NEW YORK UNIVERSITY

Big Data (DS - GA 1004)

Spring Semester 2025

Capstone Project

Project report - Group CAP 29

Group Members: Lavya Midha, Rithujaa Rajendrakumar, Vrinda Tibrewal

Introduction

In this project, we leverage user–movie interactions from MovieLens to segment viewers and build a baseline and advanced movie recommender, tackling two core tasks:

1. Customer Segmentation – finding "movie twins" via MinHash, based on Jaccard similarity and validating them by rating correlation.

2. Recommendation – partitioning 33 M ratings into train/validation/test, implementing a global popularity baseline, and tuning an ALS latent-factor model.

We evaluate every step with industry-standard ranking metrics to ensure scalable, reproducible results.

Link: https://github.com/nyu-big-data/capstone-team-29-1

Dataset

In this project, we'll use the MovieLens dataset provided by F. Maxwell Harper and Joseph A. Konstan. 2015. All the data is stored in CSV format and accessed via Dataproc's HDFS.

Methodology

1. Calculating the Top 100 Pairs:
    a. Preprocessing : Loading the **ratings.csv** file with explicit schema, filtering null values and only using *movieId* and *userId.* Next was disregarding users with less than 30 movies that were rated to not include unnecessary users.
    b. Hash Function Initialization for MinHash: to approximate Jaccard similarity, we employ *MinHash.*
    c. MinHash Signature Computation: Each user movie set mapped to a signature vector of the length 20 (number of hash tables). For each hash function, we apply it to all indexed movie IDs in the user's set and record the minimum hashed value.
    d. LSH: Each MinHash signature is divided into b bands (here, 4 bands of 5 rows). Users with matching bands are grouped as candidate pairs. This helps reduce the number of comparisons.
    e. Jaccard Similarity: Jaccard similarity was calculated
    f. The top 100 users with highest *Jaccard Similarity* were then selected and written to csv file *top100.csv.*
    g. Overall code stored in **MinHash.py**

2. Comparison between average pairwise correlation between highly similar pairs and randomly picked pairs:
    a. For each top pair Pearson Correlation was calculated based on the ratings of overlapped movies.
    b. Next random 100 distinct movie pairs were selected and the Pearson Correlation for them was computed
    c. Finally, we used the average pairwise correlation of top similar movies and random movies to see who were more correlated
    d. Overall code stored in **validate.py**

3. Data Partitioning into Train / Validation / Test:
    a. **Data Ingestion**: Loaded the full ratings file into a distributed DataFrame, allowing Spark to handle large-scale I/O and schema enforcement.
    b. **Per-Rating Random Assignment**: Assigned each rating to train/validation/test based on a reproducible random draw. This ensures that every user contributes interactions to all three splits, avoiding cold‑start users in validation or test.
    c. **Persistent Storage**: Saved each split in a compressed, columnar format (Parquet) on HDFS. This both reduced disk footprint and greatly sped up downstream queries.
    d. **Sanity-Check**: Verified that the three splits together summed to the full rating count and that their relative sizes were roughly 80 % train, 10 % validation, 10 % test.
    e. Overall code stored in **split_ratings.py**

4. Popularity-Baseline Recommendation & Evaluation:
    a. **Data & Parameters:** Loaded the train split and set two hyperparameters: a minimum-ratings threshold to drop rare movies and a pseudo-count β for smoothing.
    b. **Adjusted Popularity:** Counted distinct users per movie, filtered out low-count items, then applied β to damp each movie's raw count.
    c. **Global Top-100:** Ranked movies by the adjusted score and selected the top 100 as the universal recommendation list.
    d. **Evaluation:** For both validation and test splits, gathered each user's held-out movies and compared them to the top-100 list using Precision@100, MAP, and NDCG@100.
    e. Overall code stored in **popularity_baseline.py**

5. Latent Factor Model Recommendation & Evaluation:
    a. **Tag Feature Extraction**: Used the **Tag Genome** (genome-scores.csv) to create dense feature vectors for each movie, capturing semantic attributes like "thrilling", "romantic", etc.
    b. **Hyper-parameter Tuning**: Conducted a grid search over combinations of rank and regParam, selecting the model with the best **ranking-based performance** on the validation set. **Best Model:** rank = 30, regParam = 0.1
    c. **Regression Mapping**: Trained **30 linear regressors** (one for each ALS latent dimension) to map tag vectors → latent item embeddings, using movies that existed in the ALS training set.
    d. **Cold-Start Prediction**: For movies unseen during ALS training, predicted their embeddings using the trained regressors, enabling recommendation without prior rating data.
    e. **Evaluation**: Scored cold-start items for each user using the **dot product** between predicted item embeddings and ALS user vectors, then evaluated top-100 rankings using **Precision@100**, **NDCG@100**, and **MAP**
    f. Overall code stored in **hybrid_feature_enhanced_als.py** and **evaluate_hybrid_model.py**

Results

Top 100 Pairs

Csv files present on GitHub as **top100.csv**

Validation

Average correlation (Top-100): 0.2422

Average correlation (Random-100): 0.1118

Validation is passed, average correlation of top 100 pairs is higher.

Popularity Baseline:

Metric	**Test Set Score**	**Validation Set Score**
Precision@100	0.0182	0.0185
Mean Average Precision (MAP)	0.0382	0.0319
NDCG@100	0.1164	0.1090

- Precision@100 ≈ 0.0182
  Only about 1.8 of the 100 "most-popular" movies you recommend actually show up in each user's test set. That's low, but a popularity baseline isn't meant to be very accurate, it's just the floor.

- MAP ≈ 0.0382
  When you account for the fact that hits near the top of the list count more, you get a slightly higher number (3.8%). Again, small, because there's no personalization.

- NDCG@100 ≈ 0.1164
  This reflects how much value those few hits bring when discounted by position; scores around 0.1 – 0.2 are typical for a global-popularity list on this scale.

Latent Factor Model:

Metric	**Test Set Score**	**Validation Set Score**
Precision@100	0.0016	0.0016
Mean Average Precision (MAP)	0.0010	0.0012
NDCG@100	0.0068	0.0077

- Precision@100 ≈ 0.0016
  On average, ~0.16% of the top 100 recommendations are relevant (i.e., match

held-out user interactions).

- MAP ≈ 0.0010
  Across all users, the average quality of ranking relevant items is very modest but better than random.

- NDCG@100 ≈ 0.0068
  Relevant cold-start items occasionally appear near the top of the list, but generally rank low.

Conclusion

While the popularity baseline performs better in absolute terms, it cannot recommend cold-start items.
The hybrid model fills this gap by enabling recommendations for unseen items, a critical capability for any real-world recommender system dealing with new content.

Team Contribution

- Lavya: Calculating similar pairs, Validating top pairs using random pairs.
- Rithujaa: Data Partitioning into Train / Validation / Test, Popularity Baseline Evaluation
- Vrinda: Latent Factor Model Implementation and Evaluation

Challenges Faced

- Scaling issues were faced due to the size of the dataset and limited computational resources.
- Some randomly selected pairs had too few overlapping ratings for correlation.
- Mapping content-based features (from the tag genome) to ALS-style embeddings introduced column name ambiguity and required careful engineering to avoid Spark's errors related to duplicate or conflicting column names.
- In future work, we aim to improve scalability by consolidating regressors into a single multi-output model and enhance accuracy by incorporating additional content features such as genres or metadata from movie titles.