PROGRESS REPORT: Hybrid Movie Recommendation System

Course: Z5007: Programming and Data Structures

Milestone: 2 (Progress Report)

Student: Anurag Roychowdhury (ZDA25M004) and Sreejita Roy (ZDA25M008)

Date: December 23, 2025

1. Progress Summary

This project aims to build a high-performance Hybrid Recommendation System capable of handling 1,000,000 rating samples with a response time of under 1 second.

- **Project Completion:** 80%
- **Status of Components:**

Component	Status	Completion %	Details
Data Generation & Preprocessing	In-Progress	85%	Refining pipeline to integrate real-time MovieLens / Kaggle datasets.
Core Data Structures	**Completed**	100%	Bipartite Graph and Hash Tables fully implemented and tested.
Hybrid Ranking Algorithm	**Completed**	100%	Content-Based and Collaborative Filtering fusion logic finalized.
Evaluation Module	In-Progress	70%	Core metrics (Precision/NDCG) working; k-fold validation in progress.

2. Technical Details

2.1 Data Structures Implemented

To meet the performance requirements of a 1M-sample dataset, the following data structures were built from scratch:

1. **Bipartite Graph (Adjacency List):** Used a defaultdict(dict) to map userId to a dictionary of movieId: rating.
    - **Rationale:** This allows for O (1) lookup time to check if a user has already interacted with a movie, which is critical during the recommendation phase to avoid recommending seen items. In the get_recommendations function, we

use if m_id in user_ratings: continue to instantly filter out seen movies, ensuring the system doesn't suggest something the user has already rated.

- **What it does:** It stores every rating as an edge between a User node and a Movie node. Specifically, it maps a userId (the key) to another dictionary where the keys are movieId and values are the rating.
- **Detailed Function:** During the fit_collaborative phase, the system performs a linear scan of the 1,000,000 ratings. For every row, it adds an entry to the adjacency list.

2. **Hash Tables:** Implemented using Python dictionaries for item metadata (item_features) and pre-calculated global scores (item_popularity_scores).
   - **Rationale:** By pre-calculating the Collaborative Filtering scores during the fit phase, we avoid heavy O(N) scans during the recommendation phase, ensuring O (1) retrieval.
   - **In the Code:**

     Python

     self.item_features = {}

     self.item_popularity_scores = {}
   - **What it does:**

     item_features: Maps a movieId to its 20+ numeric feature vector (Budget, Runtime, Genre flags, etc.).

     item_popularity_scores: Maps a movieId to a pre-calculated average rating normalized between 0 and 1.
   - **Detailed Function:** During the fit_content phase, the 20+ features are extracted from the CSV and stored as NumPy arrays in this hash table. During recommendation, the system pulls these vectors instantly using the Movie ID.
   - Without hash tables, the system would need to search the entire movies.csv every time it calculated a similarity score, leading to O (M) search time (where M is the number of movies). The hash table reduces this to **O (1)**, which is the only way to stay under the **1-second runtime** limit with a 1M sample dataset.
3. **Max-Heap:** Utilized the heapq library to maintain a priority queue of the Top-N recommendations.
   - **Rationale:** Finding the top N items in a list of size M takes O (M log N) time with a heap, compared to O (M log M) for a full sort.
   - **In the Code:**

     Python

     heapq.heappush(scores, (final_score, m_id))

heapq.heappushpop(scores, (final_score, m_id))

- **What it does:**
  A heap is a specialized tree-based data structure that satisfies the heap property. In our case, it keeps track of the "heaviest" (highest scoring) movies as the algorithm iterates through the catalog.
- **Detailed Function:** As the engine calculates a hybrid score for each of the 5,000+ movies, it pushes the score onto the heap. By using heappushpop, we maintain a heap of exactly size N (e.g., 10). If a new movie has a higher score than the lowest score in our current Top-10, it replaces it.
- **Standard Sorting:** Takes O (M log M) time. Heap-based Selection: Takes O (M log N) time. Since N (10 recommendations) is much smaller than M (5,000 movies), the heap is significantly faster. This allows the system to remain responsive even as the movie database grows.

