CS:1210 Project 2

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 popular websites such as Amazon, Netflix, Pandora, youTube, New York Times, etc. These recommendations are made possible by *recommender systems* that these websites run. Recommender systems contain a lot of interesting computer science under the hood and this project aims to give you a glimpse of this.

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. In this programming project, you are asked to build a simple recommender system for movies.

One, relatively new approach to coming up with recommendations is called *collaborative filtering*. In this approach, recommendations are made on the basis of "collaboration" among many users. Typically, such a system will collect numeric ratings (e.g., from 1 through 5 with 1 being worst and 5 being best) from users on a collection of items (e.g., movies). To determine what to recommend for a user "Alice," the recommender system looks at all users who have tastes similar to Alice's tastes and uses the items they have liked as a source of recommendations for Alice. How does the system figure out who has tastes similar to Alice? It simply uses past ratings to do this – those users who have rated items in a manner similar to Alice are considered to have tastes similar to Alice.

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/`). The dataset is made available with permission from Group Lens. 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 (Due on Wednesday, 4/29)

In the first stage of the project (due on Wednesday, 4/29) 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) make rating predictions using four very simple algorithms, and

(iii) evaluate the quality of the predictions.

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 header:

  `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 header:

  `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
```

  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 header:

  `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 header:

```
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 header:

  ```
  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.

After creating these data structures you are required to write two functions to answer simple queries about the data you've processed.

- The function `meanUserRating` (see header below) should return the mean rating provided by user with given ID `u`. The second argument is the ratings data structure from the user's point of view created by the function `createRatingsDataStructure`.

  ```
  def meanUserRating(u, rLu):
  ```

- The function `meanMovieRating` (see header below) should return the mean rating for a movie with given ID `im`. The second argument is the ratings data structure from the movie's point of view created by the function `createRatingsDataStructure`.

  ```
  def meanMovieRating(m, rLm):
  ```

Simple prediction algorithms

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 4 extremely simple prediction algorithms that you will implement for Phase 1 of the project. In Phase 2, you'll implement a 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 header 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. Use the following function header.
   ```
   def meanUserRatingPrediction(u, m, userRatings):
   ```
   Here `userRatings` the data structure of movie ratings organized by user. In other words, `userRatings` is a list with one element per user, each element being a dictionary containing all movie-rating pairs associated with that user. (See the description of `createDataStructure` for more details.)

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 header.
   ```
   def meanMovieRatingPrediction(u, m, movieRatings):
   ```
   Here `movieRatings` is the data structure of movie ratings organized by movies. In other words, `movieRatings` is a list with one element per movie, each element being a dictionary containing all user-rating pairs associated with that user. (See the description of `createDataStructure` for more details.)

4. Algorithm `meanRatingPrediction`: Given a user `u` and a movie `m`, simply return the average of the mean rating that `u` gives and mean rating that `m` receives.
   ```
   def meanRatingPrediction(u, m, userRatings, movieRatings):
   ```
   Here `userRatings` and `movieRatings` are as described above.

Evaluating the Prediction Algorithm

People who design recommendation algorithms also think a lot about how their algorithms should be evaluated. 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*. 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 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 header

  ```
  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. It is expected that the user will call this function as

  ```
  [trainingSet, testSet] = partitionRatings(rawRatings, testPercent)
  ```

- A function `rmse` with header:

  ```
  def rmse(actualRatings, predictedRatings):
  ```

  that computes the RMSE given lists of actual and predicted ratings.

Putting everything together

You are required to submit two files: `project2Phase1a.py` and `project2Phase1b.py`. The first file, `project2Phase1a.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 `project2Phase1b.py` should contain a main program as well. This file should contain a program that evaluates each of the 4 simple prediction algorithms mentioned above by using an 80-20 split of the ratings into training and testing sets. The output of this program should approximately look like:

```
Random prediction RMSE: 1.8688231591
User Mean Rating prediction RMSE: 1.04495612875
Movie Mean Rating prediction RMSE: 1.02292075141
User-Movie Mean Rating prediction RMSE: 0.981469630708
```

This is the actual output of one execution of my program. This is essentially saying that the random prediction is almost two "stars" off relative to the actual ratings, whereas the other algorithms are about one "star" off. Also noteworthy is that taking the average of the mean user rating and the mean movie rating seems to improve the prediction a little bit. In Phase 2, we will implement a more sophisticated algorithm and figure out whether this improves the RMSE in any significant manner.

Phase 2 (Due on Thursday, 5/7)

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:

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

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 header:
  ```
  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 header:
  ```
  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 header:
  ```
  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 header:
  `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.
