## MOVIE RECOMMENDATION SYSTEM

By:

### Devansh Maheshwari (42976803118)

## CERTIFICATE



## ABSTRACT

Ever wonder how Netflix decides what movies to recommend for us? Or how Amazon recommends books? We can get a feel for how it works by building a simplified recommender of our own!

In this project, we will use our problem solving and Java programming skills by creating “Recommender Systems”. We will work with data for movies, including ratings, but the principle involved can easily be adapted to books, restaurants, and more.

We will write a program to answer questions about the data, including which items should be recommended to a user based on their ratings of several movies. Given input files on users’ ratings and movie titles, we will be able to read in and parse data into lists and maps, calculate average ratings, calculate how similar a given *rater* is to another user based on ratings, and recommend movies to a given user based on ratings.

**CONTENTS**

### Chapter

|  |
| --- |
| **Title Page** |
| **Certificate** |
| **Abstract** |

|  |  |  |
| --- | --- | --- |
| 1. | **Introduction** | |
|  | 1.1 A Recommendation System | |
|  | 1.2 Recommendations as per the Average Ratings | |
|  | 1.3 Recommendations as per the Average Similar Ratings | |
|  | 1.4 The Product we will be working with | |
| 2. | **Pre-requirements** | |
|  | 2.1 The Java Language | |
|  | 2.2 BlueJ | |
|  | 2.3 Duke University Packages | |
|  | 2.3.1 File Resource | |
|  | 2.4 Apache Commons CSV Classes | |
|  | 2.4.1 CSVRecord | |
|  | 2.4.2 CSVParser | |
|  | 2.5 The Data Files | |
| 3. | **The Various User-Defined Objects Created For The Project** | |
| 3.1 | | Movie |
| 3.2 | | Rating |
| 3.3 | | Rater |

1. The Rater And Movie Database
   1. [MovieDatabase](#_TOC_250001)
   2. [RaterDatabase](#_TOC_250000)

|  |
| --- |
| **Result** |
| **Summary and Conclusion** |
| **References** |
| **Appendix – A** |
| **Appendix – B** |

**Chapter One INTRODUCTION**

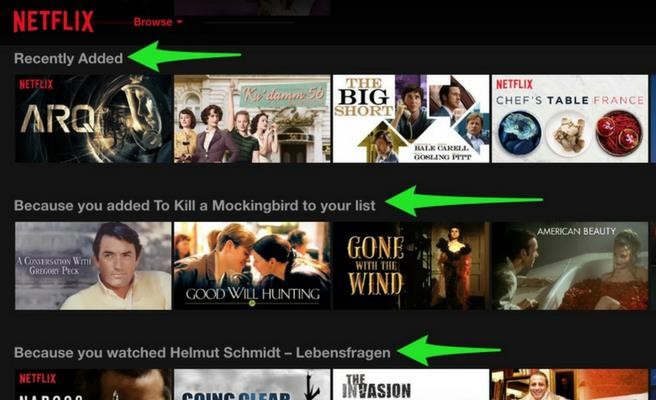
* 1. **A Recommendation System**

A *Recommender System*, or a Recommendation System (sometimes replacing 'system' with a synonym such as platform or engine), is a subclass of information filtering system that seeks to predict the “rating” or “preference” a user would give to an item. They are primarily used in commercial applications.

Recommender systems are utilized in a variety of areas and are most commonly recognized as playlist generators for video and music services, product recommenders for online stores, or content recommenders for social media platforms and open web content recommenders. These systems can operate using a single input, like music, or multiple inputs within and across platforms like news, books, and search queries. There are also popular recommender systems for specific topics like restaurants and online dating. Recommender systems have also been developed to explore research articles and experts, collaborators, and financial services.

A widely famous use of similar systems can be seen in platforms like Netflix or Amazon that recommend the user a variety of movies or books (products, in general) based upon his/her previous likings (or disliking). Platforms like *Coursera* recommend the user courses based upon the courses they have already taken.

We will be working on a similar system (though a simpler version of it) in this project.



## Recommendations as per the Average Ratings

A “Rater” is no one but a “User” who *rates* a product and how much he rates it is the “Rating”. A common and simple approach to recommend products is by sorting them according to their *average* rating, which, of course, is the sum of all the ratings the product has received divided by the number of ratings it has received.

It does indeed provide recommendations to a little extent of reliability, but the notion that all users and raters are alike is far from reality, which is why another approach to recommend products is used in most platforms, that uses, not the average ratings, but the *average similar ratings* (or Average of weighted ratings) for the same.

## Recommendations as per the Average Similar Ratings

Every one of us has different liking and disliking. What is appealing to some may not be too attractive for the others. A movie that is liked by a certain user may be hated by some other. In other words, a user seeking for recommendations might want to be recommended products that are rated well by certain raters who have similar likings as himself/herself. This is a little more tedious to implement than the implementation of simple average ratings, and is done by calculating the *weighted* averages.

