TABLE OF CONTENTS

1. INTRODUCTION

This project aims to provide an efficient and optimized infrastructure for recommendation systems by analysing user-movie rating relationships on large-scale datasets. UBCF (User-Based Collaborative Filtering) and IBCF (Item-Based Collaborative Filtering) methods are used to calculate the similarities of users or movies and predict new ratings using this information. The project is optimized with modern data structures and multi-threading, and C++11 standard library components such as **`std::unordered_map`**, **`std::priority_queue`** and **`std::thread`** are used to increase efficiency. This application, which includes the steps of processing the dataset, creating similarity matrices, making predictions and performing performance measurements, has a flexible and reusable structure. At the end of the project, a solution has been developed that aims to optimize the accuracy of predictions on a dataset and the computational time. Thanks to the structured and modular code architecture, the project is suitable for future extensions and working with different datasets. Team members are as follows:

2. CLASSES

2.1. DATAHASH2D

The class and the documentation for this part are written by 2368545 Yusuf Ersel Bal.

The **`DataHash2D`** class is intended to store the movie-user-rating triples and to perform the necessary operations on the data in accordance with the purpose of the project. The class is generally used for storing training and test data and accessing them quickly afterwards, and for analysing and processing the data. This class interacts directly with the **`Similarity`** class (for calculating the similarity matrix on the data and etc…), the **`Prediction`** class (for making predictions based on the data etc…) and the **`FileHandler`** class (for reading and converting data into objects or transferring from object to file format etc…). The class uses two nested **`std::unordered_map`** structures to store data. **`std::unordered_map`** is a hash based data structure available in the C++ standard library. The data structure used is specialised with the keyword **`using`** under the name **`RatingMap`** as follows: **`using RatingMap = std::unordered_map<int, std::unordered_map<int, float>>;`**. The key of the first **`std::unordered_map`** represents the Id of the movie, movieId, and provides access to an internal **`std::unordered_map`** containing all users who rates the movie and their ratings of the movie. In this case, the key of the second **`std::unordered_map`** represents the Id of the user, userId and accesses the rating that user has given to that movie. One of the questions that may come to mind is why this data structure was chosen. The Hash based data structure, with its average **`O(1)`** data access and data insertion time complexities, offers a great advantage in terms of speed in these areas and its use in projects that need frequent access to data, such as a recommendation engine (as in our project), makes a big difference in terms of performance. The dynamic nature of the data structure is another reason for preference. Since not every user votes for every movie, i.e. each user votes for only certain subsets of the list of all movies (or vice versa, each movie is voted for by only certain subsets of the list of all users), the data set is sparse. **`std::unordered_map`** is suitable for sparse data sets because, unlike a **`std::vector`**, it does not require allocating additional memory space for unnecessary data. In addition, since each movie and each user has a unique identifier, an Id, it is a natural choice to use a key-value type data structure. Below is a table of advantages/disadvantages for some data types that can represent the data set in the project:

Data Structures	Advantages	Disadvantages
`std::unordered_map<int, std::unordered_map<int, float>>` (RatingMap)	**O(1)** access and insertion time. Managing sparse data effectively. Dynamic data insertion support.	High memory usage. Sequential storage of keys is not guaranteed.
`std::vector<std::pair<int, std::unordered_map<int, float>>>`	Movies can be kept sequential. Provides sparse data support.	Access to movie is **O(n)** however access to user is **O(1)**. Slow on large data sets.
`std::map<int,std::map<int, float>>`	Sequential data access. Stable memory usage.	Access and insertion time **O(logn)**. Less efficient for sparse data.
`std::vector<std::vector<float>>`	Fast on fixed size data sets due to matrix structure.	Not suitable for sparse data, creates too many data holes. No dynamic data support.
`std::pair<std::pair<int,int>, float>`	Low memory usage. Sequential access is possible.	**O(n)** access to movie or user. Not fast for sparse data. Data manipulation is complex.

