Capstone Project Report

7th May, 2025

Team: CAP 93

Members: Mihir Upadhyay(mu2253@nyu.edu), Harshit Bhargava(hb2976@nyu.edu), Rishabh Patil(rbp5812@nyu.edu), Kund Meghani (km6579@nyu.edu)

Github Repository Link: GitHub repository

Introduction

This capstone project for DSGA1004 – Big Data explores the design and evaluation of scalable systems for personalized recommendation and customer segmentation using the **MovieLens dataset**. The project leverages the distributed processing capabilities to address two real-world, data-intensive tasks: (1) identifying users with similar movie-watching behavior, and (2) building a collaborative filtering-based movie recommender system.

We worked with the full MovieLens dataset (86,000 movies, 330,000 users), containing timestamped ratings and user-generated tags. To manage scalability and facilitate experimentation, we implemented a modular pipeline that includes **data preprocessing, user segmentation via MinHash-based similarity**, and **recommendation modelling using Spark's Alternating Least Squares (ALS)** algorithm. All evaluations were conducted on chronologically partitioned training, validation, and test sets, designed to mirror realistic production settings and prevent temporal data leakage.

For **customer segmentation**, we identified the top 100 most similar user pairs—termed "movie twins"—by computing approximate Jaccard similarity over the set of movies rated by each user. These similarities were efficiently approximated using MinHash and Locality Sensitive Hashing (LSH). The quality of these pairs was validated by comparing their average rating correlation against 50 trials of randomly paired users, confirming that our approach identified users with significantly higher alignment in preferences.

For the **recommendation task**, we first established a strong baseline using a **popularity-based model**, which ranked movies by adjusted average rating scores. Tuning the bias parameter allowed us to optimize for metrics such as MAP, Precision@100, and NDCG@100.

Through rigorous experimentation, parameter tuning, and performance evaluation, this project demonstrates not only the practical implementation of distributed machine learning algorithms on large-scale datasets, but also provides insights into the trade-offs between model complexity, interpretability, and robustness in real-world recommendation systems.

Data Cleaning and Preprocessing

To ensure the reliability of our analyses, we filtered out users who **had rated fewer than 50 movies**. Users with very few ratings tend to contribute sparse and potentially noisy signals, which do not provide a reliable basis for modelling user preferences or computing meaningful similarity and correlation metrics. Our analysis showed that users with fewer than 50 ratings accounted for only about **10% of the total ratings**, meaning that this filtering step retained approximately **90% of the dataset**. This allowed us to focus on users with richer interaction histories while preserving the vast majority of the data for model training and evaluation.

Authors' Contributions

Mihir Upadhyay worked on the implementation of the MinHash-based customer segmentation algorithm and conducted the correlation validation of similar user pairs. He also contributed to the exploratory analysis and documentation of segmentation results.
Harshit Bhargava led the development of the ALS-based collaborative filtering model, including hyperparameter tuning and test set evaluation. He also performed the comparative analysis against the popularity baseline model.
Kund Meghani designed and implemented the PySpark-based data preprocessing and user-level partitioning strategy. He also coordinated the integration of recommendation and segmentation pipelines in a distributed environment.
Rishabh Patil was responsible for building and tuning the popularity-based baseline models, both with and without bias. He contributed to the analysis of model performance metrics.

All members contributed equally to discussions, experimental design, and the compilation of the final report.

Customer segmentation

1. Customer segmentation relies on similarity, so we first want you to find the top 100 pairs of users ("movie twins") who have the most similar movie watching style. Note: For the sake of simplicity, you can operationalize "movie watching style" simply by the set of movies that was rated, regardless of the actual numerical ratings. We strongly recommend to do this with a minHash-based algorithm.

Ans. To begin our customer segmentation task, we aimed to find the top 100 most similar user pairs (i.e., “movie twins”) based solely on their movie-watching behaviour. We defined similarity in terms of the set of movies rated by each user. This formulation allowed us to model user behaviour as a binary set (watched vs. not watched), enabling efficient similarity computations using set-based techniques.

