CS:1210 Project 2

Due dates: **Phase 1**: Mon 4/24 at midnight
Phase 2: Mon 5/1 at midnight
Phase 3: Fri 5/5 at midnight

Introduction

As we browse the web, we are constantly being recommended stuff – videos, news articles, books, movies, music, etc. We have now come to routinely expect recommendations from youTube, Amazon, Netflix, New York Times, etc. These recommendations are made possible by *recommender systems* that these platforms use. Recommender systems provide an example of *machine learning*, one of the hottest areas in computer science. This project on producing movie recommendations aims to give you a glimpse of some basic ideas in machine learning.

A successful recommendation system has to, by some means, be able to predict a user's tastes and make recommendations accordingly. Over the years, Netflix has invested a lot of resources into improving its movie recommender system – some of you may have heard of the *Netflix prize* (`http://en.wikipedia.org/wiki/Netflix_Prize`). Between 2006 and 2009, Netflix ran a competition, with a prize of one million dollars to the team that could take a given dataset of over 100 million movie ratings and return recommendations that were 10% more accurate than those offered by the company's existing recommender system. This competition did a lot to energize the search for new and more accurate algorithms and better implementations of these algorithms.

For this programming project I provide to you a dataset that contains 100,000 movie ratings. These are real ratings by real people gathered by the Group Lens research group at the University of Minnesota (see `http://www.grouplens.org/`). Here is a nice summary of the data set from the `README` file accompanying the data set.

> The data was collected through the MovieLens web site (movielens.umn.edu) during the seven-month period from September 19th, 1997 through April 22nd, 1998. This data has been cleaned up – users who had less than 20 ratings or did not have complete demographic information were removed from this data set.

The ultimate goal of your program is to take a user (specified by an ID) and make movie recommendations for this user based on the rating history of this user and all the others in the provided data set.

Data Files

The dataset is available from the Group Lens page at

`http://files.grouplens.org/datasets/movielens/ml-100k.zip`

and consists of a `README` file along with bunch of other text files. This is a zip file that needs to be unzipped before you can start using it. I want you to focus on the `README` file and the following six data files: `u.data`, `u.info`, `u.item`, `u.genre`, `u.user`, and `u.occupation`. You can ignore the rest of the text files for this project. The data in all 6 data files is quite clearly explained in the `README` file, in the section titled "Detailed Descriptions of Data Files." So read the `README` file carefully first and then look through the data files to get a sense of how they are organized.

Phase 1

In the first phase of the project your program is expected to do the following three tasks:

(i) read from the given files and store the information in appropriate data structures,

(ii) define a function that allows users to explore and understand the dataset, and

(iii) produce bar plots that visualize demographic differences in users' tastes for movies.

Creating the data structures

Your program should start by reading the data files and creating five lists – a user list, a movie list, two ratings list, and a genre list. You will write functions for each of these tasks – these are described below in detail.

- Write a function with the signature:

  ```
  def createUserList():
  ```

  that reads from the file `u.user` and returns a list containing all of the demographic information pertaining to the users. Suppose I call this function as `userList = createUserList()`. Then `userList` should contain as many elements as there are users and information pertaining to the user with ID i should appear in slot $i - 1$ in `userList`. Furthermore, each element in `userList` should be a dictionary with keys "`age`", "`gender`", "`occupation`", and "`zip`". The values corresponding to these keys should simply be appropriate values read from the file `u.user`. For example, the first line in `u.user` is

  ```
  1|24|M|technician|85711
  ```

  and therefore `userList[0]` should be the dictionary

  ```
  {"age":24, "gender":"M", "occupation":"technician", "zip":"85711"}
  ```

  Thus `userList` is a list with 943 dictionaries, each dictionary containing 4 keys.

- Write a function with the signature:

  ```
  def createMovieList():
  ```

  that reads from the file `u.item` and returns a list containing all of the information pertaining to movies given in the file. Suppose I call this function as `movieList = createMovieList()`. Then `movieList` should contain as many elements as there are movies and information pertaining to the movie with ID i should appear in slot $i - 1$ in `movieList`. Furthermore, each element in `movieList` should be a dictionary with keys "`title`", "`release date`", "`video release date`", "`IMDB url`", and "`genre`". The values corresponding to these keys should simply be appropriate values read from the file `u.item`. For example, the first line in `u.list` is

```
1|Toy Story (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Toy%20Story%20(1995)|0|0|0|1|1|1|0|0|0|0|0|0|0|0|0|0|0|0|0
```

  and therefore `movieList[0]` should be the dictionary

  ```
  {"title":"Toy Story (1995)", "release date":"01-Jan-1995", "video release date":"",
  "IMDB url":"http://us.imdb.com/M/title-exact?Toy%20Story%20(1995)",
  "genre":[0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0]}
  ```

  Note that the value associated with the key "`genre`" is a length-19 list of zeroes and ones.

