T.C.

BAHÇEŞEHİR UNIVERSITY

FACULTY OF ENGINEERING AND NATURAL SCIENCES

DEPARTMENT OF COMPUTER ENGINEERING

CMP2003 – DATA STRUCTURES AND ALGORITHMS WITH C++

2024 – 2025 FALL TERM PROJECT

MOVIE RATING PREDICTION USING NEIGHBORHOOD-BASED COLLABORATIVE FILTERING

Emir Ismail Genc 2202780

Omer Inkaya 2103353

Berkay Sahin 2203066

Course Instructor: Dr. Tevfik Aytekin

December 2024

Contents

1 Introduction

1.1 Project Description and the Objective

Background and Motivation

Recommender systems have become a main part of digital platforms. They enable companies to deliver personalized suggestions to individual users. These systems rely on user feedback in the form of ratings or preferences, from recommending movies and music on streaming platforms to suggesting products on e-commerce websites. By analyzing these ratings, a recommender system can infer patterns of similarity—among users, among items, or both. Such insights allow the system to predict how a user might rate an item they have not experienced yet. Designing effective, efficient, and scalable recommender systems is a big challenge. Collaborative Filtering (CF), the most widely used technique in this field, works by identifying either **similar users** (in **User-Based Collaborative Filtering**, UBCF) and leveraging their ratings to predict a target user's rating for a specific item, or **similar items** (in **Item-Based Collaborative Filtering**, IBCF) and using these items' ratings from a particular user to predict that user's rating for a target item. While both approaches share the same final goal (accurate rating prediction), the IBCF approach often proves more efficient in practice, especially when dealing with large user communities but a smaller set of items, or when the item similarity is more stable than user similarity over time.

This project is an excellent example of applying theoretical concepts from data structures and algorithms to solve real-world problems. By addressing challenges like data sparsity, scalability, and computational efficiency, the project showcases the importance of algorithmic design in creating valuable software.

Scope of the Project

The project, titled **"Movie Rating Prediction using Neighborhood-Based Collaborative Filtering"**, focuses on implementing and analyzing an **Item-Based Collaborative Filtering (IBCF) system** in C++. Fundamental data structures (such as unordered_map and vector) and algorithms (such as sorting for neighbor selection) are utilized, to build a high-performance rating prediction program. The project is built in a way that highlights key data structures and concepts taught in the "**Data Structures and Algorithms** with C++" course.

In this project, a **"training" dataset** of user–item ratings is ingested first, where each user can give a numerical rating to multiple films. After that, the program **computes average ratings** for each film to help with normalization and to serve as a baseline rating. **Normalization of the film ratings** by subtracting the film-specific averages from each user's rating is also being made to center the data and mitigate rating-scale biases among different films. Later on, the **calculation of cosine similarities** between pairs of films based on their normalized ratings is being made. These similarities are also stored in a cache to avoid repeated, costly computations. Lastly, **prediction of the rating** of a given user for a target film using the **IBCF** method is being made. To do this, the user's previously rated films are identified, the similarities between these known films and the target film are retrieved, those similarities are combined by weighting each by the normalized rating the user gave to that film, and the baseline average of the target film is added back to achieve the final prediction result.

Objectives

The primary objective of this project is to **design and implement a rating prediction system** that predicts film ratings for users based on their historical preferences and the preferences of other users. This involves **developing a fast pipeline** for preprocessing user-film rating data and implementing similarity calculation methods using **cosine similarity** to measure the relationship between different films.

Optimization using caching and normalization is being done by the implementation of a caching mechanism to store previously computed similarities between films for the purpose of reducing redundant computations and improving runtime efficiency. Also, **normalization of the film ratings** by subtracting the mean rating of each film to eliminate bias introduced by differing rating scales across users, is also being made.

High scalability and performance is achieved by ensuring that the algorithm performs efficiently on large datasets by making use of great data structures such as hash tables (unordered_map) and vectors. Tree structures were also an option, which would have probably yielded even higher performance and scalability, but as we have explained in the video recording, for time's sake, readability's sake, and simplicity's sake, we've decided to utilize those data structures.