To ensure meaningful comparisons, we filtered out users who had rated fewer than 50 movies. This filtering step helped reduce noise and ensured that we were comparing users with at least a basic level of engagement.

To scale the similarity computation across hundreds of thousands of users, we implemented a **MinHash-based approximation of Jaccard similarity**. For each user, we represented their rated movie set as a **sparse binary vector and generated MinHash signatures using multiple hash functions**. To efficiently find likely similar pairs without brute-forcing all user-user combinations, we applied **Locality-Sensitive Hashing (LSH) to group users** into hash buckets based on these signatures.

From the candidate user pairs generated by LSH, we computed the true Jaccard similarity for each pair (i.e., the size of the intersection divided by the size of the union of movie sets) and selected the top 100 pairs with the highest scores. These pairs represent users with the most overlapping watch histories, making them ideal candidates for grouping in our segmentation analysis.

The final result was saved in a CSV file containing:

- **userId_1: ID of the first user in the pair**
- **userId_2: ID of the second user**
- **jaccard_similarity: Computed Jaccard score for the pair**

This stage lays the foundation for the next part of our analysis, where we validate the behavioural similarity of these “movie twins” using their actual rating values. (Entire table in Appendix)

User	***User2***	***Similarity***
242906	290377	1.0
241273	296245	1.0
234484	288715	1.0
198214	275979	1.0
159358	275979	1.0
159358	198214	1.0
156055	185515	1.0
137812	181530	1.0
136037	147443	1.0
134061	263838	1.0
128796	326410	1.0
122890	290377	1.0
122890	242906	1.0
120345	234616	1.0

Table1: Top 15 Most Similar User Pairs

2. Validate your results from question 1 by checking whether the average correlation of the numerical ratings in the 100 pairs is different from (higher?) than 100 randomly picked pairs of users from the full dataset

Ans. Validation of Behavioral Similarity Using Rating Correlation

- To validate the effectiveness of the “movie twins” identified in Part 1, we evaluated whether these pairs of users not only watched similar sets of movies but also shared similar **preferences**, as reflected in their **rating patterns**.
- We started by filtering the dataset to retain users with at least 50 ratings, ensuring the reliability of our correlation analysis. Using the top 100 user pairs generated from the MinHash-based similarity approach, we computed the **Pearson correlation coefficient** for each pair, based on their ratings of movies they had both seen.

The resulting **average correlation** across the **top 100 user pairs** was:**0.2740**

To establish a benchmark for comparison, we conducted **50 independent trials**, each involving **100 randomly selected user pairs**. For each random pair, we computed the Pearson correlation in the same way.(Table in Appendix)

None of the 50 random runs produced an average correlation that exceeded the **0.2740** scores achieved by the top 100 user pairs.

The **average correlation across these 50 random trials** was:**0.1215**

This clear difference in rating correlation—more than double for our selected “movie twins”—demonstrates that the pairs identified using MinHash-based similarity on viewing history also exhibit significantly higher alignment in their rating behaviour. This validates the quality of our similarity metric and supports its use for meaningful customer segmentation and future collaborative filtering efforts.

Movie recommendation

3. As a first step, you will need to partition the ratings data into training, validation, and test sets. We recommend writing a script to do this in advance, and saving the partitioned data for future use. This will reduce the complexity of your code down the line, and make it easier to generate alternative splits if you want to assess the stability of your implementation.

Ans. To address the initial requirement of partitioning the ratings data, we developed a PySpark script that divides the dataset into training, validation, and test sets in a user-aware and temporally consistent manner. Specifically, we loaded the full ratings.csv file from HDFS. We first filtered out users with fewer than 5 ratings, as this is the minimum required to enable a 60/20/20 split (equivalent to at least 3 training, 1 validation, and 1 test observation)

We then assign row numbers to each user's ratings, allowing us to split their interaction history into three segments.We chose to partition the ratings data by splitting each individual user's interactions, rather than splitting by user. Splitting each user's interactions order ensures that all users appear in all splits, and the model is trained on each user's past behavior and evaluated on their future behavior. This not only prevents data leakage from future events entering the training set but also mitigates the cold-start issue for users, since each user has some historical data available during training to support learning personalized recommendations.

