Collaborative Filtering for Book Recommendations

Sara Mancini - 66458A

September 9, 2025

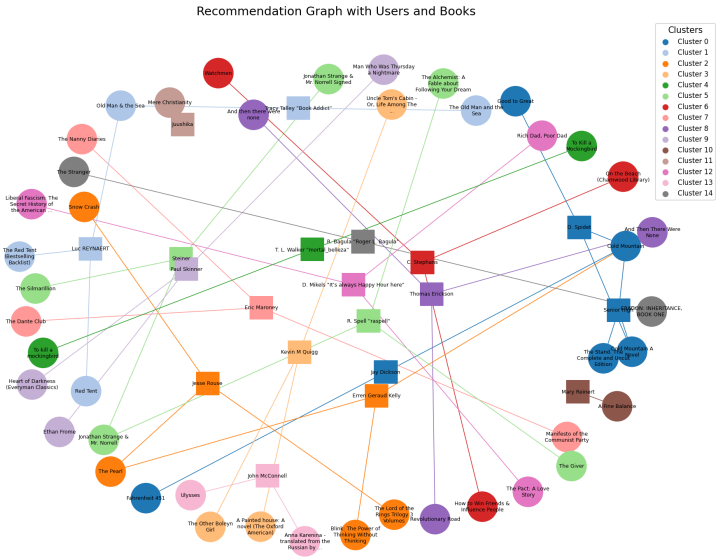

Figure 1: Graph of top-3 recommendations for the top-20 users

Disclaimer: "I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. No generative AI tool has been used to write the code or the report content."

1 Introduction

This report explores the implementation and evaluation of a recommendation system based on collaborative filtering, using the Alternating Least Squares (ALS) matrix factorization algorithm, implemented from scratch. Two solutions were differentiated in order to evaluate scalability: (i) a standard Python version on a substantially subsampled dataset, and (ii) a large-scale distributed version using Apache Spark 3.5.1.

Both models were trained on a subset of user-book rating data and evaluated against random and popularity baselines. Raw and latent spaces are compared using dimensionality reduction (PCA), cluster analysis (Figure 1), and performance metrics interpretation.

2 Data gathering and preprocessing

2.1 Data Acquisition and Cleaning

The dataset used in this project was sourced from Kaggle, specifically the *Amazon Books Reviews* dataset [1]. Access to the dataset was obtained via the Kaggle API, which was configured within the Google Colab environment by uploading the user authentication key (`kaggle.json`). Once the environment was prepared, the dataset was downloaded and unzipped into the working directory.

From the provided files, the main ratings file `Books_rating.csv` was loaded into a Pandas DataFrame, comprised of 2437900 rows. Since the raw dataset contained a wide range of metadata, only the columns directly relevant to collaborative filtering were retained. These include the user identifier (`User_id`), the associated profile name (`profileName`), the book identifier (`Id`), the title of the book (`Title`), and the rating score (`review/score`). To ensure consistency and avoid potential errors in later stages, rows containing missing values were discarded.

For the Spark-based implementation, the Pandas DataFrame was then converted into a Spark DataFrame to leverage distributed operations.

2.2 Subsampling and Normalization

To make the dataset more manageable and improve the quality of the recommendation system, two different subsampling strategies were applied depending on the framework.

Python Version (Subsampled Dataset)

A hybrid subsampling strategy (enabled via a `USE_SUBSAMPLE` flag) was applied.

First, active users and popular books were filtered, requiring each user and book to have at least 20 ratings. This reduces the risk of having extremely

sparse rows or columns in the utility matrix, that would make learning unreliable.

Next, the dataset was further limited to the top 1000 users and top 1000 books by rating frequency. This step ensures that, even within the filtered set, the focus is on the most informative interactions and maintain a manageable dataset size for computation. Importantly, the first filtering step prevents "accidental" inclusion of users who barely rated anything as the top-N selection alone would still pick users with very few ratings.

Min-max normalization was applied to ensure all ratings are non-negative, which is required for the square root calculation when initializing the expected magnitude in the ALS algorithm. The minimum value was set to a small constant ϵ to avoid confusion between actual zero ratings and non-observed entries.

After this process, the subsampled dataset contained 64854 ratings from 995 unique users and 1000 unique books, resulting in a density of ≈ 0.065, indicating that around 6.5% of all possible user-book interactions are observed.

Spark Version (Larger Dataset)

For the Spark implementation, a minimal subsampling strategy was used: only users and books with less than 5 ratings were removed. Unlike the Python version, no top-N filtering was applied.

Min-max normalization was also applied here for the same reason. In this case, however, scaling to the standard [0,1] interval was sufficient, as there won't be any actual non-observed ratings in the Spark implementation.