One of the most important objectives (and benefits to the student) of the project is certainly **the evaluation of different filtering methods.** Experimenting with different filtering methods such as the Pearson method, including IBCF and UBCF, analyzing their performance on the dataset, comparing those methods with each other to validate the choice, offers valuable insights into the world of NBCF (Neighborhood-based Collaborative Filtering) algorithms. As we have also stated in the video recording, we also explored MBCF (Model-based Collaborative Filtering) algorithms and

tried and achieved success in implementing even more successful methods such as SVD (Singular Value Decomposition) with RMSE scores as good as 0.89 and runtime as good as 0.08 seconds on HackerRank submissions, but we were late to discover that MBCF algorithms are another category and they are different than NBCF algorithms and unusable for our project, thus leading us to ditch our high-performing implementation altogether and return back to Cosine Similarity.

Achieving accuracy and robustness is also another important objective in the project. Implementation of a mechanism to predict a target user's rating for a target film using the ratings of the film's nearest neighbors (similar films). We emphasized on this objective by considering a minimum threshold of shared ratings between items and limiting the number of nearest neighbors (K) for prediction. Experimenting with the K value and other values used in weighting and normalizing gave us valuable insights about this objective in the project.

In terms of **application of data structures and algorithms,** we utilized advanced data structures like hash tables to efficiently store and retrieve ratings and average values. We applied sorting (std::sort) and selection algorithms to identify the top-K similar films for prediction.

---

2 General Overview

The program is structured to accept two types of inputs. While the **training dataset** is used to train and build internal data structures, the system predicts the likely rating for the specified user–film pair and outputs it by utilizing the **test dataset** at the same time.

2.1 Determination of Necessary Additional C++ Libraries and Data Structures

To build a fast and efficient rating prediction system, we had to use several **standard C++ libraries** and **data structures**.

Language: C++20

```
#include <iostream>            // for standard input/output operations
#include <unordered_map>     // for unordered_map hash table data structure usage
#include <vector>             // for vector data structure usage
#include <algorithm>         // for sorting and partial sorting functionality
#include <sstream>             // for istreamstring usage for parsing input
#include <cmath>             // for square root in cosine similarity calculation

```

Data Structures Employed

1. **data_map**

using data_map = unordered_map<unsigned short, unordered_map<unsigned short, float>>;

This serves as a nested hash map where the outer key is a **film ID** (unsigned short), and the inner map key is a **user ID** (unsigned short). The stored value is the user's **rating** (float). The rationale here is that we want fast lookups to see, for a given film, which users have rated it and what those ratings are. This is essential for computing film-to-film similarity and for normalizing ratings.

2. **data_smap**

using data_smap = unordered_map<unsigned short, float>;

This structure simplifies referencing an individual film's rating data. It is used in functions that compute film averages or similarities. This structure avoids overhead of repeated declarations of unordered_map<unsigned short, float>.

3. **filmAvgMap**

unordered_map<unsigned short, float> filmAvgMap;

This stores the **average rating** (float) for each film (keyed by film ID). It is used during **normalization** of a film's ratings and for calculating predicted ratings based on the film's average.

4. **similarityCache**

static unordered_map<uint64_t, float> similarityCache;

This speeds up repeated **Cosine Similarity** lookups between two films by caching previously computed similarities. The **uint64_t** key is computed by the makeCacheKey function to uniquely identify a pair of film IDs. Computing Cosine Similarities repeatedly can be expensive. With a cache, we compute each pair's similarity only once.

2.2 Detailed overview of the Source Code of the Project

Below is a thorough breakdown of each component of the source code. The code is divided into **global variables**, **function declarations**, **function definitions**, and **the main function**.

Global Variables and Type Aliases

static unordered_map<uint64_t, float> similarityCache;

data_smap filmAvgMap;

similarityCache stores already-computed **cosine similarities** between two films, identified by a 64-bit key. **filmAvgMap** stores the **average rating** for each film. This is used in normalization and final rating computations.

Function Declarations

```
void computeFilmAverages(data_map &data);

void normalizeFilmRatings(data_map &data);

float predictRatingIBCF(unsigned short userId, unsigned short targetFilmId, data_map &data, int K);

static inline uint64_t makeCacheKey(const unsigned short id1, const unsigned short id2)
{return (static_cast<uint64_t>(min(id1, id2)) << 32) | max(id1, id2); }
```

These signatures provide an **interface** for the functions that handle various aspects of the rating prediction process. **computeFilmAverages** takes the average of the user ratings for each film. **normalizeFilmRatings** subtracts the average rating from each user's rating of a film. **predictRatingIBCF** Predicts the rating of a given film for a given user, using the **K** most similar films.

makeCacheKey Function creates a **64-bit key** by combining two 16-bit film IDs. It takes the smaller of the two film IDs and places it in the **high** 32 bits and takes the larger of the two film IDs and places it in the **low** 32 bits. This function ensures that the order of id1 and id2 does not matter (i.e., makeCacheKey(2, 10) is the same as makeCacheKey(10, 2)). This is crucial for consistent lookups in similarityCache.

main Function

The lines ios::sync_with_stdio(false); and cin.tie(nullptr); disable synchronization between C and C++ I/O, improving input-reading performance.

