

# Contents

1	Basic Test Results	2
2	RecommenderSystem.h	3
3	RecommenderSystem.cpp	6

# 1 Basic Test Results

```
1 Running...
2
3 Opening tar file
4 OK
5 Tar extracted O.K.
6
7 Checking files...
8 OK
9 Making sure files are not empty...
10 OK
11 Compilation check...
12 #1
13 Compiling...
14 OK
15 Compilation seems OK! Check if you got warnings!
16
17
18 =====
19 Public test cases
20 =====
21
22 =====
23 Running test...
24 OK
25 =====
26
27
28 =====
29 = Checking coding style =
30 =====
31 ** Total Violated Rules      : 0
32 ** Total Errors Occurs      : 0
33 ** Total Violated Files Count: 0
```

## 2 RecommenderSystem.h

```
1 // RecommendSystem.h
2
3 #ifndef RECOMMENDSYSTEM_H
4 #define RECOMMENDSYSTEM_H
5
6 #include <iostream>
7 #include <cmath>
8 #include <algorithm>
9 #include <numeric>
10 #include <map>
11 #include <sstream>
12 #include <fstream>
13 #include <string>
14 #include <filesystem>
15 #include <set>
16
17 #define DEFAULT 0.0
18 #define READ_ERROR "Unable to open file "
19 #define USER_ERROR "USER NOT FOUND"
20 #define FAILURE -1
21 #define STANDART -1000
22 /**
23  *Data struct for holding the RecommenderSystem class data
24  */
25 typedef struct Data
26 {
27     std::vector<std::string> titles;
28     std::map<std::string, std::vector<double>> movies;
29     std::map<std::string, std::vector<double>> attributes;
30 } Data;
31
32 /**
33  * RecommenderSystem Class
34  */
35 class RecommenderSystem
36 {
37 public:
38     /**
39      * default constructor for RecommenderSystem;
40      */
41     RecommenderSystem();
42
43     /**
44      * A function that loads the data from given input files
45      * @param moviesAttributesFilePath: moviesAttributes file path
46      * @param userRanksFilePath: userRanks File Path
47      * @return: 1 if data load was successful 0 if not
48      */
49     int loadData(std::string moviesAttributesFilePath, std::string userRanksFilePath);
50
51     /**
52      * A function creates a movie recommendation by movie content
53      * @param userName: the user to recommend to
54      * @return: the name of the movie recommended for the specific user
55      */
56     std::string recommendByContent(const std::string &userName);
57
58     /**
59      * A function creates a movie recommendation for a user
```

```

60     * @param MovieName: an non-ranked movie that the recommendation will be based on.
61     * @param userName: the user name that the recommendation is for
62     * @param k: number of similar movies to be calculated with
63     * @return: a double that represents prediction of the movie score for the given user
64     */
65     double predictMovieScoreForUser(const std::string &MovieName,
66                                     const std::string &userName, int k);
67
68     /**
69     * A function that recommends a movie to a given user
70     * @param userName: the user name that the recommendation is for
71     * @param k: the number of seen movies to refer to while making the preference
72     * @return: a string with the recommendation
73     */
74     std::string recommendByCF(const std::string &userName, int k);
75
76
77 private:
78     Data _data;
79
80     /**
81     * An aid function that loads the user's rankings to a data map and movie titles into a
82     * vector
83     * @param userRanksFilePath : moviesAttributes file path
84     * @param usersRankingMap: the map that holds the the user's movie rankings while the user name
85     * serves as key
86     * @param titles: a vector which holds the movie titles by order
87     * @return the data map
88     */
89     static int _userLoader(std::string &userRanksFilePath,
90                           std::map<std::string, std::vector<double>> &usersRankingMap,
91                           std::vector<std::string> &titles);
92
93     /**
94     * An aid function that loads the movies attributes to a data map and movie titles into a
95     * vector
96     * @param moviesAttributesFilePath: moviesAttributes file path
97     * @param movieAttributesMap: map for storing movie attributes when the movie title serves as key
98     * @return: the data map
99     */
100    static int _movieLoader(std::string &movieFilePath,
101                            std::map<std::string, std::vector<double>> &movieAttributesMap);
102
103    /**
104    * A function for creating a preference vector
105    * @param norm : the user normalised ranking vector that will be used as the base for the preference
106    * vector
107    * @return: preference vector
108    */
109    std::vector<double> _createPreferenceVector(const std::vector<double> &vec);
110
111
112    /**
113    * A function that normalises a vector by calculating its average and decreasing it from each
114    * of the vector values
115    * @param norm : the user ranking vector to be normalised
116    * @return : a normalised vector
117    */
118    static std::vector<double> _normalisedVector(std::vector<double> &norm);
119
120    /**
121    * A function that calculates the angle between two vectors
122    * @param vecA : first vector
123    * @param vecB : second vector
124    * @return: the angle between the vectors
125    */
126    static double _getAngle(const std::vector<double> &vecA, const std::vector<double> &vecB);
127

