The Chinese University of Hong Kong

JC2104 FYP Term-End Report

Term 2

Imsong Jeon

1155087995

20 April 2022

[Abstract]

Table of Contents

Introduction

We are living in an era of data. Enormous amount of data is produced every day from various sources. A collection of raw data can transform into a valuable and useful resource once it is processed and analyzed properly. However, the size of a collection of data can be far beyond the capacity of a single computer. Storing, managing and processing data within a reasonable timeframe became a challenge. For this reason, a scalable system of multiple networked computers that communicate and coordinate to perform shared tasks called distributed systems has emerged to become a widely used environment for managing and processing large-scale data. Nowadays, distributed system technology is inseparable from big data, machine learning, AI, and other cutting-edge information technologies involving data.

My final year project is about distributed systems. The objective of my FYP is to learn how to use distributed systems for large-scale data analytics, and perform an implementation or a new development on the system side based on my learning. For the 1st term, I have studied commonly used distributed system frameworks, Hadoop and Spark. Also, I learned to use basic functions (configure, start the clusters, compile executables, and submit jobs) of Ursa framework and became familiar with high level APIs of Ursa. Then, I started to implement few distributed algorithms on Ursa. For this term (the 2nd term), I have implemented a recommendation system, based on my experience from the first term.

Background

The modern trend of analytic utilizing distributed systems started to emerge with the release of the Google File System [1] and MapReduce [2] papers published by Google in 2003 and 2004. Then, Apache Hadoop based on these papers was developed and became popular. Since then, numerous software and frameworks for big data analytics such as Apache Spark have been released.

Among these frameworks, I am using Ursa for my FYP. Ursa is a distributed system framework with improved resource scheduling and job execution based on a new system designed by Professor James Cheng and his students [3]. Ursa is written in C++ and it includes several high-level APIs for dataset transformations such as map, reduce, and broadcast. I was provided with a cluster account for accessing the Ursa cluster along with some basic information about the usage of Ursa. I was able to learn to correctly work with the framework through studying its source code and experimenting with example codes. After becoming familiar with Ursa in the first term, I decided to implement a recommendation system on the Ursa cluster for the second term.

Recommendation System: Collaborative Filtering

A recommendation system is used to predict how much a user would prefer an item in terms of rating, preference, or other measurements. Recommendation systems are closely related to people's daily life as they help users to sort through overflowing digital media offered by their computers and smartphones. Social media platforms such as Instagram and Twitter utilize a recommendation system to decide which people and posts appear on users' feeds. Video streaming platforms such as YouTube and Netflix utilize a recommendation system to choose which video or movie to offer users to watch next. Online dating applications such as Tinder utilize a recommendation system to pick which opponent to be presented to users.

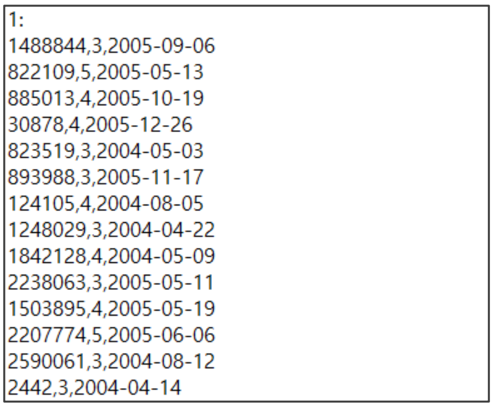
Recommendation systems can be implemented with many different methods, but the two methods that are commonly used are content-based filtering and collaborative filtering.

Content-based filtering approaches use characteristics of each user or item to recommend items with similar characteristics of the user's preferred item, or recommend items preferred by another user who shares similar characteristics with the user. On the other hand, collaborative filtering approaches aim to build a model based on the user's past behavior and use this model to recommend a new item to the user based on an assumption that a user’s preferences in the past and the present are similar. The combination of these two approaches is also possible and widely used. For instance, Toutiao, a Chinese news and content platform utilizes content-based filtering using basic data from the user’s profile to recommend contents to a new user. Then, after some usage history is collected over time, it uses collaborative filtering to refine its recommendation [4].

I chose to implement a recommendation system using a collaborative filtering approach for two reasons. First, the data for content-based filtering methods is not suitable to be managed and processed in large-scale since a profile includes a lot of subjective characteristics, which are hard to be measured by machine, and all information must be filled in before processing the data. However, the data for collaborative filtering method is more suitable for large-scale processing because it is composed of a simple history of ratings which a machine can measure without understanding the content. Second, collaborative filtering approach is more accurate with data of sufficient size compared to content-based filtering approach [5].

There are two models for performing collaborative filtering, the neighborhood model and the latent factor model. The neighborhood model uses the similarity relationship (cosine similarity or Pearson correlation coefficient) between items or users to find similar users or items for recommendation. The latent factor model transforms items and users in the same latent factor space. The latent(hidden) features in the latent space captures the relationship between users and items and latent features are inferred from the user feedback(rating). Thus, the unrated items can be predicted with the latent features of users and items. I decided to choose the latent factor model approach because the results from this model “tend to be consistently superior to those achieved by neighborhood models” [5]. In addition, the latent factor model was widely used during the Netflix Prize [6].

Before designing and implementing a program, I had to choose a dataset that is suitable for training and testing a recommendation system, and I chose to use the dataset from the Netflix Prize. Netflix Prize was an open competition launched by Netflix in 2006 where the goal was to design algorithms for predicting movie ratings to win the grand prize of $1,000,000. This dataset contains over 100 million ratings (100,480,507) from 480,189 users over 17,770 movies, and each rating is an integer scale from 1 to 5. There are four files of combined data, combined_data_1.txt ~ combined_data_4.txt, and each data file contains movie id followed by ratings from users for that movie and its date in the following format:



```
1:
1488844,3,2005-09-06
822109,5,2005-05-13
885013,4,2005-10-19
30878,4,2005-12-26
823519,3,2004-05-03
893988,3,2005-11-17
124105,4,2004-08-05
1248029,3,2004-04-22
1842128,4,2004-05-09
2238063,3,2005-05-11
1503895,4,2005-05-19
2207774,5,2005-06-06
2590061,3,2004-08-12
2442,3,2004-04-14
```

Figure 1. a screenshot of combined_data_1.txt content

The major characteristic of this dataset is the sparsity of rating. Considering a matrix *R* where each row corresponds to a user and each column corresponds to a movie, only about 1.2% of this rating matrix *R* is covered by the dataset and another 98.8% of user-movie ratings are missing.

