Building a Simple Recommender System

Using Collaborative Filtering

Wanjia Tang, June 8, 2017

Motivation

Algorithms of major technology companies control our lives now without us even paying attention to it. Eli Pariser talked in his TED speak about the “filter bubbles” that are created by search engines such as Google that try to convince people to stay on their platform by only showing results that the users like to see. The thing is, we as users don’t normally know how exactly Google or Bing respond to our queries, or how Amazon or Instagram recommend us the things we are recommended, choosing which search engines or which websites to use is like choosing whose little black box to trust, and we do it based on intuition or just blind faith. This project I did is a little peek into one simple idea in complicated recommender systems. Although for most companies, algorithms of how they do recommendations are trade secrets, Amazon did publish a paper in 2003 to give a general idea of how things work.

Methodology

Collaborative filtering is a method of predicting interests of a user by collecting and comparing interests and preferences of other people that are similar to that user. Collaborative filtering needs a rather large dataset to begin with, it assumes that people who agreed in the past are likely to agree in the future, and by analyzing the past behaviors in the dataset can we make predictions about user’s attitude towards unrated items.

In this project, I will try to implement a collaborative filtering system to recommend movies to users. I will first use a user to user collaborative filtering system and then use the same idea to

implement an item to item filtering system and see how the results would differ. In theory, the item to item approach would be much better than the user to user approach, because users can be fickle but items stay reasonably the same and they can be categorized more easily.

Dataset

The dataset I have is from the website GroupLens, it consists of 5 csv files, among them are 2 that are used in this project: one contains all the users (represented by their user IDs) and their ratings to movies (represented by movie IDs) they have watched, and another one contains mappings from movie IDs to movie names. The website updates the dataset over time, the version I have is a 1m dataset recommended for education and development, it was last updated in October 2016.

Steps in Detail

The following steps explained will be building towards a user to user collaborative filtering system. Because user to user and item to item methods use the same methodology in essence, the functions in the code I wrote can be used in both methods.

Step 1: Define the Database Structure

The dataset I have that contains users and their ratings to movies has four columns: userId, movieId, rating, and timestamp. I will use the first three and write them in Python as a dictionary of dictionaries that looks like this:

{

user1: {movie1: 3.5, movie2: 4, movie3: 5}

user2: {movie1: 5, movie2: 3.5, movie4: 5, movie6: 2}

User3: {movie1: 4, movie7: 4.5, movie9: 2}

}

This can be seen as a pivot table with user IDs as row names and movie IDs as column names.

I used pandas to merge the two csv files I have and present the pivot table in Jupyter notebook, it looked like this:

In [19]:
```
rows = ['userId', 'movieId', 'rating']
ratings01 = pd.read_csv('ratings.csv', usecols=range(3))
ratings02 = pd.read_csv('movies.csv', usecols=range(2))
ratings = pd.merge(ratings01, ratings02)
ratings.sort_values('movieId')
ratings.head(10)
```

Out[19]:

	userId	movieId	rating	title
0	1	31	2.5	Dangerous Minds (1995)
1	7	31	3.0	Dangerous Minds (1995)
2	31	31	4.0	Dangerous Minds (1995)
3	32	31	4.0	Dangerous Minds (1995)
4	36	31	3.0	Dangerous Minds (1995)
5	39	31	3.0	Dangerous Minds (1995)
6	73	31	3.5	Dangerous Minds (1995)
7	88	31	3.0	Dangerous Minds (1995)
8	96	31	2.5	Dangerous Minds (1995)
9	110	31	4.0	Dangerous Minds (1995)

```
user_movie = ratings.pivot_table(index='userId', columns='title', values='rating')
user_movie.head()
```

title	"Great Performances" Cats (1998)	$9.99 (2008)	'Hellboy': The Seeds of Creation (2004)	'Neath the Arizona Skies (1934)	'Round Midnight (1986)	'Salem's Lot (2004)	'Til There Was You (1997)	'burbs, The (1989)	'night Mother (1986)	(500) Days of Summer (2009)	...	Zulu (1964)	Zulu (2013)	[REC] (2007)	eXistenZ (1999)	loudQUI A Film A the Pixi (2006)
userId																
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN

5 rows × 9064 columns

I will only use Jupyter notebook to present the table, and use python to do the actual calculation because it's easier to access single ratings in python.

