Project 1 MapReduce Approach to Collaborative Filtering for the Netflix Challenge

Item-Item Based Collaborative Filtering Algorithm Implementation on Netflix Sub-datasets

Project Manager: Pengqiu Meng

Developer Local: Wei Wang

Key User: Jiaqi Hu

Developer Cloud: Rui Sun

Abstract

In this project, we used Collaborative Filtering (CF) algorithm to train a recommendation model on the Netflix subsets. To be more specific, we decided to take the item-item based CF after we investigated the details of the statistics of the Netflix subsets, which will be discussed in I.B section. After training, we made predictions on the test set using the similarity matrix obtained from the training process. Then, the predictions are evaluated with the true labels using the Root Mean Squared Error. In analysis section, we picked the top k similarities as the parameter to explore the influence of K on the results and found that the cumulative predictions error reaches 1.4 while k is equal to 1000, meaning the recommendation system can do a better job when given more similar movies.

I. Context

A) *Motivation.* There are a bunch of applications supported by the Recommendation System to show commercials or sell commodities to their clients based on their preferences. For instance, on Facebook, people may receive blogs about NBA because they tagged one your favorite basketball stars as Lebron James, or on YouTube, people may get a list of comedies recommended for them since they've watched a lot recently. Recommendation System utilizes a variety of different technologies generally, and these technologies can be classified as two main categories.

- Content-based algorithm. It examines the content of items and recommends it based on the key words.
- Collaborative Filtering algorithm. It recommends movies based on similarities measured between users/items.

Therefore, in this report, our team are going to explore the item-item based collaborative filtering algorithms on the Netflix subset, and the reason of such choice will be given in later sections. What's more, given the computation required to process such huge amount of data exceeds the computation power of a single personal computer, we used Spark and configured the Amazon EMR clusters to run the code.

B) *Dataset descriptions and statistics.* There are two datasets, which are training set and test set, respectively, and their statistics are given blow.

	distinct movies	distinct users
test set	1701	27555
training set	1821	28978

C) *Utility matrix.* For better understanding, we here represented the data as a utility matrix, giving each movie-user pair a double value ranging from 1.0-5.0 that represents the rating [1]. We assume that the matrix is sparse, meaning that most entries are un-rated movies which are exactly the ratings we aim to predict.

	3452	**1203**	**8374**	**3442**	**19373**
8	1.0	4.0			
101			5.0		1.0
306		3.0			
28				3.0	

II. Pre-processing

A) *Status check.* Based on the statistics, we further analyzed the Netflix data as stated in milestone2 to plan out an efficient implementation. Two criterions are used here to validate the similarities, which are overlapping items for user-user model and overlapping users for item-item model, respectively. The better the similarities the better the rating predictions, and hence, the recommendations.

For the user-user model, the steps we took to measure the overlapping items were:

1. take 10 random sampled users from TestingRatings.txt.

2. for each user in the sampled users, lookup the movies they rated, store them (in movies).

2.1 for each movie in movies, lookup the other users who rated the movie, and calculate the sum of the users(sumOverlapUsers). (if the users included the testUser himself, -1).

2.2 calculate the average per user and add it in the oneAvePerUserList.

2.3 calculate the total overlap of all users.

Similar to the user-user model, for item-item model, the steps to measure the overlapping users are:

1. take 10 sample movies from TestingRatings.txt.

2. for each movie in the sample movies, lookup the users who rated them, store them (in users).

2.1 for each user in users, lookup the other movies they rated, and calculate the sum of the movies(sumOverlapMovie). (if the movies included the testMovie itself, -1).

2.2 calculate the average per movie and add it in the oneAvePerMovieList.

2.3 calculate the total overlap of all movies.

Then we obtained the results as: (number of samples is 10)

	the estimated average overlap of items/users
User-user	10807
Item-item	254

From this table, we are going to use item-item and the reasons are as follows.

efficiency: there are fewer movies than users, so the time complexity of the algorithm will be smaller.

quality: we can find that item-item generates less overlapping. That's because items usually belong to a small set of genres compared to user's various tastes. So we think item similarity is more meaningful than user similarity.