Matrix Factorization: Alternating Least Squares Algorithm

In order to estimate these missing ratings using the latent model approach, we can factorize the sparse matrix R of dimension $n_u \times n_m$ into two lower dimensional latent spaces, a user feature matrix, *U,* of dimension $n_f \times n_u$ and a movie(item) feature matrix, *M* of dimension $n_f \times n_m$. The value, n_f, is the dimension of the feature space, which decides the number of hidden variables in the model, n_u is the number of users, and n_m is the number of movies.

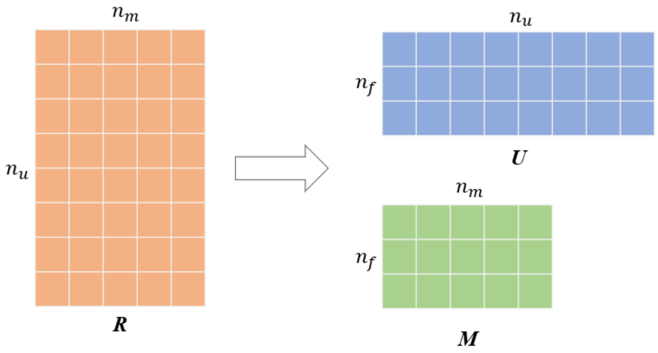


Figure 2. decomposing ***R*** into ***U*** and ***M***

With *U* and *M*, the rating r_{um} of *R* can be estimated as follows, where a vector p_u is a column of *U* and q_m is a column of M:

$$\hat{r}_{um} = p_u^T \cdot q_m \,, \qquad (1)$$

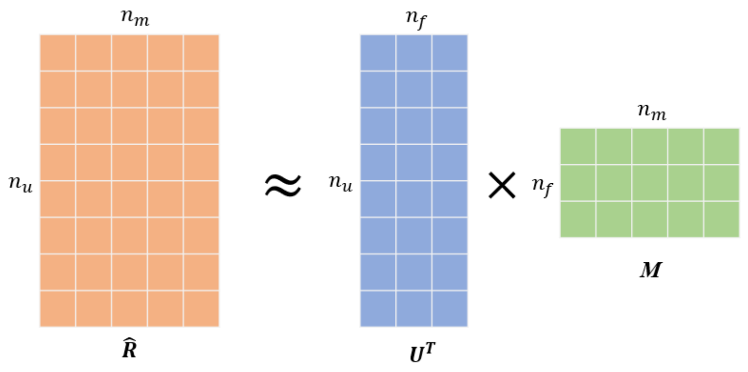


Figure 3. matrix multiplication to predict $\hat{\boldsymbol{R}}$

The SVD (singular value decomposition) is commonly used for matrix factorization, but SVD of *R* is not defined because *R* is sparse [6]. Alternatively, we can obtain optimal matrices *U* and *M* by solving an optimization problem with the following loss function f which aims to minimize the least squares error between observed ratings in *R* and estimated ratings from (1) [6]:

$$f(U,M) = \min_{U,M} \sum_{r_{um} \in R} (r_{um} - p_u^T q_m)^2 \ , \qquad (2)$$

In addition, we add a regularization so that the latent factors will not overfit the training data. For the regularizing, the following weighted-λ-regularization is added to (2) [8]:

$$f(U,M) = \min_{U,M} \sum_{r_{ui} \in R} (r_{ui} - p_u q_i)^2 + \lambda \left(\sum_{u} n_{user_u} \|p_u\|^2 + \sum_{i} n_{movie_m} \|q_i\|^2 \right) , \qquad (3)$$

Here, $n_{user_u}$is the number of ratings of user u and n_{movie_m} is the number of ratings of item i. Gradient descent is a common algorithm for optimization. However, there are two matrices *U* and *M* that have to be optimized and they are tied by a dot product. For this reason, this loss function is non-convex, and it is NP-hard to optimize a non-convex function. Thus, gradient descent would be slow and require lots of iterations [9]. Instead, I used the

ALS (alternating least squares) algorithm in order to solve this optimization problem.

The ALS algorithm fixes one of the target matrices, for example a matrix *A,* and optimizes the other matrix *B*, and then fixes the target matrix *B* and optimizes other matrix *A*, and repeats this process until convergence. Applying this approach to our loss function (3), we get the following equations for optimizing p_u for U and q_m for M by calculating derivatives with respect to p_u and q_m, respectively [8]:

$$p_u = \left(\sum_{r_{um} \epsilon r_{u*}} q_m q_m^T + \lambda n_{user_u} \right)^{-1} \sum_{r_{um} \epsilon r_{u*}} q_m r_{um}^T$$

$$= \left(M_u M_u^T + \lambda n_{user_u} I_{nf} \right)^{-1} \left(M_u R_{u_m} \right), (4)$$

$$q_m = \left(\sum_{r_{um} \epsilon r_{*m}} p_u p_u^T + \lambda n_{movie_m} \right)^{-1} \sum_{r_{um} \epsilon r_{*m}} p_u r_{um}^T$$

$$= \left(U_m U_m^T + \lambda n_{movies_m} I_{nf} \right)^{-1} \left(U_m R_{m_u} \right), (5)$$

M_u denotes the sub-matrix of M where columns for movies that the user u has rating for are selected, I_{nf} denotes the identity matrix of dimension $n_f \times n_f$, R_{u_m} denotes the row vector of user u selected from R and columns for movies that the user u has rating for selected from this row vector. Similarly, U_u denotes the sub-matrix of U where columns for users that have rating for the movie m are selected, I_{nf} denotes the identity matrix of dimension $n_f \times n_f$, R_{m_n} denotes the row vector of movie m selected from R and columns for users that has rating for the movie m selected from this row vector. With equations (4) and (5), we can iteratively update U and M, and they can be used for prediction of unobserved ratings from a user for a movie with the equation (1).