Step 2: Normalize the Ratings

As is seen from the table above, because there are over 9000 movies and most users can only rate around a hundred, for a particular user, movies not rated are far more than movies rated, and movies that are without ratings are presented as NaNs in the table. We have two ways of dealing with these NaNs, one is to simply give them values of 0, and do the rest of the calculation, but this would heavily skew the result, because the absence of a rating is not the same as rated badly, giving them ratings of 0 would, in this system, mean that they watched the movie and hated it. The second approach is to normalize the rating around 0 first and then fill the NaNs with 0. The way to do it is to calculate the average ratings of a particular user and subtract from the original rating the average, and the result is normalized rating from that user to the movie. Below is an example of how normalizing works:

The original rating:

	movie2	movie2	movie3	movie4	movie5	movie6	movie7	
								Mean
user1	5	NaN	4	NaN	2	3	NaN	3.5
user2	NaN	3.5	5	3	NaN	NaN	4.5	4.0
user3	4	5	NaN	4.5	1	3	2	3.25
user4	1	5	3	2	4	NaN	3	3.0

Subtract the mean of each row from original ratings and fill all the NaNs with 0:

	movie2	movie2	movie3	movie4	movie5	movie6	movie7
user1	1.5	0.0	0.5	0.0	-1.5	-0.5	0.0
user2	0.0	-0.5	1.0	-1.0	0.0	0.0	0.5
user3	0.75	1.75	0.0	1.25	-2.25	-0.25	-1.25
user4	-2.0	2.0	0.0	-1.0	1.0	0.0	0.0

For the table above, the sum of every row is 0, intuitively, positive ratings mean that the person likes that movie more than average, negative ratings mean the user likes the movie less than average.

Step 3: Calculate Similarity

After normalizing the results, the next step is to find a way to calculate the similarity scores between any two users, so that we can determine a group of users that are most similar to the user we want to make recommendations to. I used cosine similarity to calculate the relationship between two users. By looking at each user as a vector, we can use:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|_2 \|\mathbf{B}\|_2} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

to calculate the similarity scores.

Because numerator is the sum of the multiplications of ratings made by both users, in python, these shared movies can be identified and represented easily using set operations. The code looks like this:

```python
def similarity(objectA, objectB, pivot_table):
    '''using cosine similarity to calculate the similarity bewteen userA and userB or movieA and movieB.
    objectA, objectB: a string;
    return: a floating point number: the similarity between objectA and objectB.'''
    objectA_dict = normalize(pivot_table[objectA])
    objectB_dict = normalize(pivot_table[objectB])
    intersection = set(objectA_dict.keys()) & set(objectB_dict.keys())
    numerator = 0
    for item in intersection:
        numerator += objectA_dict[item] * objectB_dict[item]
    A_squared_sum = 0
    B_squared_sum = 0
    for item in objectA_dict.keys():
        A_squared_sum += (objectA_dict[item]) ** 2
    for item in objectB_dict.keys():
        B_squared_sum += (objectB_dict[item]) ** 2
    denominator = (A_squared_sum ** 0.5) * (B_squared_sum ** 0.5)
    similarity = numerator / denominator
    return similarity
```

Now that we can calculate any given two users' similarities, for a particular user, we can use Python to calculate the similarities between this user and all the other users, and take the likes and dislikes of the most similar users into account when we want to make recommendations for the given user, this is also called finding the neighborhood of the user.

The similarity scores between users might mean nothing intuitively, so I used the same theory and the same functions in Python to calculate the similarity between movies. I used the movie "Star Wars: Episode IV - A New Hope (1977)" as an example and found the top ten movies that are most similar to it, and the results look pretty reasonable.

The raw python results I got is a list of sets, first item of the set is the movie ID, and the second item in the set is the similarity score. The first one on the list is the Star Wars movie itself, so it has the top similarity score of 1.0.

```
april at D-10-157-150-131 in ~/GoogleDrive/CurrentClasses/ComDataScience/project/ml-lat
est-small on master [!?]
$ python recommendation.py
[('260', 1.0000000000000004), ('1210', 0.53060549785652011), ('1196', 0.46740960819923416), ('119
8', 0.27262086183139822), ('5952', 0.2268161929296299), ('1097', 0.22297935738149907), ('4993', 0
.21696344047312111), ('58559', 0.19892431762425494), ('34', 0.1954955589823997), ('8961', 0.19425
622372678894)]
```

I cross compared the movie IDs with movie names in the other data file I have and got the following list:

Movie ID	Movie Name	Similarity Score
260	Star Wars: Episode IV - A New Hope (1977)	1.000
1210	Star Wars: Episode VI - Return of the Jedi (1983)	0.530
1196	Star Wars: Episode V - The Empire Strikes Back (1980)	0.468
1198	Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981)	0.273
5952	Lord of the Rings: The Two Towers, The (2002)	0.227
1097	E.T. the Extra-Terrestrial (1982)	0.223
4993	| Lord of the Rings: The Fellowship of the Ring, The (2001)	0.217
58559	Dark Knight, The (2008)	0.199
34	Babe (1995)	0.195
8961	Incredibles, The (2004)	0.194

Step 4: Find User X's Neighborhood

For the user x and item i that user x has not yet rated, we want to find out the estimated rating for item i from user x by looking at the ratings given by a set N of n users that are most similar to user x, this set of n users are called the neighborhood of user x.

