Contents

1 Basic Test Results

```
Running...

Opening tar file
OK
Tar extracted O.K.

Checking files...
OK
Making sure files are not empty...
OK
Compilation check...
#1
Compiling...
OK
Compilation seems OK! Check if you got warnings!


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

=====================
Running test...
OK
=====================


=========================
= Checking coding style =
=========================
 ** Total Violated Rules      : 0
 ** Total Errors Occurs       : 0
 ** Total Violated Files Count: 0
```

2 RecommenderSystem.h

```cpp
// RecommendSystem.h

#ifndef RECOMMENDSYSTEM_H
#define RECOMMENDSYSTEM_H

#include <iostream>
#include <cmath>
#include <algorithm>
#include <numeric>
#include <map>
#include <sstream>
#include <fstream>
#include <string>
#include <filesystem>
#include <set>

#define DEFAULT 0.0
#define READ_ERROR "Unable to open file "
#define USER_ERROR "USER NOT FOUND"
#define FAILURE -1
#define STANDART -1000
/**
 *Data struct for holding the RecommenderSystem class data
 */
typedef struct Data
{
    std::vector<std::string> titles;
    std::map<std::string, std::vector<double>> movies;
    std::map<std::string, std::vector<double>> attributes;
} Data;

/**
 * RecommenderSystem Class
 */
class RecommenderSystem
{
public:
    /**
     * default constructor for RecommenderSystem;
     */
    RecommenderSystem();

    /**
     * A function  that loads the data from given input files
     * @param moviesAttributesFilePath: moviesAttributes file path
     * @param userRanksFilePath: userRanks File Path
     * @return: 1 if data load was successful 0 if not
     */
    int loadData(std::string moviesAttributesFilePath, std::string userRanksFilePath);

    /**
     * A function creates a movie recommendation by movie content
     * @param userName: the user to recommend to
     * @return: the name of the movie recommended for the specific user
     */
    std::string recommendByContent(const std::string &userName);

    /**
     * A function creates a movie recommendation for a user
```

```
     * @param MovieName: an non-ranked movie that the recommendation will be based on.
     * @param userName: the user name that the recommendation is for
     * @param k: number of similar movies to be calculated with
     * @return: a double that represents prediction of the movie score for the given user
     */
    double predictMovieScoreForUser(const std::string &MovieName,
                                    const std::string &userName, int k);

    /**
     * A function that recommends a movie to a given user
     * @param userName: the user name that the recommendation is for
     * @param k: the number of seen movies to refer to while making the preference
     * @return: a string with the recommendation
     */
    std::string recommendByCF(const std::string &userName, int k);


private:
    Data _data;

    /**
     *An aid function that loads the user's rankings to a data map and movie titles into a
     * vector
     * @param userRanksFilePath : moviesAttributes file path
     * @param usersRankingMap: the map that holds the the user's movie rankings while the user name
     * serves as key
     * @param titles: a vector which holds the movie titles by order
     * @return the data map
     */
    static int _userLoader(std::string &userRanksFilePath,
                           std::map<std::string, std::vector<double>> &usersRankingMap,
                           std::vector<std::string> &titles);

    /**
     *An aid function that loads the movies attributes to a data map and movie titles into a
     * vector
     * @param moviesAttributesFilePath: moviesAttributes file path
     * @param movieAttributesMap: map for storing movie attributes when the movie title serves as key
     * @return: the data map
     */
    static int _movieLoader(std::string &movieFilePath,
                            std::map<std::string, std::vector<double>> &movieAttributesMap);

    /**
     * A function for creating a preference vector
     * @param norm : the user normalised ranking vector that will be used as the base for the preference
     * vector
     * @return: preference vector
     */
    std::vector<double> _createPreferenceVector(const std::vector<double> &vec);


    /**
     * A function that normalises a vector by calculating its average and decreasing it from each
     * of the vector values
     * @param norm :the user ranking vector to be normalised
     * @return : a normalised vector
     */
    static std::vector<double> _normalisedVector(std::vector<double> &norm);

    /**
     * A function that calculates the angle between two vectors
     * @param vecA :first vector
     * @param vecB :second vector
     * @return: the angle between the vectors
     */
    static double _getAngle(const std::vector<double> &vecA, const std::vector<double> &vecB);

```

```
    /**
      * a function that finds a set of similar movies to a given movie
      * @param movie : the given movie
      * @param user : the current user
      * @param k: number of similar movies to be calculated with
      * @return : a vector of pairs that holds the set of similar ranked movies angles
      */
    std::set<std::pair<double, std::string>> _findSet(const std::vector<double> &movie,
                                                      const std::string &user, long unsigned int k);


};

#endif //RECOMMENDSYSTEM_H