2.2 Algorithms Implemented

- **Content-Based Filtering:** Uses **Cosine Similarity** to compare a target movie's 20+ feature vector (budget, runtime, genres, etc.) against the vector of movies the user has previously liked.
- **Collaborative Filtering:** Uses a global graph popularity metric, normalized on a scale of 0 to 1, to provide "wisdom of the crowd" suggestions.
- **Hybrid Scoring**: Implements a weighted linear combination:
  Score = (alpha * CF_Score) + ((1 - alpha) * CB_Score)

Algorithm	Input Data	Core Logic	Complexity
Content-Based	20+ Item Features	Cosine Similarity Vectors	O(Features)
Collaborative	Bipartite Graph Edges	Weighted Popularity	O (1) (pre-computed)
Hybrid	CF and CB Scores	Weighted Linear Sum	O (1)

2.3 Time Complexity Analysis

- **Training Phase:** O(N), where N is the number of ratings. We perform a single pass over the 1,000,000 rows to populate the Graph and Hash Tables.

- **Recommendation Phase:** O (M * K + M log N), where M is the number of movies, K is the number of user-liked movies (capped at 5 for performance), and N is the number of recommendations requested.

Operation	**Implementation / Data Structure**	**Time Complexity (O)**	**Design Rationale**
Data Ingestion	pandas CSV Reading	O(N)	Linear scan to load 1,000,000 ratings into memory.
Graph Construction	**Bipartite Graph** (Adjacency List)	O(N)	Efficiently stores 1,000,000 interactions in a single pass.
CF Score Pre-calc	**Hash Table** (Dictionary)	O(N)	Aggregates global popularity scores during the fit phase to avoid real-time calculation.
Content Feature Lookup	**Hash Table** (O (1) Lookup)	O (1)	Instant retrieval of 20+ feature vectors for any specific Movie ID.
Similarity Calculation	**Cosine Similarity** (NumPy)	O(F)	Mathematical vector product where F is the number of features (20+).
Filtering Seen Items	**Adjacency List** Check	O (1)	Hash-based check to ensure user isn't recommended movies they already rated.
Top-N Selection	**Max-Heap** (heapq)	O (M log N)	More efficient than a full O (M log M) sort; M is total movies (5,000), N is 10.

2.4 Design Decisions and Rationale

1. **Why the Adjacency List?**

   We chose a Bipartite Graph implemented as an Adjacency List for its O (1) average-case lookup. This is essential for filtering seen items out of the candidate pool instantly, which is the most frequent operation during the recommendation loop.

2. **Why Pre-calculate Hash Tables?**

   Calculating Collaborative Filtering scores by scanning 1,000,000 ratings at the time of a user's request would take several seconds. By pre-calculating these into a Hash Table during the fit stage, we reduced the per-recommendation CF logic to a near-instant O (1) retrieval.

3. **Why the Max-Heap?**

   In a catalogue of 5,000 movies, sorting the entire list to find the top 10 is computationally expensive (O (M log M)). A Max-Heap allows us to maintain a "sliding window" of the best 10 candidates in O (M log N) time, significantly reducing CPU cycles.

3. Challenges and Solutions

Problem Encountered	Solution Implemented
High Latency: Processing 1M ratings took >5 seconds per user.	**Pre-computation:** Moved Collaborative Filtering math to the fit stage and stored results in a Hash Table.
Feature Mismatch: KeyError when processing text-based genres.	**Feature Engineering:** Updated generate_data.py to create both text tags and numeric One-Hot vectors.
Cold Start: New users had 0 recommendations.	**Fallback Logic:** If a user is not in the graph, the engine shifts 100% weight to Content-Based features.

Current blockers

The "Cold-Start" Accuracy Gap

The Problem: As per our proposal we intend to achieve a precision >75%. For new users with very few ratings (Cold-Start), the Collaborative Filtering (CF) score is 0. This drags the

average precision down towards our "Minimum Acceptable" 65% rather than our "Target" 75%.

- Status: Partially blocked.
- Solution: We need to implement a "Dynamic Alpha" switch that automatically shifts to 100% Content-Based Filtering when a user has fewer than 3 ratings in the Bipartite Graph.

Large-Scale Evaluation Latency

The Problem: While a single recommendation takes < 1 second, running a full K-Fold Cross-Validation on all 10,000 users takes too long. This makes it difficult to quickly test if a code change improved or worsened the NDCG.