By the end of last step, we already have a list of users that are most similar to user x, now we need to find from this list the people who also rated item i and rank them from most similar to least similar. We can use this new list as our set N to calculate the estimated rating of user x to item i. In this project, I made n = 10 because I only want to consider of the influences of users that are most similar to user x, and also because this is a large dataset with complicated calculation, it saves some time if n is small. To get rid of users who only rated a handful of movies that falls into the subset of movies watched by user x, I also made set N to only contain users that have rated at least 100 movies.

Step 5: Calculate estimated ratings of unwatched movies

There are also two methods to do the calculation. One simply takes the average of all the ratings of item i in set N and have it be the estimated rating of user x. This approach treats every user equally. The second approach uses weighted similarity, it takes into account of the different similarity scores between different users and user x, and reflect their contribution in the estimation by how close they are with user x. This project uses the second approach.

```
97  def userToUser(user_pivot_table, movie_pivot_table, user, n):
98      '''user_pivot_table'''
99      recommend_list = []
100     all_movies = getKeys(movie_pivot_table)
101     unrated_movies = all_movies - set(user_pivot_table[user].keys())
102     for movie in unrated_movies:
103         setN = topNMostSimilar(user_pivot_table, user, movie, n)
104         numer = 0
105         denom = 0
106         for pair in setN:
107             person = pair[0]
108             sim = pair[1]
109             if not math.isnan(sim):
110                 numer += user_pivot_table[person][movie] * sim
111                 denom += sim
112         try:
113             estimated_rating = numer / denom
114         except ZeroDivisionError:
115             pass
116         if not math.isnan(estimated_rating):
117             recommend_list.append((movie, estimated_rating))
118     sorted_list = sorted(recommend_list, key=operator.itemgetter(1), reverse=True)
119     return sorted_list
```

Step 6: Make recommendations

From last step, we know how to calculate an estimated rating of one movie from user x. To make recommendations for user x, we need to use the same method to calculate all the unwatched movies, sort them by estimated ratings and recommend the top ones on the list. To test the program, I added a fake user “April” at the end of the list and have her rate some movies.

userId	movieId	movieName	rating
April	260	Star Wars: Episode IV - A New Hope (1977)	4.5
April	1196	Star Wars: Episode V - The Empire Strikes Back (1980)	5
April	8638	Before Sunset (2004)	5
April	215	Before Sunrise (1995)	4.5
April	59315	Iron Man (2008)	3
April	112552	Whiplash (2014)	5
April	119145	Kingsman: The Secret Service (2015)	2
April	102800	Frances Ha (2012)	5
April	87232	X-Men: First Class (2011)	4
April	112582	Life Itself (2014)	5
April	87234	Submarine (2010)	5
April	2571	Matrix, The (1999)	4.5
April	97921	Silver Linings Playbook (2012)	4.5
April	53123	Once (2006)	4
April	63992	Twilight (2008)	1
April	72407	Twilight Saga: New Moon, The (2009)	1
April	78772	Twilight Saga: Eclipse, The (2010)	1
April	91500	The Hunger Games (2012)	3
April	106487	The Hunger Games: Catching Fire (2013)	2
April	116823	The Hunger Games: Mockingjay - Part 1 (2014)	2
April	135133	The Hunger Games: Mockingjay - Part 2 (2015)	2.5

After running the program, this recommender system recommended user April with the following movies:

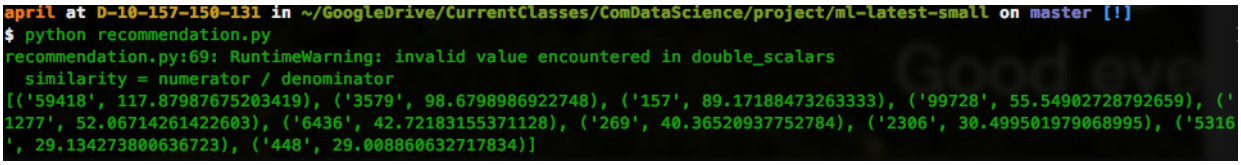
```
april at D-10-157-150-131 in ~/GoogleDrive/CurrentClasses/ComDataScience/project/ml-latest-small on master [!]
$ python recommendation.py
recommendation.py:114: RuntimeWarning: invalid value encountered in double_scalars
  estimated_rating = numer / denom
[('8656', 110.25361082819803), ('3091', 89.243486871441405), ('34534', 86.55640568627561), ('4688', 70.335793986669032)
, ('117192', 68.698940334990553), ('700', 64.208189236710297), ('6793', 47.989032732502224), ('696', 47.903628006424569
), ('27700', 34.350284735876734), ('488', 24.226329845013733)]
```