Implementation

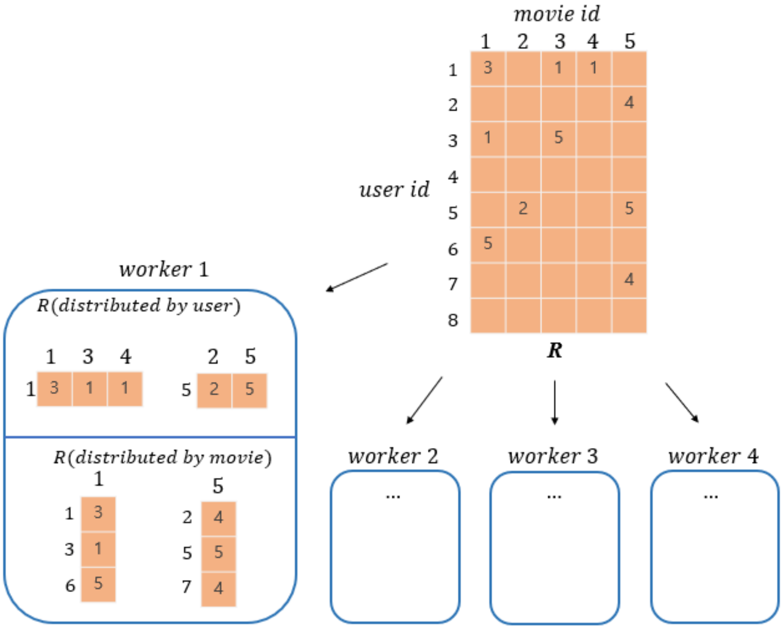
The recommendation system I have implemented goes through four major steps: data reading, data transformation, optimization, and prediction and evaluation.

Data Reading

To begin with, the ALS job is submitted to the Ursa cluster by a Python program, run_als.py. This job first gets the user configured values from the .ini file. The user configured values include the path of data files, the number of parallelisms, n_f, λ(lambda), the number of iterations, and the seed for random number generator. After that, the job reads a dataset from the local memory. Then, each line is parsed into a structure called 'Item' which includes an integer value, item_id, and a pointer to a vector of (integer, double) pairs, rate_list. For instance, a line: "18423, 4, 2005-09-06" under movie id "3" would be parsed into Item(item_id = 18423, *rate_list = (3, 4.0)). The parsed item is then stored in a distributed dataset named input_data. Four data files are read by four different workers in parallel. There were two challenges in this step. The first challenge was the reading of multiple files, which was solved by buffering data from each file on separate buffers, then combining them to a single buffer after reading was complete. The second challenge was keeping track of movie id because the data is in a format that a movie id is in the preceding line, and user ids and ratings for that movie id are placed in lines below it without movie id. This challenge was resolved by using a separate variable to store the current movie id for each worker.

Data Transformation

After reading all four data files, the dataset, input_data, is filled with 100 million items and each item includes a user id stored in item_id and a single pair of movie id and the corresponding rating for those user id and movie id stored in rate_list. This dataset is the data of each rating, r_{um}, of $n_u \times n_m$ matrix R. However, we would waste a lot of memory if we store the entire sparse matrix R in $n_u \times n_m$ space when about 98% of its space is undefined even if we store it in a distributed manner. Based on the observation of equations, (4) and (5), we use for optimization of latent factors, we only need the rating matrix R for R_{u_m} and R_{m_u}. R_{u_m} denotes the row vector of the user u selected from R and columns for movies that the user u has rating for selected from this row vector. Thus, we do not need to worry about the unrated space of R and we can store the matrix R distributed by users. Similarly, we can store another copy of the matrix R distributed by movies for R_{m_u}.



user id	1	2	3	4	5
1	3		1	1	
2					4
3	1		5		
4					
5		2			5
6	5				
7					4
8					

Figure 4. two copies of $\boldsymbol{R}$ stored in distributed system

First, the input_data dataset is transformed using the ReduceBy function. In this ReduceBy function, items are aggregated by the user id, appending their movie id and rating pairs into a single list. After this transformation, a distributed matrix R_users (matrix R distributed by users) is created in the type of dataset(vector) of items. For instance, an item in R_users for user id = 1 from the figure 3 would look like: Item(item_id = 1, *rate_list = (1, 3.0), (3, 1.0), (4, 5.0)). Second, a second distributed matrix R_movies (matrix R distributed by movies) is created from R_users. It utilizes MapPartition function to make a dataset of items similar to input_data but using movie id as the item_id. Then, the ReduceBy function is used again to gather user id and rating pairs for each movie id to form R_movies.

The next datasets we need before optimization are the user feature matrix U and the movie feature matrix M. First, a distributed dataset, movie_feature_matrix, is created from R_movies with MapPartition function; for each movie id in R_movies, it creates a row of matrix of size n_f. The first value of each row is initialized with the average rating for that movie and a small random number for other values. The dataset movie_feature_matrix is distributed by the row index (movie id) together with R_movies. However, we need the whole feature matrix for optimization. Thus, another dataset local_movie_feature_matrix is created by broadcasting all rows of movie_feature_matrix to each worker and combining them into a non-distributed (local) matrix. The row index is stored in each row of movie_feature_matrix in order to maintain the order of each row from the distributed matrix. Second, a distributed dataset, user_feature_matrix, is created from R_users with MapPartition function. The dataset user_feature_matrix does not need to be initialized because it will be calculated from the first iteration of optimization later. This dataset also needs a row index stored for each distributed row for combining them into a local_user_feature_matrix. However, the user id is not sequential like movie id. For this reason, a dataset user_id_map is built by assigning each user id with a sequential index value according to the order of the id. This index value is stored in user_id_map with the corresponding user id as a pair. For instance, for a set of user id: 12, 8, 21, 5, we will get an id map: (5, 0), (8, 1), (12, 2), (21, 3). Using this id map, we can convert between actual user id and row index value used for the feature matrix. After formulation of these feature matrices, we are set for the optimization step.