To calculate this weighted average, we need to calculate a weight; how close a rater is to me, or to some particular rater. To do this, we will look at all the other raters who have rated to the products that are rated by the user as well, and determine who is the closest to the user by performing a *dot product* of the ratings. For better clarity, the ratings, originally from 1 to 10, will be adjusted to -5 to 5. This will help us eliminate the users that are very different from the user (if the dot product is negative).

We will then multiply the original ratings with their weights and sort them. The products with the highest rating will be the ones that will more likely be appealing to the user.

## The Product we will be working with

As alluded to in the beginning too, such a recommendation system can be implemented to recommend anything from movies, to courses, to books, to restaurants, and places.

In this project, I will be using it to recommend movies to users who will rate a number of movies beforehand, and we will use those ratings to determine the movies to recommend them.

# Chapter Two

**PRE - REQUIREMENTS**

## The JAVA Language

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms. For example: J2EE for Enterprise Applications, J2ME for Mobile Applications. Java is guaranteed to be Write Once, Run Anywhere.

Java is −

* Object Oriented − In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
* Platform Independent − Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
* Simple − Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
* Secure − With Java's secure feature it enables to develop virus-free, tamper- free systems. Authentication techniques are based on public-key encryption.
* Architecture-neutral − Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
* Portable − Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
* Robust − Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
* Multithreaded − With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature

allows the developers to construct interactive applications that can run smoothly.

* Interpreted − Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
* High Performance − With the use of Just-In-Time compilers, Java enables high performance.
* Distributed − Java is designed for the distributed environment of the internet.
* Dynamic − Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

The Java Version used in the project is Java 8.

## BlueJ

BlueJ is a development environment that allows you to develop Java programs quickly and easily. Its main features are that it is:

* + - **Simple** - BlueJ has a deliberately smaller and simpler interface than professional environments like NetBeans or Eclipse. This allows beginners to get started more quickly, and without being overwhelmed.
    - **Designed for Learning -** BlueJ is deliberately designed with good pedagogy in mind. There is a popular textbook designed for teaching introductory university/college courses with BlueJ, and a site full of teaching resources.
    - **Interactive -** BlueJ allows you to interact with objects. You can inspect their value, call methods on them, pass them as parameters and more. You can also directly invoke Java expressions without compiling. Thus BlueJ is a powerful graphical shell/REPL for Java.
    - **Portable** - BlueJ runs on Windows, Mac OS X, Linux and other platforms which run Java. It can also run without installation from a USB stick.
    - **Innovative** - BlueJ has several features not seen before in other IDEs. Its object bench, code pad, and scope colouring were all original BlueJ features.

## Duke University Package

The project uses methods from the classes of the edu.duke package provided by Duke University, licensed under a *Creative Commons Attribution 4.0 International License*, such as–

### 2.3.1 FileResource

This class provides methods for accessing a file on your computer. We can create a FileResource in a variety of ways:

new FileResource() // opens a dialog box prompting you to select a file on your computer new FileResource("path/to/file.ext") // uses the given String to find a file on your computer new FileResource(existingFile) // uses the given File directly

For these examples, assume the variable *fr* has been created for a specific file

|  |  |  |
| --- | --- | --- |
| **Method name** | **Description** | **Example** |
| .lines() | returns an Iterable that provides access to the contents of this opened file one line at a time | for (String line : fr.lines())  {  // process each line in turn  } |
| .words() | returns an Iterable that provides access to the contents of this opened file one word at a time | for (String word : fr.words()) {  // process each word in turn  } |
| .asString() | returns the entire contents of this opened  file as one String | String contents = fr.asString(); |

## Apache Commons CSV Classes

Several CSV files are read and used in different ways using methods from the org.apache.commons.csv package, licensed under a *Creative Commons Attribution 4.0 International License*, such as –

### CSVRecord

This class provides methods for accessing individual data values in a line of data within a CSV formatted file. You cannot create a CSVRecord directly, instead it will be provided for you when you iterate using a CSVParser. Data values are always returned as a String, so you will need to convert any values you plan to use in your calculations to the appropriate numeric value.

For these examples, assume the variable rec has been created for the second row of data below (the first line represents the header row that names the columns of data).

Name,Food,Color,Number Fred,Pizza,Purple,13

|  |  |  |
| --- | --- | --- |
| **Method name** | **Description** | **Example** |
| .get(columnName) | returns a String, the data in this record corresponding to the column with the given columnName  it is an error if the columnName does not exist in the header row (or  does not have the same case) | rec.get("Name") is "Fred" rec.get("Food") is "Pizza" |
| .get(columnIndex) | returns a String, the data in this record corresponding to the column at the given columnIndex  note, the index of the first data value  is 0 | rec.get(0) is "Fred" rec.get(3) is "13" |
| .size() | returns the number of values in this  record | rec.size() is 4 |

### CSVParser

This class provides you the ability to iterate over each line of data within a CSV formatted file as a record of the individual data values. Most likely you will not call any methods directly on a CSVParser object, but use it as an Iterable within your loop (you do not even have to call a method to do so, just use the object itself). In any case, here is one possibly useful method.

|  |  |
| --- | --- |
| **Method name** | **Description** |
| .getCurrentLineNumber() | returns the line number of the current record in the  iteration |

## The Data Files

To go further with the project, one has to have a certain database of the raters, movies, and the ratings, and the same (Real Time) can be provided by various websites like iMDB. The one used in this project is provided by Duke University and contains data from 1916 to 2015.