Considering the table, it is seen that **RatingMap** is the most suitable data structure for the project due to its high access speed, sparse data support and dynamic insertion features. Although it takes up more memory than other data structures due to its hash based structure, it is mostly unusual to exceed the 512 MB memory limit within the project requirements. Since there is no usual need for sequential access within the project, this disadvantage can be ignored too. Compared to alternative data structures, the **O(1)** time complexity, efficiency in sparse data management and dynamic data insertion make this data structure shine in terms of performance in large-scale and dynamic data sets. Another questions that may come to mind is why the data is kept as **{ movieId → { userId → rating } }** even if the data set is in the userId, movieId, rating structure. As a result of the runtime tests with the given data sets, it was seen that keeping the data as **{ movieId → { userId → rating } }** is much faster than the reverse in terms of speed when using threads, so it was decided that the external key in the program should be movieId and the internal key should be userId. Now that we have seen why the data structure was chosen and alternative data structures with their advantages and disadvantages, we can start to analyse the **DataHash2D** class in more detail:

RatingMap DataHash2D::getRatingMap() const: It is used to obtain a complete copy of the data set.

Parameters: None.

Return Value: None.

void DataHash2D::setRatingMap(const RatingMap& newMovieRatings): Used to define a completely new data set for the object.

Parameters: None.

Return Value: None.

`void DataHash2D::addRating(int movieId, int userId, float rating):` Adds the rating value corresponding to a movie-user pair to the data set. The if block checks whether the rating is within the valid range. If it is valid, it adds the data to the data set, if it is invalid, it prints a message indicating that to the console.

Parameters:

- **`movieId:`** Unique Id of the movie.
- **`userId:`** Unique Id of the user.
- **`rating:`** Rating value corresponding to movieId-userId pair. Must be between **`0.0f`** and **`5.0f`**

Return Value: No return.

`float DataHash2D::getRating(int movieId, int userId) const:` Returns the recorded rating for a specific movie and user. The specified movieId is searched in the dataset using an iterator. If the movie is found, the userId for that movie is searched in the same way. If both searches are successful, the corresponding rating value is returned. If unsuccessful, **`-1.0f`** is returned.

Parameters:

- **`movieId:`** Unique Id of the movie.
- **`userId:`** Unique Id of the user.

Return Value: Rating (**`float`**). If score is not found, **`-1.0f`** is returned.

`std::unordered_map<int, float> DataHash2D::getMovieRatings(int movieId) const:` Returns all the rating data for a movieId. The movieId value is searched in the data set with the **`.find()`** method. If found, the corresponding user-rating matches are returned. If not found, an empty **`std::unordered_map`** is returned.

Parameters:

- **`movieId:`** Unique Id of the movie.

Return Value: If found, a **`std::unordered_map<int, float>`** including userId's and ratings, if not found an empty **`std::unordered_map<int, float>`**.

`std::unordered_map<int, float> DataHash2D::getUserRatings(int userId) const:` Returns all the rating data for a userId. Iterates through all movieId-userId pairs. If a userId exists for that movieId, it adds that movie and rating to the list.

Parameters:

- **`userId:`** Unique Id of the user.

Return Value: If found, a **`std::unordered_map<int, float>`** including movieId's and ratings, if not found an empty **`std::unordered_map<int, float>`**.

`float DataHash2D::getAverageRating(bool isMovie, int id) const:` Returns the average rating of a movie or user, based on the **`isMovie`** flag. It decides whether to run the **`getMovieRatings()`** method or the **`getUserRatings()`** method using the ternary operators according to the **`isMovie`** flag. It collects all the scores in the list obtained from these methods and divides it by the number of elements.

Parameters:

- **`isMovie:`** If **`true`**, it means that the average rating of the movie will be taken, if **`false`**, it means that the average rating of the user will be taken.
- **`id:`** The Id of the object (movie or user) whose average rating will be taken.

Return Value: Average rating (**`float`**). If no data is found, **`-1.0f`** is returned.

`std::vector<int> DataHash2D::getAllMovies() const:` Returns a list of unique movieIds in the dataset. All movies in the dataset are added to a **`std::vector`** with a **`for`** loop.

Parameters: None.

Return Value: A **`std::vector<int>`** containing all the movieIds.

`std::vector<int> DataHash2D::getAllUsers() const:` Returns a list of unique userIds in the dataset. The movies in the dataset are scanned by nested **`for`** loops to determine which users have rated them.

Parameters: None.

Return Value: A **`std::vector<int>`** containing all the userIds.

