Capstone Project

Members:	Kyeongmo Kang, Alexander Pegot-Ogier, Nikolas Prasinos
NetIDs:	kk5739, ap9283, np3106
Github repository link:	`https://github.com/nyu-big-data/capstone-bdcs-94.git`

Q1. Customer Segmentation with MinHashLSH

To understand user similarity in viewing behavior, we developed a customer segmentation pipeline using MinHash and Locality Sensitive Hashing (LSH). Our goal in this phase was to identify the top 100 user pairs who had rated the most overlapping sets of movies, regardless of the actual ratings. We refer to these highly overlapping users as "movie-twins," defining their similarity using the estimated Jaccard similarity of their rated movie sets.

Data Preparation

The preprocessing pipeline starts with converting raw CSV files into the column-oriented Parquet format using a separate script, `parquet_convertor.py`. This conversion is crucial for efficient large-scale data processing, as Parquet provides faster reads, better compression, and reduced storage footprint compared to row-based formats like CSV.

Next, the script reads the ratings data from Parquet files and casts the `userId` and `movieId` columns to integers, while the `rating` column is cast to double precision. It then filters for "active" users — those with at least 2500 ratings. This threshold reduces noise by focusing the analysis on users who are more likely to exhibit meaningful, consistent movie preferences. The filtering is achieved by grouping the data by `userId` and applying a count filter, ensuring that only high-activity users are included in the similarity calculations.

Sparse Vector Representation

To efficiently calculate approximate Jaccard similarities, the movie sets for each active user are converted into sparse vectors. This step is critical because the `MinHashLSH` model used for approximate similarity search requires vectorized inputs. The transformation is handled by a custom user-defined function (UDF) generated by the `to_sparse_factory` function, which creates sparse vectors of dimensionality equal to the maximum observed movie ID plus one. This approach captures each user's movie preferences while minimizing memory overhead.

LSH Model and Hyperparameters

For the MinHashLSH model, we used the following hyperparameters:

- **Distance Threshold:** 1.0 — captures all possible pairs, maximizing recall at the cost of potentially including some false positives.
- **Minimum Ratings:** 2500 — filters out low-activity users to reduce noise.
- **Number of Hash Tables:** 5 — balances precision and recall, but potentially increases false positives by reducing the strictness of hash collisions.

While these choices are not necessarily the most optimal, they were selected to balance speed and accuracy, allowing the pipeline to run efficiently on a large-scale dataset. For example, a lower threshold, such as 0.4, would reduce the number of false positive pairs by requiring a higher degree of overlap. Similarly, increasing the number of hash tables (or "blocks") would reduce false positives by providing a more fine-grained approximation of Jaccard similarity, though at the cost of increased memory and computation.

Pair Filtering and Final Selection

Once the data is vectorized, the script uses the `MinHashLSH` model to identify similar user pairs. It ensures that each pair is included only once by enforcing the constraint that the first user ID is always less than the second (`user_1 < user_2`). The script then computes Jaccard similarity for each pair and calculates the common movie count using the `array_intersect` function, selecting the top 100 most similar pairs based on descending Jaccard similarity and common count.

Link to the full table: `https://github.com/nyu-big-data/capstone-bdcs-94/blob/main/results/q1.csv`

Table 1: Top-100 most similar user pairs (highest estimated Jaccard).

Rank	User 1	User 2	Common Movies	Estimated Jaccard
1	77647	294432	3273	0.9979
2	25084	86967	3218	0.9844
3	25084	294432	3218	0.9814
4	86967	294432	3235	0.9800
			...	
97	101039	229135	1638	0.4601
98	31997	48689	1709	0.4598
99	243183	249280	2100	0.4594
100	233071	307430	2099	0.4582

Q2. Validating Movie-Twin Similarity Using Pearson Correlation

While the MinHash and LSH-based segmentation in Q1 effectively identified user pairs with highly overlapping movie sets, it does not consider the actual ratings those users gave. To validate the quality of these "movie-twin" pairs, we computed the Pearson correlation coefficient for each pair's ratings on their shared movies. This approach checks whether similar viewing histories also reflect similar rating patterns.

For each of the 100 most similar user pairs identified in Q1, we extracted their overlapping movies and calculated the Pearson correlation. Pairs with fewer than two shared ratings or constant rating behavior (e.g., only 5s) were excluded to ensure statistical reliability.

As a baseline, we repeated this process for 100 randomly sampled user pairs to establish a point of comparison. This allowed us to confirm that the MinHash-based pairs exhibit stronger rating alignment than arbitrary pairs.