Optimization

In the optimization step, user_feature_matrix and movie_feature_matrix are updated by a distributed ALS algorithm with equations (4) and (5) for a set number of iterations. First, we update the user_feature_matrix by calculating a new vector p_u for each row. This process is done in parallel on each worker separately. This is possible because the R_users is distributed by users and each worker carry a dataset of local_movie_feature_matrix. Thus, each p_u can be calculated on the worker containing the corresponding distributed user row of R_users for the user u. The following steps are the overall workflow for a single iteration of the user_feature_matrix optimization procedure[1]:

A. Obtain number of movies, n_m
B. Create an identity matrix $I_{nf}(n_f \times n_f)$ and multiply with λ (lambda)
C. For each user row u in the current worker:
   1. Create a sub matrix $M_u(n_{user_u} \times n_f)^2$ by extracting movie rows from movie_feature_matrix that the user u has rating for
   2. Find transpose of M_u to obtain $M_u^T(n_f \times n_{user_u})$
   3. Get $R_{u_m}(1 \times n_{user_u})$ from R_users and find transpose of $R_{u_m}$to obtain $R_{u_m}^T(n_{user_u} \times 1)$
   4. Perform matrix multiplication, $M_u^T R_{u_m}^T$, to obtain a result vector $V(n_f \times 1)$
   5. Perform matrix multiplication, $M_u^T M_u$, to obtain $A_1(n_f \times n_f)$
   6. Perform scalar product, $n_{user_u} \lambda I_{nf}$, to obtain $A_2(n_f \times n_f)$
   7. Perform matrix summation, $A_1 + A_2$, to obtain $A(n_f \times n_f)$

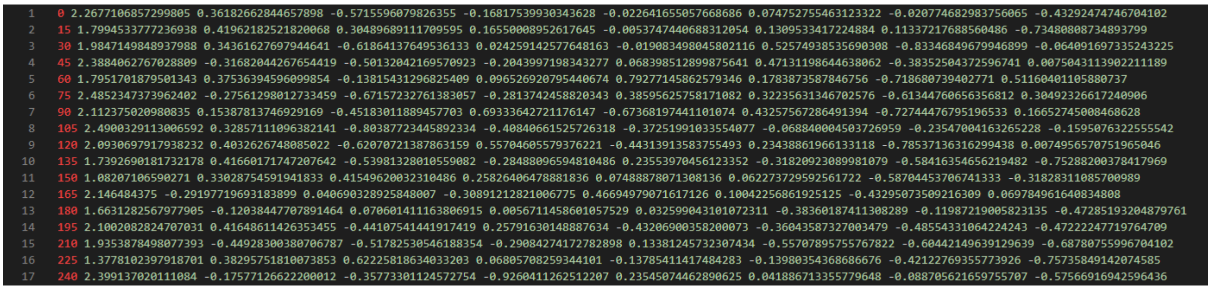
[1] the dimension of feature matrix is $n_m \times n_f$ whereas the dimension of feature matrix used in equation (4) is $n_f \times n_m$, so there is slight difference in the calculation

[2] $dimension\ of\ corresponding\ matrix;\ n_{user_u} = number\ of\ movies\ u\ has\ rating\ for$

8. Find inverse of A to obtain $A^{-1}(n_f \times n_f)$
9. Perform matrix multiplication, $A^{-1}V$ to obtain new vector $p_u(n_f \times 1)$
10. Update user_feature_matrix with p_u

After optimization of user_feature_matrix is completed, distributed rows are gathered by broadcast to update local_user_feature_matrix. The optimization of movie_feature_matrix is almost identical to the optimization of user_featrue_matrix, but the optimization of movie_feature_matrix involves R_movies instead of R_users and local_user_feature_matrix instead of local_movie_feature_matrix.

At the end of each iteration of optimization, user_feature_matrix and movie_feature_matrix is saved on the local memory as files for the record of feature matrices of each iteration, and for the prediction using feature matrices in the future. The following is an example of stored feature matrix with $n_f = 8$:



```
0 2.2677106857299805 0.36182662844657898 -0.5715596079826355 -0.16817539930343628 -0.022641655057668686 0.074752755463123322 -0.020774682983756065 -0.43292474746704102
15 1.7994533777236938 0.41962182521820068 0.30489689111709595 0.16550008952617645 -0.0053747440688312054 0.1309533417224884 0.11337217688560486 -0.73480808734893799
30 1.9847149848937988 0.34361627697944641 -0.61864137649536133 0.024259142577648163 -0.019083498045802116 0.52574938535690308 -0.83346849679946899 -0.064091697335243225
45 2.3884062767028809 -0.31682044267654419 -0.50132042169570923 -0.2043997198343277 0.068398512899875641 0.47131198644638062 -0.38352504372596741 0.0075043113902211189
60 1.7951701879501343 0.37536394596099854 -0.13815431296825409 0.096526920795440674 0.79277145862579346 0.1783873587846756 -0.718680739402771 0.51160401105880737
75 2.4852347373962402 -0.27561298012733459 -0.67157232761383057 -0.281374245820343 0.38595625758171082 0.32235631346702576 -0.61344760656356812 0.30492326617240906
90 2.112375020980835 0.15387813746929169 -0.45183011889457703 0.69333642721176147 -0.67368197441101074 0.43257567286491394 -0.72744476795196533 0.16652745008468628
105 2.4900329113006592 0.32857111096382141 -0.80387723445892334 -0.40840661525726318 -0.37251991033554077 -0.068840004503726959 -0.23547004163265228 -0.1595076322555542
120 2.0930697917938232 0.4032626748085022 -0.62070721387863159 0.55704605579376221 -0.44313913583755493 0.23438861966133118 -0.78537136316299438 0.0074956570751965046
135 1.7392690181732178 0.41660171747207642 -0.53981328010559082 -0.28488096594810486 0.23553970456123352 -0.31820923089981079 -0.58416354656219482 -0.75288200378417969
150 1.08207106590271 0.33028754591941833 0.41549620032310486 0.25826406478881836 0.07488878071308136 0.062273729592561722 -0.58704453706741333 -0.31828311085700989
165 2.146484375 -0.29197719693183899 0.040690328925848007 -0.30891212821006775 0.46694979071617126 0.10042256861925125 -0.43295073509216309 0.069784961640834808
180 1.6631282567977905 -0.12038447707891464 0.070601411163806915 0.0056711458601057529 0.032599043101072311 -0.38360187411308289 -0.11987219005823135 -0.47285193204879761
195 2.1002082824707031 0.41648611426353455 -0.44107541441917419 0.25791630148887634 -0.43206900358200073 -0.36043587327003479 -0.48554331064224243 -0.47222247719764709
210 1.9353878498077393 -0.44928300380706787 -0.51782530546188354 -0.29084274172782898 0.13381245732307434 -0.55707895755767822 -0.60442149639129639 -0.68780755996704102
225 1.3778102397918701 0.38295751810073853 0.62225818634033203 0.06805708259344101 -0.13785411417484283 -0.13980354368686676 -0.42122769355773926 -0.75735849142074585
240 2.399137020111084 -0.17577126622200012 -0.35773301124572754 -0.9260411262512207 0.23545074462890625 0.041886713355779648 -0.088705621659755707 -0.57566916942596436
```