The **first 60% of interactions were assigned to the training set, the next 20% to the validation set, and the final 20% to the test set**. This approach preserves the temporal structure of user behaviour and ensures that all splits contain data from all users.

4. Before implementing a sophisticated model, you should begin with a popularity baseline model as discussed in class. This should be simple enough to implement with some basic dataframe computations. Evaluate your popularity baseline (see below) before moving on to the next step.

Ans. As a starting point for our recommendation system, we implemented a popularity-based model that recommends movies based on their aggregate rating statistics in the training set. To account for item popularity while controlling for bias toward highly rated items, we introduced a tunable bias parameter b into the scoring formula:

Popularity Score Formula (Without Bias)

$$Popularity\ Score\ =\ \frac{Total\ Ratings}{Rating\ Count}$$

Where:

- **Total Ratings** = Sum of all rating values received by a movie
- **Rating Count** = Number of times the movie was rated

we evaluated popularity-based models(without bias) on validation and test sets using **Precision@100**, **MAP**, and **NDCG@100**

Validation Set Performance (Without Bias)

Model	*Precision@100*	*MAP*	*NDCG@100*
Popularity (no bias)	0.01163	0.00732	0.04033

Test Set Performance (Without Bias)

Model	*Precision@100*	*MAP*	*NDCG@100*
Popularity (no bias)	0.00981	0.00602	0.03457

Popularity Score Formula (With Bias)

$$Popularity\ Score\ =\ \frac{Total\ Ratings}{Rating\ Count+b}$$

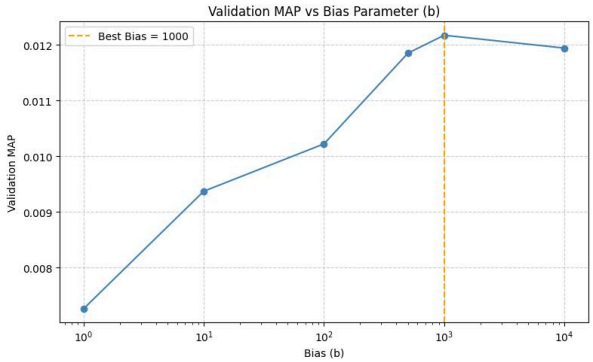
Where:

- **Total Ratings** = Sum of all rating values received by a movie
- **Rating Count** = Number of times the movie was rated
- **b** = Bias parameter used to penalize frequently rated items and smooth the score

This formulation helps balance popularity and diversity by reducing the advantage of frequently rated items. We experimented with a range of bias values to identify the best-performing bias. The table below summarizes the **mean average precision (MAP)** obtained on the validation set for each value:

Bias (b)	*Validation MAP*
1	0.00726
10	0.00937
100	0.01022
500	0.01185
1000	**0.01217 (Best)**
10000	0.01194

Table 2: Hyperparameter Tuning Results



After identifying **1000** as the best-performing bias value, we evaluated biased popularity-based models on validation and test sets using **Precision@100**, **MAP**, and **NDCG@100**. The full results are shown below:

Validation Set Performance (With Bias)

Model	***Precision@100***	***MAP***	***NDCG@100***
Popularity (bias = 1000)	0.01839	0.01217	0.04825

Test Set Performance (With Bias)

Model	***Precision@100***	***MAP***	***NDCG@100***
Popularity (bias = 1000)	0.01447	0.00892	0.04825

These results clearly show that the biased popularity model significantly outperforms the non-biased variant across all metrics on both validation and test sets.