- To process the ratings you are required to write two functions. First, you are required to write a function with signature:

  ```
  def readRatings():
  ```

  This function reads ratings from the file `u.data`. Each line in the file contains a user ID, movie ID, rating, and time stamp. You should ignore the time stamp and read the contents of each line into a a length-3 tuple of the form (user, movie, rating). The function should return a list of 100,000 length-3 tuples. For example, the first three tuples in the returned list of ratings should be

```
(196, 242, 3), (186, 302, 3), (22, 377, 1)
```

and this corresponds to the first three lines of the file `u.data` which are:

```
196     242     3       881250949
186     302     3       891717742
22      377     1       878887116
```

Second, you are required to write a function with the signature:

```
def createRatingsDataStructure(numUsers, numItems, ratingTuples):
```

that takes the rating tuple list constructed by `readRatings` and organizes the tuples in this list into two data structures. The function takes as parameters the number of users (`numUsers`), the number of movies (`numMovies`), and a list of rating tuples of the form (user, movie, rating). Suppose I call this function as `[rLu, rLm] = createRatingsList(numUsers, numMovies, ratingTuples)`. Then `rLu` is a list, with one element per user, of all the ratings provided by each user. Similarly, `rLm` is a list, with one element per movie, of all the ratings received by each movie. In particular, the ratings provided by user with ID i should appear in slot $i-1$ in `rLu` and the ratings received by movie with ID i should appear in slot $i-1$ in `rLm`. We explain `rLu` a little bit more; `rLm` is quite similar. The ratings, by a particular user, appear as a dictionary whose keys are IDs of movies that this user has rated and whose values are corresponding ratings. For example, the user with ID 1 has rated movie 61 and given it the rating 4. Hence the key-value pair `61:4` should appear in `rLu[0]`. In `rLm`, the ratings received by a movie appear as a dictionary whose keys are IDs of users who have rated that movie.

- Write a function with the signature:

  ```
  def createGenreList():
  ```

  that reads from the file `u.genre` and returns the list of movie genres listed in the file. The genres should appear in the order in which they are listed in the file.

The data exploration function

Define a function with signature

```
def demGenreRatingFractions(userList, movieList, rLu, gender, ageRange, ratingRange):
```

The arguments to this function are:

- `userList`: This is the data structure created by calling `createUserList`. It contains information about users.
- `movieList`: This is the data structure created by calling `createMovieList`. It contains information about movies.
- `rLu`: This is the first of the two data structures created by calling `createRatingsDataStructure`. It contains movie ratings organized by users.
- `gender`: This is a string that can take on one of three possible values, "`M`", "`F`", and "`A`". This argument tells the function which subpopulation of users to focus on. Here, "`A`" refers to a person of either gender.
- `ageRange`: This is a size-2 list `[age1, age2]`, where `age1` and `age2` are integers. This argument tells the function to focus on the subpopulation of users with age at least `age1` and strictly less than `age2`. For example, if this argument has value [20, 30], it is telling the function to focus on users whose age is at least 20 and strictly less than 30.
- `ratingRange`: This is a size-2 list `[r1, r2]`, where each of `r1` and `r2` can take on values 1, 2, 3, 4, or 5. This arguments tells the function to focus on ratings that are at least `r1` and at most `r2`.

The function returns a list of size 19, containing one (floating point) number for each movie genre. The number in position i in this returned list is the fraction of ratings by the specified subpopulation for movies in genre i that fall in the range `[r1, r2]`. The numerator in the fraction is the ratings by the specified subpopulation for movies in genre i that fall in the range `[r1, r2]` and the denominator is the total number of ratings provided by the specified subpopulation. For example, if we call the function as

```
demGenreRatingFractions(userList, movieList, rLu, "F", [20, 30], [5, 5])
```

the function will focus on the subpopulation of female users of age between 20 and 29 years. In the list returned by the function, the number in position 1 corresponds to the genre "Action". So this number will be the fraction of 5 ratings (which are the top ratings) that female users between 20 and 29 years have given to "Action" movies. More specifically, if female users between 20 and 29 years have collectively provided a total of 1000 ratings to all movies and of these, there are 20 ratings to "Action" movies that are 5, then the number in position 1 in the returned list should be $20/1000 = 0.02$.