Figure 5. a screenshot of saved feature matrix

The linear algebra functions used in the optimization procedure changed along the implantation. In the beginning, I wrote simple linear algebra functions for matrix transpose, multiplication and inverse. These functions worked fine with small n_f. However, as I started to test with the larger n_f, I noticed that these linear algebra functions, especially the matrix inverse function, become very time consuming as n_f value gets larger. In order to solve this delay, I tried other algorithms for matrix inverse, but I was not able to achieve major improvement. For this reason, I decided to use a library I found called Eigen. Eigen is a C++

library for linear algebra [11]. I was able to significantly lower the time it takes to process the optimization step, particularly the matrix inverse step. For example, the time used for ALS job with $n_f = 25$, lambda = 0.03, number of iterations = 20, and number of workers = 15 was 4433.206s with a naïve implementation of matrix inverse, whereas the time used for the same job was 341.797s with the matrix inverse function from Eigen library.

Finally, the job on the Ursa cluster finishes after it completes the set iterations of optimization, and successfully stores feature matrices for each iteration plus the user id map.

Prediction and Evaluation

The Python program, run_als.py, is used to make predictions based on the stored feature matrices and user id map from the previous job on the Ursa cluster, and to perform evaluation of the prediction. There are three major functions in this program. First, provided with a file that contains a list of user id and movie id, it can produce a prediction for each user id and movie id. To perform this task, it reads and builds user and movie feature matrices from the stored files and performs dot product, $p_u^T \cdot q_m$, for the corresponding user u and movie m to obtain an estimation of rating not observed in the training dataset. Second, provided with a file that contains a list of user id, movie id, and the actual rating for the pair of user and movie, it can return the RMSE calculated from the actual ratings and the estimated ratings. Third, provided with a user id u from the dataset and a value n, it can provide a recommendation of n movies that the user u would probably give a high rating. For this task, it estimates ratings for the user u and every movie in the dataset and returns top n movies with the highest estimation of ratings.

I chose to store feature matrices and use Python for prediction and evaluation instead of continuing these steps on the Ursa cluster job for three reasons. First, I wanted to create a recommendation system that a user can easily interact with. After finishing an optimizing job from the Ursa cluster with large-scale data, a user can choose to resubmit the optimizing job with an updated set of data or choose to obtain a prediction or recommendation in a short period of time using the stored feature matrices on the hard disk. Also, it is simpler to execute a Python program compared to submitting a job on the Ursa cluster. Second, prediction and evaluation procedures do not require high computing power; it only needs to perform dot product on vectors for each pair of user and movie and calculate root mean square error. Also, files containing target data for prediction are not as large as training data files. Third, Python offers a variety of utility packages to analyze and plot the results.

Experiments and Performance Results

There are many variables that can be controlled and experimented in the recommendation system. First, the number of workers can be controlled since the job is performed on a distributed system of multiple machines. Second, the variables: $n_f(\text{the number of hidden features})\,,\lambda(\text{lamda for regularization}),$ and the number of iterations can be controlled for the ALS algorithm.

I tested different numbers of workers in the Ursa cluster to compare the performance of the recommendation system with the. There were 21 machines in the cluster (1 master and 20 workers) I worked on, but I was not able to work with all the workers for some technical issues. The following figure presents the time used by the distributed ALS job for 20 iterations with different number of workers:

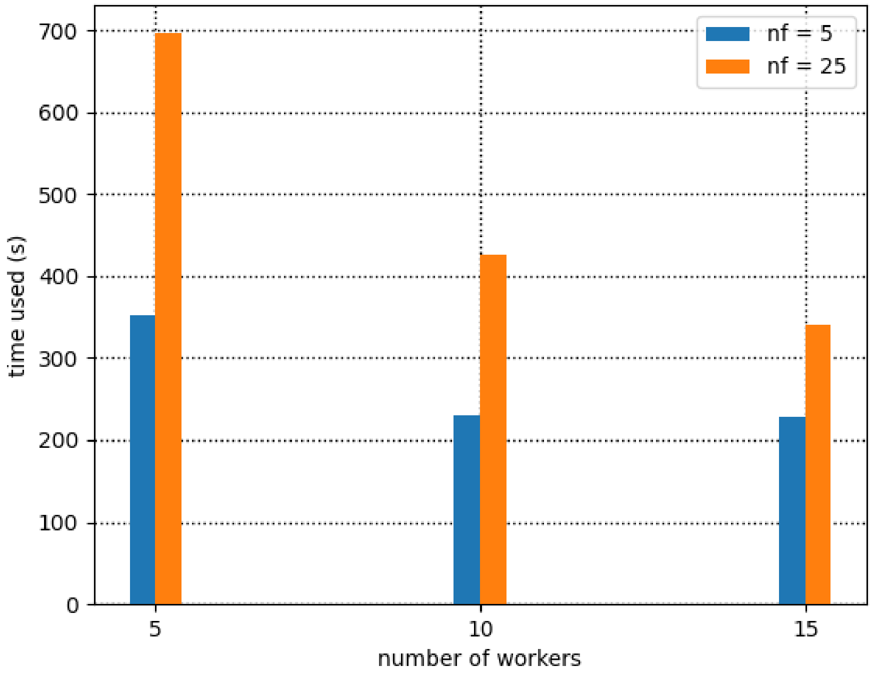


Figure 6. number of workers vs. time used

The times used by the job do not vary much with different numbers of workers when $n_f = 5$. However, the times used by the job achieves almost linear speedup with more workers when $n_f = 25$. The time used for reading data files is fixed regardless the $number\ of\ workers \geq 4$ because the file reading is done in parallel on at most 4 workers. The time used for reading is about 82s. The time used for optimization process increases along with the increase in n_f because the amount of computation needed for matrix multiplication and finding matrix inverse is directly proportional to n_f, the dimension of feature matrices. Thus, when n_f is small, the addition of workers does not significantly speed up the process since most of its running time is used for reading data files. On the other hand, when n_f is larger, the addition of workers benefits the job running time because the linear algebra computation that causes bottlenecks in optimization jobs can be run on more machines in parallel and speed up the process. 15 workers were used for all the subsequent tests.

I experimented with various values of and the number of iterations, $\lambda,$ and n_f, to test their effect on the precision of prediction from ALS. The precision of prediction model from a recommendation system is commonly measured by RMSE:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{N}(r_i - \hat{r_i})^2}{N}}\,, \qquad (6)$$

The RMSE from Netflix's own recommendation system (CineMatch) in 2006 was 0.9514 on the test dataset, and the goal for the Netflix Prize was to improve this RMSE by 10%. The prize-winning team "Bellkor's Pragmatic Chaos" scored a test RMSE of 0.8567. These RMSE are results obtained by submission to the Netflix prize website. Unfortunately, this website has been closed and there is no other way to test my recommendation system with the test dataset and compare my RMSE with those past values. I had to create my own test dataset from the training dataset; I randomly selected approximately 1% of training data, removed them from the training set, and used this dataset as the test set for calculating RMSE score. The RMSE scores from my tests are overall better than test RMSE from the Netflix Prize website. This is because the distribution of ratings per user in the test dataset provided by the Netflix Prize website and the training dataset are different [8], whereas my own test data and training data shares the same distribution.

With this in mind, I started the testing with the number of iterations. The following figure illustrates effect of the number of iterations with fixed $n_f = 8$ and $\lambda = 0.05$ on RMSE:

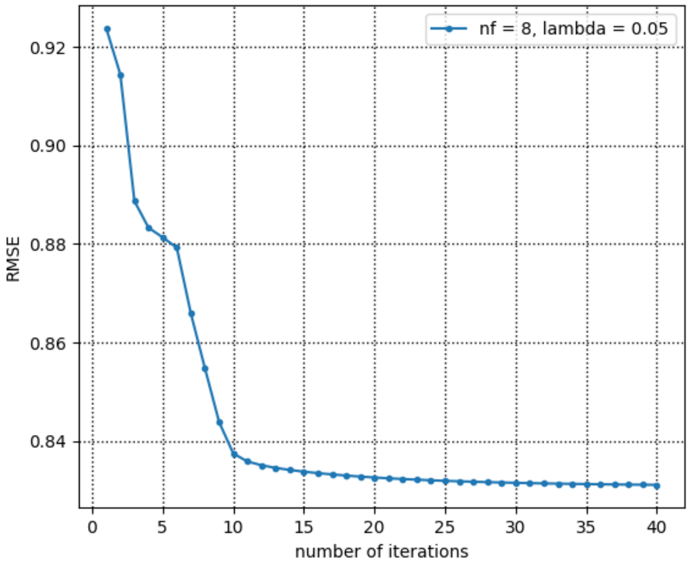


Figure 7. number of iterations vs. RMSE

As shown in the figure, RMSE is improved for each additional iteration, but the rate of improvement diminishes gradually until the model seems to converge around 30 iterations. From 0 iteration to 10th iteration, RMSE decreased from 0.92386 to 0.83744, which is about 10% improvement. From 11th iteration to 30th iteration, RMSE decreased from 0.83582 to 0.83148, which is only about 0.5% improvement. Finally, from 31st iteration to 40th iteration, RMSE still decreased from 0.83142 to 0.83106, which is only about 0.04% improvement. From this experiment, we can tell that more iterations of optimization do improve the accuracy of the prediction, but it would be inefficient to set the number of iterations to a number > 30.

