Project 1: Movie Recommender

Due: 02/21/2018, 11:59pm

Overview:

For this project, you will implement a simple movie recommender that makes recommendations to a given user based on what "similar" users liked in the past. Such approach is called *collaborative filtering.* Different algorithms can be used to compute *similarity* between a pair of users given their movie ratings. We will use Pearson correlation (https://en.wikipedia.org/wiki/Pearson_correlation_coefficient) - we discuss it later in the document. You will store movie ratings data in a custom data structure built using linked lists.

Important note: you may not use Java's built-in ArrayList and LinkedList classes for this project (and other container classes from Java standard library, except where explicitly allowed). The custom linked lists required for the assignment should be implemented completely from scratch. You are required to use the provided **starter code**. Do *not* modify signatures of methods in the starter code.

Data set[1]

You will use real movie ratings data collected by Grouplens Research from the MovieLens website. The provided files `movies.csv` and `ratings.csv` contain information about ~165K movies and ~100K ratings of 671 users.
movies.csv has the following format:

```
movieId,title,genres
1,Toy Story (1995),Adventure|Animation|Children|Comedy|Fantasy
2,Jumanji (1995),Adventure|Children|Fantasy
3,Grumpier Old Men (1995),Comedy|Romance
4,Waiting to Exhale (1995),Comedy|Drama|Romance
5,Father of the Bride Part II (1995),Comedy
6,Heat (1995),Action|Crime|Thriller
7,Sabrina (1995),Comedy|Romance
8,Tom and Huck (1995),Adventure|Children
9,Sudden Death (1995),Action
10,GoldenEye (1995),Action|Adventure|Thriller
11,"American President, The (1995)",Comedy|Drama|Romance
...
```

Each line contains information about one movie (movie id, movie title and genres of this movie). Note that the title of the movie might contain commas, in which case it would be in quotes (for instance, see the movie with id = 11).
You need to read data from this file (fill in code in `loadMovies` method in class `MovieRecommender`) and create a **HashMap (**called `movieMap` in class `MovieRecommender`**),**

[1] http://grouplens.org/datasets/movielens/
We will be using: http://files.grouplens.org/datasets/movielens/ml-latest-small.zip

where the keys are movie ids and the values are movie titles. Do not remove the year from the title - the title will include the year; for instance, the title might look like this: `Father of the Bride Part II (1995).` Ignore movie genres (unless you plan to do extra credit for the assignment).

ratings.csv has the following format:

```
userId,movieId,rating,timestamp
1,31,2.5,1260759144
1,1029,3.0,1260759179
1,1061,3.0,1260759182
1,1129,2.0,1260759185
1,1172,4.0,1260759205
1,1263,2.0,1260759151
1,1287,2.0,1260759187
1,1293,2.0,1260759148
1,1339,3.5,1260759125
1,1343,2.0,1260759131
1,1371,2.5,1260759135
1,1405,1.0,1260759203
1,1953,4.0,1260759191
1,2105,4.0,1260759139
1,2150,3.0,1260759194
1,2193,2.0,1260759198
1,2294,2.0,1260759108
1,2455,2.5,1260759113
1,2968,1.0,1260759200
1,3671,3.0,1260759117
2,10,4.0,835355493
2,17,5.0,835355681
2,39,5.0,835355604
2,47,4.0,835355552
...
```

Each line in **ratings.csv** contains four values: the user id, the movie id, the rating, and the timestamp separated by commas. For instance, in the file above, user with id=1 rated movie with movie id=31 as 2.5. The same user rated movie with movie id = 1029 as 3.0. You need to read each line and extract the first three values (user id, movie id, and rating); ignore the timestamp - we will not use it for the project. The rating value varies from **0.5 to 5.0** (where a low rating means the user did not like the movie and high rating of 5.0 means the user enjoyed the movie very much).
You need to read data from this file (fill in code in `loadRatings` method in class `MovieRecommender`) and store it in the custom data structure described below.