Table 2: Average Pearson Correlation of User Pairs

Pair Group	Average Pearson Correlation
Top 100 (LSH-based)	0.7885
100 Random Pairs	0.1523

The results indicate that the top 100 pairs identified through MinHash+LSH have significantly higher average correlation than random pairs, suggesting that these user pairs not only watched similar content but also rated it in more consistent ways.

Q3. Data Splitting–Creating Train, Validation, and Test Sets

Accurate model evaluation requires a realistic split of the user-item interaction data. To achieve this, we performed a **user-wise train/validation/test split** that preserves the temporal order of ratings, ensuring the model is evaluated on unseen data for each user.

Preprocessing and Filtering

To reduce noise and improve statistical reliability, we excluded users with fewer than 500 total ratings from the train, validation, and test sets. This threshold focuses the analysis on high-activity users who exhibit more consistent preferences. The filtering step was implemented in Spark, grouping the data by `userId` and applying a count filter to retain only users meeting this minimum threshold.

Splitting Logic

The splitting logic assigns ratings as follows:

- 80% of each active user's ratings to the training set,
- 10% to the validation set,
- 10% to the test set.

A fixed random seed was used to ensure reproducibility, and a partitioning scheme was applied to optimize the write performance. The split assignment is based on a per-user random shuffle, ensuring that each user's ratings are divided independently, which is crucial for collaborative filtering models like ALS.

Hyperparameters and Output

The main hyperparameters for this process include:

- `train_ratio` = 0.8,
- `val_ratio` = 0.1,
- `test_ratio` = 0.1,
- `min_ratings` = 500

The resulting splits were written to Parquet format for efficient downstream processing. The distribution of ratings and unique users across the splits is summarized below:

Table 3: Number of Ratings and Unique Users in Each Data Split

Split	**Number of Ratings**	**Number of Users**
Training Set	9,655,316	12,771
Validation Set	1,206,922	12,771
Test Set	1,200,576	12,771

This user-stratified approach ensures that all users in the validation and test sets also appear in the training set, a critical requirement for collaborative filtering algorithms like ALS.

Q4. Baseline Evaluation Using Popularity-Based Recommendation

To establish a simple performance benchmark, we implemented a popularity-based recommendation model. This approach ranks movies solely based on their overall frequency in the training set, ignoring user-specific preferences. While straightforward, it provides a useful baseline for assessing the effectiveness of more sophisticated, personalized algorithms.

Preprocessing and Filtering

After loading the train, validation, and test splits (see Table 3), we first identified the 100 most frequently rated movies from the training set. This step uses the full set of training ratings to calculate global popularity, without any per-user adjustments. To ensure consistency with the ALS model, we only included users with at least one rated movie in each split, filtering out users with no interactions.

Evaluation Metrics

We evaluated the popularity-based model using the following standard ranking metrics:

- **Precision@100**: The proportion of relevant items among the top 100 recommendations.
- **MAP@100**: Mean Average Precision at rank 100, reflecting both precision and rank sensitivity.
- **NDCG@100**: Normalized Discounted Cumulative Gain, which weights relevant items based on their rank.

These metrics were computed using Spark's `RankingMetrics` API to assess the relevance of the top 100 recommended movies against each user's actual rated movies in the validation and test sets.

Table 4: Performance of the Popularity-Based Baseline on Validation and Test Sets

Metric	**Test Set**
Precision@100	0.0729
MAP@100	0.0136
NDCG@100	0.0936

Q5. Collaborative Filtering with ALS

After exploring segmentation-based approaches and popularity-based baselines, we implemented a collaborative filtering model using Alternating Least Squares (ALS), a widely used matrix factorization technique for recommendation systems. ALS is particularly well-suited for large-scale, sparse datasets like MovieLens, as it efficiently handles missing entries and supports parallel computation in Spark.

Preprocessing and Data Splitting

We began by loading the preprocessed train, validation, and test splits (see Table 3), ensuring that each split contained the same set of active users. The ALS model requires that all user and item IDs in the validation and test sets also appear in the training set, aligning with our earlier data splitting strategy.

