Big Data: Capstone Project*

Tien-Leng Wu, Neil Chen, Amy Lin, Sunny Son

Abstract

It is our honour to have the opportunity to build a recommender system as well as a customer segmentation system in our capstone project. This report begins by describing data preprocessing and the cluster we choose to use. Subsequently, we address each question using our proposed methods, explaining the rationale behind and implementation of these methods for each question. This capstone project was completed by Tien-Leng Wu, Neil Chen, Amy Lin, and Sunny Son. Specifically, Tien-Leng Wu is responsible for implementing the code for every section. Neil Chen is responsible for debugging the code and executing the code for every section. Amy Lin examines implementation details and proposes improvements on code implementation. Sunny Son ensures that our project remained on suggested timeline. All team members contribute to composing and refining the final report.

0 Data Preprocessing & Cluster Choice

At the outset we convert the raw MovieLens data (CSV) to Parquet, a column-oriented format optimised for Spark. Although the dataset ships with several CSV files, our pipeline relies exclusively on `ratings.csv`, which contains the (`userId, movieId, rating`) triples required for both the customer-similarity and recommendation tasks. The remaining data (tags, titles, genres, etc.) are not strictly necessary for the objectives specified in the capstone brief, but it is preserved for potential future extensions.

All Spark jobs are executed on **BigPurple***, NYU Langone's flagship high-performance computing cluster. BigPurple is a hybrid system with 157 CPU nodes, 376 NVIDIA V100 / A100 GPUs, 200 Gb HDR Infiniband, 14PB Spectrum Scale storage, whose high memory nodes and fast scratch tier prevent I/O or RAM from limiting MinHash and ALS workloads.

1 Customer Segmentation

1.1 Top 100 pairs

The first segmentation task asks for the *100 most-similar pairs of users*. Because every user is naturally represented by the **set of movies they have rated**, the appropriate similarity measure is the *Jaccard index*

$$\text{sim}(u, v) = \frac{|M_u \cap M_v|}{|M_u \cup M_v|}, \qquad M_u = \{\text{movies rated by user } u\}.$$

Computing this quantity exactly for the full MovieLens graph would require $\mathcal{O}(|\text{users}|^2)$ set intersections—prohibitively expensive at scale. Instead we adopt a standard **MinHash + Locality-Sensitive Hashing (LSH)** pipeline that gives a tight *approximation* to the Jaccard index with sub-quadratic cost.

1. **Movie sets per user.** We group the `ratings` table by `userId` and collect the distinct `movieId` values into a set (`collect_set`).
2. **Deterministic hashing to fixed-length vectors.** Each movie identifier is converted to a string token and passed through Spark ML's `HashingTF`† with 2^{18} buckets, giving a sparse binary vector that indicates the presence of each movie. (Using the same hashing function guarantees that identical movies map to the same position for all users.)
3. **MinHash signatures.** A `MinHashLSH` model with `numHashTables = 5`‡ is trained on those vectors, producing a compact signature per user whose expected collision probability equals their Jaccard similarity.

*Team 78, GitHub: https://github.com/nyu-big-data/capstone-bdcs-78
*Official website: https://med.nyu.edu/research/scientific-cores-shared-resources/high-performance-computing-core
†`HashingTF`: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.HashingTF.html
‡`MinHashLSH`: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.MinHashLSH.html

4. **Approximate all-pairs join.** Calling `approxSimilarityJoin` (with `threshold = 0.2`) on the MinHash model performs an efficient self-join that returns *candidate* user pairs together with their estimated Jaccard *distance*. We keep only one orientation of each pair (`datasetA.userId < datasetB.userId`).

5. **Ranking and selection.** Similarity is obtained as $(1 - \text{distance})$; the result is sorted in descending order.

This LSH-based strategy reduces the search space dramatically while preserving the quality of the nearest-neighbour set with high probability, making it feasible to compute customer similarity on the full dataset within interactive time.

Our result can be accessed through this link:

https://github.com/nyu-big-data/capstone-bdcs-78/tree/main/output/top_100_pairs

According to the result above, the similarities of all 100 pairs are 1. We suspect that there are more than 100 pairs that have similarity euqals to 1. To probe deeper, we rerun the MinHash pipeline[§] while

- restricting the pool to **active users** (those who have rated at least 2,000 movies), and
- retaining only user pairs with a **meaningful overlap** of at least 10 co-rated movies.

The new ranking is available at:

https://github.com/nyu-big-data/capstone-bdcs-78/tree/main/output-2000/top_100_pairs

It shows that perfect matches disappear; the top-100 Jaccard scores now range from 0.9979 down to 0.5096. In other words, imposing tighter activity and overlap thresholds produces a markedly different and more informative similarity score.

1.2 Validation of Top 100 Pairs

The MinHash–LSH pipeline delivers an *approximate* Jaccard ranking, so we must verify that the selected pairs are indeed highly similar in terms of their *rating behaviour*. For that purpose we compute, for every pair (u, v) returned in Section 1.1, the **Pearson correlation** of their common ratings and contrast the result with the random-pair baseline.