Note that if the total number of ratings provided by the subpopulation is 0, then the denominator of the fraction is 0, and the function should return a list with all values equal to `None`.

Producing visualizations

The function `demGenreRatingFractions` can be called in a variety of ways to explore the ratings dataset. You are required to call this function and then use the Python module `matplotlib` to produce two sets of visualizations that show how movie tastes differ with demographic differences. The first set of visualizations will contain bar plots showing these differences between female and male users and the other set showing differences between younger and older adults. Here are more details.

(i) Use the 5 genres "Action", "Comedy", "Drama", "Horror", and "Romance" for your plots. For these genres, produce a bar plot that comparatively shows the fraction of high ratings (i.e., 4 or 5) that women and men provide. You can obtain the fractions you will need by calling the function `demGenreRatingFractions`. Produce a second plot that comparatively shows the fraction of low ratings (i.e., 1 or 2) that women and men provide to the 5 selected genres. Note that for these plots, we are not restricting the population by age.

(ii) Use the same 5 genres, you used in item (i) and produce similar plots, but the two subpopulations we want compared are "younger adults" (defined by the age range [20, 30]) and "older adults" (defined by age range [50, 60]).

Thus you would be producing 4 plots. You will need to make sure that the plots are labeled clearly, contain appropriate legends, and make good use of colors. To see an example of the type of bar chart you are required to produce, see "Grouped bar chart with labels" in this `matplotlib` documentation `https://matplotlib.org/stable/gallery/index.html`

What to submit

You are required to submit two Python files, `project2Phase1a.py` and `project2Phase1b.py` and one pdf file called `plots.pdf`. The first Python file, `project2Phase1a.py` should contain the implementation of all of the functions listed in subsections "Creating the data structures" and the "The dataset exploration functions", but no main program. We will use an autograder to run this file through a bunch of tests. As usual, we will provide our test file to you soon. The second Python file, `project2Phase1b.py` should contain everything that is in `project2Phase1a.py` plus a main program that produces the 4 plots. You should then place these 4 plots in a document (e.g., a word document), save this as a pdf, `plots.pdf` and submit that as well.

Phase 2

Now that we've processed and stored the given data we can turn our attention to the task of predicting ratings. The general idea is that you are given a user u and a movie m that u has not rated. Your goal is to predict a rating between 1 and 5 that user u would give movie m. This rating need not be an integer – it could be 3.5, for example. To come up with a prediction for the rating, you would use the data on past rating history that you have access to.

Here are 5 simple prediction algorithms that you will implement for Phase 2 of the project. In Phase 3, you'll implement a slightly more sophisticated prediction algorithm based on *collaborative filtering*.

1. Algorithm `randomPrediction`: Given a user `u` and a movie `m`, simply return a random integer rating in the range $[1, 5]$. Use the following signature for this function.
   ```
   def randomPrediction(u, m):
   ```

2. Algorithm `meanUserRatingPrediction`: Given a user `u` and a movie `m`, simply return the mean rating that user `u` has given to movies. In other words, consider *all* the ratings that user `u` has provided and return the mean of these. Use the following function signature.
   ```
   def meanUserRatingPrediction(u, m, rLu):
   ```
   Here `rLu` is the data structure of movie ratings organized by user, that was created by called the `createDataStructure` function.

3. Algorithm `meanMovieRatingPrediction`: Given a user `u` and a movie `m`, simply return the mean rating that movie `m` has received. Use the following function signature.
   ```
   def meanMovieRatingPrediction(u, m, rLm):
   ```
   Here `rLm` is the data structure of movie ratings organized by movies, which was created by calling the`createDataStructure` function.

4. Algorithm `demRatingPrediction` uses the following signature:
   ```
   def demRatingPrediction(u, m, userList, rLu):
   ```
   This algorithm considers the set U of all users who have the same gender as user u and whose age is in the range $[age(u) - 5, age(u) + 5]$. For example, if u is 27-year old female, then U is the set of all females of ages between 22 and 32 (including 22 and 32). The function returns the mean of all the ratings that users in U have provided for the movie m. If U is empty, the function should return `None`.