```

3 RecommenderSystem.cpp

```cpp
// RecommendSystem.cpp
//by yoav.eshed
#include <filesystem>
#include "RecommenderSystem.h"

/**
 * A default constructor to the RecommenderSystem class
 */
RecommenderSystem::RecommenderSystem()
{
    _data.titles = std::vector<std::string>();
    _data.movies = std::map<std::string, std::vector<double>>();
    _data.attributes = std::map<std::string, std::vector<double>>();
}

/**
 *An aid function that loads the movies attributes to a data map and movie titles into a
 * vector
 * @param moviesAttributesFilePath: moviesAttributes file path
 * @param movieAttributesMap: map for storing movie attributes when the movie title serves as key
 * @return: the data map
 */
int RecommenderSystem::_movieLoader(std::string &movieFilePath,
                                    std::map<std::string, std::vector<double>> &movieAttributesMap)
{
    std::ifstream file(movieFilePath);
    std::string line, title;
    double num;
    while (file.good())
    {
        getline(file, line);
        std::istringstream ss(line);
        ss >> title;
        while (ss >> num)
        {
            movieAttributesMap[title].push_back(num);
            if (ss.bad())
            {
                return FAILURE;
            }
        }
    }
    return EXIT_SUCCESS;
}

/**
 *An aid function that loads the user's rankings to a data map and movie titles into a
 * vector
 * @param userRanksFilePath : moviesAttributes file path
 * @param usersRankingMap: the map that holds the the user's movie rankings while the user name
 * serves as key
 * @param titles: a vector which holds the movie titles by order
 * @return the data map
 */
int RecommenderSystem::_userLoader(std::string &userRanksFilePath,
                                   std::map<std::string, std::vector<double>> &usersRankingMap,
                                   std::vector<std::string> &titles)
{
    std::ifstream file(userRanksFilePath);
```

```
    std::string line, title;
    std::string num;
    while (file.good())
    {
        getline(file, line);
        std::istringstream ss(line);
        while (ss >> title)
        {
            titles.push_back(title);
            if (ss.bad())
            {
                return FAILURE;
            }
        }
        while (getline(file, line))
        {
            std::istringstream iss(line);
            iss >> title;
            while (iss >> num)
            {
                usersRankingMap[title].push_back(num == "NA" ? 0 : stod(num));
                if (ss.bad())
                {
                    return FAILURE;
                }
            }
        }
    }
    return EXIT_SUCCESS;
}

/**
 * A function creates a movie recommendation by movie content
 * @param userName: the user to recommend to
 * @return: the name of the movie recommended for the specific user
 */
int RecommenderSystem::loadData(std::string moviesAttributesFilePath, std::string userRanksFilePath)
{

    if (!std::ifstream(moviesAttributesFilePath))
    {
        std::cerr << READ_ERROR + moviesAttributesFilePath << std::endl;
        return FAILURE;
    }
    if (!std::ifstream(userRanksFilePath))
    {
        std::cerr << READ_ERROR + userRanksFilePath << std::endl;
        return FAILURE;
    }
    _movieLoader(moviesAttributesFilePath, _data.movies);
    _userLoader(userRanksFilePath, _data.attributes, _data.titles);
    if (_data.movies.empty())
    {
        std::cerr << READ_ERROR + moviesAttributesFilePath << std::endl;
        return FAILURE;
    }
    if (_data.movies.empty())
    {
        std::cerr << READ_ERROR + userRanksFilePath << std::endl;
        return FAILURE;
    }
    return EXIT_SUCCESS;
}

//3.2.1
//step 1
/**
 * A function that normalises a vector by calculating its average and decreasing it from each
```

```
 * of the vector values
 * @param vec :the user ranking vector to be normalised
 * @return : a normalised vector
 */
std::vector<double> RecommenderSystem::_normalisedVector(std::vector<double> &vec)
{
    std::vector result = std::vector(vec);
    int n = vec.size();
    double average = DEFAULT;
    average = (accumulate(vec.begin(), vec.end(), average));
    int countZeros = count(vec.begin(), vec.end(), 0);
    average = average / (n - countZeros);
    for (auto &element : result)
    {
        if (element != 0)
        {
            element -= average;
        }
    }
    return result;
}

//step 2
/**
 * A function for creating a preference vector
 * @param norm : the user normalised ranking vector that will be used as the base for the preference
 * vector
 * @return: preference vector
 */
std::vector<double> RecommenderSystem::_createPreferenceVector(const std::vector<double> &norm)
{
    std::vector<double> result(_data.movies[_data.titles[0]].size(), 0.0), vec;
    for (long unsigned int j = 0; j < _data.titles.size(); j++)
    {
        std::string title = _data.titles[j];
        vec = _data.movies.at(title);
        std::transform(vec.begin(), vec.end(), vec.begin(),
                       std::bind1st(std::multiplies<double>(), norm[j]));
        std::transform(result.begin(), result.end(), vec.begin(), result.begin(),
                       std::plus<>());
