TAGLIATELA COLLEGE OF ENGINEERING

DSCI-6007
Project Report

COLLABORATIVE FILTERING USING THE NETFLIX DATA

BY:

Yesha Shah

EXECUTIVE SUMMARY

In this day and age, where a person gets to choose from so many things to watch, hear, or do, it could sometimes become an extremely difficult task on deciding what to choose. It could be challenging to find something that one is guaranteed to like, irrespective of going through the reviews. Thus, automated tools can come in very handy. Using various algorithms and user data, one can build machine learning models which can recommend the users what to watch. This project is in collaboration of recommending movies to users based on past Netflix data (subset).

For the given data, we as students, are expected to build a model that can predict the ratings of movies based on previous data. The predictions were evaluated against the true ratings using two metrics, root mean squared error (RMSE) and mean absolute error (MAE). I addition to that, we were to build and model a recommender system which looks at the user's current data, i.e. the ratings given by the user to things they have already seen and then, suggest movies to the user accordingly. This recommendation system is fulfilled using the approach of collaborative filtering. This means that along with item attributes, a user's behavior is also taken into consideration for suggesting movies. In my approach, when I say user behavior, my model refers to the movies that the particular user likes the most, i.e. their highly rated movies.

Collaborative filtering uses the information collected by the system about the interactions of a user with different items. In our case, this information is the rating a particular user has provided to a particular item/movie. Collaborative filtering can be implemented either using a user-based approach or an item-based approach. The user based recommends movies based on similar users as the current user, whereas item-based recommends movies based on similar items (here, movies) between all items and the target users previously watched/rated movies. This project is fully implemented in jupyter notebook, on an AWS EMR cluster with the use of spark. The project report is a brief guide and explanation of the steps I took to design and model the recommender system for the Netflix data.

SYSTEM CONFIGURATION

To be able to use distributed computing with spark and jupyter, the system first needs to be set up. Below is a step by step guide to do the same.

1. Create and configure a new EMR cluster:
   a. go to EMR
   b. Click on create cluster
   c. At quick options page:
      - Give a name to the cluster
      - For software configuration, select spark as application
      - For hardware config, I have used m4.xlarge with 3 instances in total
      - Select an EC2 key pair
      - Leave everything else as default
      - Click on create cluster
      - Wait for cluster to be up
2. Add the port 22 to the inbound rules of the security group corresponding to the master node through EC2 as anywhere IPv4 – to be able to SSH
3. SSH into the master node
4. Switch to super user mode
   ```
   $ sudo su
   ```
5. Install the required packages as
   ```
   $ pip install pyyaml ipython jupyter ipyparallel pandas boto -U
   ```
6. Leave super user mode
   ```
   $ exit
   ```
7. Run the following to modify the .bashrc file and instruct to run pyspark on jupyter
   ```
   $ export PYSPARK_DRIVER_PYTHON=/usr/local/bin/jupyter
   $ export PYSPARK_DRIVER_PYTHON_OPTS="notebook --no-browser  --ip=0.0.0.0 --port=8888"
   ```
8. Refresh the .bashrc file
   ```
   $ source ~/.bashrc
   ```
9. Start pyspark
   ```
   $ pyspark
   ```
10. Add the port 8888 to the inbound rules of the security group corresponding to the master node through EC2 as anywhere IPv4 – to be able to access jupyter
11. From the SSH terminal, user the provided URL (along with token) and replace localhost with public IP or DNS to open the jupyter notebook in any browser you like

Problem 2 – Analyzing the Netflix data

The goal of this step was to analyze the provided data so as to figure out what would be the right approach to for this particular data. To avoid any overheads, I have done the analysis on a smaller dataset, which would represent the whole data. This smaller data is simply a random subset, containing about 20% of the original data (train and test). Before I began, I made sure that the datasets (TrainingRatings.txt, TestingRatings.txt, and movie_titles.txt) were uploaded to a bucket in S3 which I can read in my code.

