quality of movie recommendations, leading to a more sophisticated and user-customized experience.

For future improvements, other recommendation algorithms can also be considered. Content-based filtering, besides collaborative filtering, is also a viable method that focuses on item attributes, recommending items similar to what a user has previously interacted with, thereby personalizing suggestions based on the content of the items themselves. Additionally, deep learning approaches, such as autoencoders, could be utilized for their ability to learn

efficient representations of data, while reinforcement learning algorithms might be employed to continually adapt recommendations based on user feedback, optimizing the recommendation policy over time. Integrating these methods could potentially provide a more robust and nuanced recommendation system, enhancing user satisfaction through improved relevance and personalization of content.

6. Conclusion

6.1 EDA Conclusion

The year of release, the genres of the films, and the phrasing of the movie titles are factors that entice viewers to leave ratings, especially positive ones. The most well-ranked movie genres aren't always the highest rated ones. Based on the available statistics, a significant fraction of evaluations are still negative, therefore this aligns with reality. We are aware that nearly everyone is familiar with and enjoys the films in the highest grossing franchises. This is also in line with reality, as it makes natural that well-received films would receive positive reviews. That's what makes these films what they are. But just because a movie does well financially doesn't mean that every film in the same collection does as well. We can presume that it's not always the case to suggest the entire series to a user while making movie recommendations. After a thorough exploratory data analysis, we can conclude that factors that attract users to give ratings, especially good ratings, include year of release, movie genres, movie title wordings, and etc. Movie genres that receive the most ratings are not necessarily rated the highest. This suits the reality because bad ratings are still a big portion of given ratings. For movies that are the highest grossing franchises made, we know that almost everyone knows

those names and have positive comments about them. This also makes sense in the reality world that popular movies get a lot of good feedback.

6.2 Modeling Conclusion

In conclusion, our MovieLens Big Data Analysis project has shown encouraging results in creating a dynamic movie recommendation system by utilizing both Python's SVD model and PySpark's ALS model. The Root Mean Squared Error (RMSE) of the PySpark ALS model was 1.4378826830223226, but the Python-based SVD model had a lower RMSE of 0.44781990220337586. Based on past movie ratings, these figures indicate how well our recommendation engines forecast customer preferences. With an RMSE of 1.44, the PySpark ALS model predicts user-liked movies fairly well. Its ability to handle large amounts of video data rapidly comes in handy when there's a lot of data to process. Conversely, the Python SVD model accurately predicts user preferences for movies, with an RMSE of 0.45. It's similar to having a deeper comprehension of user behavior when it comes to movie selection. However, when working with really large datasets, it may become somewhat slower. In conclusion, each model has advantages. In case you have a large amount of data and require speedy processing, the PySpark ALS model is the best option. However, the Python SVD model is the best option if you want really precise recommendations and don't mind a slight slowness while working with large datasets. It all comes down to the specific requirements that your movie recommendation system has.

References

1. <https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset/data>
2. <https://grouplens.org/datasets/movielens/latest/>
3. <https://spark.apache.org/docs/latest/ml-collaborative-filtering.html>
4. <https://medium.com/@jonahflateman/building-a-recommender-system-in-pyspark-using-als-18e1dd9e38e6>
5. <https://machinelearningmastery.com/singular-value-decomposition-for-machine-learning/>
6. <https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html>
7. https://python.quantecon.org/svd_intro.html