

Contents

1	Basic Test Results	2
2	ex5/README.md	4
3	ex5/Movie.h	5
4	ex5/Movie.cpp	7
5	ex5/RecommendationSystem.h	8
6	ex5/RecommendationSystem.cpp	10
7	ex5/RecommendationSystemLoader.h	15
8	ex5/RecommendationSystemLoader.cpp	16
9	ex5/User.h	17
10	ex5/User.cpp	19
11	ex5/UsersLoader.h	21
12	ex5/UsersLoader.cpp	22

1 Basic Test Results

```
1
2 Running presubmission script...
3
4
5 Opening tar file
6 OK
7 Tar extracted O.K.
8 For your convenience, the MD5 checksum for your submission is 12e50b412dbbb0fb2540e1c45520ab2f
9 Checking files...
10 OK
11 Making sure files are not empty...
12 OK
13 Checking CodingStyle...
14 Checking file RecommendationSystem.cpp...
15 Checking file RecommendationSystemLoader.cpp...
16 Checking file UsersLoader.cpp...
17 Checking file Movie.cpp...
18 Checking file User.cpp...
19 Checking file RecommendationSystem.h...
20 Checking file RecommendationSystemLoader.h...
21 Checking file UsersLoader.h...
22 Checking file Movie.h...
23 Checking file User.h...
24 Passed codingStyle check.
25 Compilation check...
26 Compiling...
27
28 Compilation looks good!
29
30
31 =====
32 Public test cases
33 =====
34
35 Running test...
36 -----
37 Test Movie get_name
38 Test Movie get_name succeeded
39 Test Movie get_name done
40 Test Movie get_year
41 Test Movie get_year succeeded
42 Test Movie get_year done
43 -----
44 Test Movies with different years
45 Test Movies with different years succeeded
46 Test Movies with different years done
47 Test Movies with same years
48 Test Movies with same years succeeded
49 Test Movies with same years done
50 Test Movies symmetrically
51 Test Movies symmetrically succeeded
52 Test Movies symmetrically done
53 -----
54 Test RecommendationSystemadder and getter pointer
55 Test RecommendationSystemadder and getter pointer succeeded
56 Test RecommendationSystemadder and getter pointer done
57 -----
58 Test RecommendationSystemadd movie for the movie
59 Test RecommendationSystemadd movie for the movie succeeded
```

```

60 Test RecommendationSystemadd movie for the movie done
61 m1:A(1998)
62
63 m2:B(1996)
64
65 m1 < m2:0
66 m2 < m1:1
67 m3:C(1999)
68
69 m1 < m3:1
70 m3 < m1:0
71
72
73 OK
74 *****
75 *           ***           *
76 *           Passed all tests!!           *
77 *           Good Job!           *
78 *           ***           *
79 *****

```

2 ex5/README.md

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

3 ex5/Movie.h

```
1
2  #ifndef INC_23B_C_C_EX5_MOVIE_H
3  #define INC_23B_C_C_EX5_MOVIE_H
4
5  #include <iostream>
6  #include <vector>
7  #include <memory>
8  #include <cstring>
9
10 #define HASH_START 17
11
12 using std::ostream;
13 using std::string;
14 using std::vector;
15 using std::endl;
16
17 class Movie;
18
19 typedef std::shared_ptr<Movie> sp_movie; // define your smart pointer
20
21 /**
22  * those declarations and typedefs are given to you and should be used in the ex
23  */
24 typedef std::size_t (*hash_func)(const sp_movie& movie);
25 typedef bool (*equal_func)(const sp_movie& m1, const sp_movie& m2);
26 std::size_t sp_movie_hash(const sp_movie& movie);
27 bool sp_movie_equal(const sp_movie& m1, const sp_movie& m2);
28
29 class Movie
30 {
31 private:
32     string name;
33     int year;
34 public:
35     /**
36      * constructor
37      * @param name: name of movie
38      * @param year: year it was made
39      */
40     Movie(const string& name, int year);
41
42     /**
43      * returns the name of the movie
44      * @return const ref to name of movie
45      */
46     string get_name() const;
47
48     /**
49      * returns the year the movie was made
50      * @return year movie was made
51      */
52     int get_year() const;
53
54     /**
55      * operator< for two movies
56      * @param rhs: right hand side
57      * @param lhs: left hand side
58      * @return returns true if (lhs.year) < rhs.year or
59      * (rhs.year == lhs.year & lhs.name < rhs.name) else return false
```

```

60     */
61     bool operator< (const Movie &rhs) const;
62
63     /**
64     * operator< for movie
65     * @param os ostream to output info with
66     * @param movie movie to output
67     */
68     friend ostream &operator<< (ostream &os, const Movie &movie);
69 };
70
71
72 #endif //INC_23B_C_C_EX5_MOVIE_H