# Chapter Three

**THE VARIOUS USER-DEFINED OBJECTS CREATED FOR THE PROJECT**

## Movie

The class Movie is a Plain Old Java Object (POJO) class for storing the data about one movie. It includes the following items:

* + - Eight private variables to represent information about a movie including:

1. id• a String variable representing the IMDB ID of the movie
2. title­ a String variable for the movie’s title
3. year• an integer representing the year
4. genres• one String of one or more genres separated by commas
5. director• one String of one or more directors of the movie separated by commas
6. country• one String of one or more countries the film was made in, separated by commas
7. minutes• an integer for the length of the movie
8. poster• a String that is a link to an image of the movie poster if one exists, or “N/A” if no poster exists
   * + A constructor with eight parameters to initialize the private variables
     + Eight getter methods to return the private information such as the method getGenresthat returns a String of all the genres for this movie.
     + A toStringmethod for representing movie information as a String so it can easily be printed

## Rating

The class Ratingis also a POJO class for storing the data about one rating of an item. It includes:

* + - Two private variables to represent information about a rating

1. item• a String description of the item being rated (for this assignment you should use the IMDB ID of the movie being rated)
2. value• a double of the actual rating
   * + A constructor with two parameters to initialize the private variables
     + Two getter methods getItemand getValue
     + A toStringmethod to represent rating information as a String
     + A compareTo method to compare this rating with another rating

## Rater

The class Rater keeps track of one rater and all their ratings. This class includes:

* + - Two private variables

1. myID• a unique String ID for this rater
2. myRatings• HashMap<String,Rating> (The key in the HashMap is a movie ID, and its value is a rating associated with this movie)
   * + A constructor with one parameter of the ID for the rater
     + A method addRating that adds a new Rating to the HashMap with the value associated with the movie ID String item as the key in the HashMap.
     + A method getIDwith no parameters to get the ID of the rater
     + A method hasRatingthat has one parameter item. This method returns true if this item is in myRatings, and false otherwise.
     + A method getRatingthat has one parameter item. This method returns the double rating of this item if it is in myRatings.Otherwise this method returns

•1

* + - A method numRatingsthat returns the number of ratings this rater has
    - A method getItemsRatedthat has no parameters. This method returns an ArrayList of Strings representing a list of all the items that have been rated.

# Chapter Four

**THE RATER AND MOVIE DATABASE**

## Movie Database

The class MovieDatabase —This class is an efficient way to get information about movies. It stores movie information in a HashMap for fast lookup of movie information given a movie ID. The class also allows filtering movies based on queries. All methods and fields in the class are static. This means we’ll be able to access methods in MovieDatabase without using new to create objects, but by calling methods like MovieDatabase.getMovie(“0120915”). This class includes:

* + - A HashMap named ourMoviesthat maps a movie ID String to a Movie object with all the information about that movie.
    - A public initialize method with one String parameter named moviefile. You can call this method with the name of the file used to initialize the movie database.
    - A private initialize method with no parameters that will load the movie file ratedmoviesfull.csv if no file has been loaded. This method is called as a safety check with any of the other public methods to make sure there is movie data in the database.
    - A private loadMovies method to build the HashMap.
    - A containsID method with one String parameter named id. This method returns true if the idis a movie in the database, and false otherwise.
    - Several getter methods including getYear, getTitle, getMovie, getPoster, getMinutes, getCountry, getGenres, and getDirector. Each of these takes a movie ID as a parameter and returns information about that movie.
    - A size method that returns the number of movies in the database.
    - A filterBy method that has one Filter parameter named f. This method returns an ArrayList of type String of movie IDs that match the filtering criteria

## Rater Database

The class RaterDatabase – This class is very similar to the MovieDatabase.java but for the raters. The class includes:

* + - A HashMap named ourRaters that maps a rater ID String to a Rater object that includes all the movie ratings made by this rater.
    - A public static initialize method with one String parameter named filename. You can call this method with the name of the file used to initialize the rater database.
    - A private initialize method with no parameters that initializes the HashMap ourRaters if it does not exist.
    - A public static void addRatings method that has one String parameter named filename. One could alternatively call this method to add rater ratings to the database from a file.
    - A public static void addRaterRating method that has three parameters, a String named raterID representing a rater ID, a String named movieID that represents a movie ID, and a double named rating that is the rating the rater raterID has given to the movie movieID. This function can be used to add one rater and their movie rating to the database. Notice that the method addRatings calls this method.
    - A method getRater has one String parameter named id. This method returns a Rater that has this ID.
    - A method getRaters that has no parameters. This method returns an ArrayList of Raters from the database.
    - A method size that has no parameters. This method returns the number of raters in the database.