What's more, we decided to apply normalization to the training data because after that, missing values replaced by 0 would be treated as 'average', compared to scenarios where 0 represents 'negative' without normalization.

(The code for this part is in milestone2/prob2b.py)

B) *Normalization.* Replace missing values as 0 is equivalently treating missing ratings as negative [1]. Therefore, normalization comes into play. We normalized ratings by subtracting the row mean for each row. After that, 0 becomes the average rating and positive ratings means users are more positive than the average and negative means users are more negative than the average. Therefore, we normalized the data by calculating the average ratings which the users rated it for each movie, and then store the average ratings into a dictionary call normTrainRatings. (normTrainRatings = {"trainMovie": average rating})

(The code and file for this part are in src/normalize.py and src/similarity.txt)

III. Collaborative Filtering Algorithms (item-based)

When wrote the code, we broke the Collaborative filtering implementation down to 2 jobs. Job 1 is to compute similarities between movies. Job2 is using to predict a movie's rating to users based on other rated items and the similarities we got from job1. To normalize ratings, we first created a Map to store the average rating of each movie (the 'normTrainRatings' mentioned in last section). Then compute the normalized rating on the fly.

We have written three python files to fulfill the job.

- job1.py
- job2.py
- Util.py

job1.py takes in 2 arguments. To execute job1.py, in the command line:

pyspark job1.py [training_set] [out_put_file]

job2.py takes in 4 arguments. To execute job2.py, in the command line:

pyspark job2.py [testing_set] [training_set] [job1_output] [out_put] [out_put_k_and_error]

Job1 focuses on these jobs:

- RDD structure in each stage user_id, (movie_id, rating)
    - user_id, [(movie_id, rating), ...]
    - (movie_id_1, movie_id_2), ((r1, $r1^2$), (r2, $r2^2$), r1 * r2)
    - (movie_id_1, movie_id_2), [((r1, $r1^2$), (r2, $r2^2$), r1 * r2), ((r1, $r1^2$), (r2, $r2^2$), r1 * r2), ...]]
    - (movie_id_1, movie_id_2, similarity)
- Save the result as a text file called 'similarity.txt'

Job2 takes roles as:

- read the output of job1 and create a RDD (movie_id_1, movie_id_2), similarity)
- create a map to store the most similar movies in descending order { movie_id: [(movie, similarity), ...] }
- create a map to store user rated movies { user_id: [(movie, rating), ...] }
- compute the prediction rating according to the user_id and movie_id in the testing set
- iterate through movies the user has rated, if the movie is in k most similar movies of the movie you are going to predict, compute the weighted rating.
- Sum all weighted rating and similarity, compute the prediction
- compute the square error by (prediction - true_rating)^2
- compute average square error

(The code files for this part are src/job1.py, src/job2.py, src/util.py)

IV. Small Data/Pseudo Cluster and AWS EMR Configuration and Execution

A. *Small Data on Pseudo Cluster*. In development phase, we sampled 90 distinct movies from the testing set and the first 100,000 lines of data from the training set, and then tested out the program on the pseudo cluster with only a single node in the Virtual Machine.

We got the k values and the corresponding errors as below. From the result, we found that the error increase rapidly after k is bigger than 90. It's because the overlapping of users is small considering the sampled size of the training set. Therefore, the prediction error became larger since we used un-similar users' ratings to predict. This issue can be addressed by using the complete training dataset.

