Project 2:

Movie Recommendation System with PySpark

Tianhao Guan, Kunbo Zhang, Weiyao Li, Chongzhi Li

Content

- Introduction
- Data Preprocessing
- EDA
- PySpark
- Modeling
    - Collaborative Filtering with ALS
    - Hybrid Filtering (collaborative and content-based)

Introduction

- Dataset Source: https://www.kaggle.com/code/alfarias/movie-recommendation-system-with-als-in-pyspark/input

- The dataset contains information on 45,000 movies featured in the Full MovieLens dataset, with movies released on or before July 2017, and also includes 26 million ratings from 270,000 users for all 45,000 movies, with ratings on a scale of 1-5 that were obtained from the official GroupLens website.

- Our project aims to build a movie recommendation system using PySpark, which is a type of information filtering system that provides personalized recommendations to users based on their movie preferences and behavior.

Data Overview

- Dealing with three datasets
  - Ratings: data on each user's rating of a single movie. Columns include **user id, movie id, and rating**
  - Metadata: information about each movie. Columns include **movie title, genre, popularity, budget, revenue, average rating, counts of rating, etc.**
  - Links: a linkage dataset which links the id's in *Metadata* to the movie id specified in *Ratings*

- Relationship between the datasets
  - Dataset *Links* includes id in both *Ratings* and *Metadata*
  - 'id' and 'imdb_id' in metadata refers to 'tmdbId' and 'imdbId' links respectively

```
ratings.head()
```

	userId	movieId	rating	timestamp
0	1	110	1.0	1425941529
1	1	147	4.5	1425942435
2	1	858	5.0	1425941523
3	1	1221	5.0	1425941546
4	1	1246	5.0	1425941556

```
links.head()
```

	movieId	imdbId	tmdbId
0	1	114709	862.0
1	2	113497	8844.0
2	3	113228	15602.0
3	4	114885	31357.0
4	5	113041	11862.0

```
metadata[['id','imdb_id']].head()
```

	id	imdb_id
0	862	tt0114709
1	8844	tt0113497
2	15602	tt0113228
3	31357	tt0114885
4	11862	tt0113041

Data Preprocessing: metadata - columns

Columns dropped:

'adult', 'belongs_to_collection', 'video', 'homepage', 'tagline', 'poster_path'

- Because either too unbalanced or too many missing values
- URLs (homepage, tagline, poster_path)

```
metadata.isna().sum().sort_values(ascending=False)

belongs_to_collection    40972
homepage                 37684
tagline                  25054
```

Data type:

- Popularity -> float
- Release date -> datetime
- Budget -> float

```
metadata['video'].value_counts()

False    45367
True        93
Name: video, dtype: int64
```

Data Preprocessing: metadata - rows

Invalid Data:

- Three rows that have data in the wrong column
- For example:
  - Budget: poster path
  - Id: released date

	budget	id
19730	/ff9qCepilowshEtG2GYWwzt2bs4.jpg	1997-08-20
29503	/zV8bHuSL6WXoD6FWogP9j4x80bL.jpg	2012-09-29
35587	/zaSf5OG7V8X8gqFvly88zDdRm46.jpg	2014-01-01

Replacing values:

- Many 0's in numeric columns that shouldn't be 0
- 0's in **budget, runtime, revenue** -> NA
- 0's in vote average should be NA when vote count is 0 (undefined)
- Missing values dropped at the end

	vote_count	vote_average
83	0.0	0.0
107	0.0	0.0
126	0.0	0.0
132	0.0	0.0
137	0.0	0.0

	vote_count	vote_average
83	0.0	NaN
107	0.0	NaN
126	0.0	NaN
132	0.0	NaN
137	0.0	NaN

Data Preprocessing: links

Adjust the format of ‘tmdbId’ to align the corresponding column in the metadata:

- Floating-point numbers in ‘tmdbId’ need to be converted to integers then to string
- However, ‘tmdbId’ has null values (of type float), which cannot be converted to integers
  - Null values are replaced with -1
  - Once the conversion is complete, the -1 placeholders are replaced back with null values

```
links.head()
```

	movieId	imdbId	tmdbId
0	1	114709	862.0
1	2	113497	8844.0
2	3	113228	15602.0
3	4	114885	31357.0
4	5	113041	11862.0

tmdbId
862
8844
15602
31357
11862
949

Data Preprocessing: linking datasets

- Adjust the format of ‘imdb_id’ to align the corresponding column in the links dataset
  - String subsetting
  - Null values, handled the same way as links
- Map the ‘movieId’ from the links dataset to the metadata dataset

```
metadata[['id','imdb_id']].head()
```

	id	imdb_id
0	862	tt0114709
1	8844	tt0113497
2	15602	tt0113228

```
meta_eda[['id', 'imdb_id', 'movieId']].head()
```

	id	imdb_id	movieId
0	862	114709	1
1	8844	113497	2
2	15602	113228	3
3	31357	114885	4
4	11862	113041	5

```
links_clean.head()
```

	movieId	imdbId	tmdbId
0	1	114709	862
1	2	113497	8844
2	3	113228	15602
3	4	114885	31357
4	5	113041	11862