# RESULT

## RESULT

After submitting a .zip file of all the .class files involved in the project to the portal provided by Duke University on Coursera, the java program I wrote was automatically implemented on web. The project has provided an inside knowledge of developing and understanding of working of a recommendation system. It provided me with a great insight into problem-solving and algorithms-building, along with a learning of core and the clearing of a lot of the basics of Java Language. The project helped me understand the importance of extendibility and reusability in OOP programming, through the use of features like Interfaces. Building a, though simple, yet working recommendation system was a greatly creative process that involved working with a lot of classes and methods, and it hugely helped me the process of working in the programming field.

# SUMMARY AND CONCLUSION

## Summary and Conclusion

In this project, I was able to build a working recommendation system that successfully analyses the ratings submitted by the user and recommends them movies based upon the ratings given by other users and the closeness of taste of the user and those raters, using the weighted averages technique. We earned a great insight into how to understand and solve problems through programming and algorithm designing.

The project has kept a certain level of simplicity till the end, and only a certain output has been focused on. There are further various alterations that might be done to the program to make it more appealing, like using API to run trailers along with the poster too.

The approach towards the project has throughout considered the possibility to extend and reuse the code. For instance, the *Filter* interface, which is used in the project to only facilitate *TrueFilter* that passes all movies, can be used to make more filters like *DirectorsFilter* or *GenreFilter.*

The project may also be extended in a way that can display more information about the product, such as how many raters rated it or the rating it holds. One can search for all the movies directed by a certain director, or movies that are of a certain duration, as the objects we created hold such information already.

The program can of course access the data from web to get hold of real time data and display the results based upon that, which will help the program be more relevant and universal.

# REFERENCES

## REFERENCES

1. <https://www.dukelearntoprogram.com/course5/doc/>
2. <https://en.wikipedia.org/wiki/Java_(programming_language)>
3. <https://www.tutorialspoint.com/java/java_overview.html>
4. <https://www.bluej.org/about.html>
5. <https://stackoverflow.com/>
6. <https://www.geeksforgeeks.org/>

