22C:16 (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 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*. 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 we 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 in the directory

`http://homepage.cs.uiowa.edu/ sriram/16/spring13/projects/project2/dataset/`

and consists of a `README` file along with 6 other text files. The data in all 6 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.

Stage 1 (Due on Monday, 4/29)

In the first stage of the project (due on Monday, 4/29) your program is expected to read from the given files store the information in appropriate data structures, and answer simple queries about the data. Specifically, it is required to create five lists – a user list, a movie list, two ratings list, and a genre list – 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 we 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 we 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.

- Write a function with the header:

  ```
  def createRatingsList(numUsers, numMovies):
  ```

  that reads from the file `u.data` and returns two lists containing all 100,000 ratings provided in `u.data`. The function takes as arguments the number of users and the number of movies in the data set. Suppose we call this function as `[rLu, rLm] = createRatingsList(numUsers, numMovies)`. 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 `ratingsList[0]`. In `rLm`, the ratings received by a movie appear as a dictionary whose keys are IDs of users who have rated that movie. Note that at this point we are ignoring the time stamp values provided in the `u.data` file.

- 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 a bunch of functions to answer simple queries. Below we specify the headers of the required functions along with documentation describing what the function is expected to do.

```
# returns the mean rating provided by user with id u. The second argument is
# the ratings list containing ratings per user.
def meanUserRating(u, userRatings):

# returns the mean rating received by a movie with id u. The second argument is
# the ratings list containing ratings per movie. Only movies that have received
# at least 10 ratings are include in the mean computation.
def meanMovieRating(u, movieRatings):

# Given a genre ID and a movie list, this function returns the list of all movie titles in the given
# genre, sorted in alphabetical order.
def moviesInGenre(genre, movieList):

# This function takes a genre ID, movie list and the ratings list containing ratings per movies.
# It returns a list of tuples containing movies titles and associated mean ratings. This list should
# only contain movie titles of the given genre that have received at least 20 ratings and should be
# sorted in decreasing order of mean rating received.
def popularMoviesInGenre(genre, movieList, movieRatings):

# This function takes a genre ID, movie list and the ratings list containing ratings per movies.
# It returns the mean rating for the given genre, which is the mean taken over all ratings of all
movies in the genre.
def meanGenreRating(genre, movieList, movieRatings):

# This function takes a genre list, movie list and the ratings list containing ratings per movies.
# It returns a sorted list of genres, sorted in decreasing order of mean rating.
def popularGenres(genreList, movieList, movieRatings):
```

Stage 2 (Due on Friday, 5/10)

In this stage of the project, you will be required to implement algorithms that predict users tastes in movies based on the available ratings data, demographic data, etc. You will also be required to evaluate the performance of the implemented algorithms.

The Collaborative Filtering Algorithm

The first algorithm you are required to implement 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.

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.
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:

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

Here U is the set of users that have rated movie m and are very similar to i. To define this more precisely, let $N(i, k)$ be the k users that are most similar to i. 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, U is 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.

You are required to implement the above definitions via the following functions.

```
# Given the IDs of two users, u and v, and 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.
def similarity(u, v, userRatings):

# Returns the list of (user ID, similarity)-pairs for the k users who are most similar
# to user u. The returned list of tuples is in decreasing order of similarity.
def kNearestNeighbors(u, userRatings, k):

# Predicts a rating by user u for movie m. If u has already rated m, then it simply uses
# that rating. Otherwise it uses the ratings of the list of friends (the 4th parameter) to come up
# with a rating by u of m. Here, as usual, userRatings is the list of movie ratings that
# contains one ratings-dictionary per user. Typically the argument corresponding to friends
# would have been computed by a call to the kNearestNeighbors function.
def predictedRating(u, m, userRatings, friends):
```

Once the function `predictedRating` has been implemented, it is easy to implement the following function that picks out the top few movie recommendations for a given user.

```
# Returns the k movies with highest predicted ratings by the given user u. The returned
# list is a list of (movie, rating)-tuples. The function uses the provided list
# of friends to compute the ratings. You should think of friends as something that would
# have been computed by the kNearestNeighbors function. In other words, friends is a list
# of (user ID, similarity) pairs.
def topKMovies(u, userRatings, k, friends):
```

Evaluating the Prediction Algorithm

So we can now make movie recommendations! But, how do we know if these are any good? People who design such prediction algorithms also think a lot about how these 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 test set, come up with a predicted rating for every item in the test set and compare the predicted rating with the actual rating. Here are more details of this process. An item in the test 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. Momentarily hide the rating r and use the function `predictedRating` 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.

