Contents

1 Basic Test Results

```

Running presubmission script...


Opening tar file
OK
Tar extracted O.K.
For your convenience, the MD5 checksum for your submission is 12e50b412dbbb0fb2540e1c45520ab2f
Checking files...
OK
Making sure files are not empty...
OK
Checking CodingStyle...
Checking file RecommendationSystem.cpp...
Checking file RecommendationSystemLoader.cpp...
Checking file UsersLoader.cpp...
Checking file Movie.cpp...
Checking file User.cpp...
Checking file RecommendationSystem.h...
Checking file RecommendationSystemLoader.h...
Checking file UsersLoader.h...
Checking file Movie.h...
Checking file User.h...
Passed codingStyle check.
Compilation check...
Compiling...

Compilation looks good!


=====================
 Public test cases
=====================

Running test...
-------------------------
Test Movie get_name
Test Movie get_name succeeded
Test Movie get_name done
Test Movie get_year
Test Movie get_year succeeded
Test Movie get_year done
-------------------------
Test Movies with different years
Test Movies with different years succeeded
Test Movies with different years done
Test Movies with same years
Test Movies with same years succeeded
Test Movies with same years done
Test Movies symmetrically
Test Movies symmetrically succeeded
Test Movies symmetrically done
-------------------------
Test RecommendationSystemadder and getter pointer
Test RecommendationSystemadder and getter pointer succeeded
Test RecommendationSystemadder and getter pointer done
-------------------------
Test RecommendationSystemadd movie for the movie
Test RecommendationSystemadd movie for the movie succeeded
Test RecommendationSystemadd movie for the movie done
m1:A(1998)

m2:B(1996)

m1 < m2:0
m2 < m1:1
m3:C(1999)

m1 < m3:1
m3 < m1:0


OK
*****************************************
*                  ***                  *
*          Passed all tests!!           *
*              Good Job!                *
*                  ***                  *
*****************************************
```

2 ex5/README.md

```
# ex5-ido.azulai
```

3 ex5/Movie.h

```

#ifndef INC_23B_C_C__EX5_MOVIE_H
#define INC_23B_C_C__EX5_MOVIE_H

#include <iostream>
#include <vector>
#include <memory>
#include <cstring>

#define HASH_START 17

using std::ostream;
using std::string;
using std::vector;
using std::endl;

class Movie;

typedef std::shared_ptr<Movie> sp_movie; // define your smart pointer

/**
 * those declartions and typedefs are given to you and should be used in the ex
 */
typedef std::size_t (*hash_func)(const sp_movie& movie);
typedef bool (*equal_func)(const sp_movie& m1,const sp_movie& m2);
std::size_t sp_movie_hash(const sp_movie& movie);
bool sp_movie_equal(const sp_movie& m1,const sp_movie& m2);

class Movie
{
 private:
  string name;
  int year;
 public:
  /**
   * constructor
   * @param name: name of movie
   * @param year: year it was made
   */
  Movie(const string& name, int year);

  /**
   * returns the name of the movie
   * @return const ref to name of movie
   */
  string get_name() const;

  /**
   * returns the year the movie was made
   * @return year movie was made
   */
  int get_year() const;

  /**
   * operator< for two movies
   * @param rhs: right hand side
   * @param lhs: left hand side
   * @return returns true if (lhs.year) < rhs.year or
   * (rhs.year == lhs.year & lhs.name < rhs.name) else return false
```

```
     */
    bool operator< (const Movie &rhs) const;

    /**
     * operator<< for movie
     * @param os ostream to output info with
     * @param movie movie to output
     */
    friend ostream &operator<< (ostream &os, const Movie &movie);
};


#endif //INC_23B_C_C__EX5_MOVIE_H
```