# Appendix A SCREENSHOTS

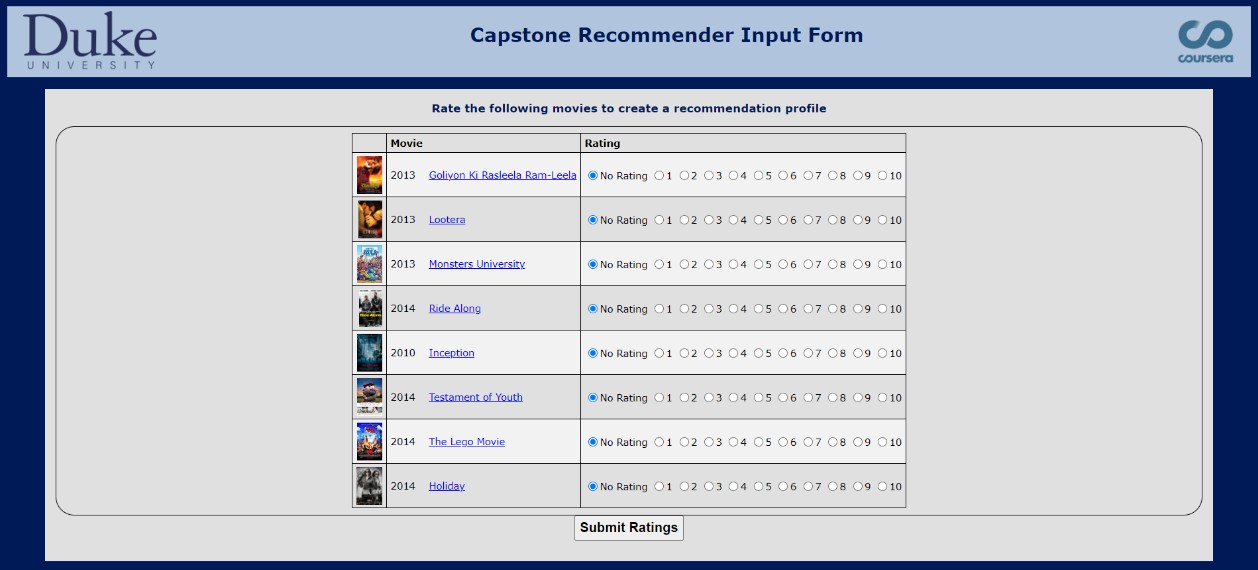
Fig AA1. Home Screen Main Menu

Fig AA2. The user can easily rate the movies displayed on the screen

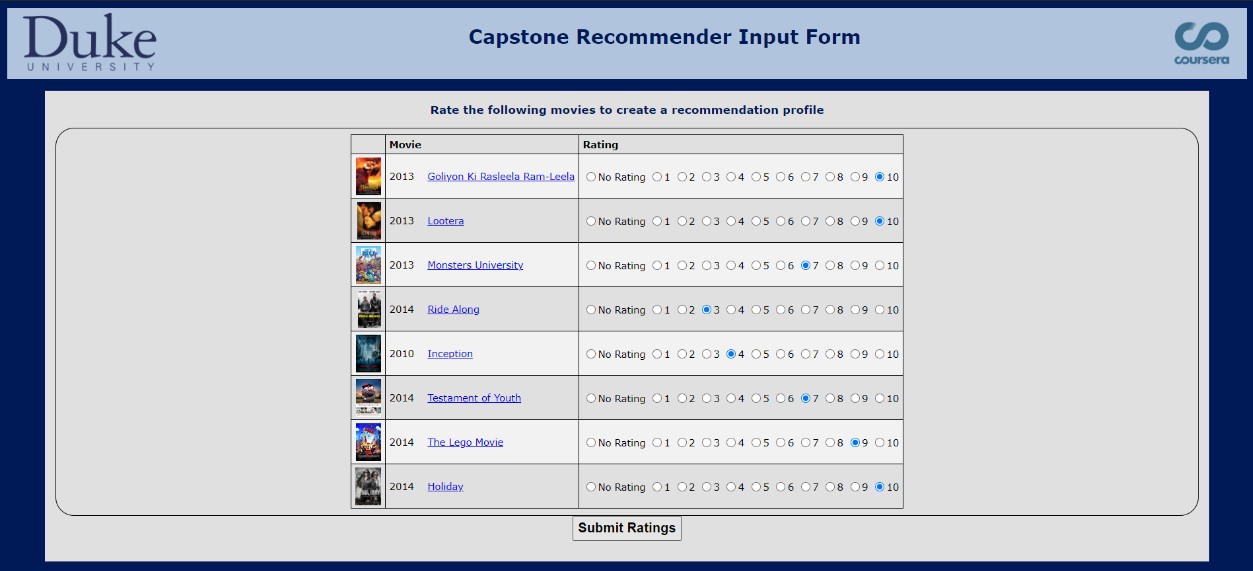


Fig AA3.1. The output displayed after the user submits ratings

Fig AA3.2. The output displayed after the user submits ratings

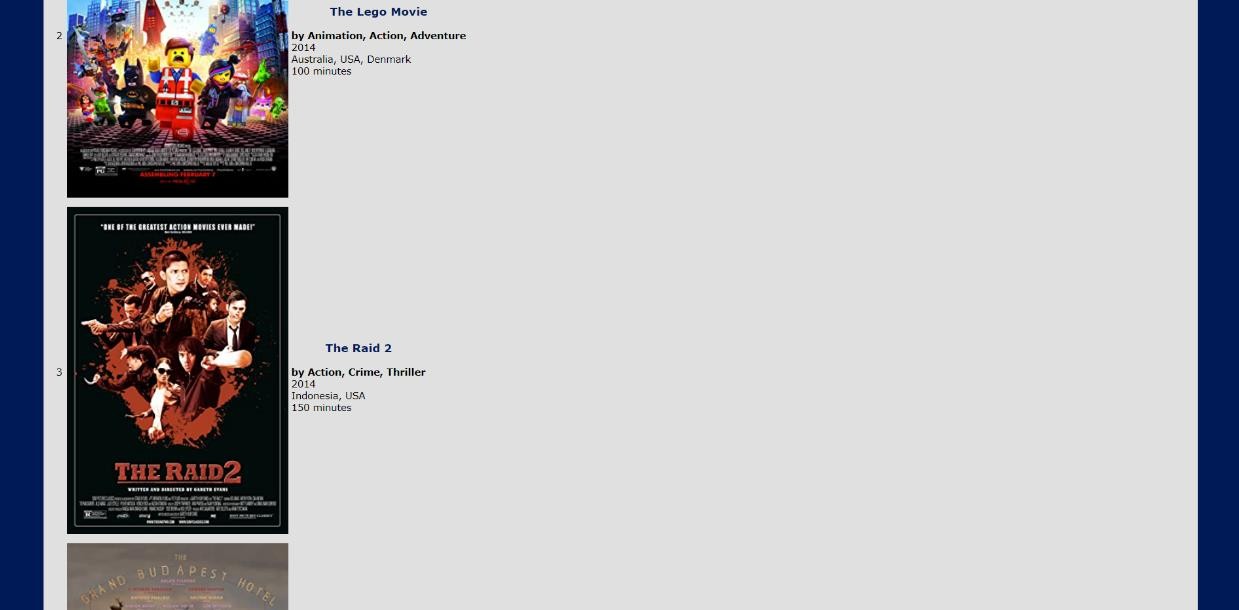
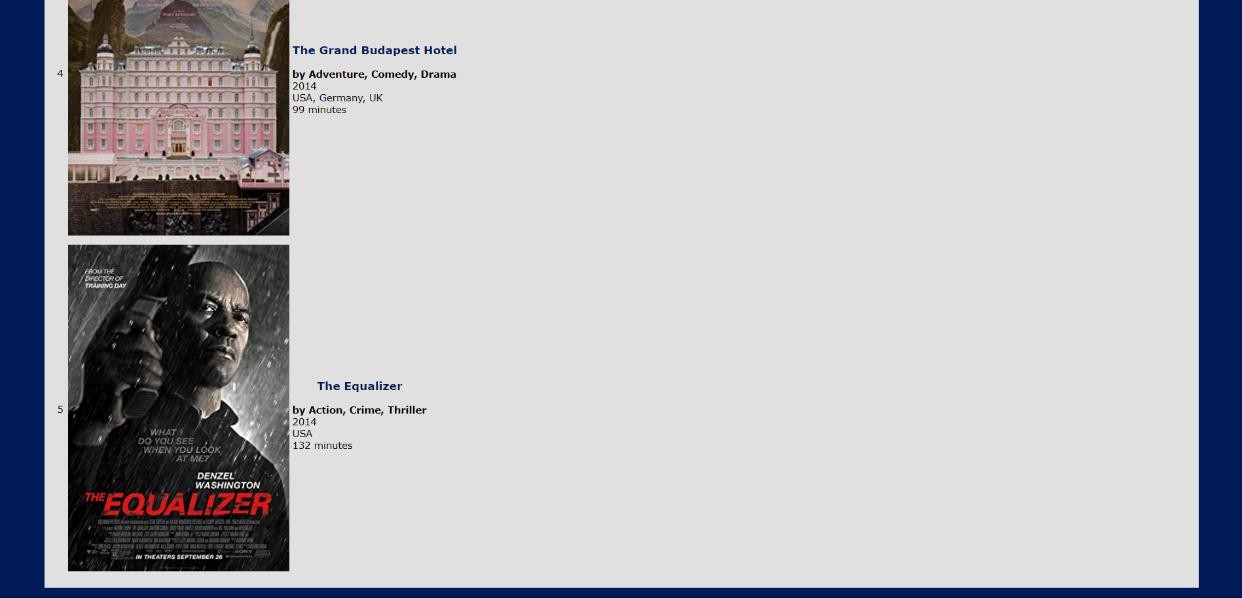