Storing users' ratings

For this project, you need to store information about users and movie ratings in the data structure shown in Figure 1. A singly linked list displayed vertically in *black /light blue color* is **UsersList** (you need to fill in code in this class); it stores a node (**UserNode**) for *each* user from the data set. Each UserNode contains the following information:

- A user **id**
- A reference to the **next** UserNode

- A reference to the **MovieRatingsList** (a different linked list that contains this user's movie ratings).

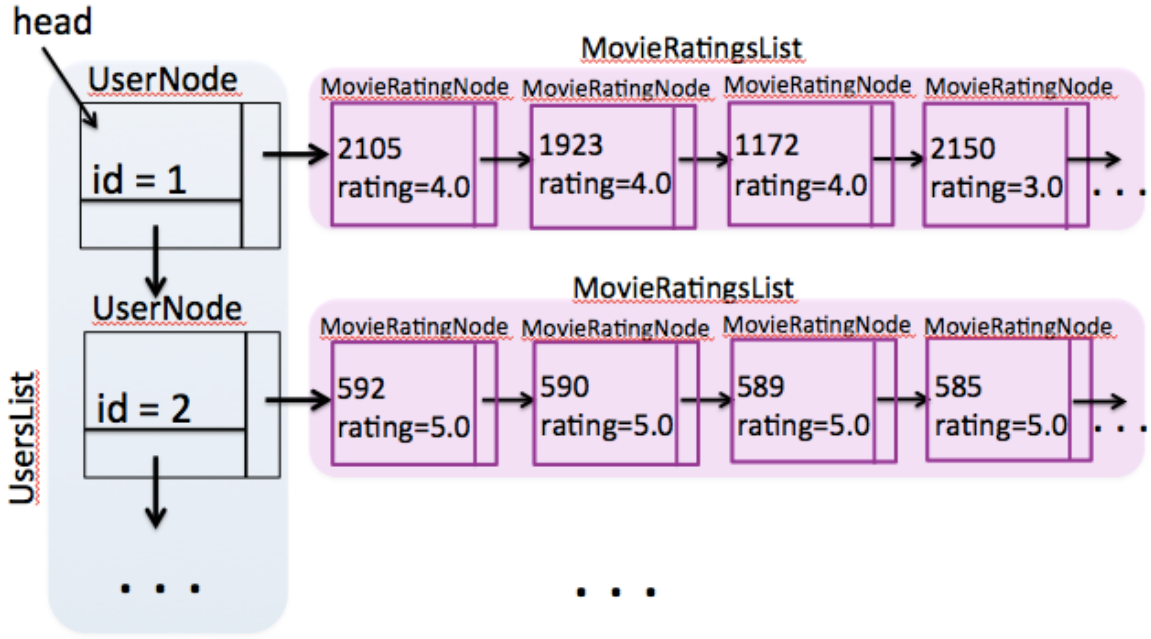


Figure 1: Custom data structure to store movie ratings data for project 1.

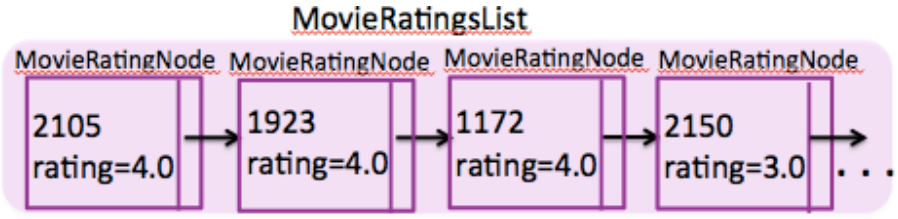
```
public class UserNode {
	private int userId;
	private MovieRatingsList movieRatings;
	private UserNode nextUser;

	// Other code (methods..)
}
```

Most of the methods of class `UserNod` have been provided to you. You just need to fill in methods `getFavoriteMovies` and `getLeastFavoriteMovies` (see comments above each method for details).