5. Algorithm `genreRatingPrediction` uses the following signature:
   ```
   def genreRatingPrediction(u, m, movieList, rLu):
   ```
   This algorithm considers the set M of all movies who have the same genre as movie m. The function returns the mean of all the ratings that user u has provided for movies in the set M. If M is empty, the function should return `None`.

Evaluating the Prediction Algorithm

People who design recommendation algorithms also think a lot about how their algorithms should be evaluated in order to determine if these algorithms are making good predictions. One standard approach is called *cross-validation*. The main idea here is that we take a fraction of the rating data, say 20%, and call it our *testing set*. The remaining 80% of the rating data will form our *training set*. The idea is that we will then "train" our prediction algorithm on our training set and test it on our testing set. More specifically, we will "hide" our testing set and come up with predicted ratings based on our training set alone. We will then walk through our testing set, come up with a predicted rating for every item in the testing set and compare the predicted rating with the actual rating. Here are more details of this process. An item in the testing set will have the form (u, m, r), where u is a user, m is a movie, and r is the actual rating (or "ground truth" rating) that user

u has assigned to movie m. Suppose that we momentarily hide the rating r and use one of the rating prediction algorithms described above to come up with a predicted rating, say r', by user u for movie m. Note that the predicted rating r' is based on the training set alone. How well our algorithm does depends on how close the predicted rating r' is to the actual user rating, r. We will do this for the entire testing set and output a measure of how far our predicted ratings are compared to the actual ratings. Several different measures are used for this; one common measure is the *root mean squared error (RMSE)* defined as

$$\sqrt{\frac{\sum_i (r_i - r'_i)^2}{T}}.$$

Here r_i and r'_i are the actual and predicted rating of the i-th element in the testing set and T is the total number of elements in the testing set. Thus, RMSE is computed by first taking the mean of the squares of differences between actual and predicted ratings and then taking the square root of this quantity.

Here are two functions you are required to implement in order to implement the evaluation process described above.

- A function `partitionRatings` with signature
  ```
  def partitionRatings(rawRatings, testPercent):
  ```
  that partitions ratings into a training set and a testing set. This function takes a list of raw ratings in the form of (user, movie, rating)-tuples. In addition it takes the percentage of the raw ratings that should be placed in the testing set of ratings. For example, the percentage could be 20%. The testing set is obtained by *randomly* selecting the given percent of the raw ratings. The remaining unselected ratings are returned as the training set. The testing set is a list with each element having the form (user, movie, rating). The training set has a similar form. For example, if we want a 80-20 split of the `rawRatings` into a training set and a testing set, we would will call this function as
  ```
  [trainingSet, testSet] = partitionRatings(rawRatings, 20)
  ```

- A function `rmse` with signature:
  ```
  def rmse(actualRatings, predictedRatings):
  ```
  that computes the RMSE given lists of actual and predicted ratings. You can assume that these two lists of ratings have the same length and are non-empty.

How to run your experiments

You are required to submit two files: `project2Phase2a.py` and `project2Phase2b.py`. The first file, `project2Phase2a.py` should contain the implementation of all of the functions mentioned above, but no main program. Specifically, this file should contain the 7 new functions described in Phase 2, plus any helper functions that these functions depend on, plus all your functions from Project 2 Phase 1a. As usual, we will use an autograder to test all these functions.

The second file `project2Phase1b.py` should contain a main program as well. The purpose of this main program is to run experiments comparing the 5 prediction algorithms mentioned above using an 80-20 split of the ratings into training and testing sets. The specific steps your main program should implement are as follows:

1. Call the function `partitionRatings` to randomly partition the 100,000 ratings into a training set with 80,000 ratings and a test set with 20,000 ratings.

2. Call the function `createRatingsDataStructure` on just the training set of rating tuples (constructed in Step 1 above) to create two data structures. Let us call these `trainingRLu` and `trainingRLm` respectively. Note that `trainingRLu` and `trainingRLm` are just like `rLu` and `rLm`, but only for the training set of ratings.

3. Walk through the testing set and for each user u and movie m in the testing set, compute the 5 predicted ratings obtained by calling each of the 5 prediction algorithms described above. So at the end of this step, for each of the 20,000 (u, m) pairs in the testing set, we have 5 predicted ratings, one from each algorithm.
4. Compute the RMSE for each of the 5 algorithms, where each RMSE is computed over the 20,000 predicted and actual ratings.
5. Repeat Steps 1-4, 10 times and report results over the 10 repititions. By repeating this 10 times, and considering the 10 RMSEs, for each of the 5 algorithms, you are making sure that a particular 80-20 split did not bias the results against or in favor of any particular algorithm.