(10, 12.72202731797141)
(20, 11.652480018806534)
(30, 10.185964984621291)
(40, 8.0777965247649774)
(50, 6.4409362147166611)
(60, 4.5592427537037441)
(70, 3.1307731981997331)
(80, 1.90300250314362)
(90, 35.795092990567035)
(100, 35.795092990567035)
(200, 35.795092990567035)
(300, 35.795092990567035)
(400, 35.795092990567035)
(k, error) tuples

B. *AWS EMR Configuration and Execution.*We used Amazon ElasticMapReduce (EMR) to execute the analysis in large scale. Amazon EMR is a hosted distribution of Hadoop and related software packages, meaning we do not need to worry about configuring the cluster. The user provides job code as well as defines necessary parameters, and EMR takes care of the process from provisioning to tearing down a cluster. The use of Amazon EMR has several important implications:

1. The use of Amazon Simple Storage Service (S3) over Hadoop File System (HDFS). Amazon S3 is a hosted object storage service that provides flexibility and resiliency, and it persists data between each cluster. On the other hand, HDFS is created with a specific cluster, and when that cluster is terminated, the respective HDFS will be deleted together. Since most Amazon EMR clusters are created for a job (not for standby and waiting for tasks) and will be terminated when the last step of the job is finished (we also cost concerns), the use of Amazon S3 is strongly encouraged over HDFS.
2. The use of Amazon S3 over local file system. Local file system to the cluster is the same as its HDFS, in terms of persistence between different clusters. Thus, the above analysis of HDFS also applies to local file system.

We used a cluster with following configuration (cost per hour: approx. $2.50):

Node Type	#	Spec
Master	1	**m4.large**: 4 vCore, 8 GiB memory, EBS only storage EBS Storage:32 GiB
Worker	8	**m4.xlarge**: 8 vCore, 16 GiB memory, EBS only storage EBS Storage:32 GiB

Spark-submit options for the two steps are as follows:

Job 1:

spark-submit --deploy-mode cluster --py-files s3://cse427s-fl18/ww/util.py s3://cse427s-fl18/ww/job1.py s3://cse427s-fl18/input/TrainingRatings.txt s3://cse427s-fl18/ww/job1

Job 2:

spark-submit --deploy-mode cluster --py-files s3://cse427s-fl18/ww/util.py s3://cse427s-fl18/ww/job2.py s3://cse427s-fl18/input/TestingRatings.txt s3://cse427s-fl18/input/TrainingRatings.txt s3://cse427s-fl18/ww/job1 s3://cse427s-fl18/ww/job2 s3://cse427s-fl18/ww/k

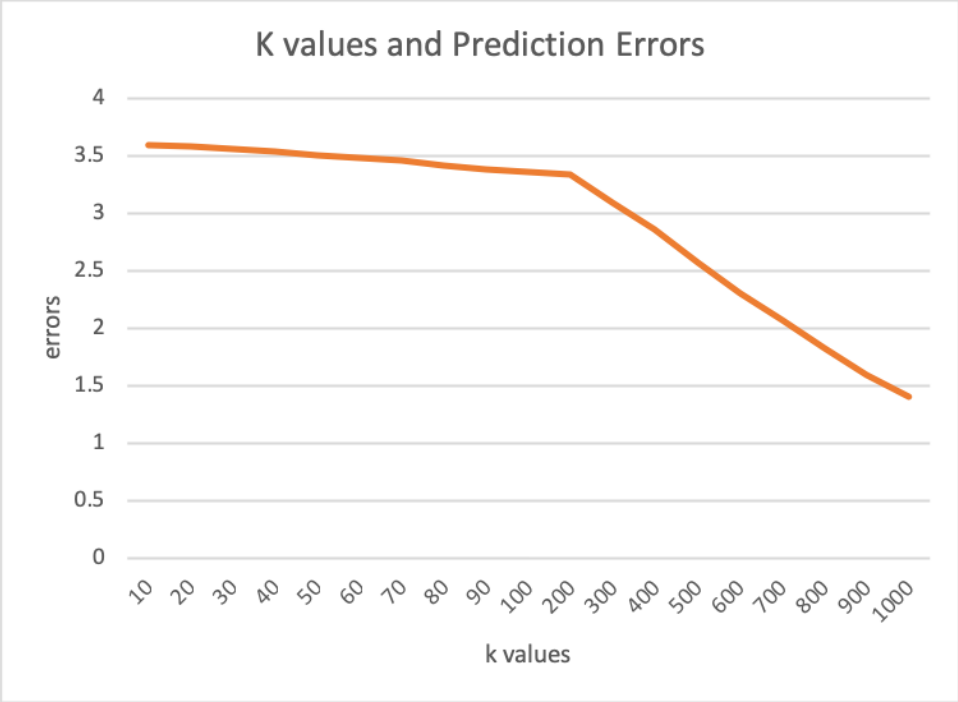
We have the following tuning:

- Setting Amazon EMR property *maximizeResourceAllocation* to true. This maximizes available resources allocated to Spark.
- Trial execution to determine the number of worker machines and their specs.
- Use Amazon S3 instead of HDFS for performance and data persistence.

Cluster provisioning takes around 10 minutes. Job 1 needs 10 minutes to complete; job 2 needs 4 minutes to complete

V. Experiments and Analysis

As we stated in the beginning, we mainly interested in the relationship between the top K similarity values and the prediction errors. In the graph below, we found that the prediction errors decrease along with the increasing of top k values from 10 to 1000, basically meaning that the more similar movies we can use, the more accurate the result will be. What's more, we can also found that the error keeps decreasing while k value increase. When k is large, even though there may be some movies that have very small similarities, we are in fact using similarity as weight, and the denominator is the sum of all similarities so that the result should be more accurate than small k. With more accurate prediction for each movie, the error will decrease accordingly, so that larger k results in smaller errors.



VI. Conclusions, Lessons, and Future Work

According to the work we devoted to this project, we concluded that item-item based collaborative filtering algorithm can catch the implicit relations between movies, which is the 'genre', as we demonstrated in the beginning. What's more, we also found that the more movies we used when did the prediction, the more accurate the predictions would be, confirming the assumption that movies share some implicit connections among each other. We would like to explain this phenomenon as a group of users share similar interest, and doing the predictions based on them would balance the bias of their preferences.

Eventually, during the development of this project, we did learn a lot. In terms of a project manager, keeping the paces of the group members is never easy, you need distribute reasonable time to each member to make sure the whole project goes smoothly. In terms of a local developer, it's important to have some test cases to test my implementation. Otherwise it would take too long to execute the codes on all data. With such big data, I need to reduce the time complexity of my algorithm, to do so, extra memory like hash maps can help. Refactor codes so that the logic is clear, and functions can be imported from another file. In terms of a key users, she learned that the implement of milestone 2 is easy. But she misunderstood the meaning of overlapping at first and be familiar with the Spark operations can make

code easily. She thought it's too slow to run especially item overlap and therefore she had to use the simple sample. Maybe better way to implement it and reduce the time complexity. Job1/Job2 is the same as the local developer said. We should use a test case to save the test time in the beginning. In terms of a cloud developer, we learned that the use of Amazon Simple Storage Service (S3) over Hadoop File System (HDFS). Amazon S3 is a hosted object storage service that provides flexibility and resiliency, and it persists data between each cluster. On the other hand, HDFS is created with a specific cluster, and when that cluster is terminated, the respective HDFS will be deleted together. Since most Amazon EMR clusters are created for a job (not for standby and waiting for tasks) and will be terminated when the last step of the job is finished (we also cost concerns), the use of Amazon S3 is strongly encouraged over HDFS. Moreover, the use of Amazon S3 over local file system. Local file system to the cluster is the same as its HDFS, in terms of persistence between different clusters. Thus, the above analysis of HDFS also applies to local file system.

This work partially reveals that the implicit relationship could exist among different kinds of genres of movies, and the future work could focus on the strengh of such relationship, or group movies to make better recommendations.

VII. References

[1] http://infolab.stanford.edu/~ullman/mmds/ch9.pdf.