```

4 ex5/Movie.cpp

```
1
2  #include "Movie.h"
3  #define HASH_START 17
4  #define RES_MULT 31
5
6  /// Constructor
7  Movie::Movie(const string &movie_name, int movie_year): name(movie_name),
8                                                         year(movie_year) {}
9
10 /// Movie name getter
11 string Movie::get_name() const
12 {
13     return name;
14 }
15
16 /// Movie year getter
17 int Movie::get_year() const
18 {
19     return year;
20 }
21
22 /// returns false if (movies are the same, or rhs is totally bigger than lhs)
23 /// true if lhs is totally bigger than rhs
24 bool Movie::operator<(const Movie& rhs) const
25 {
26     return year < rhs.year || (year <= rhs.year && name < rhs.name);
27 }
28
29
30 /// prints a movie by the format
31 ostream &operator<<(ostream &os, const Movie &movie)
32 {
33     return os << movie.name << "(" << movie.year << ")" << endl;
34 }
35
36
37 /**
38  * hash function used for a unordered_map (implemented for you)
39  * @param movie shared pointer to movie
40  * @return an integer for the hash map
41  */
42 std::size_t sp_movie_hash(const sp_movie& movie){
43     std::size_t res = HASH_START;
44     res = res * RES_MULT + std::hash<std::string>()(movie->get_name());
45     res = res * RES_MULT + std::hash<int>()(movie->get_year());
46     return res;
47 }
48
49 /**
50  * equal function used for an unordered_map (implemented for you)
51  * @param m1
52  * @param m2
53  * @return true if the year and name are equal else false
54  */
55 bool sp_movie_equal(const sp_movie& m1, const sp_movie& m2){
56     return !(*m1 < *m2) && !(*m2 < *m1);
57 }
```

5 ex5/RecommendationSystem.h

```
1  //
2  // Created on 2/20/2022.
3  //
4
5  #ifndef RECOMMENDATIONSYSTEM_H
6  #define RECOMMENDATIONSYSTEM_H
7  #include "User.h"
8
9  using std::map;
10
11 typedef bool (*movies_less_than)(sp_movie, sp_movie);
12 typedef map<sp_movie, vector<double>, movies_less_than> rec_sys;
13
14 class RecommendationSystem
15 {
16 private:
17     rec_sys rec_sys_ins;
18
19     // comperator for sp_movie in the map so it would know how to order it
20     static bool movies_less_than(sp_movie m1, sp_movie m2);
21
22     vector<double> calc_pref_vec(const User &user_rankings, double avg);
23
24     sp_movie find_most_similar(const User &user_rankings,
25                               const vector<double> &pref_vec);
26
27     double calc_angle (const vector<double> &watched,
28                       const vector<double> &possible_movie);
29
30     static bool has_rank (const User &user, const sp_movie &movie);
31
32 public:
33
34     /**
35      * Default constructor, initializes an empty map.
36      */
37     RecommendationSystem();
38
39     /**
40      *
41      * @param user_rankings
42      * @return
43      */
44     sp_movie get_recommendation_by_content (const User& user_rankings);
45
46     //explicit RecommendationSystem()
47     /**
48      * adds a new movie to the rec_sys_ins
49      * @param name name of movie
50      * @param year year it was made
51      * @param features features for movie
52      * @return shared pointer for movie in rec_sys_ins
53      */
54     sp_movie add_movie(const string& name,int year,
55                       const vector<double>& features);
56
57
58     /**
59      * a function that calculates the movie with highest score
```



```

60     * based on movie features
61     * @param ranks user ranking to use for algorithm
62     * @return shared pointer to movie in rec_sys_ins
63     */
64     sp_movie recommend_by_content(const User& user);
65
66     /**
67     * a function that calculates the movie with highest predicted score
68     * based on ranking of other movies
69     * @param ranks user ranking to use for algorithm
70     * @param k
71     * @return shared pointer to movie in rec_sys_ins
72     */
73     sp_movie recommend_by_cf(const User& user, int k);
74
75
76     /**
77     * Predict a user rating for a movie given argument using item cf
78     * procedure with k most similar movies.
79     * @param user_rankings: ranking to use
80     * @param movie: movie to predict
81     * @param k:
82     * @return score based on algorithm as described in pdf
83     */
84     double predict_movie_score(const User &user, const sp_movie &movie,
85                               int k);
86
87     /**
88     * gets a shared pointer to movie in rec_sys_ins
89     * @param name name of movie
90     * @param year year movie was made
91     * @return shared pointer to movie in rec_sys_ins
92     */
93     sp_movie get_movie(const string &name, int year) const;
94
95
96     friend ostream &operator<< (ostream &os, const RecommendationSystem &sys);
97
98 };
99
100
101 #endif //RECOMMENDATIONSYSTEM_H