4 ex5/Movie.cpp

```cpp

#include "Movie.h"
#define HASH_START 17
#define RES_MULT 31

/// Constructor
Movie::Movie(const string &movie_name, int movie_year): name(movie_name),
                                                        year(movie_year) {}

/// Movie name getter
string Movie::get_name() const
{
  return name;
}

/// Movie year getter
int Movie::get_year() const
{
  return year;
}

/// returns false if (movies are the same, or rhs is totally bigger than lhs)
/// true if lhs is totally bigger than rhs
bool Movie::operator<(const Movie& rhs) const
{
  return year < rhs.year || (year <= rhs.year && name < rhs.name);
}


/// prints a movie by the format
ostream &operator<<(ostream &os, const Movie &movie)
{
  return os << movie.name << "(" << movie.year << ")" << endl;
}


/**
 * hash function used for a unordered_map (implemented for you)
 * @param movie shared pointer to movie
 * @return an integer for the hash map
 */
std::size_t sp_movie_hash(const sp_movie& movie){
  std::size_t res = HASH_START;
  res = res * RES_MULT + std::hash<std::string>()(movie->get_name());
  res = res * RES_MULT + std::hash<int>()(movie->get_year());
  return res;
}

/**
 * equal function used for an unordered_map (implemented for you)
 * @param m1
 * @param m2
 * @return true if the year and name are equal else false
 */
bool sp_movie_equal(const sp_movie& m1, const sp_movie& m2){
  return !(*m1 < *m2) && !(*m2 < *m1);
}
```

5 ex5/RecommendationSystem.h

```
//
// Created on 2/20/2022.
//

#ifndef RECOMMENDATIONSYSTEM_H
#define RECOMMENDATIONSYSTEM_H
#include "User.h"

using std::map;

typedef bool (*movies_less_than)(sp_movie, sp_movie);
typedef map<sp_movie, vector<double>, movies_less_than> rec_sys;

class RecommendationSystem
{
 private:
  rec_sys rec_sys_ins;

  // comperator for sp_movie in the map so it would know how to order it
  static bool movies_less_than(sp_movie m1, sp_movie m2);

  vector<double> calc_pref_vec(const User &user_rankings, double avg);

  sp_movie find_most_similar(const User &user_rankings,
                             const vector<double> &pref_vec);

  double calc_angle (const vector<double> &watched,
                     const vector<double> &possible_movie);

  static bool has_rank (const User &user, const sp_movie &movie);

 public:

  /**
   * Default constructor, initializes an empty map.
   */
  RecommendationSystem();

  /**
   *
   * @param user_rankings
   * @return
   */
  sp_movie get_recommendation_by_content (const User& user_rankings);

  //explicit RecommendationSystem()
  /**
   * adds a new movie to the rec_sys_ins
   * @param name name of movie
   * @param year year it was made
   * @param features features for movie
   * @return shared pointer for movie in rec_sys_ins
   */
  sp_movie add_movie(const string& name,int year,
                     const vector<double>& features);


  /**
   * a function that calculates the movie with highest score
```

```
     * based on movie features
     * @param ranks user ranking to use for algorithm
     * @return shared pointer to movie in rec_sys_ins
     */
    sp_movie recommend_by_content(const User& user);

    /**
     * a function that calculates the movie with highest predicted score
     * based on ranking of other movies
     * @param ranks user ranking to use for algorithm
     * @param k
     * @return shared pointer to movie in rec_sys_ins
     */
    sp_movie recommend_by_cf(const User& user, int k);


    /**
     * Predict a user rating for a movie given argument using item cf
     * procedure with k most similar movies.
     * @param user_rankings: ranking to use
     * @param movie: movie to predict
     * @param k:
     * @return score based on algorithm as described in pdf
     */
    double predict_movie_score(const User &user, const sp_movie &movie,
                               int k);

    /**
     * gets a shared pointer to movie in rec_sys_ins
     * @param name name of movie
     * @param year year movie was made
     * @return shared pointer to movie in rec_sys_ins
     */
    sp_movie get_movie(const string &name, int year) const;


    friend ostream &operator<< (ostream &os, const RecommendationSystem &sys);

};


#endif //RECOMMENDATIONSYSTEM_H
```