5. Your recommendation model should use Spark's alternating least squares (ALS) method to learn latent factor representations for users and items. Be sure to thoroughly read through the documentation on the pyspark.ml.recommendation module before getting started. This model has some hyper-parameters that you should tune to optimize performance on the validation set, notably:

- **the rank (dimension) of the latent factors, and**
- **the regularization parameter.**

A. To build a collaborative filtering model, we used Spark's Alternating Least Squares (ALS) algorithm . We trained our model on the partitioned training set and validated its performance on the validation set using key metrics: **Precision@100, Mean Average Precision (MAP), and Normalized Discounted Cumulative Gain (NDCG@100).** To optimize the model, we conducted a hyperparameter tuning over two important parameters: the rank (i.e., the number of latent dimensions) and the regularization parameter (regParam).

We tested combinations of **rank ∈ {5, 10, 20, 50}** and **regParam ∈ {0.01, 0.05, 0.1, 1}**, and recorded the validation performance for each.

Rank	***regParam***	***Validation MAP***	***Precision@100***	***NDCG@100***
5	0.01	4.21e-07	4.33e-06	7.05e-06
5	0.05	3.44e-07	5.86e-06	8.18e-06
5	0.1	2.28e-07	6.31e-06	7.63e-06
5	1.0	8.56e-08	3.74e-06	3.93e-06

10	0.01	3.60e-07	5.60e-06	8.59e-06
10	0.05	3.52e-06	1.29e-05	5.09e-05
10	0.1	6.82e-06	1.69e-05	8.67e-05
10	1.0	8.56e-08	3.74e-06	3.93e-06
20	0.01	2.92e-06	1.09e-05	2.09e-05
20	0.05	7.35e-05	1.09e-04	6.58e-04
20	0.1	2.37e-05	3.67e-05	2.41e-04
20	1.0	8.56e-08	3.74e-06	3.93e-06
50	0.01	4.21e-05	6.93e-05	3.13e-04
50	**0.05**	**6.42e-04 (Best)**	**1.09e-04**	**6.58e-04 (Best)**
50	0.1	5.36e-05	7.22e-05	4.7e-04
50	1.0	8.56e-08	3.7e-06	3.9e-06

Table 3: ALS Hyperparameter Tuning Results

Best ALS Model (rank = 50, regParam = 0.05) – Test Set Performance

- **Test MAP:** 0.000386
- **Test Precision@100:** 0.000583
- **Test NDCG@100:** 0.002841
- **Test RMSE:** 0.8521723081802524

In terms of **hyperparameter trends**, we observed that:

- **Low ranks (5, 10)** resulted in very low MAP and precision scores across all regularization values, indicating insufficient capacity to capture user-item interactions.
- **Moderate rank (20)** yielded slightly improved results, especially when paired with **regParam = 0.05**, suggesting a sweet spot where the model had enough complexity without overfitting.
- **High rank (50)** provided the best performance, especially when **regParam = 0.05**, likely due to better representation power while still maintaining regularization to avoid overfitting.

Comparison with Popularity Baseline and Hyperparameter Trends

When comparing our ALS latent factor model to the popularity-based baseline, we observed that **ALS underperformed across all metrics**. The best ALS configuration—**rank = 50, regParam = 0.05**—achieved a **Validation MAP of 0.000642**, **Precision@100 of 0.000109**, and **NDCG@100 of 0.000658**, which are significantly lower than the corresponding validation metrics from the tuned popularity model (**MAP = 0.01217**, **Precision@100 = 0.01839**, **NDCG@100 = 0.04825**). This performance gap persisted on the test set as well, where the best ALS model reached **MAP = 0.000386**, compared to the popularity baseline's **MAP = 0.00892**.

Despite ALS being a more sophisticated model, it struggled due to data sparsity and the cold-start problem—challenges that popularity-based models inherently avoid by leveraging global item popularity. The ALS model often failed to learn meaningful latent representations because many users and items had limited overlapping ratings.