1. Read the data from S3 and store into cache

```
training_data = sqlContext.read.format('csv').options(header=False, inferSchema=False).schema(df_schema).load(s3PathTrainData)
testing_data = sqlContext.read.format('csv').options(header=False, inferSchema=False).schema(df_schema).load(s3PathTestData)

training_data.cache()
testing_data.cache()

print('Total train data = ', training_data.count())
print('Total test data = ', testing_data.count())
```

```
Total train data =  3255352
Total test data =  100478
```

2. Creating a sample:

   Randomly selecting 20% of the data to create sample datasets

```
from pyspark.sql.functions import rand

sample_train = training_data.orderBy(rand(seed=5)).limit(651071)
sample_test = testing_data.orderBy(rand(seed=5)).limit(20096)
```

3. Problem 2a)

   Number of distinct items and users in train set

```
distinct_users = training_data.select(countDistinct('userID'))
distinct_items = training_data.select(countDistinct('movieID'))
```

```
Distinct users in training set =  28978
Distinct items/movies in training set =  1821
```

4. Problem 2b)

   Calculating estimated average overlap of items for users:

   Randomly select 10 users from test set

   For each user in above list:

   Fetch all items rated by user

   Count number of users that rated same movies

   Average over the above count for each item

   Average over the item overlap average for each user

Calculating estimated average overlap of users for items:

Randomly select 10 items from test set

For each item in above list:

Fetch all users that rated current item

Count number of items that were rated by same user

Average over the above count for each user

Average over the user overlap average for each item

```
estimated average overlap of items for users =  2051.319
estimated average overlap of users for items =  48.74359
```

5. Problem 2c)

From 2a:

- Distinct items/movies in test set = 1701
- Distinct users in test set = 27555

- Distinct items/movies in training set = 1821
- Distinct users in training set = 28978

Since number of items << number of users,
it would be expected that the overlap of items >> overlap of users.
This is because the SAME items could have been rated by MANY users
however, the SAME users might or might not have rated the SAME items

From 2b:

- estimated average overlap of items for users ~ 2000
- estimated average overlap of users for items ~ 40

As expected, the overlap of items is much higher than the overlap of users

From the calculations, **higher overlap of items** indicates the data has many **similar users**. Thus, the predictions would likely be way more accurate if we were to use the **user-user** model.

However, due to system limitations, (unable to pivot for such a huge number of users), I decided to go ahead with item-item model. It is important to note that user-based model is basically the same implementation but replacing item with user and vice-versa in all the code.

6. Pearson correlation (item-based model)
   Group data by userID and pivot by movieID – while aggregating over each rating
   Created a rating vector – to be used for building correlation matrix
   Build the correlation matrix using Pearson method
   The final correlation matrix looks as follows (shown as a dataframe for better viz)

movieID	8	28	43	48	61	64	66	92	96	111	...	17654	17660	17689	17693
8	1.000000	-0.042146	0.005975	-0.014675	-0.000750	0.012873	0.012119	-0.000779	0.022855	-0.023613	...	-0.033358	0.036393	0.007312	-0.030664
28	-0.042146	1.000000	0.014497	0.110906	-0.009361	0.001573	0.004635	0.003757	0.009357	0.123198	...	0.101463	0.004686	0.003791	0.157262
43	0.005975	0.014497	1.000000	0.037607	0.025534	0.050806	0.046491	0.005333	0.028172	-0.008141	...	0.015749	0.005587	0.030161	0.013084
48	-0.014675	0.110906	0.037607	1.000000	0.022967	0.028974	0.050921	0.052250	0.031889	-0.000721	...	0.034976	0.012718	0.030831	0.078753
61	-0.000750	-0.009361	0.025534	0.022967	1.000000	0.051848	0.086510	0.079832	0.037186	0.000258	...	0.005865	0.044564	0.116404	0.023404
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
17725	0.025023	-0.025760	0.009742	0.030731	0.024588	0.008251	0.013599	0.023541	0.030360	0.016149	...	0.006480	0.011732	0.026723	0.009913
17728	0.043411	-0.015913	0.025317	0.001624	0.028530	0.036705	0.028801	0.008176	0.053833	-0.011111	...	-0.001303	0.012518	0.059656	-0.003611
17734	0.026431	0.002747	0.016258	0.015319	0.042600	0.075105	0.078206	0.066338	0.051135	0.002920	...	-0.005808	0.085926	0.072782	0.015864
17741	0.005980	0.015772	0.024677	0.042453	0.014583	0.032127	0.054287	0.052028	0.006804	0.039247	...	0.002104	0.023743	0.015954	0.014572
17742	0.018934	0.003481	0.020845	0.012585	0.027635	0.057850	0.057945	0.036698	0.045001	0.002495	...	0.016719	0.042896	0.053636	0.028344