Here are two functions you are required to implement in order to partition the rating data into a training set and a test set and then use the training set to construct rating predictions.

```
# This function reads from the file u.data and returns a trainingSet of
# ratings and a test set of ratings. The test set is obtained by randomly
# selecting 20,000 ratings. The remaining 80,000 ratings are returned as
# the training set. The test set is a list of size 20,000, each element
# having the form (user, movie, rating). The training set has a similar
# form, but it is a list of 80,000 elements. it is expected that the user
# will call this function as [trainingSet, testSet] = partitionRatings()
def partitionRatings():


# Takes the raw list of rating-triples and converts this into data
# structures containing the ratings, one from the point of view of users
# and one from the point of view of movies. In addition, the
# function takes the number of users and movies as parameters. It is expected
# that this function will be called with the trainingSet constructed from a
# call to partitionRatings(). This function no longer reads directly from the
# file u.data.
def createTrainingRatingsList(rawRatings, numUsers, numItems):
```

The function `createTrainingRatingList` is just like the function `createRatingList` from Stage 1, except that this function does not read from a file and builds the data structures from the raw list of ratings, `rawRatings`.

We use variants of two standard measures, namely *precision* and *recall*, to measure the performance of the prediction algorithm on the test set. In general, we would like our algorithm to predict high ratings for movies that were actually rated highly by users and we would like our algorithm not to predict high ratings for movies that were actually rated low by users. Precision and recall measure how well our algorithm does along these two measures. Consider elements in the test set with predicted rating at least 4. The fraction (percent) of these elements for which the actual rating is at least 3.5 is the *precision* of our prediction algorithm. Thus if the precision of the prediction algorithm is high, then it means that the algorithm does not usually assign high ratings to movies that users don't like. Consider elements in the test set with actual rating at least 4. The fraction (percent) of these elements for which our algorithm has a predicted rating of at least 3.5 is the *recall* of our prediction algorithm. Thus if the recall of the prediction algorithm is high, then it means that the algorithm does not usually assign low ratings to movies that users like. The numbers 4 and 3.5 that you see in these definitions are made up – there could be many reasonable alternative definitions, but this is what we will use in this project. Based on these definitions or precision and recall, implement the following function to evaluate the collaborative filtering prediction algorithm.

```
# This function takes a test set of ratings and the ratings list that was built from the
# training set and returns precision and recall. This function will be called as
# [precision, recall] = evaluateCF(testSet, rLu)
def evaluateCF(testSet, rLu, userList, occList):
```

Finally, write a main program that appropriately reads from files, sets up data structures, and evaluates the collaborative filtering algorithm. You should perform 10 repetitions of a process that does an 80-20 training-test split of the ratings data and computes precision and recall. Your main program should output the average precision and recall over these 10 repetitions.

As comparison, you should implement two other, simple, prediction algorithms.

1. Given a user u and a movie m, we could completely ignore both u and m and simply return a random integer in the range 1 through 5 as the the predicted rating.

2. Given a user u and a movie m, we could completely ignore the user and simply return movie m's mean rating as the predicted rating.

Repeat the above described evaluation process (i.e., computing average precision and recall) on the two simple prediction algorithm. The output from your main program should be informative, but not overly verbose. It should contain the average precision and recall numbers for the collaborative filtering prediction algorithm and also these numbers for the other two simple algorithms.

Demographic Similarity, Naive Bayes, and other Cool Stuff!

By the time you finish this project, you will hopefully have gotten a sense of how recommendation algorithms work and how they are evaluated. More importantly, we hope that you have gotten a sense that the methods we have implemented for this project have just scratched the surface. For example, we have not used any of the user demographic or movie genre information available in the data files. The title of this section alludes to other interesting additional methods you could implement to improve predictions, but we will not ask you to any of this because it is almost May and the weather outside is getting to be so nice! In any case, if these ideas interest you and you want to explore all of this more on your own, come by and see one of the instructors and we will give you a few pointers to get started.