Collaborative Filtering on Netflix Dataset using Matrix Factorization

Harsh Uttam Mehta[1] and Khusaal Narendra Giri[1]

[1]*School of Informatics,Computing & Engineering*
[1]*Indiana University Bloomington,IN,US*

***Abstract*— The Netflix Movie Data is a very popular dataset in the Data Mining & Machine Learning Community.It comprises of about 100 million anonymous movie ratings by the user.This dataset has many attributes;which can be effectively used in building a good recommender system for the user.Due to the presence of many features,the dataset is very sparse which makes it difficult to apply Collaborative Filtering algorithm for making recommendations to the user. Also,processing such a large dataset requires a lot of resources such as time & memory.In this report,we briefly describe how to tackle these problems using Factorization Machines & compare the performance of these machines on a single node(locally) with that in a distributed environment using various algorithms such as Alternating Least Squares & Singular Value Decomposition.**

Keywords: Factorization Machine, Collaborative Filtering Algorithm, Singular Value Decomposition, Alternating Least Squares, Apache Spark.

I. Introduction

Recommendation systems suggest items of interest & enjoyment to people based on their preferences.Nowadays,we have more options and choices in our life and this increase in options and choices will even increase in the near future. What to eat, which movie to watch, what book to read are the questions that we find ourselves to answer all the time. If we just consider the information space that has the answer of these questions let alone answering the questions in the first place, we could find ourselves in an immense decision domain, which may cripple our decision making.Therefore,Recommender Systems will have a great impact in our life in the near future.These systems have started playing an important role in many e-commerce sites,notably Amazon,Yahoo, Netflix & Coursera.

Netflix,an online movie subscription rental service,allows people to rent movies for a fixed monthly fee,maintaining a prioritized list of movies they wish to view.The length of service of subscribers is related to the number of movies they watch & enjoy.If subscribers fail to find movies that interest & engage them,they tend to abandon the service.Connecting subscribers to movies that they will love is therefore critical to both the subscribers & the company.

Also,Making recommendations on a sparse data set is a challenging problem to solve as the traditional collaborative filtering methods compute distance relationships between items or users;which are generally thought of as 'neighborhood' methods.But,this approach has two major issues which are as follows:

- It doesn't scale particularly well to massive data sets like the Netflix Data set.
- It might overfit on the sparse ratings matrices & model the noisy representations of the user tastes & preferences in the sparse settings.

To avoid these issues,we employ the method of Matrix Factorization Machines.Matrix factorization is defined as the breaking down of one matrix into a product of multiple matrices.Factorization machines can factor the matrices in many different ways; but algorithms such as Alternating Least Squares(ALS) & Singular Value decomposition(SVD) is particularly useful for Matrix factorization.

Also,since the Netflix data is very large & wide,It will take a lot of time to perform computations for Factorization Machine on a single machine.This motivates us to implement the Matrix Factorization Machine in a distributed environment.By hosting it on multiple nodes,we can achieve the highest performance by reducing the compute time taken for fitting the model & recommending the movies to the users.

Many algorithms have been developed for Matrix factorization.It therefore becomes imperative to determine which algorithms perform the best for the Netflix Dataset under different parameter settings.We have decided to choose ALS & SVD for performance comparison as they are the most famous & the recent algorithms for Factorization Machines.

II. Architecture & Implementation

A. *Singular Value Decomposition*

Singular Value decomposition is an algorithm that decomposes a matrix **R** into the best lower rank(i.e. simpler) approximation of the original matrix **R**.Mathematically,it decomposes **R** into two unitary matrices & a diagonal matrix:

$$R = U \sum V^T$$

where **R** is the user ratings matrix,**U** is the user features matrix,$\sum$ is the diagonal matrix of singular values(essentially weights) & V^T is the movie features matrix.**U** & V^T are orthogonal.

Intuitively,**U** represents how much users 'like' each feature & V^T represents how relevant each feature is to each movie.**U** will help us in building an abstract model of every user in the database.This matrix will give us a feature vector for every user & can be thought of representing the similarities or dissimilarities between every user in the data.

On the other hand,**V** will help us in building an abstract model of every movie in the database.This matrix will provide us a feature vector for every movie where each feature can correspond to features assumed for the user

model in the **U** matrix.It can be thought of representing the similarities or dissimilarities between every movie in the data set.