Exact similarity metric.

1. We reshape the `ratings` table into one map per user (`movieId → rating`). This yields a sparse user–vector that preserves the original 0–5 star values.
2. For a candidate pair we extract the set of movies rated by *both* users. If the overlap has fewer than two movies the Pearson coefficient is undefined and we record a `null`. When all ratings are identical (zero variance) we follow recommender- literature practice and set the coefficient to 0.

Average correlation of the Top 100. This is our primary figure of interest, which serves as a quantitative validation of the MinHash ranking.

Random-pair baseline. 100 pairs are drawn uniformly at random from the user set (with seed = 78).

The result is shown as follows:

```
Avg Pearson (top pairs) = 0.0720  |  Avg Pearson (random) = 0.2531
```

Unexpectedly, the average correlation for the MinHash pairs is *lower* than for a random sample. Two artefacts can account for this:

- **Extremely sparse users.** Many users have rated only a handful of films; when MinHash compares such tiny sets, hash collisions can spuriously suggest high Jaccard similarity.
- **Insufficient overlap.** Several user pairs share just one or two movies—far too few to yield a reliable Pearson estimate.

[§]Minor attributes & parameters adjustments: `numFeatures`: $2^{18} \to 2^{20}$ in `HashingTF`, `numHashTables`: $5 \to 15$ in `MinHashLSH`, and `threshold`: $0.2 \to 1$ in `approxSimilarityJoin`

Similar to the previous section, we impose the constraints (by only keeping the active users and meaningful overlap) and compute the average correlation again. This time, the result is the following:

```
Avg Pearson (top pairs) = 0.9036  |  Avg Pearson (random) = 0.2436
```

The new outcome contrasts sharply with the previous one: the average correlation for the top pairs soars from below 0.1 to above 0.9. We believe this finding is far more plausible—when two users have rated many of the same films, one would expect their tastes (and thus their ratings) to align, yielding a high Pearson coefficient.

2 Movie Recommendation

2.1 Data Partitioning

Our goal is to obtain three *reproducible*, *balanced* subsets (train / validation / test) that every experiment can reuse.

1. Load the master table. We start from the Parquet copy of `ratings.csv`(`userId`, `movieId`, `rating`) that we created during preprocessing.

2. Balance helper (cold-start guard). Rows whose user or movie occurs fewer than five times are discarded:

```
user_counts = ratings.groupBy("userId").count().filter("count >= 5")
movie_counts = ratings.groupBy("movieId").count().filter("count >= 5")
ratings = (ratings
.join(user_counts, "userId")
.join(movie_counts, "movieId")
.select("userId", "movieId", "rating"))
```

- Removes extreme sparsity that can destabilise ALS optimisation.
- Guarantees every user and every movie remains visible in all three splits, preventing accidental cold-start during evaluation.

3. 60/20/20 random split. With a fixed seed (78) each interaction is sampled in 60% train, 20% validation, and 20% test:

```
train, val, test = ratings.randomSplit([0.6, 0.2, 0.2], seed=78)
```

Because the ≥ 5 filter already enforces a minimal history per user, a simple global split is statistically sound and lighter than leave-k-out alternatives in Spark.

This pipeline therefore delivers compact, balanced and perfectly reproducible datasets for reliable hyper-parameter tuning and final benchmarking.

2.2 Baseline

Our first yard-stick is the *bias–adjusted popularity* baseline introduced in class, which augments raw popularity with additive bias terms:

$$\widehat{R}_{ui} \;=\; \mu \;+\; b_i \;+\; b_u.$$

Global component μ. The damped global mean captures the overall rating level while limiting the influence of outliers:

$$\mu \;=\; \frac{\sum_{(u,i)\in\mathcal{T}} r_{ui}}{|\mathcal{T}| + \beta_g}, \qquad \beta_g = 25.$$

Item bias b_i. Each movie receives its own offset that expresses how well it is rated *on average*:

$$b_i \;=\; \frac{\sum_{u\in\mathcal{U}_i} (r_{ui} - \mu)}{|\mathcal{U}_i| + \beta_i}, \qquad \beta_i = 25.$$

User bias b_u. Analogously we correct users who systematically rate higher or lower than average:

$$b_u = \frac{\sum_{i \in \mathcal{I}_u} (r_{ui} - \mu) - \sum_{i \in \mathcal{I}_u} b_i}{|\mathcal{I}_u| + \beta_u}, \qquad \beta_u = 25.$$

Large damping constants $(\beta_g, \beta_i, \beta_u)$ shrink estimates towards zero, protecting items or users with only a few observations from extreme (and unreliable) biases. The βs are set to 25 in default, but we choose (25, 25, 50) for our baseline.