    }
    return result;
}

//step 3
/**
 * A function that calculates the angle between two vectors
 * @param vecA :first vector
 * @param vecB :second vector
 * @return: the angle between the vectors
 */
double
RecommenderSystem::_getAngle(const std::vector<double> &vecA, const std::vector<double> &vecB)
{
    std::vector<double> tempA = vecA;
    std::vector<double> tempB = vecB;
    double product, norms, angle;
    product = inner_product(tempA.begin(), tempA.end(), tempB.begin(), 0.0);
    norms = sqrt(inner_product(tempA.begin(), tempA.end(), tempA.begin(), 0.0));
    norms *= sqrt(inner_product(tempB.begin(), tempB.end(), tempB.begin(), 0.0));
    angle = (product / norms);
    return (angle);
}

/**
 * A function creates a movie recommendation by movie content
 * @param userName: the user to recommend to
 * @return: the name of the movie recommended for the specific user
```

```
 */
std::string RecommenderSystem::recommendByContent(const std::string &userName)
{
    if (_data.attributes.find(userName) == _data.attributes.end())
    {
        return USER_ERROR;
    }
    //step 1
    std::vector<double> norVec = _normalisedVector(_data.attributes[userName]);
    //step 2
    std::vector<double> Preference = _createPreferenceVector(norVec);
    //step 3
    std::string result;
    double res = DEFAULT, max = STANDART;
    for (unsigned long int i = 0; i < _data.titles.size(); i++)
    {
        std::string title = _data.titles[i];
        if (_data.attributes[userName][i] == 0)
        {
            res = _getAngle(_data.movies.at(title), Preference);
            if (res > max)
            {
                max = res;
                result = title;
            }
        }
    }
    return result;
}


//3.2.2
//step 1
/**
  * a function that finds a set of similar movies to a given movie
  * @param movie : the given movie
  * @param user : the current user
  * @param k: number of similar movies to be calculated with
  * @return : a vector of pairs that holds the set of similar ranked movies angles
  */
std::set<std::pair<double, std::string>> RecommenderSystem::_findSet(const std::vector<double>
                                                                     &movie,
                                                                     const std::string &user,
                                                                     long unsigned int k)
{
    std::set<std::pair<double, std::string>> simSet;
    std::string title;
    double angle = 0.0;
    for (unsigned long int i = 0; i < _data.titles.size(); i++)
    {
        if (_data.attributes[user][i] != 0)
        {
            title = _data.titles[i];
            angle = _getAngle(movie, _data.movies.at(title));
            simSet.emplace(std::pair(angle, title));
            if (simSet.size() > k)
            {
                simSet.erase((simSet.begin()));
            }
        }
    }
    return simSet;
}
//step 2
//finding the predicted rank

// steps 2 and  3  predicting the movie score
/**
```

```
 * A function creates a movie recommendation for a user
 * @param MovieName: an non-ranked movie that the recommendation will be based on.
 * @param userName: the user name that the recommendation is for
 * @param k: number of similar movies to be calculated with
 * @return: a double that represents prediction of the movie score for the given user
 */
double RecommenderSystem::predictMovieScoreForUser(const std::string &movieName,
                                                   const std::string &userName, int k)
{
    if (_data.attributes.find(userName) == _data.attributes.end() or
        _data.movies.find(movieName) == _data.movies.end())
    {
        return FAILURE;
    }
    double multSum = DEFAULT, similaritySum = DEFAULT;
    std::set<std::pair<double, std::string>> simSet = _findSet(_data.movies.at(movieName),
                                                               userName, k);
    for (auto &elem : simSet)
    {
        ptrdiff_t pos = find(_data.titles.begin(), _data.titles.end(),
                             elem.second) - _data.titles.begin();
        multSum += (elem.first * _data.attributes[userName][pos]);
        similaritySum += (elem.first);
    }
    double sum = multSum / similaritySum;
    return sum;
}

/**
 * A function that recommends a movie to a given user
 * @param userName: the user name that the recommendation is for
 * @param k: the number of seen movies to refer to while making the preference
 * @return: a string with the recommendation
 */
std::string RecommenderSystem::recommendByCF(const std::string &userName, int k)
{
    if (_data.attributes.find(userName) == _data.attributes.end())
    {
        return USER_ERROR;
    }
    std::string result, title;
    double max = DEFAULT, res;
    for (unsigned long int i = 0; i < _data.titles.size(); i++)
    {
        title = _data.titles[i];
        if (_data.attributes[userName][i] == 0)
        {
            res = predictMovieScoreForUser(title, userName, k);
            if (res > max)
            {
                max = res;
                result = title;
            }
        }
    }
    return result;
}
```