Recommendation System Exercise

Over the course of the next weeks we will be developing a single user movie recommendation system. You will work in groups throughout the project.

The following are the **mandatory** functional requirements to the system:

- It must have a graphical user interface
- As a user you must be able to add new movies to the system
    - You should be able to do all C.R.U.D.[1] operations on movies
- As a user you must be able to rate all movies on a scale from 1-5 based on how much you like the movie (-5 (bad), -3, 1 (neutral), 3, 5 (good))
    - On the UI you can optionally display this 5 point scale differently: ★★★☆☆
- As a user you must be able to receive recommendation for movies to watch based on:
    - … all users top rated movies you haven't seen (rated) yet
    - … your personal rating history
- As a user you may be able to search for movies based on their title

Besides the requirements above the following are **nice to have** features.

- The application should **automatically** search movies when you enter a key in the search field
- The application should **automatically** recommend movies whenever you have rated a new one
- Movies must always be displayed **the same way** and with an **image of the cover**.
- Movies should also contain categories
    - As a user you should be able to do all the mandatory operations limiting the results to certain categories (Search / recommendation)

The following are the non-functional requirements:

[1] **C**reate **R**ead **U**pdate **D**elete

- The application should have a **layered architecture**
- The code must not be smelly...
- It must **store data** between executions
- Operations should be relatively **fast** when performing operations

Building a recommendation system is no trivial task. In 2010 the streaming giant Netflix issued a competition to see who could create the most efficient recommendation system[2]. To that end they also released a dataset containing millions of actual user recommendations. We will be using data from that dataset to create our own recommendation system. The dataset contains several files:

movie_titles.txt
ratings.txt
users.txt

The files are both structured with one data item per row. Thus the first line of the movie titles dataset is:

1,2003,Dinosaur Planet

This means that there is a movie with the ID of 1, released in 2003 and the title is “Dinosaur Panet”. The following is the first line of the TrainingRatings.txt:

8,1744889,-5

This means that the movie with the ID 8, was rated by the user with the ID 1744889, and the rated value was -5. There is no data on the users and the only thing we know about them is their ID.

Over the course of the weeks we will extend the system so that we at the end of the semester have met all the mandatory requirements.

[2] https://www.netflixprize.com/

Stage 1 - File Operations

First we must get familiar with our data so that we can effectively perform CRUD operations. First we will do some simple read operations on the movie_titles file. Created a temporarily main method in the provided DAO class. Use this method to test your output.

The resources below together with in-class instructions should enable you to solve the exercises.

Relevant resources:
Big Java:

- 11.1 - 11.2

Online

- A quick hands on file reading tutorial
- Java file writing example
- The Buffered reader (Has a new line method…)
- Java's own (thorough) IO documentation
- The try with resources statement

Exercise 0: Prepping the MovieDAO class

A. Implement the methods in the MovieDAO class.
B. Run them in the temporarily main method for validation.

Exercise 1: Getting all Movies on the small screen

A. Change from ListView<?> to ListView<Movie> in the controller.
B. Create a Movie Model class with an ObservableArrayList<Movie> instance field in it.
C. Connect the ListView to the Model's ObservableArrayList so updates happens automatically.
D. Connect your Model to BLL → DAL and fetch all Movies and add them to your ObservableArrayList.
E. Run and rejoice.

Exercise 2: Searching through titles

A. Add a Button with the text search, and add an event handler to it.
B. Let the event handler use a search method you create in the model.
C. Let the search method in the model use the unimplemented search method in the BLL.
D. Implement the search method in the BLL.
    a. Run the associated test!

    b. Make them pass.

E. In the Model, when your search method returns, store the result to a local variable. Clean the ObservableArrayList. Add all results from the local variabel to the ObservableArrayList.
F. If the search query is null or an empty string, load all movies instead of performing the search.
G. Run and rejoice.

Exercise 3: Adding Movie CRUD to the GUI

This exercise is about expanding the GUI so that you can create, update, and delete Movies from the persistence storage. It's important that you follow the architecture. So go from GUI → BLL → DAL and use the unimplemented methods in the MovieDAO class.

Exercise 4: User and Ratings

Implement all the undone methods in the UserDAO and RatingsDAO classes. Test them using a main method. See that they work.

Exercise 5: Extend the GUI

Add the following functionality, from the GUI to the DAL, to the system:

- As a user you should be able to log in by choosing your user from a searchable list (Searches names and ID)
- As a user you should be able to rate a movie
- **As a user you should receive movie recommendations based on your own ratings. This is a hard exercise. Look at the appendix on help to solve the challenge.**

Stage 2 - Database operations

Using files for persistence can be great. But it has some limitations. Greatest of the limitations is the fact that we can no easily share them between systems. In the following we will get familiar with relational databases and how we can use them for storing data.

Relevant resources:
Big Java:

- Ch. 22 (online)
  - 22.1.1, 22.1.2, 22.2.1-22.2.4, 22.2.6, 22.4
- Ch. 23.1.1, 23.1.2, 23.2.1-23.2.4, 23.2.6, 23.4

Online

- One article about SQL injetions and another on the same topic

Exercise 0: Database IO

Mirror the DAL layer so that we have two versions that supply identical functionality. One that uses files, and one that uses a database.

Exercise 1: Data mitigation

Create a script that reads data from your files, and pushes it to your database.

Exercise 2: Tying the knots

Change your implementation so that it uses the database DAL. Validate that your program still works.

Appendix A - Creating recommendations[3]

The component of this project that sets it apart from the implementation of a basic database is the ability of the program to make predictions about which other books a given user might like. Implementing this feature provides a wide range of opportunities for differentiated learning and a number of openings for discussing broader Computer Science concepts. In particular, it provides a hook to current research in artificial intelligence that could be explored by strong students. Below is an explanation of three different prediction algorithms.