```

```

128     /**
129     * a function that finds a set of similar movies to a given movie
130     * @param movie : the given movie
131     * @param user : the current user
132     * @param k: number of similar movies to be calculated with
133     * @return : a vector of pairs that holds the set of similar ranked movies angles
134     */
135     std::set<std::pair<double, std::string>> _findSet(const std::vector<double> &movie,
136                                                    const std::string &user, long unsigned int k);
137
138
139 };
140
141 #endif //RECOMMENDSYSTEM_H
142

```

### 3 RecommenderSystem.cpp

```
1 // RecommenderSystem.cpp
2 //by yoav.eshed
3 #include <filesystem>
4 #include "RecommenderSystem.h"
5
6 /**
7  * A default constructor to the RecommenderSystem class
8  */
9 RecommenderSystem::RecommenderSystem()
10 {
11     _data.title = std::vector<std::string>();
12     _data.movies = std::map<std::string, std::vector<double>>();
13     _data.attributes = std::map<std::string, std::vector<double>>();
14 }
15
16 /**
17  * An aid function that loads the movies attributes to a data map and movie titles into a
18  * vector
19  * @param moviesAttributesFilePath: moviesAttributes file path
20  * @param movieAttributesMap: map for storing movie attributes when the movie title serves as key
21  * @return: the data map
22  */
23 int RecommenderSystem::_movieLoader(std::string &movieFilePath,
24                                     std::map<std::string, std::vector<double>> &movieAttributesMap)
25 {
26     std::ifstream file(movieFilePath);
27     std::string line, title;
28     double num;
29     while (file.good())
30     {
31         getline(file, line);
32         std::istringstream ss(line);
33         ss >> title;
34         while (ss >> num)
35         {
36             movieAttributesMap[title].push_back(num);
37             if (ss.bad())
38             {
39                 return FAILURE;
40             }
41         }
42     }
43     return EXIT_SUCCESS;
44 }
45
46 /**
47  * An aid function that loads the user's rankings to a data map and movie titles into a
48  * vector
49  * @param userRanksFilePath : moviesAttributes file path
50  * @param usersRankingMap: the map that holds the the user's movie rankings while the user name
51  * serves as key
52  * @param titles: a vector which holds the movie titles by order
53  * @return the data map
54  */
55 int RecommenderSystem::_userLoader(std::string &userRanksFilePath,
56                                    std::map<std::string, std::vector<double>> &usersRankingMap,
57                                    std::vector<std::string> &titles)
58 {
59     std::ifstream file(userRanksFilePath);
```

```

60     std::string line, title;
61     std::string num;
62     while (file.good())
63     {
64         getline(file, line);
65         std::istringstream ss(line);
66         while (ss >> title)
67         {
68             titles.push_back(title);
69             if (ss.bad())
70             {
71                 return FAILURE;
72             }
73         }
74         while (getline(file, line))
75         {
76             std::istringstream iss(line);
77             iss >> title;
78             while (iss >> num)
79             {
80                 usersRankingMap[title].push_back(num == "NA" ? 0 : stod(num));
81                 if (ss.bad())
82                 {
83                     return FAILURE;
84                 }
85             }
86         }
87     }
88     return EXIT_SUCCESS;
89 }
90
91 /**
92  * A function creates a movie recommendation by movie content
93  * @param userName: the user to recommend to
94  * @return: the name of the movie recommended for the specific user
95  */
96 int RecommenderSystem::loadData(std::string moviesAttributesFilePath, std::string userRanksFilePath)
97 {
98
99     if (!std::ifstream(moviesAttributesFilePath))
100     {
101         std::cerr << READ_ERROR + moviesAttributesFilePath << std::endl;
102         return FAILURE;
103     }
104     if (!std::ifstream(userRanksFilePath))
105     {
106         std::cerr << READ_ERROR + userRanksFilePath << std::endl;
107         return FAILURE;
108     }
109     _movieLoader(moviesAttributesFilePath, _data.movies);
110     _userLoader(userRanksFilePath, _data.attributes, _data.titles);
111     if (_data.movies.empty())
112     {
113         std::cerr << READ_ERROR + moviesAttributesFilePath << std::endl;
114         return FAILURE;
115     }
116     if (_data.movies.empty())
117     {
118         std::cerr << READ_ERROR + userRanksFilePath << std::endl;
119         return FAILURE;
120     }
121     return EXIT_SUCCESS;
122 }
123
124 //3.2.1
125 //step 1
126 /**
127  * A function that normalises a vector by calculating its average and decreasing it from each