```

6 ex5/RecommendationSystem.cpp

```
1  //
2  // Created by ido.azulai on 3/24/24.
3  //
4  #include "RecommendationSystem.h"
5
6  using std::make_shared;
7  using std::transform;
8  using std::sqrt;
9
10 // Comparison function for movies
11 bool
12 RecommendationSystem::movies_less_than (sp_movie m1, sp_movie m2)
13 {
14     return *m1 < *m2;
15 }
16
17 RecommendationSystem::RecommendationSystem (): rec_sys_ins(movies_less_than){}
18
19 sp_movie RecommendationSystem::add_movie(const string &name,
20                                           int year,
21                                           const vector<double> &features)
22 {
23     sp_movie movie_to_add = make_shared<Movie>(name, year);
24     rec_sys_ins[movie_to_add] = features;
25     return movie_to_add;
26 }
27
28 /// Calculates average of rankings in an unordered map
29 double avg_ranks(const rank_map& user_rankings)
30 {
31     double sum = 0;
32     for (const auto& it : user_rankings)
33     {
34         sum += it.second;
35     }
36     if (sum == 0)
37     {
38         return 0;
39     }
40     return sum / user_rankings.size();
41 }
42
43
44 /// Function to add two vectors element-wise
45 vector<double>& operator+=(vector<double>& lhs, const vector<double>& rhs)
46 {
47     // Ensure both vectors have the same size
48     if (lhs.size() != rhs.size())
49     {
50         throw std::invalid_argument("Vectors must have "
51                                     "the same size for element-wise addition.");
52     }
53
54     // Element-wise addition
55     for (size_t i = 0; i < lhs.size(); ++i)
56     {
57         lhs[i] += rhs[i];
58     }
59     return lhs;
60 }
```

```

60 }
61
62 /// creates the pref vector by adding each normalized movie vec to pref vec
63 vector<double>
64 RecommendationSystem::calc_pref_vec (const User &user_rankings, double avg)
65 {
66     /// normalize the rankings map (original minus avg), and add to pref vector
67     /// init pref vector
68     vector<double> pref_vec(rec_sys_ins.begin()->second.size(), 0);
69     for (const auto& it : user_rankings.get_rank())
70     {
71         /// TODO check if normalized_ranks[it.first] is not NA
72         /// TODO if so, calc normalized value and mult in the specific
73         /// movie's vector, and add it to pref_vec
74         /// hold scalar value
75         double scalar = it.second - avg;
76
77         /// mult scalar and movie's ranking vector manually
78         /// and update pref vector by element-wise addition manually
79         for (size_t i = 0; i < rec_sys_ins[it.first].size(); ++i)
80         {
81             pref_vec[i] += rec_sys_ins[it.first][i] * scalar;
82         }
83     }
84     return pref_vec;
85 }
86
87
88 /// Calculate the dot product
89 double vec_dot(const vector<double>& v1, const vector<double>& v2)
90 {
91     double result = 0.0;
92     for (size_t i = 0; i < v1.size (); ++i)
93     {
94         result += v1[i] * v2[i];
95     }
96     return result;
97 }
98
99 /// Function to calculate the norm (magnitude) of a vector
100 double vector_norm(const vector<double>& vec)
101 {
102     double sum_of_squares = 0.0;
103     for (double value : vec) {
104         sum_of_squares += value * value;
105     }
106     return sqrt(sum_of_squares);
107 }
108
109 /// Checks if user already ranked this movie (from rec_sys)
110 bool RecommendationSystem::has_rank(const User& user, const sp_movie& movie)
111 {
112     try
113     {
114         /// Attempt to retrieve the ranking for the movie from the user
115         user.get_rank().at(movie);
116
117         /// If the ranking retrieval didn't throw an exception,
118         /// it means the user has ranked the movie
119         return true;
120     }
121     catch (const std::out_of_range&)
122     {
123         /// If the movie is not found in the user's rankings, return false
124         return false;
125     }
126 }
127