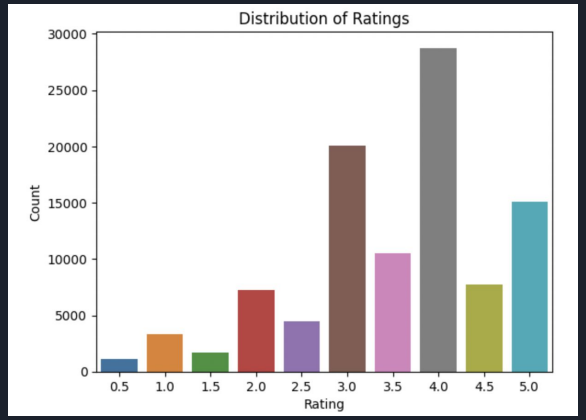
EDA: ratings

- Basic statistics

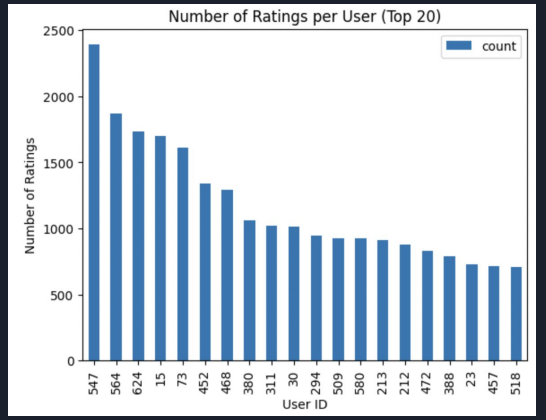
```
ratings.describe().show()
```

```
+-------+------------------+------------------+------------------+------------------+------------------+
|summary|               _c0|            userId|           movieId|            rating|         timestamp|
+-------+------------------+------------------+------------------+------------------+------------------+
|  count|            100004|            100004|            100004|            100004|            100004|
|   mean|           50001.5| 347.0113095476181|12548.664363425463| 3.543608255669773|1.1296390869392424E9|
| stddev|28868.812497226136|195.16383797819535|26369.198968815268|1.0580641091070326|1.9168582602710962E8|
|    min|                 0|                 1|                 1|               0.5|         789652009|
|    max|            100003|               671|            163949|               5.0|        1476640644|
+-------+------------------+------------------+------------------+------------------+------------------+
```

- Majority of the ratings being between 3 and 5



- Certain users are more active in providing ratings, which could influence the recommendation system.



EDA: metadata - correlation

- Average vote and revenue is negative related, budget and revenue is positive related.
- These findings could be useful for understanding the impact of budgets and user ratings on a movie's success.

```
+--------------------------+
|corr(vote_average, revenue)|
+--------------------------+
|       -0.09166003952107951|
+--------------------------+
```

```
+--------------------+
|corr(budget, revenue)|
+--------------------+
|    0.726754201236627|
+--------------------+
```

- On the right is the top ten movie with the highest average rating score and the movie title

movieId	title	rating
163949	The Beatles: Eight Days a Week - The Touring Y...	5.0
4088	The Big Town	5.0
91673	Albert Nobbs	5.0
91690	Friends with Kids	5.0
1859	Taste of Cherry	5.0
4076	Two Ninas	5.0
92210	The Disappearance of Haruhi Suzumiya	5.0
92494	Dylan Moran: Monster	5.0
25801	She Done Him Wrong	5.0
93320	Trailer Park Boys	5.0

PySpark

- PySpark can handle large-scale data processing tasks by distributing the workload across a cluster of computers.
- Compared to other big data processing methods, PySpark offers significant advantages such as lightning-fast processing speed, fault-tolerance, and ability to manage massive datasets across multiple nodes, making it an optimal solution for cost-effective analytics.
- Read data using PySpark and split the data into training and test datasets

```
data_schema = StructType([
    StructField("userId", IntegerType(), True),
    StructField("movieId", IntegerType(), True),
    StructField("rating", FloatType(), True),
    StructField("timestamp", IntegerType(), True)
])

ratings_spark = spark.read.csv('/content/drive/MyDrive/project2/ratings_small.csv', header=True, schema=data_schema).cache()
ratings_df = (ratings_spark.select(
    'userId',
    'movieId',
    'rating'
)).cache()
```

```
# Split the data into training and test datasets
(training, test) = ratings_df.randomSplit([0.8, 0.2], seed=42)
```