```

```

128  * of the vector values
129  * @param vec :the user ranking vector to be normalised
130  * @return : a normalised vector
131  */
132  std::vector<double> RecommenderSystem::_normalisedVector(std::vector<double> &vec)
133  {
134      std::vector result = std::vector(vec);
135      int n = vec.size();
136      double average = DEFAULT;
137      average = (accumulate(vec.begin(), vec.end(), average));
138      int countZeros = count(vec.begin(), vec.end(), 0);
139      average = average / (n - countZeros);
140      for (auto &element : result)
141      {
142          if (element != 0)
143          {
144              element -= average;
145          }
146      }
147      return result;
148  }
149
150  //step 2
151  /**
152   * A function for creating a preference vector
153   * @param norm : the user normalised ranking vector that will be used as the base for the preference
154   * vector
155   * @return: preference vector
156   */
157  std::vector<double> RecommenderSystem::_createPreferenceVector(const std::vector<double> &norm)
158  {
159      std::vector<double> result(_data.movies[_data.titles[0]].size(), 0.0), vec;
160      for (long unsigned int j = 0; j < _data.titles.size(); j++)
161      {
162          std::string title = _data.titles[j];
163          vec = _data.movies.at(title);
164          std::transform(vec.begin(), vec.end(), vec.begin(),
165                        std::bind1st(std::multiplies<double>(), norm[j]));
166          std::transform(result.begin(), result.end(), vec.begin(), result.begin(),
167                        std::plus<>());
168      }
169      return result;
170  }
171
172  //step 3
173  /**
174   * A function that calculates the angle between two vectors
175   * @param vecA :first vector
176   * @param vecB :second vector
177   * @return: the angle between the vectors
178   */
179  double
180  RecommenderSystem::_getAngle(const std::vector<double> &vecA, const std::vector<double> &vecB)
181  {
182      std::vector<double> tempA = vecA;
183      std::vector<double> tempB = vecB;
184      double product, norms, angle;
185      product = inner_product(tempA.begin(), tempA.end(), tempB.begin(), 0.0);
186      norms = sqrt(inner_product(tempA.begin(), tempA.end(), tempA.begin(), 0.0));
187      norms *= sqrt(inner_product(tempB.begin(), tempB.end(), tempB.begin(), 0.0));
188      angle = (product / norms);
189      return (angle);
190  }
191
192  /**
193   * A function creates a movie recommendation by movie content
194   * @param userName: the user to recommend to
195   * @return: the name of the movie recommended for the specific user