```

```

128 /// calc the angle between each movie and return a sp_movie of the closest
129 sp_movie
130 RecommendationSystem::find_most_similar(const User &user_rankings,
131                                         const vector<double>& pref_vec)
132 {
133     double max_angle = -1.0; /// Any angle calculated will be greater than this
134     double v1_size = vector_norm(pref_vec);
135     sp_movie closest_movie = nullptr; /// Initialize with nullptr
136
137     /// Iterate over the movies the user didn't watch
138     for (const auto& movie_entry : rec_sys_ins)
139     {
140         /// Check if the user has not ranked this movie
141         if (!has_rank(user_rankings, movie_entry.first))
142         {
143             /// Calculate dot product and vector size for the current movie
144             double dot_product = vec_dot(pref_vec, movie_entry.second);
145             double v2_size = vector_norm(movie_entry.second);
146
147             /// Calculate angle between preference vector and current movie's vector
148             double angle = dot_product / (v1_size * v2_size);
149
150             /// Update closest movie if the current movie has a higher angle
151             if (angle > max_angle)
152             {
153                 max_angle = angle;
154                 closest_movie = movie_entry.first;
155             }
156         }
157     }
158     return closest_movie;
159 }
160
161 /// Calculates the closest movie for user, by rating similarity
162 sp_movie RecommendationSystem::
163 get_recommendation_by_content(const User &user_rankings)
164 {
165     /// calc average ranking of all movies the user ranked
166     double avg = avg_ranks(user_rankings.get_rank()); /// calc average
167
168     /// normalize the rankings map (original minus avg), and add to pref vector
169     vector<double> pref_vec = calc_pref_vec(user_rankings, avg);
170
171     /// calculate the difference between pref_vec and other movies vectors
172     return find_most_similar(user_rankings, pref_vec);
173 }
174
175 ///calc angle between 2 movies, for the cf recommendation system
176 double RecommendationSystem::calc_angle (const vector<double> &watched,
177                                           const vector<double> &possible_movie)
178 {
179     double v1_size = vector_norm(watched); /// pre-calc the norm
180     double v2_size = vector_norm(possible_movie);
181     double dot_product = vec_dot(watched, possible_movie);
182
183     return dot_product / (v1_size * v2_size);
184 }
185
186 /// predicts a score for a movie, based on user's rank map
187 double RecommendationSystem::predict_movie_score (const User &user,
188                                                    const sp_movie &movie, int k)
189 {
190     /// calculate angle between each movie user didn't watch to all movies he did
191     map<double, sp_movie> movie_recs; /// an ordered map, holds all angles
192     double top = 0.0, base = 0.0;
193     int cnt = 0;
194
195     for (const auto& watched_movie : user.get_rank())