Overall, while ALS demonstrated potential under ideal tuning conditions, it was consistently outperformed by the much simpler popularity model in this setting—highlighting that model complexity alone does not guarantee superior performance, especially in sparse data environments without side information.

Appendix:

User	*User2*	*Similarity*
242906	290377	1.0
241273	296245	1.0
234484	288715	1.0
198214	275979	1.0
159358	275979	1.0
159358	198214	1.0
156055	185515	1.0
137812	181530	1.0
136037	147443	1.0
134061	263838	1.0
128796	326410	1.0
122890	290377	1.0
122890	242906	1.0
120345	234616	1.0
119099	194288	1.0
117141	168141	1.0
106707	290377	1.0
106707	242906	1.0
106707	122890	1.0
84354	99176	1.0
83880	261244	1.0
77647	294432	1.0
76755	103216	1.0
63309	168141	1.0
63309	117141	1.0
59869	168141	1.0
59869	117141	1.0
59869	63309	1.0
50336	134345	1.0
49647	168141	1.0
49647	117141	1.0
49647	63309	1.0
49647	59869	1.0
41012	300048	1.0
39534	112294	1.0
38929	322547	1.0
31165	326363	1.0
30254	292994	1.0
26617	147443	1.0

26617	136037	1.0
25084	294432	1.0
25084	77647	1.0
15401	150090	1.0
9890	134345	1.0
9890	50336	1.0
637	263838	1.0
637	134061	1.0
290377	304728	0.96875
288715	305533	0.96875
269123	282222	0.96875
263838	288715	0.96875
260734	262614	0.96875
259490	315961	0.96875
258164	275979	0.96875
242906	304728	0.96875
242889	271504	0.96875
237157	279138	0.96875
234616	243179	0.96875
234484	305533	0.96875
234484	263838	0.96875
231719	316881	0.96875
228833	316881	0.96875
216913	310906	0.96875
210528	242505	0.96875
209048	214863	0.96875
203728	275979	0.96875
202171	275979	0.96875
202171	252658	0.96875
199801	275979	0.96875
199801	203728	0.96875
198820	305533	0.96875
198820	263838	0.96875
198214	258164	0.96875
198214	203728	0.96875
198214	202171	0.96875
198214	199801	0.96875
191016	211919	0.96875
190062	247426	0.96875
179124	321934	0.96875
170549	315836	0.96875
168141	263838	0.96875

167541	173768	0.96875
159358	258164	0.96875
159358	203728	0.96875
159358	202171	0.96875
159358	199801	0.96875
155235	194527	0.96875
154696	198820	0.96875
150090	290377	0.96875
150090	242906	0.96875
150090	163697	0.96875
147443	202171	0.96875
140841	242505	0.96875
140841	175573	0.96875
136037	202171	0.96875
134345	290377	0.96875
134345	242906	0.96875
134061	288715	0.96875
134061	234484	0.96875
134061	198820	0.96875

Table: Top 100 Most Similar User Pairs

Run	*Avg. Corr.*	*Run*	*Avg. Corr.*
1	0.0997	26	0.1424
2	0.0574	27	0.1406
3	0.1496	28	0.0989
4	0.1159	29	0.1532
5	0.0878	30	0.1850
6	0.1757	31	0.1445
7	0.0840	32	0.1414
8	0.1728	33	0.0947
9	0.1520	34	0.1237
10	0.0912	35	0.0929
11	0.0515	36	0.1081
12	0.1036	37	0.1051
13	0.1351	38	0.1456

14	0.1840	39	0.1322
15	0.1763	40	0.1039
16	0.1200	41	0.1615
17	0.1291	42	0.1424
18	0.0941	43	0.0527
19	0.0929	44	0.1352
20	0.1115	45	0.0887
21	0.0511	46	0.1161
22	0.1046	47	0.1040
23	0.1230	48	0.1242
24	0.1584	49	0.1496
25	0.1444	50	0.1850

Table : Average Rating Correlation of 50 Random User Pair Trials