```



```

196  */
197  std::string RecommenderSystem::recommendByContent(const std::string &userName)
198  {
199      if (_data.attributes.find(userName) == _data.attributes.end())
200      {
201          return USER_ERROR;
202      }
203      //step 1
204      std::vector<double> norVec = _normalisedVector(_data.attributes[userName]);
205      //step 2
206      std::vector<double> Preference = _createPreferenceVector(norVec);
207      //step 3
208      std::string result;
209      double res = DEFAULT, max = STANDART;
210      for (unsigned long int i = 0; i < _data.titles.size(); i++)
211      {
212          std::string title = _data.titles[i];
213          if (_data.attributes[userName][i] == 0)
214          {
215              res = _getAngle(_data.movies.at(title), Preference);
216              if (res > max)
217              {
218                  max = res;
219                  result = title;
220              }
221          }
222      }
223      return result;
224  }
225
226
227  //3.2.2
228  //step 1
229  /**
230   * a function that finds a set of similar movies to a given movie
231   * @param movie : the given movie
232   * @param user : the current user
233   * @param k: number of similar movies to be calculated with
234   * @return : a vector of pairs that holds the set of similar ranked movies angles
235   */
236  std::set<std::pair<double, std::string>> RecommenderSystem::_findSet(const std::vector<double>
237                                                                    &movie,
238                                                                    const std::string &user,
239                                                                    long unsigned int k)
240  {
241      std::set<std::pair<double, std::string>> simSet;
242      std::string title;
243      double angle = 0.0;
244      for (unsigned long int i = 0; i < _data.titles.size(); i++)
245      {
246          if (_data.attributes[user][i] != 0)
247          {
248              title = _data.titles[i];
249              angle = _getAngle(movie, _data.movies.at(title));
250              simSet.emplace(std::pair(angle, title));
251              if (simSet.size() > k)
252              {
253                  simSet.erase((simSet.begin()));
254              }
255          }
256      }
257      return simSet;
258  }
259  //step 2
260  //finding the predicted rank
261
262  // steps 2 and 3 predicting the movie score
263  /**

```

```

264  * A function creates a movie recommendation for a user
265  * @param MovieName: an non-ranked movie that the recommendation will be based on.
266  * @param userName: the user name that the recommendation is for
267  * @param k: number of similar movies to be calculated with
268  * @return: a double that represents prediction of the movie score for the given user
269  */
270  double RecommenderSystem::predictMovieScoreForUser(const std::string &movieName,
271                                                    const std::string &userName, int k)
272  {
273      if (_data.attributes.find(userName) == _data.attributes.end() or
274          _data.movies.find(movieName) == _data.movies.end())
275      {
276          return FAILURE;
277      }
278      double multSum = DEFAULT, similaritySum = DEFAULT;
279      std::set<std::pair<double, std::string>> simSet = _findSet(_data.movies.at(movieName),
280                                                            userName, k);
281      for (auto &elem : simSet)
282      {
283          ptrdiff_t pos = find(_data.titles.begin(), _data.titles.end(),
284                             elem.second) - _data.titles.begin();
285          multSum += (elem.first * _data.attributes[userName][pos]);
286          similaritySum += (elem.first);
287      }
288      double sum = multSum / similaritySum;
289      return sum;
290  }
291
292  /**
293   * A function that recommends a movie to a given user
294   * @param userName: the user name that the recommendation is for
295   * @param k: the number of seen movies to refer to while making the preference
296   * @return: a string with the recommendation
297   */
298  std::string RecommenderSystem::recommendByCF(const std::string &userName, int k)
299  {
300      if (_data.attributes.find(userName) == _data.attributes.end())
301      {
302          return USER_ERROR;
303      }
304      std::string result, title;
305      double max = DEFAULT, res;
306      for (unsigned long int i = 0; i < _data.titles.size(); i++)
307      {
308          title = _data.titles[i];
309          if (_data.attributes[userName][i] == 0)
310          {
311              res = predictMovieScoreForUser(title, userName, k);
312              if (res > max)
313              {
314                  max = res;
315                  result = title;
316              }
317          }
318      }
319      return result;
320  }

```