```

```

196 {
197     // calc angle
198     if (watched_movie.first != movie){
199         double angle = calc_angle(rec_sys_ins[watched_movie.first],
200                                 rec_sys_ins[movie]);
201         movie_recs[angle] = watched_movie.first;
202         cnt++;
203     }
204 }
205
206 // calculate predicted rank for the movie we got, for k top user rating's
207 auto possible_rank = movie_recs.end();
208 --possible_rank;
209 k = (k > cnt) ? cnt : k; // if k is bigger than amount of movies user ranked
210 for (int i = 0; i < k; ++i)
211 {
212     top += possible_rank->first *
213           user.get_rank()[possible_rank->second];
214     base += possible_rank->first;
215     --possible_rank;
216 }
217
218 // calc rank (sum(similar * rank) / sum(similar))
219 return top/base;
220 }
221
222 /// recommends a movie from the movies the user didn't watch, by cf
223 sp_movie RecommendationSystem::recommend_by_cf (const User &user, int k)
224 {
225     double max_rate = 0.0, angle = 0.0;
226     sp_movie recommended_movie = nullptr;
227     for (const auto& it : rec_sys_ins)
228     {
229         try {
230             user.get_rank().at (it.first); // look for the sp_movie in user's ranks
231         }
232         catch (const std::out_of_range& e) {
233             // means we have in it.first a movie the user didn't rate
234             angle = predict_movie_score (user, it.first, k);
235             if (angle > max_rate)
236             {
237                 max_rate = angle;
238                 recommended_movie = it.first;
239             }
240         }
241     }
242
243     // return the sp+movie that got the highest score
244     return recommended_movie;
245 }
246
247
248 ///// return a smart pointer to a movie, by it's name and year
249 sp_movie RecommendationSystem::get_movie(const string& name, int year) const
250 {
251     sp_movie movie = make_shared<Movie>(name, year);
252
253     // Loop through each entry in rec_sys_ins to find the movie
254     for (const auto& pair : rec_sys_ins)
255     {
256         const sp_movie& current_movie = pair.first;
257         if (current_movie->get_name() == name && current_movie->get_year() == year)
258         {
259             // Movie found, return the smart pointer to the movie
260             return current_movie;
261         }
262     }
263 }

```

```

264     // Movie not found, return nullptr
265     return nullptr;
266 }
267
268 /// output stream operator (friend)
269 ostream &operator<< (ostream &os, const RecommendationSystem &sys)
270 {
271     for (const auto &it : sys.rec_sys_ins)
272     {
273         os << *it.first;
274     }
275     return os;
276 }

```

7 ex5/RecommendationSystemLoader.h

```
1
2 #ifndef RECOMMENDATIONSYSTEMLOADER_H
3 #define RECOMMENDATIONSYSTEMLOADER_H
4 #include "RecommendationSystem.h"
5
6 class RecommendationSystemLoader {
7
8     private:
9
10    public:
11        RecommendationSystemLoader () = delete;
12        /**
13         * loads movies by the given format for movies with their feature's score
14         * @param movies_file_path a path to the file of the movies
15         * @return smart pointer to a RecommendationSystem which was created with
16         * those movies
17         */
18        static std::unique_ptr<RecommendationSystem> create_rs_from_movies
19            (const std::string &movies_file_path) noexcept (false);
20    };
21
22 #endif //RECOMMENDATIONSYSTEMLOADER_H
```

8 ex5/RecommendationSystemLoader.cpp

```
1
2 #include "RecommendationSystemLoader.h"
3 #include <fstream>
4 #include <sstream>
5 #define YEAR_SEPARATOR '-'
6 #define ERROR_MSG "input file is incorrect"
7
8
9 std::unique_ptr<RecommendationSystem>
10 RecommendationSystemLoader::create_rs_from_movies(
11     const std::string &movies_file_path) noexcept(false)
12 {
13     std::unique_ptr<RecommendationSystem> rs =
14         std::make_unique<RecommendationSystem>();
15     std::ifstream in_file;
16     in_file.open(movies_file_path);
17     std::string buffer;
18
19     while (getline(in_file, buffer))
20     {
21         std::string movie_det;
22         double ranking;
23         std::istringstream splitted_line(buffer);
24         splitted_line >> movie_det;
25         std::vector<double> vec;
26         while (splitted_line >> ranking)
27         {
28             if (ranking <= 0){
29                 throw std::invalid_argument(ERROR_MSG);
30             }
31             vec.push_back(ranking);
32         }
33         size_t end = buffer.find(YEAR_SEPARATOR);
34         rs->add_movie(buffer.substr(0, end),
35             std::stoi(buffer.substr(end + 1, buffer.length())),vec);
36     }
37     in_file.close();
38     return rs;
39 }
```