The finalized dataset contained 1065959 rows, 82427 unique users, and 63548 unique books, resulting in a much sparser matrix with density of ≈ 0.0002, highlighting that only 0.02% of all potential interactions are observed, which significantly increases the sparsity challenges for recommendation.

2.3 Utility matrix construction

Python Version (Subsampled Dataset)

In the Python version, to structure the data for collaborative filtering, a utility matrix was constructed from the subsampled dataset. In this matrix, each row represents a user, each column represents a book, and the entries

correspond to the rating that the user gave to that book. Ratings that were not provided by the user were filled with zeros. For computational convenience, each user and book was assigned an internal index, enabling efficient access to specific matrix entries.

The locations of the observed ratings were recorded separately, ensuring that model training, error computation, and evaluation metrics focus only on actual user-item interactions, ignoring missing entries which represent unknown preferences rather than negative feedback. The number of observed ratings is 57447. This is slightly lower than the total number of ratings in the subsampled dataset (64854) because the pivot operation aggregates duplicate ratings by the same user for the same book, typically taking their mean.

Spark Version (Larger Dataset)

In the Spark version, the full utility matrix isn't actually explicitly built, as that would be unfeasible to store in memory with millions of rows. Instead, the ratings remain as observed triplets (u, i, r_{ui}) with u the user index, i the item index, and r_{ui} the normalized rating. Unique integer indices are assigned to both users and items to enable efficient mapping of these triplets to the corresponding latent factor vectors. During training, Spark dynamically reconstructs the necessary slices of the utility matrix via `map-reduce` operations.

For example, the user update step is implemented as follows:

```
train_df.rdd \
    .map(lambda row: (row['user_idx'], (row['book_idx'], row['r
     ating_normalized']))) \
    .groupByKey() \
    .mapValues(lambda ratings: update_user_vector(list(ratings
     ), v, k_fixed, lambda_fixed))
```

Here, each observed triplet is mapped to a key-value pair with the user index as the key and (i, r_{ui}) pairs as the values. The `groupByKey` stage collects all ratings for a single user, effectively reconstructing the u-th row of the utility matrix on demand. The subsequent `mapValues` applies the ALS update rule to compute the new latent factor vector w_u.

A symmetric process is used for the item updates, grouping by i instead of u:

```
train_df.rdd \
    .map(lambda row: (row['book_idx'], (row['user_idx'], row['r
     ating_normalized']))) \
    .groupByKey() \
    .mapValues(lambda ratings: update_item_vector(list(ratings
     ), w, k_fixed, lambda_fixed))
```

3 Algorithms and Implementation

3.1 ALS Matrix Factorization

To predict unknown ratings in the utility matrix, the Alternating Least Squares (ALS) algorithm was implemented from scratch for matrix factorization. The observed user-item rating matrix $\mathbf{R} \in \mathbb{R}^{n_{\text{users}} \times n_{\text{items}}}$ is approximated as the product of two lower-dimensional latent factor matrices, $\mathbf{W} \in \mathbb{R}^{n_{\text{users}} \times k}$ and $\mathbf{V} \in \mathbb{R}^{n_{\text{items}} \times k}$:

$$\mathbf{R} \approx \mathbf{W}\mathbf{V}^\top$$

where k is the number of latent factors. Each user r is represented by a row vector $\mathbf{w}_r$ and each item j by a row vector $\mathbf{v}_j$. The predicted rating for user r and item j is

$$\hat{R}_{rj} = \sum_{s=1}^{k} w_{rs} v_{js}.$$

Initialization: All entries in $\mathbf{W}$ and $\mathbf{V}$ are initialized to

$$a = \sqrt{\frac{\text{average observed rating}}{k}}, \quad w_{rs}, v_{js} \leftarrow a,$$

which reflects the magnitude of observed ratings and facilitates convergence.

ALS Updates: ALS alternates between updating each latent factor. For a given user r and factor s, we can define the partial prediction excluding factor s as

$$P_{rj} = \sum_{k \neq s} w_{rk} v_{jk}.$$

The squared error for user r is

$$E_r = \sum_{j \in I_r} \big(R_{rj} - (P_{rj} + w_{rs} v_{js})\big)^2,$$

where I_r is the set of items rated by user r. Taking the derivative with respect to w_{rs} and setting it to zero gives the update:

$$w_{rs} = \frac{\sum_{j \in I_r} (R_{rj} - P_{rj}) v_{js}}{\sum_{j \in I_r} v_{js}^2 + \lambda},$$

where λ is a regularization parameter, included in the denominator to prevent overfitting by penalizing large latent factor values.