Ranking strategy. For a fixed user u the personal term b_u is constant, so ranking reduces to sorting items by b_i. Consequently *all* users see the same top-k list:

$$\text{rank}_u(i) \equiv b_i.$$

Evaluation protocol. We adopt the ranking metrics (from `RankingEvaluator`)[¶] used throughout the project and report them for **both** validation and held-out test splits:

1. **MAP@k** — mean average precision at the chosen cut–off k;
2. **NDCG@k** — normalised discounted cumulative gain at k.

The accompanying Spark routine

1. trains the bias model on the *train* partition;
2. generates the universal top-k recommendation list;
3. evaluates on *validation* and *test* partitions in turn, printing both MAP@k and NDCG@k.

These scores furnish the performance bar that our ALS must be comparable with. The results are the following:

`VALIDATION-SET MAP@k = 0.0086 | NDCG@k = 0.0496`

`TEST-SET MAP@k = 0.0085 | NDCG@k = 0.0499`

2.3 Alternating Least Squares

To move beyond a popularity baseline we factorise the sparse user–item matrix with implicit *Alternating Least Squares* (ALS). Formally we seek two low-rank matrices $U \in R^{|\mathcal{U}| \times k}$ and $V \in R^{|\mathcal{I}| \times k}$ such that

$$\widehat{R} = U V^\top \qquad \Longleftrightarrow \qquad \widehat{r}_{ui} = \langle \mathbf{u}_u, \mathbf{v}_i \rangle,$$

and minimise the regularised squared-error objective

$$\min_{U,V} \sum_{(u,i) \in \mathcal{T}} \big(r_{ui} - \langle \mathbf{u}_u, \mathbf{v}_i \rangle\big)^2 + \lambda \left(\|U\|_F^2 + \|V\|_F^2\right),$$

where λ is the Tikhonov (ridge) penalty and $k = \textit{rank}$ controls model capacity. Spark ML's implementation alternates closed-form updates for U and V until convergence (We use `maxIter = 20`).[‖]

Practical details.

- **Cold-start strategy.** Predictions that involve an unseen user or item are dropped (`coldStartStrategy="drop"`), preventing artificial deflation of ranking metrics.
- **Non-negativity.** We allow latent factors to take on negative values (`nonnegative=False`) after empirical tests showed no benefit from the constraint.

[¶]: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.evaluation.RankingEvaluator.html
[‖] ALS: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.recommendation.ALS.html

Hyper-parameter search. A coarse grid is explored:

$$k \in \{10, 20, 40, 80, 160\}, \qquad \lambda \in \{0.001, 0.005, 0.01, 0.05, 0.1\}.$$

For every (k, λ) pair we train on the *train* split and compute **MAP@**k on the *validation* split. The model with the highest MAP is retained and additionally scored with **NDCG@**k to gauge ranking quality at the head of the list. The following $(k^\star, \lambda^\star)$ pair is chosen:

$$\texttt{rank=}k^\star\texttt{=160} \quad | \quad \texttt{reg=}\lambda^\star\texttt{=0.05}$$

The MAP@k and NDCG@k are as follows:

$$\texttt{MAP@}k\texttt{ = 0.01} \quad | \quad \texttt{NDCG@}k\texttt{ = 0.0467}$$

Final test evaluation. Using the chosen $(k^\star, \lambda^\star)$ the model is retrained on the combined *train+val* data and evaluated on the held-out *test* partition:

$$\texttt{TEST-SET MAP@}k\texttt{ = 0.01} \quad | \quad \texttt{NDCG@}k\texttt{ = 0.0468}$$

We can see that the ALS surpasses the baseline model in MAP@k when evaluating the test set after hyper-parameter tuning, although the ALS performs slightly worse than the baseline in NDCG@k.

3 Future Work

Although our implementation for each question (section) is carefully crafted, each could be extended for refined, listed as follows:

Optimizing MinHash pipeline. Although our MinHash pipeline yields solid results, its computational cost remains prohibitive; further optimisation is needed to bring the runtime down. This is a concern for ALS as well.

Definitions for active users and meaningful overlap. Currently, our thresholds for active users and meaningful overlap are 2,000 and 10 respectively. Varying the thresholds for them may yield different and/or more reliable results.

Data Partitioning. 3 possible avenues can be explored to refine our data partitioning.

- **Threshold modification.** Our current method removes rows whose user and movie occurs fewer than five times. Future experiments will raise this threshold (e.g. 10 or 20).
- **Richer feature space.** Only `ratings.csv` is consumed at present. Incorporating data from the companion files (genres, timestamps, tags, titles) may support subsequent analyses.
- **Smarter splits.** A temporal or leave–one-user-out split would more closely imitate an online recommender's deployment scenario than a simple i.i.d. random split.

β search for baseline. Our βs are fixed for our baseline. A grid search or Bayesian optimisation over β-triples could yield a stronger baseline.

Expanded hyper-parameter grid for ALS. Additional ranks and regularization strengths for ALS will be explored.

Acknowledgments

We thank the entire Big Data course instructional team—specifically Prof. Pascal Wallisch, the lab leaders, and the tutors—for their clear guidance on both the underlying concepts and practical implementations that enabled us to complete this project seamlessly.