When **Reading the Training Dataset**, the code looks for the **"train dataset"** marker in the beginning. It continues reading lines until it encounters the **"test dataset"** marker. Each line is parsed into user_id, film_id, and rating. It stores the rating in data[film_id][user_id].

Computing Averages and Normalizing Ratings with computeFilmAverages(data); normalizeFilmRatings(data);
These calls ensure that each film's **average rating** is computed and stored in filmAvgMap and each user's rating for a film is **shifted** by subtracting that film's average rating, making the ratings zero-centered for more accurate similarity computations.

Reading the Test Dataset: For each new line, we parse the **user_id** and **film_id**. We compute a **predicted rating** using IBCF with **K = 40**. The predicted rating is then **output**. The procedures are:

cosine_similarity Function computes the **cosine similarity** between two films, filmIdA and filmIdB, given their respective **normalized** rating maps (filmA and filmB).

When it comes to **caching**, a unique cacheKey is generated using makeCacheKey(...). If a similarity was computed before, it is immediately returned from similarityCache. If the **number of common ratings** is < 7, or the **denominator** is effectively zero, the similarity is set to 0.0. The final similarity is stored in similarityCache and returned to the caller.

computeFilmAverages Function sums the ratings from all users and **computes an average for each film**. The function uses a range-based for loop, iterating over the nested user-rating map. Each film is iterated over exactly once, making it O(F×UF), where F is the number of films, and UF is the number of users per film.

normalizeFilmRatings Function subtracts the average rating of each film from every user's rating for that film. After normalization, a film's ratings might have positive, negative, or zero values. This zero-centering helps **Cosine Similarity** focus on the shape of ratings rather than absolute scales.

predictRatingIBCF Function Predicts the rating a **user** would give to a **target film** using **Item-Based Collaborative Filtering**. The steps are:

1. We iterate over every other film in the data map to see if the user has rated it. We compute the cosine similarity only if the user has a rating for that other film. We store (similarity value, otherFilmId) in similarities. We sort the vector of (similarity, filmId) pairs in descending order of similarity.
2. We only consider the top KK neighbors (films). Each neighbor film's *normalized* rating is multiplied by the similarity and summed. We also keep track of the sum of similarities, to normalize the result.

3. We **re-center** the rating by adding back the mean rating of the **target** film. If there were not enough neighbors (i.e., simSum is negligible), we default to the **film's average** rating.

The system's design ensures that rating prediction is **modular** (via separate functions for each step), **efficient** (using unordered_map and a similarityCache), and **accurate** (through normalization and top-KK similarity-based scoring).

3 Conclusion

This project demonstrated the practical application of data structures and algorithms in solving a real-world problem: predicting movie ratings for users through an efficient implementation of Item-Based Collaborative Filtering (IBCF). By employing advanced C++ techniques and data structures such as hash maps, vectors, and caching mechanisms, we successfully designed a scalable and accurate system.

The implementation achieved several milestones. The first one is **efficient similarity calculations:** Through the use of cosine similarity and caching, we minimized redundant computations, improving runtime performance. Another one is **normalization for bias elimination.** By centering user ratings around film averages, we reduced biases from individual user rating scales, ensuring fairness in similarity measures.Lastly, there is **adaptability with key parameters.** Experimenting with parameters like the number of neighbors (K) provided insights into the trade-offs between accuracy and computational efficiency.

Challenges and Solutions

Throughout the project, we encountered challenges such as **data sparsity** which was addressed by limiting similarity computations to films with sufficient shared ratings, **scalability** which was resolved through optimized data structures and modular design and **methodological misalignment**, meaning that despite initial success with Model-Based Collaborative Filtering (MBCF) methods like SVD, they were abandoned to align with the project's Neighborhood-Based Collaborative Filtering (NBCF) scope.

Learnings and Future Work

This project reinforced the importance of algorithmic optimization in handling large datasets efficiently, evaluating multiple methods to validate design choices and iterative testing and parameter tuning for high performance. Future work could explore extending the project to hybrid

models combining NBCF and MBCF approaches, implementing real-time prediction capabilities and incorporating additional data, such as user demographics, to enhance personalization.