1821 rows × 1821 columns

Essentially, this is an item x item matrix where the relation of each item with every other item is indicated. Thus, the relation of any item with itself is 1 (all diagonal elements)

7. Problem 2d)

```
+------+-------+
|rating|  count|
+------+-------+
|   1.0| 169886|
|   2.0| 374452|
|   3.0|1048538|
|   4.0|1044293|
|   5.0| 618183|
+------+-------+
```

With the count of each rating in the training set, it is significant that all of the ratings can be significant (although ratings 3 and 4 have the most weightage)

If I were to normalize the ratings, it would mean changing the rating scale from 1-5 to any other target rating. I would not like to do so since the 1-5 ratings are way more natural and easier to understand (than suppose a 1-3 rating).

In addition to that, since the test set could still have lower rated movies, it would be unfair to remove them from the predictions. However, the lower rated movies can definitely be filtered out during recommendation. This is because it is always much better to suggest highly rated movies rather than poorly rated ones.

Problem 3: Collaborative Filtering Implementation

For better understanding the process of collaborative filtering and recommendation, I decided to test and try two approaches – item-based filtering, as well as ALS (which is a model-based collaborative filtering) provided by spark. It uses the Alternating Least Squares algorithm to learn the latent factors. By implementing both approaches, I would understand the differences and similarities between the two and can compare their accuracies (measured by error rates)

○ Approach 1 – item-based model

➢ For prediction:

Fetch top 15 movies correlated to current movie
Fetch ratings of current movie to the 15 items from above
If current user has less than 5 ratings,
    Fetch average rating of all 15 movies for predicted rating
Else, Calculate average of all movies (out of 15) rated by current user as predicted rating

Pyspark does not support accessing other rdds, or dataframes (here, training data – for fetching average ratings) in parallelized functions such as map, foreach, etc. Thus, I had to manually iterate over each row in test data to fetch its prediction. Since this process is obviously going to be slow, I performed the predictions on a controlled set (i.e., a subset of the test data). The run time, MAE (Mean Absolute Error) and RMSE (Root Mean Square Error) for different lengths of test data are as below:

Number of records	Run time	MAE	RMSE
100 records	44 s	0.91	1.67
350 records	2 m 14 s	0.86	1.05
700 records	5 m 13 s	0.74	0.95

*Since the records are randomly selected, the errors are approximate

Clearly, as the number of records increased, the MAE and the RMSE keep decreasing, which leads me to believe that the model would perform well with high number of records but at a cost of spending too much time, unless the resources were increased.

➢ For recommendation:

Initially, I created a dataframe for my user ratings – with id 4 (unused in train or test data)
To recommend movies to current (my user),
    Fetch user's top-rated movies (movies with rating > 3)
    For each top-rated movie, fetch the 3 most correlated items
    Calculate the average rating for each item to be shown to the user
    Provide the user with a list of movie names and average ratings as recommendations

Since there is still some manual processing done, the time taken to generate recommendations using the item-based model was 40 seconds.

My rated movies are:

movieID	userID	rating	releaseYear	title
10676	4	4	1933	The Kennel Murder Case / Nancy Drew
14810	4	4	2000	Dolphins: IMAX
16162	4	2	2002	Kim Possible: The Secret Files
11340	4	5	1988	Johnny Be Good
4556	4	2	2001	Stealing Time
6250	4	4	1997	Female Perversions
13334	4	1	2000	Catfish in Black Bean Sauce
11312	4	1	1998	Mystery Kids
15731	4	5	2002	Roxy Music: Live at the Apollo
10109	4	4	1994	Major League II