Similarly, for an item j and factor s, we can define

$$Q_{rj} = \sum_{k \neq s} w_{rk} v_{jk}, \quad U_j = \{r \mid R_{rj} \text{ observed}\},$$

and the update rule is

$$v_{js} = \frac{\sum_{r \in U_j} (R_{rj} - Q_{rj}) w_{rs}}{\sum_{r \in U_j} w_{rs}^2 + \lambda}.$$

These updates are applied for multiple epochs until convergence. Notation clarification: P_{rj} and Q_{rj} conceptually represent the partial predictions excluding the current latent factor, though in the Python and Spark implementations these sums were computed inline as `pred_others`.

Convergence Monitoring: After each epoch, the root mean squared error (RMSE) is computed on observed entries:

$$\text{RMSE} = \sqrt{\frac{1}{|\text{observed indices}|} \sum_{(r,j)\in\text{observed}} (R_{rj} - \hat{R}_{rj})^2}.$$

Early stopping is applied if the RMSE on a validation set stops decreasing, generating a local optimum solution. The final predicted matrix is

$$\hat{\mathbf{R}} = \mathbf{W}\mathbf{V}^\top.$$

3.1.1 Implementation differences

In the Python implementation, ALS iterates explicitly over all users, items, and latent factors, with each update computed via nested for-loops. The user and item matrices $\mathbf{W}$ and $\mathbf{V}$ are stored in memory as NumPy arrays, and updates are applied sequentially. In contrast, the Spark version leverages distributed computation: user and item indices are stored in a Spark DataFrame, and recommendations are computed for a selected subset of users using vectorized operations to avoid looping over all items, as discussed in Section 2.3.

3.2 Hyperparameter tuning

To evaluate the performance of the ALS recommendation model, the available ratings were split into training and test sets. The observed ratings, were first randomly shuffled and then divided such that 80% were assigned to the training set and the remaining 20% to the test set, keeping the original proportions.

A grid search strategy was then employed to tune the key hyperparameters of ALS:

- the number of latent factors $k \in \{2, 5, 10\}$,
- the regularization strength $\lambda \in \{0.01, 0.1, 1, 10\}$,
- and the number of training epochs (here fixed at 5 for time limitations).

For each hyperparameter combination, the ALS factorization was run on the training data and the Root Mean Squared Error (RMSE) on the test set was computed. RMSE was chosen as the evaluation metric because it directly

measures the magnitude of prediction errors in the same scale as the ratings, heavily penalizing larger deviations.

In the Python subsampled version, the optimal parameters were found to be $k = 10$, $\lambda = 1$, yielding a test RMSE of 0.7006. This version's results were summarized in a grid and visualized as a heatmap (Figure 2), showing how the test RMSE varies with k and λ.

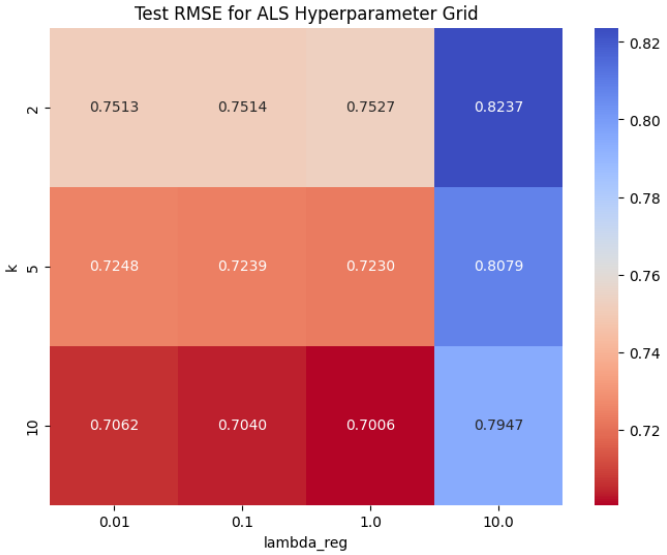

Figure 2: Hyperparameter tuning results (subsampled version)

For the Spark implementation, due to computational constraints, the parameter grid was reduced. The results indicated that smaller latent dimensions and lighter regularization provided the best performance, with the combination $k = 2$, $\lambda = 0.1$, achieving a test RMSE of 0.3038. Other tested configurations, such as $k = 2, \lambda = 0.01$ or $k = 5, \lambda = 0.01$, resulted in slightly higher RMSE values, but still notably lower compared to the Python subsampled version.

3.3 Training

With these optimal hyperparameters, both implementations of ALS were trained for approximately ten minutes.