6 ex5/RecommendationSystem.cpp

```cpp
//
// Created by ido.azulai on 3/24/24.
//
#include "RecommendationSystem.h"

using std::make_shared;
using std::transform;
using std::sqrt;

// Comparison function for movies
bool
RecommendationSystem::movies_less_than (sp_movie m1, sp_movie m2)
{
  return *m1 < *m2;
}

RecommendationSystem::RecommendationSystem (): rec_sys_ins(movies_less_than){}

sp_movie RecommendationSystem::add_movie(const string &name,
                                         int year,
                                         const vector<double> &features)
{
  sp_movie movie_to_add = make_shared<Movie>(name, year);
  rec_sys_ins[movie_to_add] = features;
  return movie_to_add;
}

/// Calculates average of rankings in an unordered map
double avg_ranks(const rank_map& user_rankings)
{
  double sum = 0;
  for (const auto& it : user_rankings)
  {
    sum += it.second;
  }
  if (sum == 0)
  {
    return 0;
  }
  return sum / user_rankings.size();
}


/// Function to add two vectors element-wise
vector<double>& operator+=(vector<double>& lhs, const vector<double>& rhs)
{
  // Ensure both vectors have the same size
  if (lhs.size() != rhs.size())
  {
    throw std::invalid_argument("Vectors must have "
                                "the same size for element-wise addition.");
  }

  // Element-wise addition
  for (size_t i = 0; i < lhs.size(); ++i)
  {
    lhs[i] += rhs[i];
  }
  return lhs;
}

/// creates the pref vector by adding each normalized movie vec to pref vec
vector<double>
RecommendationSystem::calc_pref_vec (const User &user_rankings, double avg)
{
  // normalize the rankings map (original minus avg), and add to pref vector
  // init pref vector
  vector<double> pref_vec(rec_sys_ins.begin()->second.size(), 0);
  for (const auto& it : user_rankings.get_rank())
  {
    /// TODO check if normalized_ranks[it.first] is not NA
    /// TODO if so, calc normalized value and mult in the specific
    /// movie's vector, and add it to pref_vec
    // hold scalar value
    double scalar = it.second - avg;

    // mult scalar and movie's ranking vector manually
    // and update pref vector by element-wise addition manually
    for (size_t i = 0; i < rec_sys_ins[it.first].size(); ++i)
    {
      pref_vec[i] += rec_sys_ins[it.first][i] * scalar;
    }
  }
  return pref_vec;
}


/// Calculate the dot product
double vec_dot(const vector<double>& v1, const vector<double>& v2)
{
  double result = 0.0;
  for (size_t i = 0; i < v1.size (); ++i)
  {
    result += v1[i] * v2[i];
  }
  return result;
}

/// Function to calculate the norm (magnitude) of a vector
double vector_norm(const vector<double>& vec)
{
  double sum_of_squares = 0.0;
  for (double value : vec) {
    sum_of_squares += value * value;
  }
  return sqrt(sum_of_squares);
}

/// Checks if user already ranked this movie (from rec_sys)
bool RecommendationSystem::has_rank(const User& user, const sp_movie& movie)
{
  try
  {
    // Attempt to retrieve the ranking for the movie from the user
    user.get_rank().at(movie);

    // If the ranking retrieval didn't throw an exception,
    // it means the user has ranked the movie
    return true;
  }
  catch (const std::out_of_range&)
  {
    // If the movie is not found in the user's rankings, return false
    return false;
  }
}