Modelling: Collaborative Filtering with ALS

- PySpark, when used with the Alternating Least Squares (ALS) algorithm, provides a powerful and scalable solution for collaborative filtering in recommendation systems.
- The ALS algorithm is a model-based collaborative filtering method that uses ratings data to generate recommendations.
- It factorizes the user-item interaction matrix, which consists of user ratings for items, into lower-dimensional user and item latent factor matrices. These matrices capture underlying patterns and preferences in user-item interactions.
- In contrast, memory-based collaborative filtering methods like user-based and item-based approaches compute similarities directly from the ratings data without building a model.
- So, while ALS utilizes the ratings data, it differs from user-based or item-based collaborative filtering in its approach to generating recommendations.

Modelling: Collaborative Filtering with ALS

- Considers only the user-item rating data for making recommendations
- Training and evaluating a recommendation model using Alternating Least Squares (ALS) algorithm in Spark
- On average, the predicted ratings deviate from the actual ratings by about 0.71 on a scale of 0 to 5, which is relatively low.

```
svd = ALS(
          rank=30,
          maxIter=4,
          regParam=0.1,
          userCol='userId',
          itemCol='movieId',
          ratingCol='rating',
          coldStartStrategy='drop',
          implicitPrefs=False
         )
svd_model = svd.fit(training)

svd_predictions = svd_model.transform(test)
svd_evaluator = RegressionEvaluator(metricName='mae', labelCol='rating',
                                    predictionCol='prediction')

svd_mae = svd_evaluator.evaluate(svd_predictions)
print(f'MAE (Test) = {svd_mae}')
```

```
MAE (Test) = 0.7144480259975858
```

Modelling: Collaborative Filtering with ALS

- Films recommended for users by the ALS model along with their ratings

```
svd_model.recommendForAllUsers(1).show(5)
```

```
+------+--------------------+
|userId|     recommendations|
+------+--------------------+
|     1|  [{1172, 3.490329}]|
|     2| [{1192, 5.0077834}]|
|     3| [{83411, 4.675861}]|
|     4|  [{1192, 5.794789}]|
|     5|[{89904, 4.8696694}]|
+------+--------------------+
only showing top 5 rows
```

- Titles for the recommended films

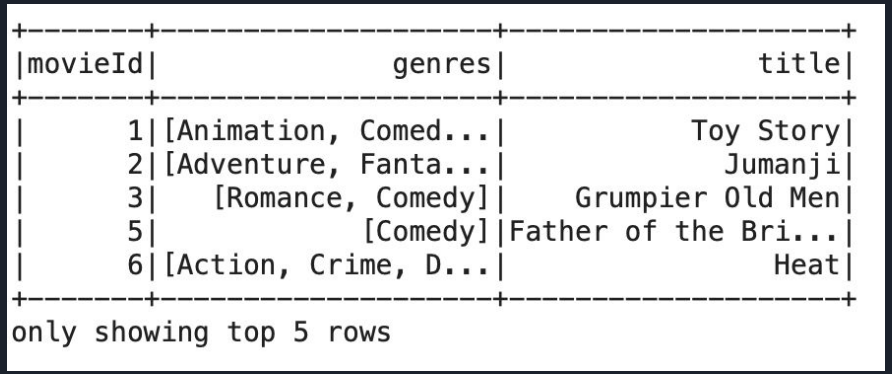
```
movie_ids_to_find = [1172, 1192, 83411, 89904]
filtered_movies = movies_spark.filter(col("movieId").isin(movie_ids_to_find))
movie_titles = filtered_movies.select("movieId", "title")
movie_titles.show()
```

```
+-------+----------------+
|movieId|           title|
+-------+----------------+
|   1172| Cinema Paradiso|
|   1192|Paris is Burning|
|  83411|            Cops|
|  89904|      The Artist|
+-------+----------------+
```

Modelling: Hybrid Filtering (collaborative + content-based)

Considers both user-item rating data and genre of the film for making recommendations:

- Original data in column ‘genres’ is in json format, where each film's genres are listed under the 'name' key
- Extracts the genre for each film from the 'genres' column and stores them in a list



```
+-------+--------------------+--------------------+
|movieId|              genres|               title|
+-------+--------------------+--------------------+
|      1|[Animation, Comed...|           Toy Story|
|      2|[Adventure, Fanta...|             Jumanji|
|      3|   [Romance, Comedy]|    Grumpier Old Men|
|      5|            [Comedy]|Father of the Bri...|
|      6|[Action, Crime, D...|                Heat|
+-------+--------------------+--------------------+
only showing top 5 rows
```

Modelling: Hybrid Filtering (collaborative + content-based)

Prepare the movie data for building a content-based recommendation model that uses the movie genres as features to make recommendations:

- PipelineModel, which is a result of fitting a sequence of transformations (Pipeline) to the movie data. In our project, the pipeline consists of two stages: CountVectorizer and IDF (Inverse Document Frequency)
    - CountVectorizer: converts the 'genres' column, which contains a list of genres for each movie, into a term frequency (TF) vector that represents how often each genre appears in the movie's list of genres.
    - IDF: transformer calculates the inverse document frequency for each genre in the dataset, giving more weight to less common genres and less weight to more common genres.
    - Together, these transformers produce a new column named "features" that contains the final feature vectors for each movie, which can be used for making content-based recommendations.

```
cv = CountVectorizer(inputCol="genres", outputCol="tf")
idf = IDF(inputCol="tf", outputCol="features")
pipeline = Pipeline(stages=[cv, idf])
```

Modelling: Hybrid Filtering (collaborative + content-based)

- The pipeline_model is created by fitting the pipeline to the movies_spark DataFrame, which contains the movie data. The fit method applies the pipeline stages sequentially, resulting in a feature vector for each movie in the dataset. Once the pipeline is fitted, the pipeline_model can be used to transform any new data or make predictions using the learned feature vectors.
- In the context of the hybrid recommendation system, the pipeline_model is used to generate content-based recommendations for users by calculating the similarity between the movies' feature vectors and the user's preferences (user profile).

```
cv = CountVectorizer(inputCol="genres", outputCol="tf")
idf = IDF(inputCol="tf", outputCol="features")
pipeline = Pipeline(stages=[cv, idf])
pipeline_model = pipeline.fit(movies_spark)
movies_features = pipeline_model.transform(movies_spark)
```

Modelling: Hybrid Filtering (collaborative + content-based)

Hybrid recommendation model that combines collaborative filtering and content-based approaches

```
def recommend_hybrid(userId, n_recommendations=10):
    # Collaborative filtering recommendations
    user_ratings = ratings_df.filter(ratings_df['userId'] == userId)
    user_unrated_movies = movies_features.alias("movies").join(user_ratings.alias("ratings"), col("movies.movieId") == col("ratings.movieI
    user_unrated_movies = user_unrated_movies.select("movies.movieId", "movies.features").withColumn("userId", lit(userId))
    cf_predictions = als_model.transform(user_unrated_movies).select("movieId", "prediction")

    # Content-based recommendations
    user_profile = pipeline_model.transform(user_ratings.join(movies_spark, on="movieId"))
    user_profile = user_profile.withColumn("product", dot_product(col("features"), col("rating")))
    user_profile = user_profile.groupBy("userId").agg(expr("sum(product)").alias("features_sum"), expr("sum(rating)").alias("rating_sum"))
    user_profile = user_profile.withColumn("features", col("features_sum") / col("rating_sum"))

    content_based_recommendations = pipeline_model.transform(movies_spark).withColumn("userId", lit(userId)).alias("movies_features")
    content_based_recommendations = content_based_recommendations.join(user_profile.alias("user_profile"), on="userId", how="inner")
    content_based_recommendations = content_based_recommendations.withColumn("content_based_score", dot_product(col("movies_features.featu

    # Combine both recommendations
    hybrid_recommendations = cf_predictions.join(content_based_recommendations, on="movieId")
    hybrid_recommendations = hybrid_recommendations.withColumn("hybrid_score", col("prediction") * 0.5 + col("content_based_score") * 0.5)
    hybrid_recommendations = hybrid_recommendations.orderBy("hybrid_score", ascending=False).limit(n_recommendations)

    return hybrid_recommendations.select("movieId")
```

Modelling: Hybrid Filtering (collaborative + content-based)

Display 10 recommendations for a specific user

movieId	title
148	An Awfully Big Ad...
463	Guilty as Sin
471	The Hudsucker Proxy
496	What Happened Was...
833	High School High
1088	Dirty Dancing
1238	Local Hero
1342	Candyman
1580	Men in Black
1591	Spawn