Producing visualizations

Using the above steps you have computed 10 RMSEs for each of the 5 algorithms. One way to compare the performance of the 5 algorithms would be to simply compare the RMSE means or medians. But, a better way to do this is to show a visual comparision by using a *box and whisker* plot. A *box and whisker* plot shows the minimum, maximum, the median, the first quartile, and the third quartile, etc., in one plot thus allowing us to better compare our algorithms. You will need to use `matplotlib` to produce these box and whisker plots. We will provide functions for you to do this.

What to submit

As mentioned earlier, you are required to submit two files: `project2Phase2a.py` and `project2Phase2b.py`. Details about what should go into each file have already been discussed. In addition, you should submit a pdf file called `Phase2Plots.pdf`, containing the box and whisker plot described above.

Phase 3

DON'T READ ANYTHING BEYOND THIS. I AM STILL ORGANIZING PHASE 3. In this phase you are required to implement an algorithm that predicts ratings of a given user i by taking into account ratings of users whose tastes are similar to i's tastes. The algorithm largely depends on the following definitions.

Definition of similarity. The *similarity* between two users i and j is defined as:

$$sim(i,j) = \frac{\sum_{m \in C}(r_{i,m} - r_i) \cdot (r_{j,m} - r_j)}{\sqrt{\sum_{m \in C}(r_{i,m} - r_i)^2} \cdot \sqrt{\sum_{m \in C}(r_{j,m} - r_j)^2}}.$$

Here C is the set of movies that *both* i and j have rated, $r_{i,m}$ is user i's rating of movie m, $r_{j,m}$ is user j's rating of movie m, and r_i is user i's mean rating and r_j is user j's mean rating. This definition guarantees that $sim(i,j)$ will always be between -1 and +1. Some of you may know this formula as the *Pearson correlation coefficient* and may also recognize the two terms that appear in the denominator as standard deviations. This definition of $sim(i,j)$ views the "similarity" of users i and j as a correlation between their ratings. If it turns out that user i and j have similar tastes and they have both rated common movies in a similar manner, then $sim(i,j)$ will be close to 1; on the other hand if their tastes are "opposite" then $sim(i,j)$ will be closer to -1.
Note that if C is empty then it means that we have no basis for figuring out the correlation between users i and j and in this case we assume that i and j are uncorrelated and set $sim(i,j)$ to be 0. Also, if the denominator in the above expression is 0, it means that the numerator will also be 0 (convince yourself of this) and in this case also we set $sim(i,j)$ to 0.

Predicting ratings via collaborative filtering. Once similarity between users is defined as above, we can predict the rating that a user i gives to a movie m by taking the "weighted" average of ratings that movie m has received from users who are similar to i. Specifically, for a user i and a movie m, define the *predicted rating* of movie m by user i as:

$$p(i,m) = r_i + \frac{\sum_{j \in U}(r_{j,m} - r_j) \cdot sim(i,j)}{\sum_{j \in U} |sim(i,j)|}. \tag{1}$$

Here U is the set of users that have rated movie m and are very similar to i. For example, let $N(i,k)$ be the k users that are most similar to i (using the similarity measure defined earlier). Think of $N(i,k)$ as user i's k best "friends," namely those k users whose tastes in movies is closest to i's tastes. Then, for an appropriately chosen positive constant k, U might be the subset of users in $N(i,k)$ that have rated movie m.

The following might help you gain some intuition into what the above formula is saying. The formula starts with r_i, which is user's i's mean movie rating. It then and increases this value if other users who are similar to i have rated m highly; otherwise, if other similar users have rated m poorly, r_i is decreased in order to obtain a predicted rating. Also note that the term in the formula corresponding to a user j is weighted by $sim(i,j)$ implying that the more similar j is to i, the more "weight" j's rating gets in the prediction.

Functions you need to define. You are required to implement the above definitions via the following functions.

- A function called `similarity` with the following function signature:
  ```
  def similarity(u, v, userRatings):
  ```
  that takes the IDs of two users, `u` and `v`, and the ratings list (containing a ratings-dictionary per user). This function computes the similarity in ratings between the two users, using the movies that the two users have commonly rated. It might help you understand the context for this function to note that we expect `userRatings` to be derived from the training set.