```

```cpp
/// calc the angle between each movie and return a sp_movie of the closest
sp_movie
RecommendationSystem::find_most_similar(const User &user_rankings,
                                        const vector<double>& pref_vec)
{
  double max_angle = -1.0; // Any angle calculated will be greater than this
  double v1_size = vector_norm(pref_vec);
  sp_movie closest_movie = nullptr; // Initialize with nullptr

  // Iterate over the movies the user didn't watch
  for (const auto& movie_entry : rec_sys_ins)
  {
    // Check if the user has not ranked this movie
    if (!has_rank(user_rankings, movie_entry.first))
    {
      // Calculate dot product and vector size for the current movie
      double dot_product = vec_dot(pref_vec, movie_entry.second);
      double v2_size = vector_norm(movie_entry.second);

      // Calculate angle between preference vector and current movie's vector
      double angle = dot_product / (v1_size * v2_size);

      // Update closest movie if the current movie has a higher angle
      if (angle > max_angle)
      {
        max_angle = angle;
        closest_movie = movie_entry.first;
      }
    }
  }
  return closest_movie;
}

/// Calculates the closest movie for user, by rating similarity
sp_movie RecommendationSystem::
get_recommendation_by_content(const User &user_rankings)
{
  // calc average ranking of all movies the user ranked
  double avg = avg_ranks(user_rankings.get_rank()); // calc average

  // normalize the rankings map (original minus avg), and add to pref vector
  vector<double> pref_vec = calc_pref_vec(user_rankings, avg);

  // calculate the difference between pref_vec and other movies vectors
  return find_most_similar(user_rankings, pref_vec);
}

///calc angle between 2 movies, for the cf recommendation system
double RecommendationSystem::calc_angle (const vector<double> &watched,
                                         const vector<double> &possible_movie)
{
  double v1_size = vector_norm(watched); // pre-calc the norm
  double v2_size = vector_norm(possible_movie);
  double dot_product = vec_dot(watched, possible_movie);

  return dot_product / (v1_size * v2_size);
}

/// predicts a score for a movie, based on user's rank map
double RecommendationSystem::predict_movie_score (const User &user,
                                                  const sp_movie &movie,int k)
{
  // calculate angle between each movie user didn't watch to all movies he did
  map<double, sp_movie> movie_recs; // an ordered map, holds all angles
  double top = 0.0, base = 0.0;
  int cnt = 0;

  for (const auto& watched_movie : user.get_rank())
  {
    // calc angle
    if (watched_movie.first != movie){
      double angle = calc_angle(rec_sys_ins[watched_movie.first],
                                rec_sys_ins[movie]);
      movie_recs[angle] = watched_movie.first;
      cnt++;
    }
  }

  // calculate predicted rank for the movie we got, for k top user rating's
  auto possible_rank = movie_recs.end();
  --possible_rank;
  k = (k > cnt) ? cnt : k; // if k is bigger than amount of movies user ranked
  for (int i = 0; i < k; ++i)
  {
    top += possible_rank->first *
           user.get_rank()[possible_rank->second];
    base += possible_rank->first;
    --possible_rank;
  }

  // calc rank (sum(similar * rank) / sum(similar))
  return top/base;
}

/// recommends a movie from the movies the user didn't watch, by cf
sp_movie RecommendationSystem::recommend_by_cf (const User &user, int k)
{
  double max_rate = 0.0, angle = 0.0;
  sp_movie recommended_movie = nullptr;
  for (const auto& it : rec_sys_ins)
  {
    try {
      user.get_rank().at (it.first); // look for the sp_movie in user's ranks
    }
    catch (const std::out_of_range& e) {
      // means we have in it.first a movie the user didn't rate
      angle = predict_movie_score (user, it.first, k);
      if (angle > max_rate)
      {
        max_rate = angle;
        recommended_movie = it.first;
      }
    }
  }

  // return the sp+movie that got the highest score
  return recommended_movie;
}


///// return a smart pointer to a movie, by it's name and year
sp_movie RecommendationSystem::get_movie(const string& name, int year) const
{
  sp_movie movie = make_shared<Movie>(name, year);

  // Loop through each entry in rec_sys_ins to find the movie
  for (const auto& pair : rec_sys_ins)
  {
    const sp_movie& current_movie = pair.first;
    if (current_movie->get_name() == name && current_movie->get_year() == year)
    {
      // Movie found, return the smart pointer to the movie
      return current_movie;
    }
  }

  // Movie not found, return nullptr
  return nullptr;
}

/// output stream operator (friend)
ostream &operator<< (ostream &os, const RecommendationSystem &sys)
{
  for (const auto &it : sys.rec_sys_ins)
  {
    os << *it.first;
  }
  return os;
}
```

