Lab 6 - Recommender Systems

Kyle Swanson

January 17, 2018

0 Introduction

It turns out that building recommender systems is actually kind of hard, so this lab is going to be optional. The Naive Algorithm (Part 1) is not too difficult to implement, but it also doesn't provide very good recommender model. The Nearest-Neighbor Prediction method (Part 2) is a significant improvement, but is definitely a challenge to implement. The Low-Rank Matrix Factorization method (Part 3) is especially difficult, though it should theoretically produce the best result. I would recommend doing Part 1 just as a good coding exercise, but parts 2 and 3 are optional and are left as a challenge if you're interested.

0.1 Recommender Systems

In lecture we learned about several recommender systems. The goal of a recommender system is to learn to predict content that you might like based on features of the content and based on ratings that you and other users have provided for some of the content.

In this lab, we're going to be working with a set of movie ratings, and the goal will be to predict how users would rate movies which they have not yet seen.

0.2 Movie Data

The movie data in this lab comes from GroupLens (`https://grouplens.org/datasets/movielens/latest/`). For convenience, we're going to be working with their small dataset, which consists of 671 users and 9,066 movies with a total of 100,004 known ratings. These ratings are going to be split into two sets: a training set with 80,004 known ratings and a test set with 20,000 known ratings.

0.3 Data Format

Take a look in `main.py`. You'll see in the Data Loading section that all the movie data is loaded into a matrix Y. Each row of Y represents a user while each column represents a movie. Therefore, the entry Y_{ai} represents the rating given by the a^{th} user to the i^{th} movie. Ratings can be {`0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0`}. A rating of `-1.0` indicates that the user has not rated that movie.

0.4 Train and Test Splits

After loading the matrix `Y` with all the known ratings, it is split into two matrices, `Y_train` and `Y_test`. The `Y_train` matrix contains 80% of the known ratings with the other 20% replaced with `-1.0` while the `Y_test` matrix contains the other 20% of the known ratings with the training 80% replaced with `-1.0`. We will learn to predict `Y_train` and evaluate our resuts by comparing our predictions to the known ratings in `Y_test`.

0.5 Metrics

The metric we will use to evaluate our predictions is root mean squared error. If Y is a matrix with the correct ratings and X is a matrix with the predicted ratings, root mean squared error is defined as follows:

$$\text{rmse} = \sqrt{\sum_{ai \in D} (Y_{ai} - X_{ai})^2}$$

where D is the set of all user/movie pairs for which we know the rating that the user has given to the movie (i.e. Y_{ai} is not `-1.0`).

The root mean squared error is computed by calling the `root_mean_squared_error` function from `utils.py` and passing it the test ratings matrix `Y_test` and your matrix of predicted ratings `X`.

Since root mean squared error measures how far away your predictions are from the correct ratings, your goal is to minimize the root mean squared error.

1 Naive Recommendation Algorithm

First we will implement a naive algorithm in order to get a baseline root mean squared error against which we can compare our more advanced models.

1.1 Algorithm

The naive algorithm works as follows. (Note: n_u is the number of users and n_m is the number of movies.)

Algorithm 1 Naive Recommendation Algorithm

```
1: procedure NAIVE
2:     for $a = 1, 2, \ldots, n_u$ do
3:         for $i = 1, 2, \ldots, n_m$ do
4:             if $Y_{ai} \neq -1.0$ then
5:                 $X_{ai} = Y_{ai}$
6:             else if at least one user has rated movie $i$ then
7:                 $X_{ai} =$ average rating of movie $i$
8:             else
9:                 $X_{ai} =$ average rating of all movies
10:    return $\theta, \theta_0$
```

In words, the algorithm works as follows. If we know the rating user a gives to movie i, then we just predict that rating (and we know we'll be correct). If we don't know the rating, then we'll simply use the average rating that other users give to movie i. If no users have rated the movie, then we just give the movie the average rating across all movies.

1.2 Implementation

Your task is to implement the function `predict_ratings_naive` in `lab6.py`, which takes in a matrix `Y` with the known training ratings and outputs a matrix `X` with the predicted ratings for all users and movies.

Once your implementation is complete, uncomment Part 1 in `main.py` and run the code. You should see a test root mean squared error (test rmse) of around 1.0.

2 Nearest-Neighbor Prediction (Challenge)

The naive algorithm only makes use of the known predictions and the average rating of the movies without using any qualities of the user. Therefore, such predictions will not be personalized and will in fact work very poorly for users who are not like the average user.

In order to personalize predictions, we will use a more advanced method called nearest-neighbor prediction. In order to make movie rating predictions for user a, the nearest-neighbor method will try to find other users who have similar preferences to user a (i.e. the other user generally likes the same movies and dislikes the same movies as user a), and it will use the ratings of these similar (neighbor) users to predict the rating that user a would give for a movie.

Below I will briefly describe the algorithm. If you'd like more details, you can read section 2.3.1 "User-Based Neighborhood Models" in this textbook: `http://www.springer.com/cda/content/document/cda_downloaddocument/9783319296579-c1.pdf?SGWID=0-0-45-1554478-p179516130`

2.1 Algorithm

The core steps of the nearest-neighbor algorithm are as follows:

1. Compute the similarity between users based on how users rate the same movies.
2. For each user a and movie i:
    (a) Look at the other users who have rated i and select the k users who are most similar to user a.
    (b) Use the ratings of those k users to predict the rating that user a would give to movie i.