In the Python subsampled version, the model was retrained for 50 epochs, in which a steady decrease was exhibited in both training and test RMSE,

with the test RMSE dropping from 0.7692 at epoch 0 to 0.5814 at epoch 49. Notably, the RMSE was still gradually decreasing at the last epoch, indicating that the model had not fully converged and could potentially improve further with additional training.

In contrast, the Spark version, which was trained for 10 epochs, reached a near-stable test RMSE of 0.3038 by epoch 4–5, after which further training produced negligible improvement.

4 Evaluation

4.1 Performance Metrics

4.1.1 Precision, Recall, Hit Rate and Coverage at different Top-k

Table 1 summarizes the recommendation performance of the ALS models under different top-k cutoffs. Four metrics were considered: Precision, Recall, Hit Rate and Coverage.

Table 1: Recommendation metrics for the subsampled Python and Spark ALS implementations at different top-k values.

	Subsampled Python				**Spark**			
k	Precision	Recall	Hit Rate	Coverage	Precision	Recall	Hit Rate	Coverage
3	0.0128	0.0028	0.0341	0.3410	0.0024	0.0028	0.0072	0.0004
5	0.0112	0.0059	0.0455	0.4500	0.0015	0.0028	0.0073	0.0006
10	0.0102	0.0135	0.0703	0.6070	0.0007	0.0028	0.0073	0.0011
20	0.0113	0.0295	0.1458	0.8040	0.0015	0.0117	0.0302	0.0019

The subsampled Python implementation demonstrates consistent improvements in recall, hit rate, and coverage as k increases. For example, recall rises from 0.0028 at $k = 3$ to 0.0295 at $k = 20$, showing that the model retrieves a larger share of relevant items when more recommendations are allowed. Similarly, hit rate grows steadily from 0.0341 to 0.1458, meaning users are increasingly likely to see at least one relevant item in their top-k list. Coverage also expands significantly, reaching 80% at $k = 20$, which indicates broad exposure of items across the catalog rather than repeatedly recommending a narrow subset. Precision, however, remains relatively flat (around 0.01), reflecting the common trade-off in recommender systems: as k

grows, the model finds more relevant items (higher recall) but also introduces more non-relevant ones, diluting precision.

In contrast, the Spark implementation yields extremely low values across all metrics. Precision never exceeds 0.0024, hit rate plateaus around 0.0073 for most k, and coverage remains below 0.2% even at $k = 20$. Recall does improve modestly at higher k (up to 0.0117), but the gains are minimal compared to the Python model.

4.1.2 Comparison with random and popular baselines

To contextualize the performance of ALS, its recommendation quality was compared against two simple baselines: a *random recommender*, which selects items uniformly at random, and a *popularity-based recommender*, which always recommends the most frequently rated items. The evaluation was conducted at $k = 5$ (Table 2) and $k = 10$ for both the subsampled Python implementation and the Spark implementation.

Table 2: Comparison of ALS, Random, and Popularity recommenders at $k = 5$ for the Python (subsampled) and Spark implementations.

	Python (subsampled)			Spark		
Metric	ALS	Random	Popularity	ALS	Random	Popularity
Precision	0.0112	0.0105	0.0273	0.0015	0.0000	0.0087
Recall	0.0059	0.0037	0.0117	0.0028	0.0001	0.0167
Hit Rate	0.0455	0.0517	0.0962	0.0073	0.0002	0.0433
Coverage	0.4500	0.9950	0.0050	0.0006	0.9948	0.0001

In the subsampled setting, the ALS model outperforms random recommendations on recall at both cutoffs, indicating that it is better at retrieving relevant items. However, its precision remains similar to random, and popularity dominates both metrics, especially in terms of hit rate. The key strength of ALS lies in coverage: while popularity only ever recommends a small set of the same items (coverage $< 1\%$), ALS explores a much broader portion of the catalog, covering up to 61% of items at $k = 10$. This trade-off highlights that ALS can recommend more diverse items, at the cost of slightly lower accuracy compared to a simple popularity heuristic.

The Spark implementation shows much lower absolute values but consistently outperforms the random baseline in precision, recall and hit rate, with ran-

dom only leading in coverage, while popularity maintains stronger accuracy but minimal diversity.

The same trend was observed at $k = 10$, with recall and hit rate increasing across all methods but precision decreasing.

4.2 Latent Space analysis

The latent factor matrices $\mathbf{W}$ and $\mathbf{V}$ obtained from ALS provide a compact representation of users and books in a k-dimensional space, capturing the underlying preferences and item characteristics.