9 ex5/User.h

```
1  //
2  // Created on 2/20/2022.
3  //
4
5  #ifndef USER_H
6  #define USER_H
7  #include <unordered_map>
8  #include <map>
9  #include <vector>
10 #include <string>
11 #include <memory>
12 #include "Movie.h"
13 #include <algorithm>
14 #include <cmath>
15
16 #define MAX_RATE 10
17 #define MIN_RATE 1
18
19 using std::unordered_map;
20
21 class RecommendationSystem;
22 typedef unordered_map<sp_movie, double, hash_func, equal_func> rank_map;
23
24 using std::string;
25
26 class User
27 {
28 private:
29     string user_name;
30     rank_map user_rank_map;
31     std::shared_ptr<RecommendationSystem> rec_sys;
32
33 public:
34
35     /**
36      * Constructor for the class
37      */
38     User(const string& name, const rank_map &rank_map_in,
39          std::shared_ptr<RecommendationSystem> &rec_sys_input);
40
41     /**
42      * function for adding a movie to the DB
43      * @param name name of movie
44      * @param year year it was made
45      * @param features a vector of the movie's features
46      * @param rate the user rate for this movie
47      */
48     void add_movie_to_rs(const string &name, int year,
49                          const vector<double> &features,
50                          double rate);
51
52     /**
53      * a getter for the user's name
54      * @return the username
55      */
56     string get_name() const;
57
58
59     /**
```

```

60     * a getter for the ranks map
61     * @return the user's rank map
62     */
63     rank_map get_rank() const;
64
65     /**
66     * returns a recommendation according to the movie's content
67     * @return recommendation
68     */
69     sp_movie get_recommendation_by_content() const;
70
71     /**
72     * returns a recommendation according to the similarity
73     * recommendation method
74     * @param k the number of the most similar movies to calculate by
75     * @return recommendation
76     */
77     sp_movie get_recommendation_by_cf(int k) const;
78
79     /**
80     * predicts the score for a given movie
81     * @param name the name of the movie
82     * @param year the year the movie was created
83     * @param k the parameter which represents the number of the most
84     * similar movies to predict the score by
85     * @return predicted score for the given movie
86     */
87     double get_prediction_score_for_movie(const string& name,
88                                           int year, int k) const;
89
90     /**
91     * output stream operator
92     * @param os the output streamRecommendationSystem
93     * @param user the user
94     * @return output stream
95     */
96     friend ostream &operator<< (ostream &os, const User &user);
97 };
98
99
100
101 #endif //USER_H