Now we will fill in the details.

2.1.1 Similarity

The similarity between users a and b is defined as the correlation between between the ratings of user a and the ratings of user b for movies that both a and b have rated.

$$\mathrm{sim}(a,b) = \mathrm{corr}(a,b) = \frac{\sum_{j\in CR(a,b)}(Y_{aj}-\widetilde{Y}_a)(Y_{bj}-\widetilde{Y}_b)}{\sqrt{\sum_{j\in CR(a,b)}(Y_{aj}-\widetilde{Y}_a)^2}\sqrt{\sum_{j\in CR(a,b)}(Y_{bj}-\widetilde{Y}_b)^2}}$$

The notation is as follows:

- Y_{aj} is the rating user a gave to movie i
- Y_{bj} is the rating user b gave to movie j
- $CR(a,b)$ is the set of all movies that both a and b have rated
- $\widetilde{Y}_a = \frac{1}{|CR(a,b)|}\sum_{j\in CR(a,b)} Y_{aj}$ is the average rating user a gave to movies rated by both a and b
- $\widetilde{Y}_b = \frac{1}{|CR(a,b)|}\sum_{j\in CR(a,b)} Y_{bj}$ is the average rating user b gave to movies rated by both a and b

Note that $\mathrm{sim}(a,b) \in [-1,1]$, with a similarity closer to -1 meaning that the users are dissimilar (user a likes movies that user b dislikes and vice versa) while a similarity closer to 1 means that users are similar (users a and b generally like and dislike the same movies).

2.1.2 Predicting Ratings

To predict X_{ai} (the rating that user a would give to movie i), our first step is to find all users who have rated movie i. We then look at the similarity between user a and each of these users, and we select the k users who are most similar to a. Let $KNN(a,i)$ be the top k most similar users to a who have rated movie i.

Once we have the top k most similar users, we can predict the rating that user a would give to movie i:

$$X_{ai} = \overline{Y}_a + \frac{\sum_{b \in KNN(a,i)} \text{sim}(a,b)(Y_{bi} - \overline{Y}_b)}{\sum_{b \in KNN(a,i)} |\text{sim}(a,b)|}$$

Note that now we are using $\overline{Y}_a$ and $\overline{Y}_b$ rather than $\widetilde{Y}_a$ and $\widetilde{Y}_b$. Earlier we defined $\widetilde{Y}_a$ to be the average rating that user a gave to movies rated by *both* a and b. Here we are using $\overline{Y}_a$, which is defined to be the average rating given by user a to *all* movies that user a has rated ($\overline{Y}_b$ is defined similarly for user b).

The prediction for X_{ai} works by looking at the similar users' ratings for movie i and determining how much those ratings differ from those users' average ratings ($Y_{bi} - \overline{Y}_b$). This is an indication of whether movie i is better or worse than average according to the similar users. Then, since every user has a different idea of what "average" means on a $0-5$ scale, we take user a's idea of average ($\overline{Y}_a$) and we add to it the amount that movie i deviates from average according to the similar users (with the deviations weighted in importance by how similar the other users are).

2.1.3 Algorithm statement

In the algorithm below, S is a matrix such that S_{ab} is the similarity between users a and b. Note that since similarity is symmetric, $S_{ab} = S_{ba}$. As before, n_u is the number of users and n_m is the number of movies.

Algorithm 2 Nearest-Neighbor Algorithm

```
procedure NEAREST-NEIGHBOR
    $S = 0$ ($n_u \times n_u$ matrix)
    $X = 0$ ($n_u \times n_m$ matrix)
    for $a = 1, 2, \ldots, n_u$ do
        for $b = a, a+1, \ldots, n_u$ do
            $S_{ab} = S_{ba} = \frac{\sum_{j \in CR(a,b)} (Y_{aj} - \widetilde{Y}_a)(Y_{bj} - \widetilde{Y}_b)}{\sqrt{\sum_{j \in CR(a,b)} (Y_{aj} - \widetilde{Y}_a)^2} \sqrt{\sum_{j \in CR(a,b)} (Y_{bj} - \widetilde{Y}_b)^2}}$
    for $a = 1, 2, \ldots, n_u$ do
        for $i = 1, 2, \ldots, n_m$ do
            $KNN(a, i)$ = top $k$ users most similar to user $a$ (i.e. largest $S_{ab}$) who have rated movie $i$
            $X_{ai} = \overline{Y}_a + \frac{\sum_{b \in KNN(a,i)} \text{sim}(a,b)(Y_{bi} - \overline{Y}_b)}{\sum_{b \in KNN(a,i)} |\text{sim}(a,b)|}$
    return $X$
```

2.2 Implementation

Implement the function `predict_ratings_nearest_neighbor` in `lab6.py`. You may find it helpful to define other functions to help implement the algorithm.

Once your implementation is complete, uncomment Part 2 in `main.py` and run the code. You should see a test root mean squared error (test rmse) of around 0.93. This is a significant improvement over the rmse of 1.0 from the naive algorithm (remember: smaller rmse is better), indicating that this algorithm is making better predictions.

3 Low-Rank Matrix Factorization (Challenge x2)

Read the original paper describing the low-rank matrix factorization algorithm and implement their solution: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.173.2797&rep=rep1&type=pdf`

I couldn't get it to work very well, but maybe you can. Good luck!