`std::unordered_map<int, int> DataHash2D::countRatings(bool isMovie) const:` According to the **`isMovie`** flag, it returns the total rating count of all movies or users. According to flag, **`getAllMovies()`** or **`getAllUsers()`** is called with the help of ternary operators. The received list is divided into equal parts to be executed by the thread. **`numEntities`** are calculated to distribute the workload equally. Then, the rating count is revealed by scanning operations with the help of **`for`** loops on a certain range with each thread. The result of each thread is added to the global counts variable at the end. Race conditions are prevented with the **`lock()`** and **`unlock()`** methods of the **`ThreadHandler`** class. Since it is a private method that should help the getTopUsers() and getTopMovies() methods, it returns data of type **`std::unordered_map<int, int>`**.

Parameters:

- **`isMovie:`** If **`true`**, it means that the number of ratings of a movie will be counted, if **`false`**, it means that the number of ratings of a user will be taken.

Return Value: A **`std::unordered_map<int, int>`** where objects (movie or user) are keys and their rating counts are values.

`std::vector<std::pair<int, int>> DataHash2D::getTopMovies(int topN) const:` Returns the top **`topN`** movies with the highest rating counts from the dataset, along with their total ratings. **`CountRatings()`** is called to get the total number of ratings for all movies. Then, the movies with the highest ratings are moved to the top with the help of a **`std::priority_queue`**. The **`topN`** movies with the highest value in the **`std::priority_queue`** are taken and added to the **`topMovies`** list. In each iteration, the top movie is removed from the queue.

Parameters:

- **`topN:`** Determines how many top-rated movies will be returned.

Return Value: A **`std::vector<std::pair<int, int>>`** containing the top rating users and their total rating count is returned.

`std::vector<std::pair<int, int>> DataHash2D::getTopUsers(int topN) const:` It is based on the same logic as the getTopMovies() method. Returns the top **`topN`** users with the highest rating counts from the dataset, along with their total ratings.

Parameters:

- **`topN:`** Determines how many top-rated movies will be returned.

Return Value: A **`std::vector<std::pair<int, int>>`** containing the top rating users and their total rating count is returned.

`size_t DataHash2D::getMovieCount() const:` Returns the total number of movies recorded in the dataset.

Parameters: None.

Return Value: Returns the total number of movies in the dataset, of **`size_t`** data type. **`size_t`** is a standard type used in C++ to represent values such as the size of an object or the number of elements in a list. It can only take positive values and is platform independent.

`size_t DataHash2D::getUserCount() const:` Returns the total number of users recorded in the dataset.

Parameters: None.

Return Value: Returns the total number of users in the dataset, of **`size_t`** data type.

`std::vector<int> DataHash2D::getUnratedMoviesByUser(int userId) const:` Returns the movieIds that a given user has not rated yet. First, it calls the **`getUsertRatings()`** method. Then, all the movies in the dataset are scanned with a **`for`** loop. In this scan, it is checked whether the movieIds are among the movies rated by the user. If not, the movie is added to the **`unratedMovies`**, which is a **`std::vector<int>`**.

Parameters:

- **`userId:`** Unique Id of the user.

Return Value: A `std::vector<int>` containing the movieIds of all movies that the user has not rated.

`std::vector<int> DataHash2D::getUsersWhoDidNotRateMovie(int movieId) const:` Returns userIds that have not yet rated a given movie. It is based on exactly the same logic as the `getUnratedMoviesByUser()` method, but here the same operations are performed to create a list of users who have not rated a movie.

Parameters:

- `movieId:` Unique Id of the movie.

Return Value: A `std::vector<int>` containing the userIds of all users that that the movie has not rated by.

`size_t DataHash2D::getDatasetSize() const:` It calculates and returns the total number of user-movie matches (ratings) in the dataset. So, it determines how many different ratings were made. All moviesin the dataset are scanned with a `for` loop. For each movie, the `.size()` method of the `std::unordered_map` corresponding to that movieId is called to get the number of ratings given to that movie. These values are added to the `totalSize` variable.

Parameters: None.

Return Value: Returns the size of the dataset in `size_t`.

`void DataHash2D::printDataset() const:` Prints all movie-user-rating triples in the dataset to the console using two nested `for` loops. This method is used to visualize the content of the dataset.

Parameters: None.

Return Value: None.