```

10 ex5/User.cpp

```
1
2
3 // don't change those includes
4 #include "User.h"
5 #include "RecommendationSystem.h"
6
7
8 /// Constructor
9 /// TODO: do i need to deepcopy rankmap? if so, a for loop
10 /// TODO: that copies each key:val from rank_map_in
11 User::User(const string& name,
12            const rank_map &rank_map_in,
13            std::shared_ptr<RecommendationSystem> &rec_sys_input):
14     user_name(name),
15     user_rank_map(rank_map_in),
16     rec_sys(rec_sys_input)
17 {}
18
19 void User::add_movie_to_rs(const string &name, int year,
20                           const vector<double> &features,
21                           double rate)
22 {
23     if (rate < MIN_RATE || rate > MAX_RATE || std::isnan(rate)){
24         //name = nullptr || year < 0 || features = nullptr
25         return;
26     }
27     sp_movie new_m = rec_sys->add_movie(name, year, features);
28     user_rank_map[new_m] = rate;
29 }
30
31 /// return the username
32 string User::get_name() const
33 {
34     return user_name;
35 }
36
37 /// returns user's rank map
38 rank_map User::get_rank() const
39 {
40     return user_rank_map;
41 }
42
43 /// returns a pointer to a recommended movie by the algorithm
44 sp_movie User::get_recommendation_by_content() const
45 {
46     return rec_sys->get_recommendation_by_content(*this);
47 }
48
49 /// returns a recommendation according to the similarity recommendation method
50 sp_movie User::get_recommendation_by_cf(int k) const
51 {
52     return rec_sys->recommend_by_cf (*this, k);
53 }
54
55 /// predicts the score for a given movie
56 double User::get_prediction_score_for_movie(const string &name,
57                                              int year, int k) const
58 {
59     sp_movie movie_ptr = rec_sys->get_movie(name, year);
```

```
60     return rec_sys->predict_movie_score(*this, movie_ptr, k);
61 }
62
63 /// output stream operator (friend)
64 ostream &operator<< (ostream &os, const User &user){
65     return os << user.user_name << endl << *user.rec_sys << endl;
66 }
```

11 ex5/UsersLoader.h

```
1  //
2  // Created on 2/21/2022.
3  //
4
5
6
7
8  #ifndef USERFACTORY_H
9  #define USERFACTORY_H
10
11  #include <sstream>
12  #include <fstream>
13  #include <vector>
14  #include "User.h"
15  #include "RecommendationSystem.h"
16
17  #define YEAR_SEPARATOR '-'
18
19
20  class UsersLoader
21  {
22  private:
23
24
25  public:
26      UsersLoader() = delete;
27      /**
28       *
29       * loads users by the given format with their movie's ranks
30       * @param users_file_path a path to the file of the users and their
31       * movie ranks
32       * @param rs RecommendingSystem for the Users
33       * @return vector of the users created according to the file
34       */
35      static std::vector<User> create_users(const std::string& users_file_path,
36                                           std::unique_ptr<RecommendationSystem> rs)
37                                           noexcept(false);
38
39  };
40
41
42  #endif //USERFACTORY_H
```

12 ex5/UsersLoader.cpp

```
1
2  #include "UsersLoader.h"
3
4  #define INIT_BUCKET_SIZE 8
5  #define ERROR_MSG "input file is incorrect"
6
7
8  std::vector<User>
9  UsersLoader::create_users
10 (const std::string &users_file_path, std::unique_ptr<RecommendationSystem> rs)
11 noexcept(false)
12 {
13     std::shared_ptr<RecommendationSystem> s_rs = std::move(rs);
14     std::ifstream in_file;
15     std::vector<User> users;
16     in_file.open(users_file_path);
17     std::string buffer;
18     getline(in_file, buffer);
19     std::istringstream movies_names(buffer);
20     std::vector<sp_movie> movies;
21     while (movies_names >> buffer)
22     {
23         size_t end = buffer.find(YEAR_SEPARATOR);
24         sp_movie m = s_rs->get_movie(buffer.substr(0, end),
25                                     std::stoi(buffer.substr(end + 1,
26                                                         buffer.length())));
27         movies.push_back(m);
28     }
29     while (getline(in_file, buffer))
30     {
31         std::string user_name;
32         std::string ranking;
33         std::istringstream splitted_line(buffer);
34         splitted_line >> user_name;
35         int i = 0;
36         rank_map ranks(INIT_BUCKET_SIZE, sp_movie_hash, sp_movie_equal);
37         while (splitted_line >> ranking)
38         {
39
40             if (ranking != "NA")
41             {
42                 int rating = std::stoi(ranking);
43                 if (rating <= 0){
44                     throw std::invalid_argument(ERROR_MSG);
45                 }
46                 ranks[movies[i]] = rating;
47             }
48             i++;
49         }
50         users.emplace_back(user_name, ranks, s_rs);
51     }
52     in_file.close();
53     return users;
54 }
```