In `UsersList` class, you will need to fill in code in several methods such as appending a new node to the list; inserting new information into the list for a given user id, movie id and rating; getting the node by the user id, computing the most similar user, printing the data structure into the file in a given format etc. *Please read comments above each method in the provided starter code to see what the method is supposed to do.*

Each of the singly linked lists displayed "horizontally" in pink color (**MovieRatingsList**) stores movie ratings for a particular user:



A `MovieRatingNode` represents one node in the movie ratings list. It contains:

- the movie id (such as 2105),
- the rating value (between 0.5 and 5.0),
- the reference to the next node.

`MovieRatingNode` class has been provided to you. Do no modify this class.

MovieRatingsList class (where you need to fill in code in many methods) should store **only the head** of the list (**no** tail, **no** variable that stores the size; note: there will be a deduction if you store the tail or the size since it makes some methods easier to implement). The nodes in the MovieRatingsList should be **sorted by rating**. `insertByRating` method in MovieRatingsList should take a movieId and a rating, create a new node with these values and insert it in the correct place into the sorted list so that the list is still sorted by rating after the insertion.

In addition to `insertByRating`, **MovieRatingsList** class contains various other methods to manipulate this custom linked list (you need to fill in code of these methods). For instance, we want to be able to change the rating of a particular movie, to return the sublist that contains ratings in a certain range, to find N best ranked movies, as well as N worst ranked movies, to compute similarity of "this" list with another MovieRatingsList (discussed below) etc.

Please see the starter code (read comments above each method) for a full list of methods in the class. Note: Not all of these methods are required for the movie recommender, but you are still required to implement all of them. Make sure to test all these methods separately. For each method, there are specific requirements on how it should be implemented (for instance, you are required to use fast&slow pointers to find the middle node in the list). The requirements are provided in the comments above each method.

Please note that there is a typo in the comment above `getNWorstRankedMovies method` - it should say "to find the n-th node from the end of the list, use two pointers, where first you move one pointer so that pointers are n-nodes apart, and then move both together until the first pointer reaches null; when it happens, the second pointer would be pointing at the right node".

As mentioned above, the nodes in the MovieRatingsList should be ***sorted by rating (from the largest to the smallest).*** When you implement different operations on this linked list, make sure this order is maintained (apart from `reverse`). If several movies *have the same rating*, insert a node with a higher movieId *before* the node with the lower movieId.

Computing Recommendations

As we mention earlier, your recommender will make recommendations to a given user based on what "similar" users liked. Specifically, to compute recommendations for a given user X, you will

- Compute the "similarity" between X and all the other users (see below).
- Find the user S with the highest similarity.
- Take several movies that user S rated highly (5.0) and recommend them to user X (if X hasn't seen these movies yet).

For this project, you will use a *Pearson correlation coefficient*[2] as the measure of similarity between two vectors of movie ratings. If we have two users, and X and Y are their vectors of movie ratings that include ratings for movies *they both saw and rated,* then their similarity score (using Pearson correlation coefficient) can be computed using the following formula[2]*:*

$$r_{xy} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\,\sqrt{n\sum y_i^2 - (\sum y_i)^2}}$$

where **n** is the number of ratings for each vector. Note that n is the same for both vectors X and Y since these vectors contain ratings of movies **both** users rated; if a particular movie has been rated by only one of the two users, it's rating is not included in the vector for computing the similarity score (although it *is* stored in the data structure in Figure 1).
The coefficient varies from -1.0 to 1.0.

Example: Suppose we have four users, Alice, Hao, Pablo and David; and six movies with movie ids from 0 to 5. Assume that the users rated movies as following (the first number in parentheses is movie id, the second is the rating of this movie):

Alice's ratings:	(1, 3.0), (2, 4.0), (3, 5.0), (5, 1.0)
Hao's ratings:	(0, 5.0), (1, 2.0), (2, 5.0), (3, 5.0), (4, 2.0), (5, 2.0)
Pablo's ratings:	(0,1.0), (2, 2.0), (3, 5.0), (4, 5.0), (5, 5.0)
David's ratings:	(0, 3.0), (2, 1.0), (3, 3.0), (4, 5.0), (5, 5.0)

You can see that Alice did not see movie with id = 0, she thought movie with id=1 was just ok (rated as 3.0), she really liked the movie with id = 3 etc.