For my rated movies, the recommendations were:

title	movieID	releaseYear	avgRating
Sherlock Holmes: Terror by Night	15676	1946	3.69
Charlie Chan: The Chinese Cat	10195	1944	3.42
Wildcats	6523	1986	3.38
Youngblood	2352	1986	3.21
Encino Man	3670	1992	3.03
Search for the Great Sharks: IMAX	8121	1999	2.99
Genesis: Live at Wembley Stadium	5921	2004	2.97
Niagara: Miracles	834	1999	2.96
Teen Wolf / Teen Wolf Too (Double Feature)	2284	1985	2.91
The Story of O	1252	1975	2.9
Young Adam	12679	2004	2.71
Lenny Kravitz: Live	10187	2002	2.23

○ Approach 2 – ALS model

Libraries needed –
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import RegressionEvaluator

ALS is used for building the actual model and the RegressionEvaluator will be used to evaluate the model based on MAE and RMSE

- For prediction:

  Build ALS model as follows:

```
als = ALS(maxIter=maxIter, regParam=regParam,
          userCol="userID", itemCol="movieID", ratingCol="rating",
          coldStartStrategy="drop")
```

Cold strategy set as drop indicates that if there are any NaN values in the dataset, they can be eliminated implicitly. I tried building models for different values of maxIter and regParam, so as to fetch the best model with lowest error rates. The MAE (Mean Absolute Error) and RMSE (Root Mean Square Error) for different parameters and ranks tested are as shown.

Parameters	MAE	RMSE
`maxIter = 5` `regParam = 0.10`	Rank – 4 : 0.6914 Rank – 8 : 0.6877 Rank – 12: 0.6908	Rank – 4 : 0.8708 Rank – 8 : 0.8643 Rank – 12: 0.8666
`maxIter = 5` `regParam = 0.08`	Rank – 4 : 0.6920 Rank – 8 : 0.6848 Rank – 12: 0.6881	Rank – 4 : 0.8708 Rank – 8 : 0.8604 Rank – 12: 0.8627
`maxIter = 10` `regParam = 0.01`	Rank – 4 : 0.6783 Rank – 8 : 0.6620 Rank – 12: 0.6621	Rank – 4 : 0.8595 Rank – 8 : 0.8436 Rank – 12: 0.8480

Since the prediction is performed on the original test dataset, these error values are the true values and not just approximations. The entire process of building all the 9 models and calculating their errors took nearly 3 minutes, which, as expected, is a lot faster than the manual approach.

- For recommendation:

  Initially, I created a dataframe for my user ratings – with id 4 (unused in train or test data)
  To recommend movies to current (my user),
  - Join / add my user ratings to training data
  - Build model using best parameters with new training data
  - Use model.recommendForUserSubset() method to recommend items for subset (my user)

  For my rated movies, the recommendations were:

```
Linkin Park
Maxim: The Real Swimsuit DVD: Vol. 1
Vietnam: We Were Heroes 1st Cavalry Division Airmobile
Kiss: Unauthorized Kiss
The Deviants
Learning HTML: No Brainers
The Cars: Live
Raging Bull: Collector's Edition: Bonus Material
Secrets of War: Nazi Warfare
Dance for Camera
```

CONCLUSION

Although the item-based approach definitely took way more time (could perform better with more resources), its predictions were still quite close to the predictions made by the ALS model. This shows that even an item-based approach, if implemented rightly, can be used to predict ratings.

As for the recommendations, although the movies from item-based model and ALS model do not match, they are still actually close to the movies my user had rated. For example, the item-based model suggested Youngblood which is very similar to the movie Johnny Be Good, which is in turn similar to The Deviants, which was recommended by the ALS model. Thus, although the model made different recommendations, the movies recommended by both models are still very relevant to the user's preferences.

It was interesting to note that the user-based model would be very similar to the item-based model (code wise). The only difference would be to swap movieID and userID used anywhere in the code. Additionally, it would also be highly useful if I was to store the correlation matrix so an to be used over and over again rather than having to calculate it each time. The item-based model that I implemented was very basic and required quite a lot of manual computations. In the future, the approach could be improved and better functionalities could be made use of. This would ensure a shorter run time for the similar results.