The following figure shows how different λ resulted in different RMSE with fixed $n_f = 8$ and number of iterations $= 20$:

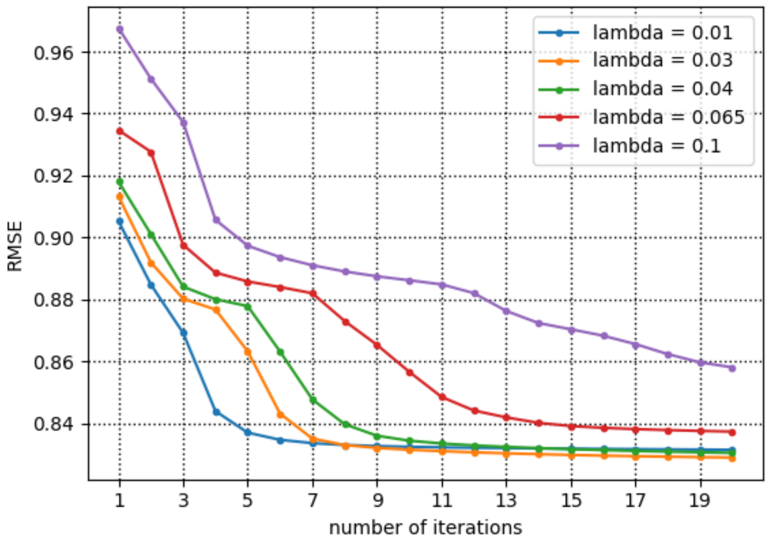


Figure 8. number of iterations vs. RMSE (lambda)

The optimization with $\lambda = 0.03$ scored the lowest RMSE: 0.82893 among different lambda values.

Next, the following figure shows the effect of different n_f on RMSE with fixed lambda, $\lambda = 0.03$ and 20 iterations:

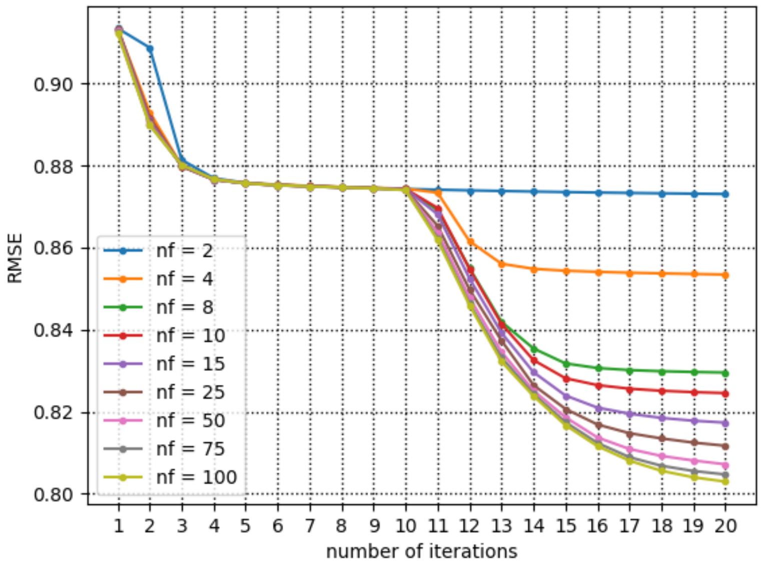


Figure 9. number of iterations vs. RMSE (nf)

From this figure, it is clear that a higher n_f results in a better RMSE. The lowest RMSE score of 0.8029 was obtained with $n_f = 100$. We should also observe the time used for optimization of different n_f.

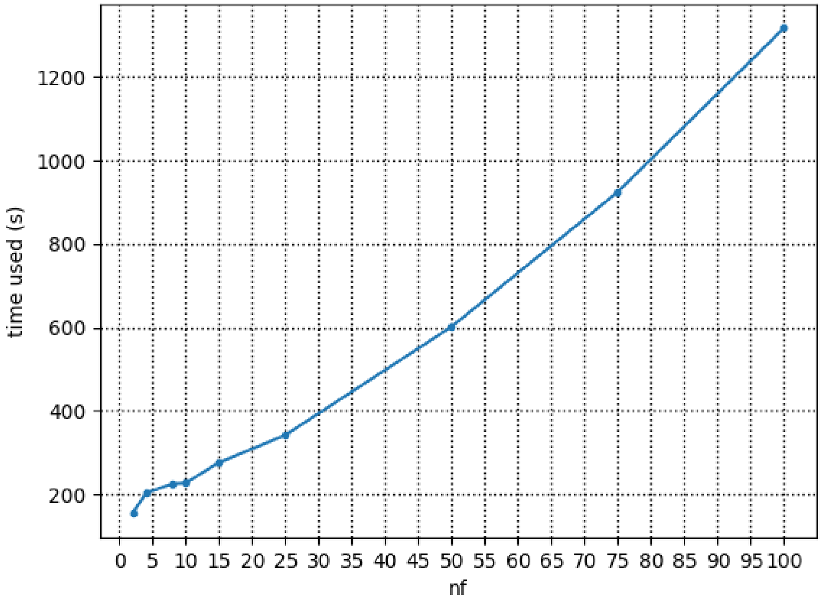


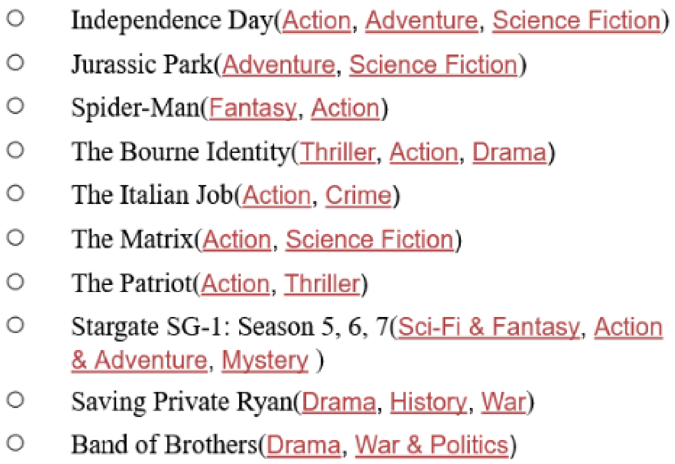
Figure 10. nf vs. time used

As shown in the figure, the time used for optimization increases exponentially as n_f increases. The ALS job with $n_f = 100$ used 1318.421s (about 22 minutes) whereas the ALS job with $n_f = 25$ used 341.797s (about 6 minutes) which is about a quarter of the time used for the job with $n_f = 100$. This increase in time for higher n_f could be a significant drawback for ALS algorithm on a single machine, but this problem can be partially resolved by adding more machines to the cluster for the distributed ALS algorithm.