2.2. THREADHANDLER

It is very important to use parallel process management effectively in order to increase performance in projects with high processing requirements and critical performance such as recommendation systems. The `ThreadHandler` class was created for this purpose, that is, to facilitate parallel processing within the project and to provide a common interface to all classes in this regard. The maximum number of threads that can be created with this class is limited by `std::thread::hardware_concurrency()` (inside the constructor method of the class). The minimum number of threads that can be executed is 2. This serves to automatically determine the number of threads based on the number of cores available. The default number of threads is also `std::thread::hardware_concurrency()`.

The class is based on `std::thread` structure in the standard library. Unlike `std::async` or `std::future`, `std::thread` provides a low level of thread control. The biggest factors in choosing this method are the manual division of the workload, the advantage of data integrity by allowing the use of customised locking mechanisms, i.e. `std::mutex`, and its cross-platform portability, unlike a C-based POSIX method such as `pthread`. Although `std::thread` features have not been used with fine details until now, it was preferred to use structures suitable for a larger scale project since the project is desired to be sustainable in the future.

After these explanations, we can start to examine the general structure and methods of the class. **`ThreadHandler`** class contains three private variables. These are as follows:

- **`size_t numThreads:`** Determines how many threads will be created. Since it is of **`size_t`** type, it cannot be set to negative. In the constructor method, if the value is less than 2, it is set to two, if it is 2 or more, the number of threads is decided by **`std::min(numThreads, static_cast<size_t>(std::thread::hardware_concurrency())`**.
- **`std::mutex mutex:`** A global mutex variable to be used in locking operations.
- **`std::vector<std::thread> threads:`** A list where threads are kept. Required for thread management.

Since we explain the variables, we can also explain the methods of the class:

`ThreadHandler::ThreadHandler(size_t numThreads):` It determines the number of threads depending on the **`numThreads`** parameter. If **`numThreads`** is less than 2, 2 threads are created; if **`numThreads`** is greater than 2, as many threads are created as the smaller of **`numThreads`** and **`std::thread::hardware_concurrency()`**.

Parameters:

- **`numThreads:`** The number of threads to be created.

Return Value: None.

`void ThreadHandler::runParallel(const std::function<void(size_t, size_t)>& task, size_t totalWork):` It allows to create threads and execute the given task in parallel. First the workload is divided into equal chunks, then each chunk is distributed to a thread. After all threads are completed, they are combined with the **`join`** method of **`std::thread`**.

Parameters:

- **`task:`** The function to be executed by the threads and the information of the function.
- **`totalWork:`** Total workload.

Return Value: None.

`void ThreadHandler::lock():` Executes the lock operation necessary to prevent race conditions and other concurrency problems in common data. Performs the locking operation using **`std::mutex`**.

Parameters: None.

Return Value: None.

`void ThreadHandler::unlock():` It releases **`std::mutex`**, which was previously locked with the **`lock()`** method, allowing other threads to access the locked resource. However, the lock should not be released before a thread finishes its work on the critical resource.

Parameters: None.

Return Value: None.

2.3. SIMILARITY

The **`Similarity`** class is designed to calculate the similarities between users and movies. This class establishes relationships between users and movies to create a similarity matrix. The Similarity class mainly offers several methods for calculating the similarity between two entities (that is, in this case users or movies). Most of these methods are used in recommendation systems, especially in Collaborative Filtering algorithms. One of them is the **`cosineSimilarity`** function. **`cosineSimilarity`** calculates the similarity by measuring the angle between two vectors. It is based on the direction and magnitude of the vectors. This method is widely used when measuring similarities between data points. The **`cosineSimilarity`** method is basically a transcription of the following formula:

$$\frac{A \cdot B}{\| A \| \| B \|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Another function of the class is **`similarityMatrix`**. It creates a matrix that calculates the similarity between all pairs of items in a dataset. This matrix contains the similarity values between each entity and other entities, and these values are used in the IBCF and UBCF methods to predict the ratings. Within this class, with the help of the **`ThreadHandler`** class, the calculation of the matrix is divided into threads. Thus, speed is gained in terms of performance. In addition, since similarity matrices are symmetric by nature, it is sufficient to calculate only the upper triangular part of the matrix to calculate the entire matrix. This is another factor that increases performance.

The Similarity class improves efficiency by performing parallel computations on large datasets. This class provides a reusable structure. The same similarity calculation functions can be used in different projects or with different datasets. For example, the same **`cosineSimilarity`** function can be used in both user-based and item-based recommendation systems. The Similarity class is useful in many areas such as recommendation systems, personalized recommendations, and user analysis by calculating the similarity between items.