- A function called `kNearestNeighbors` with function signature:
  ```
  def kNearestNeighbors(u, userRatings, k):
  ```
  This function returns the list of (user ID, similarity)-pairs for the `k` users who are most similar to user `u`. The user `u` herself should be excluded from candidates being considered by this function. Ties can be broken arbitrarily. (For example, if $k = 1$ and there are two users `v` and `w` who are most similar to `u` and both have similarity 0.9, then it does not matter whether your function returns `[(v, 0.9)]` or `[(w, 0.9)]`).

- A function called `CFRatingPrediction` with the following function signature:
  ```
  def CFRatingPrediction(u, m, userRatings, friends):
  ```
  This function predicts a rating by user u for movie m. It uses the ratings of the list of `friends` (the 4th parameter) to come up with a rating by u of m according to formula (1). Typically the argument corresponding to friends would have been computed by a call to the `kNearestNeighbors` function. Here, as usual, `userRatings` is the list of movie ratings that contains one ratings-dictionary per user.

- A function called `CFMMRatingPrediction` with the following function signature:
  ```
  def CFMMRatingPrediction(u, m, userRatings, movieRatings, friends):
  ```
  This function is very similar to `CFRatingPrediction`. To come up with a rating, the function computes a number using the formula in (1) and then returns the average of this and mean rating of movie `m`.

Once the function `collaborativeFilteringRatingPrediction` has been implemented, you can perform experiments to compare the performance of collaborative filtering with the simpler prediction algorithms you implemented for Phase 1.

Experiments

To compare the algorithms implemented in the two phases, write a main program (similar to Phase 1 main program) that appropriately reads from files, sets up data structures, creates the testing and training data sets, and evaluates all of the prediction algorithms.

Note that the performance of the collaborative filtering algorithm may depend on the number of "friends" used. So I would like you to run the algorithms `CFRatingPrediction` and `CFMMRatingPrediction` with 0 friends, 25 friends, 300 friends, 500 friends, and the friends consisting of the entire population of users. This gives 5 variants of each of the collaborative-filtering-based algorithms for a total of 10 algorithms. Your main program should evaluate the 4 algorithms implemented in Phase 1 and the 10 algorithms implemented in Phase 2.

As before, the evaluation will simply be via rmse scores. To make sure that the reported rmse values are reliable, your program should perform 10 repetitions of the above process, by generating 10 different 80-20 splits of the data into training and testing sets, computing the rmse values of all of the prediction algorithms and then reporting the average rmse value of each prediction algorithm (averaged over the 10 repetitions).

The output from your main program should be informative, but not overly verbose. Note that your program will be reporting 14 numbers with simple accompanying messages.

What to turn in?

You are required to submit three program files: `project2Phase2a.py`, `project2Phase2b.py`, and `project2Phase2bOneRun.py`. The first file, `project2Phase2a.py` should contain the implementation of all of the functions mentioned above, but no main program. The graders will run this file through a bunch of tests (as in Project 1, we will provide our test file to you) in order to evaluate the functions. The second file `project2Phase2b.py` should contain a main program that performs the experiments described above. The third program file, `project2Phase2bOneRun.py`, is very similar to `project2Phase2b.py`, except that it contains code for just one run instead of 10 runs. Finally, you are required to submit a file called `output.txt`. This file is a simple text file containing the output produced by your program `project2Phase2b.py`.

Cautionary Note

Phase 2 is somewhat computationally intensive and so unless you are a bit thoughtful about your implementation, it is possible that it will take too much time to perform the experiments. Keep this in mind as your implementing Phase 2.

Extra Credit

As you may have noticed, we've only scratched the surface as far as building a good recommender system is concerned. The data set we are working with contains a lot of relevant information that we've ignored. We have also been somewhat simple-minded in a number of choices we've made. For example, algorithms `meanRatingPrediction` and `CFMMRating` compute the simple average of two different scores. It is possible that weighting this scores differently (e.g., using a formula such as $0.7 * x_1 + 0.3 * x_2$) might lead to better predictions. You'll receive up to 20 points extra credit if (i) you implement an algorithm that consistently improves the best rmse score from the algorithms described in this handout and (ii) briefly describe what you did and what output you got in 2-3 paragraphs. You'll need to submit two files: `project2EC.py` and `project2ECWriteUp.pdf` to receive any extra credit.