Finally, I used the most optimal number of iterations, n_f, and lambda value from the experiments to try to get the best RMSE score within a reasonable time. In the end, I managed to get RMSE of 0.79 with the number of iterations = 25, $n_f = 200$, and $\lambda = 0.03$.

The time used for this job is 5426.531s, which is about 1.5 hours.

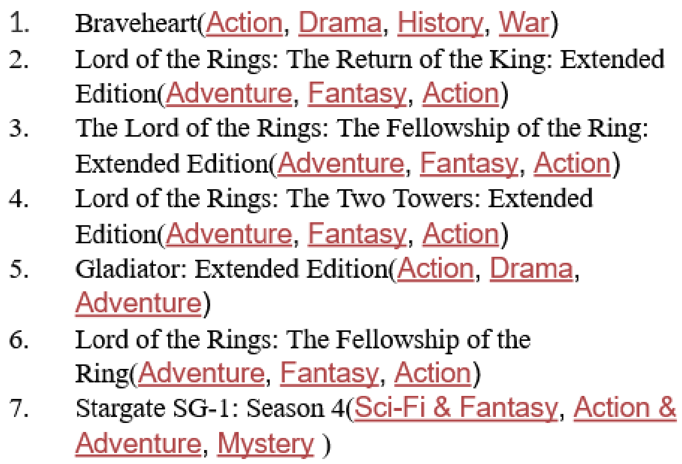
After testing the precision of my ALS program with RMSE scores, I also observed the program's actual recommendation of movies based on its optimized feature matrices. Movies are recommended using the predicted ratings for the specified user and each movie in the training dataset. The program not only prints the top 7 movies with the highest predicted rating, but also saves the whole list of predictions as a text file. I used the data of a user with the User ID = 1583391. The following list of movies is the list of 10 movies watched and rated 5 by the user. Genres of each movie to depict the characteristics of each movie.



- ○ Independence Day(Action, Adventure, Science Fiction)
- ○ Jurassic Park(Adventure, Science Fiction)
- ○ Spider-Man(Fantasy, Action)
- ○ The Bourne Identity(Thriller, Action, Drama)
- ○ The Italian Job(Action, Crime)
- ○ The Matrix(Action, Science Fiction)
- ○ The Patriot(Action, Thriller)
- ○ Stargate SG-1: Season 5, 6, 7(Sci-Fi & Fantasy, Action & Adventure, Mystery)
- ○ Saving Private Ryan(Drama, History, War)
- ○ Band of Brothers(Drama, War & Politics)

Figure 11. 5-rated watched movies by User ID = 1583391

Then, the following list of movies is the top 7 recommended movies based on the optimized feature matrices.



1. Braveheart(Action, Drama, History, War)
2. Lord of the Rings: The Return of the King: Extended Edition(Adventure, Fantasy, Action)
3. The Lord of the Rings: The Fellowship of the Ring: Extended Edition(Adventure, Fantasy, Action)
4. Lord of the Rings: The Two Towers: Extended Edition(Adventure, Fantasy, Action)
5. Gladiator: Extended Edition(Action, Drama, Adventure)
6. Lord of the Rings: The Fellowship of the Ring(Adventure, Fantasy, Action)
7. Stargate SG-1: Season 4(Sci-Fi & Fantasy, Action & Adventure, Mystery)

Figure 12. top 7 recommended movies for User ID = 1583391

As shown in these lists, many of the movies that received ratings of 5 from the User ID = 1583391 have the genres: action, adventure, or science fiction. Likewise, the movies recommended by the recommendation system have similar genres. This example illustrates the recommendation capability of the system.

Conclusion

Through this experience of working with a distributed system and large-scale data in FYP, I learned about programing techniques on how to use distributed systems for large-scale data analytics. In addition, I learned four lessons regarding large scale data analytics on a distributed system. The first lesson I learned was that a well-designed distributed algorithm on a distributed system can improve performance significantly. Also, I was able to see the further increase in performance would be possible if more machines were added to the system. I felt the powerfulness of a distributed system from this experience. The second lesson was that data cleaning and data transformation take up a large portion of large-scale data analytics. A lot of effort was needed to transform the collected data into a correct format that can be used for data analytics. The third lesson was that testing and debugging each small building block of a distributed or a large-scale program is important. It was difficult to debug through intertwined functions and data that goes through multiple transformation. In addition, testing using large-scale data along the development is helpful because some bugs are not detected with test data of small size. The fourth lesson was that planning before starting to implement a system is crucial for saving time and effort. I wasted a lot of time and effort writing codes that turn out to be useless or wrong during the project because I did not plan properly. Overall, I have both enjoyed and struggled with FYP, but in the end, I have learned a lot through the project.

References

[1] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. 2003.

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. 2004.

[3] Tatiana Jin, Zhenkun Cai, Boyang Li, Chenguang Zheng, Guanxian Jiang, James Cheng. Improving Resource Utilization by Timely Fine-Grained Scheduling. April 2020.

[4] Anu Hariharan. The Hidden Forces Behind Toutiao: China's Content King. https://www.ycombinator.com/blog/the-hidden-forces-behind-toutiao-chinas-content-king. October 2017.

[5] Yifan Hu, Yehuda Koren, Chris Volinsky. Collaborative Filtering for Feedback Datasets.

[6] Nicolas Hug. Understanding matrix factorization for recommendation (part 1) – preliminary insights on PCA. http://nicolas-hug.com/blog/matrix_facto_1. June 2017.

[7] Netflix. Netflix Prize data. https://www.kaggle.com/datasets/netflix-inc/netflix-prize-data. 2020.

[8] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber and Rong Pan. Large-scale Parallel Collaborative Filtering for the Netflix Prize.

[9] Reza Zadeh. 14 Matrix Completion via Alternating Least Square(ALS). http://stanford.edu/~rezab/classes/cme323/S15/notes/lec14.pdf. May 2015.

[10] Eigen, https://eigen.tuxfamily.org/index.php?title=Main_Page. January 2022