Approach A: Basic
Let's consider a user named Rabia. How is it that the program should predict other books Rabia might like? The simplest approach would be to make almost the same prediction for every customer. In this case the program would simply calculate the average rating for all the books in the database, sort the books by rating and then from that sorted list, suggest the top 5 books that Rabia hasn't already rated.

Approach B: More Sophisticated
In Approach A, the only information unique to Rabia used by the prediction algorithm was whether or not Rabia had read a specific book. We could make a better prediction about what Rabia might like by considering her actual ratings in the past and how these ratings compare to the ratings given by other customers. Consider how you decide on movie recommendations from friends. If a friend tells you about a number of movies that s(he) enjoyed and you also enjoyed them, then when your friend recommends another movie that you have never seen, you probably are willing to go see it. On the other hand, if you and a different friend always tend to disagree about movies, you are not likely to go to see a movie this friend recommends.

The program can calculate how similar two users are by treating each of their ratings as a vector and calculating the dot product of these two vectors. You may have to remind (or even teach) students the definition of a dot product. It is just the sum of the products of each of the corresponding elements. Suppose we had 3 books in our data-base and Rabia rated them
[5,3,-5], Suelyn rated them [1,5,-3], Bob rated them [5,-3,5], and Kalid rated them [1, 3, 0].

The similarity between Rabia and Bob is calculated as: (5 X 5) + (3 X -3) + (-5 x 5) = 25 -9 -25 = -9
The similarity between Rabia and Suelyn is: (5x1) + (3x 5) + (-5x -3) = 5 + 15+ 15 = 35
The similarity between Rabia and Kalid is (5x1) + (3x3) + (-5x0) = 5 + 9 + 0 = 14

We see that if both people like a book (rating it with a positive number) it increases their similarity and if both people dislike a book (both giving it a negative number) it also increases their similarity.

[3] Kindly borrowed from http://nifty.stanford.edu/2011/craig-book-recommendations/cs1/handout.shtml

Once you have calculated the pair-wise similarity between Rabia and every other customer, you can use the ratings of the user who is most similar to Rabia to make your sorted list. Then, you can use this list to make some recommendations to Rabia. In this case Rabia is most similar to Suelyn, so we would sort for the books according to Suelyn's ratings. Then we would recommend to Rabia the top books from Suelyn's list that Rabia hadn't already rated.

Approach C: Most Sophisticated
In Approach B, the algorithm used the ratings of every customer to calculate how similar Rabia is to every other user in the database and find the single user who is most similar to Rabia. Once the algorithm finds that Suelyn is the closest match to Rabia, it only uses Suelyn’s ratings to make suggestions for Rabia and doesn’t consider the ratings of anyone else at all. Wouldn’t it be more interesting to consider the ratings of a number of different people who are similar to Rabia? But how many and how much weight should we give to the different recommendations? Think again about the friends telling you about a movie you should see. If one friend Irfan (whose movie sense you tend to trust) tells you he hated a certain movie, but 5 other friends (whose movie sense you also trust) enjoyed the movie, you might decide to go see it. However, if you really trust Irfan’s judgement and you often don’t agree with the other 5 friends, that might change your opinion. The more you trust a person’s ratings, the more you will factor them into your final decision.

For this approach, your algorithm will look at every user and calculate how similar that person is to Rabia just as it did in Approach A. Then, it will calculate a final prediction rating for each book by weighting the ratings given by each user and according to how similar each user’s ratings are to Rabia’s.

Let’s look again at our example. The final rating of the books for Rabia is calculated as:

(Bob's ratings) X (Bob's similarity to Rabia) + (Kalid's ratings) X (Kalid's similarity to Rabia) + (Suelyn's ratings) X (Suelyn's similarity to Rabia)

= [5,-3,5] * x -9 + [1, 3, 0] x 14 + [1,5,-3] * 35
= [-45,27,-45] + [13,42,0] + [35,175,-105]
= [3,244,-150]

So we would recommend books to Rabia in the order book 2, book 1, and book 3. Of course, this doesn't make much sense since Rabia has already read all the books in our system. In order for the algorithm to work, the user has to have some books that they have not yet read.

Determining a Rating System
In the student handout, the rating system is not prescribed. Students are allowed to design their own system. It is likely that without guidance students will invent a scheme that uses only non-negative numbers. Many will propose using a scale of 0-5 or maybe 1-5 with 5 being “I really liked this” and 1 being “I didn’t like it”. They might then use 0 to be “I didn’t read this”. This is an obvious choice and will work sufficiently well for prediction Approach A (the basic one). Unfortunately, it won't work nearly as well if the student is calculating the

similarity of two users using the dot product of rating vectors. A much better rating scheme would be something that uses both positive and negative ratings, centred around 0 but not using 0 for a rating itself. Zero is saved for indicating that this user hasn't rated this book.

Here is the example rating scheme used in the prediction example above:

-5	: Hated it!
-3	: Didn't like it
1	: ok – neither hot nor cold about it
3	: Liked it
5	: Really liked it!
0	: Didn't read it

If the student insists on using a strictly positive rating scheme, the similarity calculation can be adjusted to take this into account by shifting all the ratings to make the range centre around zero before taking the dot product. This also means a few extra conditional statements to consider the "Didn't read it" case and to avoid including this value in the similarity calculations. The ideal practice of using zero to indicate a book that wasn't read means the similarity calculation effectively ignores the rating of one user for any book that the other user hasn't read.

Notice that you may need to remind students that 0 isn’t really a rating so if the program calculates average ratings, the zeros themselves can be added into the total (since they have no effect) but the number of zeros isn’t part of the count.