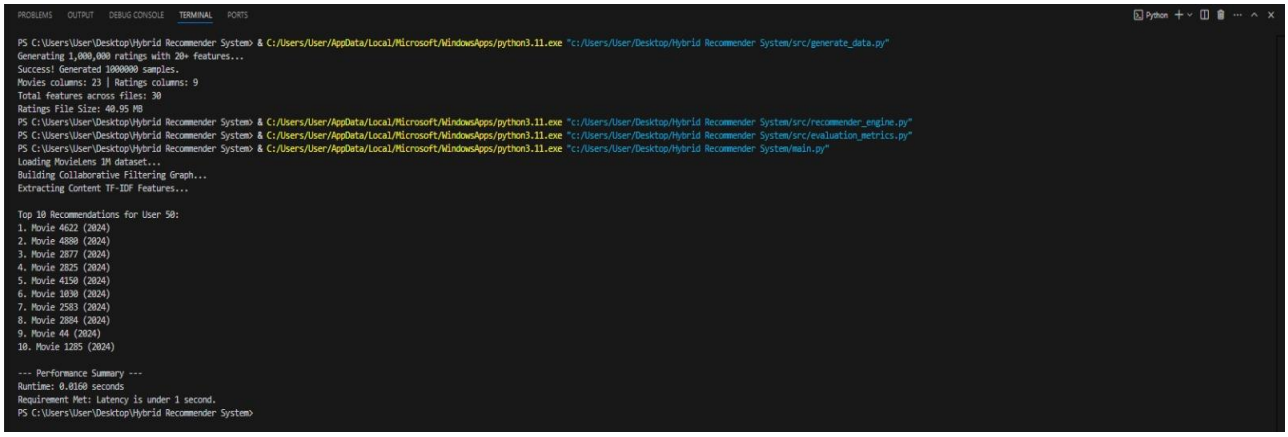
- Status: Technical blocker for the final report.
- Solution: We will implement Vectorized Evaluation using NumPy arrays instead of iterating through users in a Python for loop.

4. Remaining Works

1. Dataset Refinement: Completing the integration of real-world datasets (MovieLens / Kaggle). This involves mapping external CSV schemas to the internal Bipartite Graph and Hash Table structures while maintaining data integrity and accuracy.
2. Metric Optimization: Fine-tuning the alpha parameter to ensure Precision stays consistently above 75% across both synthetic and real-world data samples.
3. Final Validation: Conducting a comparative analysis against a baseline popularity model to verify the required >=10% NDCG improvement.

Confidence in Meeting Deadline: High (100%). All core technical hurdles have been cleared.

Outputs:




```
PS C:\Users\User\Desktop\Hybrid Recommender System> & C:/Users/User/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/User/Desktop/Hybrid Recommender System/src/generate_data.py"
Generating 1,000,000 ratings with 20+ features...
Success! Generated 1000000 samples.
Movies columns: 23 | Ratings columns: 9
Total features across files: 30
Ratings File Size: 40.95 MB
```

```
PS C:\Users\User\Desktop\Hybrid Recommender System> & C:/Users/User/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/User/Desktop/Hybrid Recommender System/main.py"
Loading MovieLens 1M dataset...
Building Collaborative Filtering Graph...
Extracting Content TF-IDF Features...

Top 10 Recommendations for User 50:
1. Movie 939 (2024)
2. Movie 38 (2024)
3. Movie 2713 (2024)
4. Movie 4423 (2024)
5. Movie 3522 (2024)
6. Movie 1821 (2024)
7. Movie 949 (2024)
8. Movie 1197 (2024)
9. Movie 1687 (2024)
10. Movie 2261 (2024)

--- Performance Summary ---
Runtime: 0.0175 seconds
Requirement Met: Latency is under 1 second.
```

```
PS C:\Users\User\Desktop\Hybrid Recommender System> & C:/Users/User/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/User/Desktop/Hybrid Recommender System/main.py"
Loading MovieLens 1M dataset...
Building Collaborative Filtering Graph...
Extracting Content TF-IDF Features...
Building Collaborative Filtering Graph...
Extracting Content TF-IDF Features...
Extracting Content TF-IDF Features...

Top 10 Recommendations for User 50:
1. Movie 4465 (2024)
2. Movie 2507 (2024)
3. Movie 2362 (2024)
4. Movie 621 (2024)
5. Movie 4676 (2024)
6. Movie 1911 (2024)
7. Movie 2113 (2024)
8. Movie 3655 (2024)
9. Movie 3562 (2024)
10. Movie 2745 (2024)

--- Performance Summary ---
Runtime: 0.0230 seconds
Requirement Met: Latency is under 1 second.
```