Let's say we want to recommend a movie to Alice, based on how she and other users rated movies in the past. We can compute the similarity between Alice and all the other users using the formulas above. Note that we only need to include the terms that correspond to the movies that **both** people saw.
For instance, let us see how we would compute the similarity between Alice and Pablo:
Alice's ratings vector X = {4.0, 5.0, 1.0)
Pablo's ratings vector Y = {2.0, 5.0, 5.0)
n here is 3. Note that vectors X and Y contains movie ratings for movies with ids = 2, 3, 5 because these are the movies *both* Alice and Pablo saw and rated. To compute similarity r_{xy}, we first compute:

---

[2] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

$\sum x_i y_i$ = 4.0*2.0 + 5.0*5.0 + 1.0*5.0 = 38.0
$\sum x_i$ = 4.0 + 5.0 + 1.0 = 10.0
$\sum y_i$ = 2.0 + 5.0 + 5.0 = 12.0
So in the numerator of the formula, we get: 3*38 - 10*12 = 114 - 120 = -6;
$\sum x_i^2$ = 4.0*4.0 + 5.0*5.0 + 1.0*1.0 = 16.0 + 25.0 + 1.0 = 42.0
$\sum y_i^2$ = 2.0*2.0 + 5.0*5.0 + 5.0*5.0 = 4.0 + 25.0 + 25.0 = 54.0
Then in the denominator, we will get:
sqrt(3*42.0 - 10^2) * sqrt(3*54.0 - 12^2) = sqrt(126-100)*sqrt(162 - 144) = sqrt(26)*sqrt(18) ~ 5.09 * 4.24 ~ 21.58
Hence, r_{xy} = -6 / 21.58 ~ -0.27

We can repeat this process and compute the similarity coefficient between Alice and all the other users, and then pick the user with *the highest similarity score*. Then we can take several movies this user rated as 5 and recommend them to Alice if she has not seen these movies.

Implementation notes: as shown in Figure 1, MovieRatingsList for each user contains nodes only for the movies that the user has rated. To compute the Pearson correlation coefficient for two users whose ratings lists are represented as linked lists, you would need to first find *common movies* that *both* users saw. You may use a temporary HashMap for this: first iterate over the fist user's list of ratings and map each movie id to a rating. Then iterate over the second user's list of ratings, and for each movie id in the second list, check if it's the key in the HashMap. If it is, then it means that both users saw this movie, so we should include the ratings for this movie in vectors X and Y and use it in the calculation of Pearson coefficient.

To find **k movie recommendations** for a particular user (let's say, with userId = 3), you need to
- Find this user in the UsersList,
- Compute the similarity scores (Pearson coefficient) of this user with all the other users, and find the user that has the highest similarity
- Take k movies that the "most similar user" rated as 5 (if there are *m* such movies and *m<k,* then use *m* movies)
- Recommend them to the original user (in the example, userId=3) (only including the movies that the original user has **not** seen).

There is also an anti-recommend method that lists the "movies to avoid" for a particular user. It would return the lowest rated movies of "the most similar user" (only include movies that the original user has **not** seen).

Important notes on implementation

You are welcome to add helper methods to the classes above, but **don't change the signatures of the methods provided in the starter code.** Project should be written in Java. You may not use any third-party libraries for the project. You may *not* use LinkedList and ArrayList classes from Java's standard library for this project. If in doubt whether you can use a particular class, ask the instructor.

Project must be completed individually. You may *not* discuss it with other students, copy any code from others or the web. Any student might be asked to come in for a code review of the project.

Submission

Submit project1 code to your private github repository created for you automatically by Github Classroom when you click on the assignment link on github. The project must be submitted to this github repository by the deadline. No credit for late projects.

Testing

You are responsible for testing your code. The instructor will provide a test file that tests some of the methods of the project.

Extra credit:

This project has many opportunities for extra credit, you need to discuss them with the instructor in person. Complete required functionality and show it to the instructor before you start working on the extra credit.

.