For the subsampled dataset ($k = 10$), the 10-dimensional latent vectors were projected to 2D using Principal Component Analysis (PCA) to allow visualization. In contrast, for the full Spark dataset ($k = 2$), the latent space was already two-dimensional, so only a random subset of users and books was plotted for clarity.

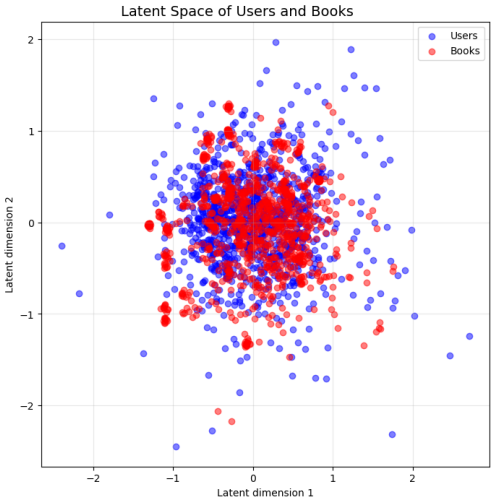

Figure 3: 2D PCA projection of user and book latent vectors for the subsampled dataset. Users in blue, books in red.

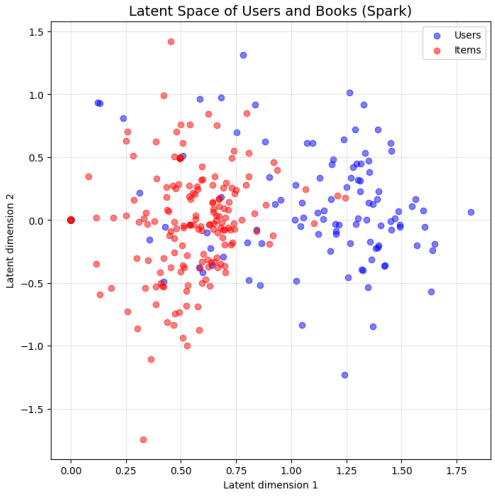

Figure 4: 2D latent space of a random subset of users and books from the Spark dataset. Users in blue, books in red.

The differing geometries of the latent spaces for the subsampled and Spark datasets provide insight into the observed performance of the ALS models. In the subsampled dataset (Figure 3), user and book vectors are highly concentrated and largely overlapping near the center. This dense overlap sug-

gests that most users and books share very similar latent features, which can make it challenging for the model to differentiate between individual preferences. While this contributes to moderate predictive performance, it also explains the higher test RMSE: the latent space does not fully capture the fine-grained distinctions necessary for accurate predictions, and additional training epochs could further refine these vectors.

In contrast, the Spark dataset (Figure 4) exhibits two clearly separated clusters corresponding to users and books, with only minor overlap at the center. The separation reflects that ALS was able to learn distinct latent representations for users and items despite the extreme sparsity of the dataset. However, the sparse interactions mean that many users and books are represented by very few observed ratings, limiting the algorithm's ability to accurately model preferences. Consequently, although the latent vectors are more structurally distinct, the precision and recall metrics are lower than the subsampled version, as the model struggles to make confident predictions for most user-item pairs. This sparsity-induced limitation also explains why baseline methods perform poorly in Spark: the underlying data provides insufficient signal for either random or popularity-based recommendations to be reliable.

Overall, these latent space structures illustrate the trade-off between dataset density and model generalization. A dense, overlapping latent space as in the subsampled set can provide moderate accuracy across many predictions but lacks fine-grained discrimination, whereas a sparse, well-separated latent space can clearly encode differences but may leave most predictions uncertain, leading to lower evaluation metrics.

4.3 On Scalability and Conclusions

The results across the subsampled and Spark datasets reveal a striking contrast in model behavior, that I hypothesize to be largely attributed to differences in dataset sparsity. Given the fact that the Spark dataset is $\approx$300 times sparser than the subsampled one, the weaker performance metrics observed across all methods, including popularity and random, suggest a structural limitation of the data itself.

These observations underline the fact that despite achieving lower RMSE values, a larger dataset does not automatically yield better results, as increased size combined with extreme sparsity can hinder the model's ability to learn meaningful patterns.

Nonetheless, there is clear potential for improvement. With increased computational resources, longer training times, and more extensive hyperparameter tuning, both the subsampled and Spark implementations could achieve better latent representations and improved recommendation quality. Extending training epochs could help the subsampled model converge further, while more sophisticated approaches or data augmentation could mitigate the sparsity challenges in the Spark dataset, leading to more robust and accurate recommendations overall.

References

[1] Mohamed Bakhet. *Amazon Books Reviews Dataset*. 2023. `https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews`. Accessed: 2025-09-03.