Fig AA3.3. The output displayed after the user submits ratings



# Appendix B SOURCE CODE

**AB1. Recommender**

import java.util.\*;

public interface Recommender {

public ArrayList<String> getItemsToRate ();

public void printRecommendationsFor (String webRaterID);

}

**AB2. RecommendationRunner**

import java.util.\*;

public class RecommendationRunner implements Recommender { public ArrayList<String> getItemsToRate(){

ArrayList<String> moviesToRate = new ArrayList<String>(); String [] movies ={"2215477","2224317","56993","1453405",

"365748","1408253","1375666","1441953",

"1490017","87843","833476","2556308"};

for(int i=0;i<12;i++){

moviesToRate.add(movies[i]);

}

return moviesToRate;

}

public void printRecommendationsFor(String webRaterID){ FourthRatings fr = new FourthRatings();

int numSimilarRaters = 5; int minimalRaters = 3 ;

ArrayList<Rating> recList = fr.getSimilarRatings(webRaterID,numSimilarRaters,minimalRaters);

if(recList.size()==0){

printError();

}

int i=0;

else{

printUpperPart();

for(Rating r: recList){ i++;

if((i+1)%2 == 0){

System.out.println("<tr class=\"even\_rows\"><td>" + i + "</td>");

}

else{

}

System.out.println("<tr class=\"odd\_rows\"><td>" + i + "</td>");

String URL = MovieDatabase.getPoster(r.getItem()); String title = MovieDatabase.getTitle(r.getItem());

String director = MovieDatabase.getDirector(r.getItem()); String country = MovieDatabase.getCountry(r.getItem()); int year = MovieDatabase.getYear(r.getItem());

String genre = MovieDatabase.getGenres(r.getItem()); int minutes = MovieDatabase.getMinutes(r.getItem());

System.out.println("<td><table><tr><td class = \"pic\">"); if(URL.length()>3){

System.out.println("<img src = \""+URL+"\" target=\_blank></td>");

}

System.out.println("<td><h3>"+ title+"</h3>"); System.out.println("<b>by "+ genre+"</b><br>"); System.out.println(year+"<br>"); System.out.println(country+"<br>");

System.out.println(minutes+" minutes</td></tr></table></td></tr>"); if(i>12) break;

}

printLowerPart();

}

}

private void printError(){

System.out.println("This is system error, please try again!");

}

private void printUpperPart(){

System.out.println("<link href=\"https://fonts.googleapis.com/css?family=Tangerine\" rel=\"stylesheet\"><link href=\"https://fonts.googleapis.com/css?family=Roboto|Syncopate\" rel=\"stylesheet\"><div id=\"header\"><h2>Recommended Movies:</h2></div><table class=\"outside\_table\"><tr class=\"table-header\"><th>&nbsp</th><th class=\"movie\_title\">Title</th></tr>");