Hyperparameter Tuning

ALS has several critical hyperparameters that significantly impact its performance:

- **Rank** (latent factors): 50, 100, 160, 200
- **Regularization Parameter** (`regParam`): 0.05, 0.1
- **Max Iterations** (`maxIter`): 10, 20
- **Cold Start Strategy**: "drop" (to avoid evaluating unknown items)

We performed grid search over these parameters, selecting the model with the lowest validation RMSE as the best configuration. This approach balances the model's ability to generalize while avoiding overfitting. We found that increasing the rank beyond 200 led to diminishing returns, suggesting that most user-item interactions in this dataset can be effectively captured with a smaller latent space.

Best Model Configuration

- **Rank:** 100
- **regParam:** 0.05
- **maxIter:** 20
- **Validation RMSE:** 0.7344

Evaluation Metrics We evaluated the best ALS model on the test set using two types of metrics:

- **Regression Metrics:** RMSE to measure rating prediction accuracy.
- **Ranking Metrics:** Precision@100, MAP@100, and NDCG@100 to assess the quality of top-N recommendations.

Table 5: Performance of the ALS Model on the Test Set

Metric	**Value**
RMSE	0.7345
Precision@100	0.0119
MAP@100	0.0009
NDCG@100	0.0137

These results indicate that while ALS is effective at predicting individual ratings, it struggles to surface the most relevant items when evaluated in a top-N recommendation setting. This gap is typical for matrix factorization models, which optimize for rating prediction rather than ranking.

Potential Improvements To further improve this model, future work could consider:

- **Incorporating Implicit Feedback:** Using side information from `tags.csv` or user interaction frequency, which ALS can handle through implicit matrix factorization.
- **Expanded Hyperparameter Tuning:** Testing a wider range of ranks, regularization values, and iteration counts.

Together, these extensions could significantly enhance both the accuracy and relevance of the recommendations, moving beyond the limitations of purely collaborative filtering meth

Software Components and Installation

This project was developed in Python using several open-source libraries. Below is a summary of key software components used across our deliverables, along with installation guidance:

- **PySpark**: Used for implementing and evaluating the ALS collaborative filtering model and computing ranking metrics such as Precision@100, MAP@100, and NDCG@100. `pip install pyspark`
- **scikit-learn**: Used for splitting data into train, validation, and test sets in a reproducible and user-wise manner.
  `pip install scikit-learn`
- **NumPy**: Used for computing Pearson correlation coefficients between movie-twin rating vectors.
  `pip install numpy`
- **pandas**: Used for data manipulation, particularly for preprocessing the ratings and formatting outputs.
  `pip install pandas`
- **datasketch**: Used to construct MinHash signatures and perform efficient similarity search using MinHashLSH.
  `pip install datasketch`

All scripts were written to run on Python 3.x and can be executed in standard Python or PySpark environments. Each deliverable corresponds to the following core scripts:

- `customer_segmentation.py` – Implements MinHash and LSH to find highly similar user pairs based on movie overlap and calculates the correlations. (Question 1 and 2).
- `split.py` – Performs reproducible user-wise train/validation/test splits (Question 3).
- `popularity_model.py` – Computes and evaluates a non-personalized popularity-based recommendation baseline (Question 4).
- `als_model.py` – Trains the ALS collaborative filtering model, tunes hyperparameters, and evaluates performance (Question 5).
- `parquet_convertor.py` – Optional utility script to convert CSV files into Parquet format for optimized I/O.

Team Contributions

All members of Team 94 collaborated closely on the capstone project, providing mutual support throughout all phases. While we distributed specific tasks to ensure efficiency and accountability, we reviewed and debugged each other's work regularly. Below is a breakdown of primary responsibilities based on the five required deliverables:

- **Kyeongmo Kang** was responsible for Deliverable 3, where he implemented the user-wise train/validation/test data splitting strategy. He also took charge of writing and structuring the documentation across all five sections (D1–D5), ensuring consistency, clarity, and completeness of the final report.
- **Alexander Pegot-Ogier** led Deliverables 1 and 2. He implemented the MinHash and Locality Sensitive Hashing (LSH) pipeline to identify the top 100 most similar user pairs based on their rated movie sets. He also validated these pairs by computing Pearson correlation coefficients and comparing them against a control group of random pairs.
- **Nikolas Prasinos** led Deliverables 4 and 5. He implemented the popularity-based baseline model and evaluated it using ranking metrics (Precision@100, MAP@100, NDCG@100). He then developed the collaborative filtering model using Spark's ALS, performed hyperparameter tuning, and evaluated its predictive and ranking performance.

Despite the division of labor, all members contributed to code review, testing, and interpretation of results across the project. This collaborative workflow ensured high-quality implementation and a cohesive final submission.