Movie ID	**Movie Name**	**Estimated Rating**
8656	Short Film About Killing, A (1988)	110.25
3091	Kagemusha (1980)	89.24
3934	Four Brothers (2005)	86.56
34534	Black Robe (1991)	70.34
4688	Doctor Who: The Time of the Doctor (2013)	68.70

117192	Angus (1995)	64.21
700	Beethoven (1992)	47.99
6793	Bent (1997)	47.90
696	Evil (Ondskan) (2003)	34.35
27700	M. Butterfly (1993)	24.23

I also used the same theory and implemented an item to item collaborative filtering system,

and the result is:



Movie ID	**Movie Name**	**Estimated Rating**
59418	American Crime, An (2007)	117.88
3579	I Dreamed of Africa (2000)	98.68
157	Canadian Bacon (1995)	89.17
99728	Gangster Squad (2013)	55.55
1277	Cyrano de Bergerac (1990)	52.07
6436	This Boy's Life (1993)	42.72
269	My Crazy Life (Mi vida loca) (1993)	40.36
2306	Holy Man (1998)	30.50
5316	Enigma (2001)	29.13
448	Fearless (1993)	29.01

Limitations and Challenges

Cold Start

Because collaborative filtering systems are intrinsically using the past to predict the future, the more data about the past user behaviors we have, the better our predictions will be. It would be hard to implement this recommender system without a large set a data, and similarly it would also be hard to make recommendations to a new user.

Data Sparsity

As we can see from the user-movie matrix in the second step, because the movies we have in the dataset is far more than any single person could rate, the matrix is very sparse. In the real world, this problem could only be worse. When there are two users that have no shared rating history, because of the lack of numerator, their similarity is NaN. Although in the fourth step, we tried to filter out the NaNs by only taking the top most similar users, it could happen than within the top ten most similar users, there still are people that no similarity with said user. Having NaNs as similarity score would eventually be a real problem in step five, so in the final function, I added an extra step to check if similarity score is NaN, and get rid of the ones that are. This does not skew the final result, as their presence cannot contribute to the calculation anyway.

Scalability

The dataset is represented as a dictionary of dictionaries, step five even involves looping through all the movies and calculate a set N of users that have watched this movie, the program would have a polynomial complexity at least. As the dataset grows, the runtime would increase considerably. For big companies, this would not only mean a lot of resources and money, but it would cause them to lose clients due to prolonged waiting time. One way to improve on this is to use tools that are fit to process big data, like apache Spark, or to improve the algorithm itself, and this has been done by scientists in most major tech companies.

Success is Hard to Measure

It is hard to know how good the recommendations are until the users watched all the recommended items and give feedback. It takes time to get the feedback, and as we said before, people are fickle and sometimes it could be difficult to pinpoint where things went wrong.

Problems I had

Once I understand the theory and the flow of steps, the code was not difficult to write. However, I did learn more about Python through debugging, and this process took me much longer than writing the code itself.

One of the major problems I had was how to get rid of the NaNs, and more importantly, in which step should I get rid of them. At first I wanted to keep them until the very end because I

was afraid the lack of data would skew the result. My original thinking was, in the last step, I would loop through all the unwatched film by user x, calculate the estimated rating for each of them, sort them by the ratings into a list, and take the first 10 as recommendations to present. In theory it should work fine, but the sorting method I used did not work in the presence of NaNs. I wasn't able to locate the problem and was perplexed about kept getting random 10 movies, it was not until I printed out the whole list several times and confirmed they are the same did I realize the list was not sorted and that it was in random order. So I added a line to append only the non-NaNs to the list and finally got the right result. However, the last step was not the only step that involved sorting, so I went back to the earlier step of sorting users by similarity scores and picked out the NaNs too to make sure the list returned was properly sorted.

This bug was not easy to find because although in the final step, failing to sort brought a noticeable difference in the result returned, not sorting in the middle steps was not as significant.

Appendix

Code and dataset file:

https://drive.google.com/drive/folders/0B391C-zqKVB3MEwxQ0pFU1o1Tjg?usp=sharing

Algorithmic sources:

https://www.cs.umd.edu/~samir/498/Amazon-Recommendations.pdf

https://www.youtube.com/watch?v=h9gpufJFF-0&t=395s