}

private void printCSS(){

System.out.println("<style>\* {margin: 0;padding: 0;}img{height: 100px;margin- right:10px;}#header{background-color: ##5689f0;margin-top: 0;height: 100px;}h2{padding-left: 15px;padding-top: 40px;color: #FFFFFF;}h3{}body{margin-top: 0;font-family: 'Arial'}th{text-align: left;font-family: 'Arial', sans-serif;padding-top:15px;padding-bottom: 7px;}td{padding-top: 10px;padding-right: 10px;padding-left: 10px;padding-bottom: 5px;}tr{padding-bottom: 10px;}.table- header{background-color: #FFB97F;}.odd\_rows{background-color: #FFE4CC;}.even\_rows{background-color: #FFFFFF;}.outside\_table{width: 100%;border-collapse: collapse;}.movie\_title{width = 40%;}</style>");

}

private void printLowerPart(){ System.out.println("</table>");

}

}

**AB3. FourthRatings**

import edu.duke.\*; import java.util.\*;

import org.apache.commons.csv.\*; public class FourthRatings {

private double dotProduct(Rater me,Rater r){

double similarValue=0;

ArrayList<String> itemsRatedByMe = me.getItemsRated(); for(String movieID:itemsRatedByMe){

if(r.hasRating(movieID)){

double rRating = r.getRating(movieID)-5; double myRating = me.getRating(movieID)-5;

similarValue = similarValue + (rRating\*myRating);

}

}

return similarValue;

}

private ArrayList<Rating> getSimilarities(String id){ ArrayList<Rating> list = new ArrayList<Rating>(); ArrayList<Rater> allRaters = RaterDatabase.getRaters(); Rater me = RaterDatabase.getRater(id);

for(Rater r: allRaters){

String raterID = r.getID(); if(raterID.equals(id)== false){

double similarValue = dotProduct(me,r);

Rating similarRating = new Rating(raterID,similarValue); list.add(similarRating);

}

}

Collections.sort(list, Collections.reverseOrder()); return list;

}

public ArrayList<Rating> getSimilarRatings(String id, int numSimilarRaters, int minimalRaters){ return getSimilarRatingsByFilter(id, numSimilarRaters, minimalRaters, new TrueFilter());

}

public ArrayList<Rating> getSimilarRatingsByFilter(String id,int numSimilarRaters, int minimalRaters, Filter filterCriteria){

ArrayList<Rating> movieSimRatings = new ArrayList(); ArrayList<Rating> raterSimList = getSimilarities(id); ArrayList<String> movieIDList = new ArrayList();

HashMap<String,Double> similarMap = new HashMap(); int mapSize = getSimilarities(id).size();

int minIndex = Math.min(mapSize, numSimilarRaters);

for(Rating similar : getSimilarities(id).subList(0,minIndex)){ if(similar.getValue()>0){

similarMap.put(similar.getItem(), similar.getValue());

}

}

for(String movieID:MovieDatabase.filterBy(filterCriteria)){ int count =0;

double total =0;

for(Rater curRater:RaterDatabase.getRaters()){ double rating = -1;

if(similarMap.containsKey(curRater.getID()) && curRater.hasRating(movieID)){

rating = curRater.getRating(movieID) \*

similarMap.get(curRater.getID());

if(rating ==-1){}

else{

}

}

count++;

total = total + rating;

if(count< minimalRaters || total==0){} else{

movieSimRatings.add(new Rating(movieID, total/count));

}

}

Collections.sort(movieSimRatings, Collections.reverseOrder()); return movieSimRatings;

}

}

**AB4. FirstRatings**

import edu.duke.\*; import java.util.\*;

import org.apache.commons.csv.\*; public class FirstRatings {

public ArrayList<Movie> loadMovies(String filename){ FileResource fr = new FileResource(filename); ArrayList<Movie> movieList = new ArrayList<Movie>(); for(CSVRecord record:fr.getCSVParser()){

Movie currMovie = new Movie(record.get("id"),record.get("title"),record.get("year"),record.get("genre"),record.get("director")

,record.get("country"),record.get("poster"),Integer.parseInt(record.get("minutes")));; movieList.add(currMovie);

}

return movieList;

}

}

**AB5. Rater**

import java.util.\*; public class Rater{

private String myID;

private HashMap<String,Rating> myRatings;

public Rater(String id) { myID = id;

myRatings = new HashMap<String,Rating>();

}

public void addRating(String item, double rating) { myRatings.put(item,(new Rating(item,rating)));

}

public boolean hasRating(String item) { return myRatings.containsKey(item);

}

public String getID() { return myID;

}

public double getRating(String item) { if(hasRating(item)){

return myRatings.get(item).getValue();

}

else{

return -1;

}

}

public int numRatings() { return myRatings.size();

}

public ArrayList<String> getItemsRated() { ArrayList<String> list = new ArrayList<String>(); for(String movieID:myRatings.keySet()){

list.add(movieID);

}

return list;