7 ex5/RecommendationSystemLoader.h

```cpp

#ifndef RECOMMENDATIONSYSTEMLOADER_H
#define RECOMMENDATIONSYSTEMLOADER_H
#include "RecommendationSystem.h"

class RecommendationSystemLoader {

 private:

 public:
  RecommendationSystemLoader () = delete;
  /**
   * loads movies by the given format for movies with their feature's score
   * @param movies_file_path a path to the file of the movies
   * @return smart pointer to a RecommendationSystem which was created with
   * those movies
   */
  static std::unique_ptr<RecommendationSystem> create_rs_from_movies
      (const std::string &movies_file_path) noexcept (false);
};

#endif //RECOMMENDATIONSYSTEMLOADER_H
```

8 ex5/RecommendationSystemLoader.cpp

```cpp

#include "RecommendationSystemLoader.h"
#include <fstream>
#include <sstream>
#define YEAR_SEPARATOR '-'
#define ERROR_MSG "input file is incorrect"


std::unique_ptr<RecommendationSystem>
    RecommendationSystemLoader::create_rs_from_movies(
        const std::string &movies_file_path) noexcept(false)
{
    std::unique_ptr<RecommendationSystem> rs =
        std::make_unique<RecommendationSystem>();
    std::ifstream in_file;
    in_file.open(movies_file_path);
    std::string buffer;

    while (getline(in_file, buffer))
    {
        std::string movie_det;
        double ranking;
        std::istringstream splitted_line(buffer);
        splitted_line >> movie_det;
        std::vector<double> vec;
        while (splitted_line >> ranking)
        {
            if (ranking <= 0){
                throw std::invalid_argument(ERROR_MSG);
            }
            vec.push_back(ranking);
        }
        size_t end = buffer.find(YEAR_SEPARATOR);
        rs->add_movie(buffer.substr(0, end),
                       std::stoi(buffer.substr(end + 1, buffer.length())),vec);
    }
    in_file.close();
    return rs;
}
```

9 ex5/User.h

```cpp
//
// Created on 2/20/2022.
//

#ifndef USER_H
#define USER_H
#include <unordered_map>
#include <map>
#include <vector>
#include <string>
#include <memory>
#include "Movie.h"
#include <algorithm>
#include <cmath>

#define MAX_RATE 10
#define MIN_RATE 1

using std::unordered_map;

class RecommendationSystem;
typedef unordered_map<sp_movie, double, hash_func, equal_func> rank_map;

using std::string;

class User
{
 private:
  string user_name;
  rank_map user_rank_map;
  std::shared_ptr<RecommendationSystem> rec_sys;

 public:

  /**
   * Constructor for the class
   */
  User(const string& name, const rank_map &rank_map_in,
       std::shared_ptr<RecommendationSystem> &rec_sys_input);

  /**
   * function for adding a movie to the DB
   * @param name name of movie
   * @param year year it was made
   * @param features a vector of the movie's features
   * @param rate the user rate for this movie
   */
  void add_movie_to_rs(const string &name, int year,
                       const vector<double> &features,
                       double rate);

  /**
   * a getter for the user's name
   * @return the username
   */
  string get_name() const;


  /**
```

```
     * a getter for the ranks map
     * @return the user's rank map
     */
    rank_map get_rank() const;

    /**
     * returns a recommendation according to the movie's content
     * @return recommendation
     */
    sp_movie get_recommendation_by_content() const;

    /**
     * returns a recommendation according to the similarity
     * recommendation method
     * @param k the number of the most similar movies to calculate by
     * @return recommendation
     */
    sp_movie get_recommendation_by_cf(int k) const;

    /**
     * predicts the score for a given movie
     * @param name the name of the movie
     * @param year the year the movie was created
     * @param k the parameter which represents the number of the most
     * similar movies to predict the score by
     * @return predicted score for the given movie
     */
    double get_prediction_score_for_movie(const string& name,
                                          int year, int k) const;

    /**
     * output stream operator
     * @param os the output streamRecommendationSystem
     * @param user the user
     * @return output stream
     */
    friend ostream &operator<< (ostream &os, const User &user);
};



#endif //USER_H
```