}

}

**AB6. Movie**

import java.util.ArrayList; import java.util.Arrays;

public class Movie { private String id; private String title; private int year; private String genres; private String director; private String country; private String poster; private int minutes;

public Movie (String anID, String aTitle, String aYear, String theGenres) { id = anID.trim();

title = aTitle.trim();

year = Integer.parseInt(aYear.trim()); genres = theGenres;

}

public Movie (String anID, String aTitle, String aYear, String theGenres, String aDirector, String aCountry, String aPoster, int theMinutes) {

id = anID.trim(); title = aTitle.trim();

year = Integer.parseInt(aYear.trim()); genres = theGenres;

director = aDirector; 22

country = aCountry; poster = aPoster; minutes = theMinutes;

}

public String getID () { return id;

}

public String getTitle () { return title;

}

public int getYear () { return year;

}

public String getGenres () { return genres;

}

public String getCountry(){ return country;

}

public String getDirector(){ return director;

}

public String getPoster(){ return poster;

}

public int getMinutes(){ return minutes;

}

public String toString () {

String result = "Movie [id=" + id + ", title=" + title + ", year=" + year; result += ", genres= " + genres + "]";

return result;

}

}

**AB7. Rating**

public class Rating implements Comparable<Rating> { private String item;

private double value;

public Rating (String anItem, double aValue) { item = anItem;

value = aValue;

}

public String getItem () { return item;

}

public double getValue () { 23

return value;

}

public String toString () {

return "[" + getItem() + ", " + getValue() + "]";

}

public int compareTo(Rating other) { if (value < other.value) return -1; if (value > other.value) return 1;

return 0;

}

}

**AB8. RaterDatabase**

import edu.duke.\*; import java.util.\*;

import org.apache.commons.csv.\*;

public class RaterDatabase {

private static HashMap<String,Rater> ourRaters;

private static void initialize() { if (ourRaters == null) {

ourRaters = new HashMap<String,Rater>();

}

}

public static void initialize(String filename) {

if (ourRaters == null) {

ourRaters= new HashMap<String,Rater>(); addRatings("data/" + filename);

}

}

public static void addRatings(String filename) { initialize();

FileResource fr = new FileResource(filename); CSVParser csvp = fr.getCSVParser(); for(CSVRecord rec : csvp) {

String id = rec.get("rater\_id"); String item = rec.get("movie\_id"); String rating = rec.get("rating");

addRaterRating(id,item,Double.parseDouble(rating));

}

}

public static void addRaterRating(String raterID, String movieID, double rating) { initialize();

Rater rater = null;

if (ourRaters.containsKey(raterID)) { rater = ourRaters.get(raterID);

}

else {

rater = new EfficientRater(raterID); ourRaters.put(raterID,rater);

}

rater.addRating(movieID,rating); 24

}

public static Rater getRater(String id) { initialize();

return ourRaters.get(id);

}

public static ArrayList<Rater> getRaters() { initialize();

ArrayList<Rater> list = new ArrayList<Rater>(ourRaters.values());

return list;

}

public static int size() {

return ourRaters.size();

}

}

**AB9. MovieDatabase**

import java.util.\*;

import org.apache.commons.csv.\*; import edu.duke.FileResource;

public class MovieDatabase {

private static HashMap<String, Movie> ourMovies;

public static void initialize(String moviefile) { if (ourMovies == null) {

ourMovies = new HashMap<String,Movie>(); loadMovies("data/" + moviefile);

}

}

private static void initialize() { if (ourMovies == null) {

ourMovies = new HashMap<String,Movie>(); loadMovies("data/ratedmoviesfull.csv");

}

}

private static void loadMovies(String filename) { FirstRatings fr = new FirstRatings(); ArrayList<Movie> list = fr.loadMovies(filename); for (Movie m : list) {

ourMovies.put(m.getID(), m);

}

}

public static boolean containsID(String id) { initialize();

return ourMovies.containsKey(id);

}

public static int getYear(String id) { initialize();

return ourMovies.get(id).getYear();

} 25

public static String getGenres(String id) { initialize();

return ourMovies.get(id).getGenres();

}

public static String getTitle(String id) { initialize();

return ourMovies.get(id).getTitle();

}

public static Movie getMovie(String id) { initialize();

return ourMovies.get(id);

}

public static String getPoster(String id) { initialize();

return ourMovies.get(id).getPoster();

}

public static int getMinutes(String id) { initialize();

return ourMovies.get(id).getMinutes();

}

public static String getCountry(String id) { initialize();

return ourMovies.get(id).getCountry();

}

public static String getDirector(String id) { initialize();

return ourMovies.get(id).getDirector();

}

public static int size() { return ourMovies.size();

}

public static ArrayList<String> filterBy(Filter f) { initialize();

ArrayList<String> list = new ArrayList<String>(); for(String id : ourMovies.keySet()) {

if (f.satisfies(id)) { list.add(id);

}

}

return list;

}

}

**AB10.1. Filter**

public interface Filter {

public boolean satisfies(String id);

}

**AB10.2 TrueFilter**

public class TrueFilter implements Filter { @Override

public boolean satisfies(String id) { return true;

}

}