10 ex5/User.cpp

```cpp


// don't change those includes
#include "User.h"
#include "RecommendationSystem.h"


/// Constructor
/// TODO: do i need to deepcopy rankmap? if so, a for loop
/// TODO: that copies each key:val from rank_map_in
User::User(const string& name,
           const rank_map &rank_map_in,
           std::shared_ptr<RecommendationSystem> &rec_sys_input):
user_name(name),
user_rank_map(rank_map_in),
rec_sys(rec_sys_input)
{}

void User::add_movie_to_rs(const string &name, int year,
                           const vector<double> &features,
                           double rate)
{
    if (rate < MIN_RATE || rate > MAX_RATE || std::isnan(rate)){
        //name = nullptr || year < 0 || features = nullptr
        return;
    }
    sp_movie new_m = rec_sys->add_movie(name, year, features);
    user_rank_map[new_m] = rate;
}

/// return the username
string User::get_name() const
{
    return user_name;
}

/// returns user's rank map
rank_map User::get_rank() const
{
    return user_rank_map;
}

/// returns a pointer to a recommended movie by the algorithm
sp_movie User::get_recommendation_by_content() const
{
    return rec_sys->get_recommendation_by_content(*this);
}

/// returns a recommendation according to the similarity recommendation method
sp_movie User::get_recommendation_by_cf(int k) const
{
  return rec_sys->recommend_by_cf (*this, k);
}

/// predicts the score for a given movie
double User::get_prediction_score_for_movie(const string &name,
                                            int year, int k) const
{
    sp_movie movie_ptr = rec_sys->get_movie(name, year);
    return rec_sys->predict_movie_score(*this, movie_ptr, k);
}

/// output stream operator (friend)
ostream &operator<< (ostream &os, const User &user){
    return os << user.user_name << endl << *user.rec_sys << endl;
}
```

11 ex5/UsersLoader.h

```cpp
//
// Created on 2/21/2022.
//




#ifndef USERFACTORY_H
#define USERFACTORY_H

#include <sstream>
#include <fstream>
#include <vector>
#include "User.h"
#include "RecommendationSystem.h"

#define YEAR_SEPARATOR '-'


class UsersLoader
{
private:


public:
    UsersLoader() = delete;
    /**
     *
     * loads users by the given format with their movie's ranks
     * @param users_file_path a path to the file of the users and their
     * movie ranks
     * @param rs RecommendingSystem for the Users
     * @return vector of the users created according to the file
     */
    static std::vector<User> create_users(const std::string& users_file_path,
                                      std::unique_ptr<RecommendationSystem> rs)
                                              noexcept(false);

};


#endif //USERFACTORY_H
```

12 ex5/UsersLoader.cpp

```cpp

#include "UsersLoader.h"

#define INIT_BUCKET_SIZE 8
#define ERROR_MSG "input file is incorrect"


std::vector<User>
UsersLoader::create_users
(const std::string &users_file_path, std::unique_ptr<RecommendationSystem> rs)
noexcept(false)
{
    std::shared_ptr<RecommendationSystem> s_rs = std::move (rs);
    std::ifstream in_file;
    std::vector<User> users;
    in_file.open(users_file_path);
    std::string buffer;
    getline(in_file, buffer);
    std::istringstream movies_names(buffer);
    std::vector<sp_movie> movies;
    while (movies_names >> buffer)
    {
        size_t end = buffer.find(YEAR_SEPARATOR);
        sp_movie m = s_rs->get_movie(buffer.substr(0, end),
                                     std::stoi(buffer.substr(end + 1,
                                                              buffer.length())));
        movies.push_back(m);
    }
    while (getline(in_file, buffer))
    {
        std::string user_name;
        std::string ranking;
        std::istringstream splitted_line(buffer);
        splitted_line >> user_name;
        int i = 0;
        rank_map ranks(INIT_BUCKET_SIZE,sp_movie_hash,sp_movie_equal);
        while (splitted_line >> ranking)
        {

            if (ranking != "NA")
            {
                int rating = std::stoi(ranking);
                if (rating <= 0){
                    throw std::invalid_argument(ERROR_MSG);
                }
                ranks[movies[i]] = rating;
            }
            i++;
        }
        users.emplace_back(user_name, ranks, s_rs);
    }
    in_file.close();
    return users;
}
```