So,our main goal is to determine the matrix U & V from the Netflix User Ratings matrix using SVD.These matrices will help us to recommend the correct movies to the user & build a very strong Recommender System.

B. *Alternating Least Squares*

Alternating Least Squares is a method of Matrix Factorization which uses Gradient descent to find the factors of a matrix.In this method,we define a cost function having the factors of the matrix as the parameters & try to minimize the cost using the Gradient descent update rules.The name 'alternating' comes from the fact that the update rules are being applied in an alternating fashion wherein we keep one factor constant & update the second factor followed by updating the first factor by using the updated second factor.In this way,we end up getting the factors of a matrix which will be used for prediction in the recommender system.

We have users rating matrix in the Netflix dataset,which can be defined as follows:

$$Q_{ui} = \begin{cases} r, & \text{if user u rate item i} \\ 0, & \text{if user u did not rate item i} \end{cases} \quad (1)$$

where r is the rating provided by the user u on movie i.If we have m users and n items, then we want to learn a matrix of factors which represent movies. That is, the factor vector for each movie would represent that movie in the feature space. We also want to learn a factor vector for each user in a way similar to how we represent the movie.Factor matrix for movies $Y \in R^{fXn}$ (each movie is a column vector) & factor matrix for users $X \in R^{mXf}$ (each user is a row vector) are the unknown matrices, where f is the total number of factors into consideration.By adopting an alternating least squares approach with regularization,we estimate Y using X & estimate X using Y.After enough number of iterations, we are aiming to reach a convergence point where either the matrices X & Y are no longer changing or the change is quite small.

Let,the weight matrix w_{ui} be defined as follows:

$$w_{ui} = \begin{cases} 0, & if\ q_{ui} = 0 \\ 1, & \text{else} \end{cases} \quad (2)$$

Then,the cost function to be minimized is as follows:

$$J(x_u) = (q_u - x_u Y) W_u (q_u - x_u Y)^T + \lambda x_u x_u^T$$

$$J(y_i) = (q_i - X y_i) W_i (q_i - X y_i)^T + \lambda y_i y_i^T$$

We also need the regularization terms in order to prevent overfitting the data.The regularization parameters need to be tuned using cross-validation in the dataset for algorithm to generalize better.Thus,the solution for the factor vectors is,then,given as follows:

$$x_u = (Y W_u Y^T + \lambda I)^{-1} Y W_u q_u$$

$$y_i = (X^T W_i X + \lambda I)^{-1} X^T W_i q_i$$

where $W_u \in R^{nXn}$ & $W_i \in R^{mXm}$ are the diagonal matrices.In this way,we can factorize the user-ratings matrix using the ALS approach.

C. *Spark Architecture*

Spark uses a master/worker architecture. There is a driver that talks to a single coordinator called master that manages workers in which executors run.The driver & the executors run in their own Java processes. We can run them all on the same (horizontal cluster) or separate machines (vertical cluster) or in a mixed machine configuration.Physical machines are called as hosts or nodes.

The spark architecture is described in the following figure:

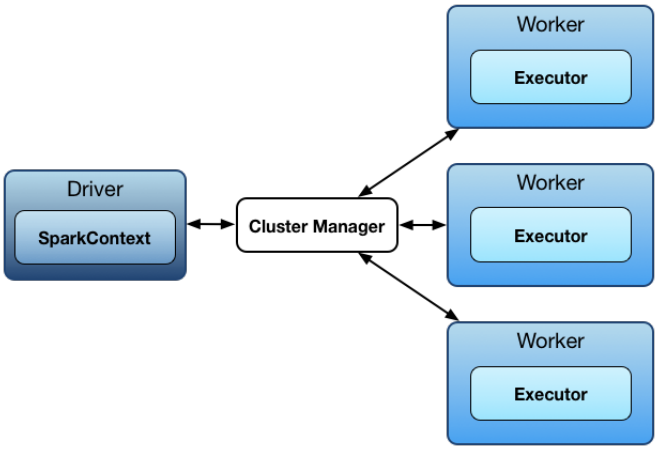

Fig. 1. Spark architecture

The communication between master & slave nodes is described in the following figure:

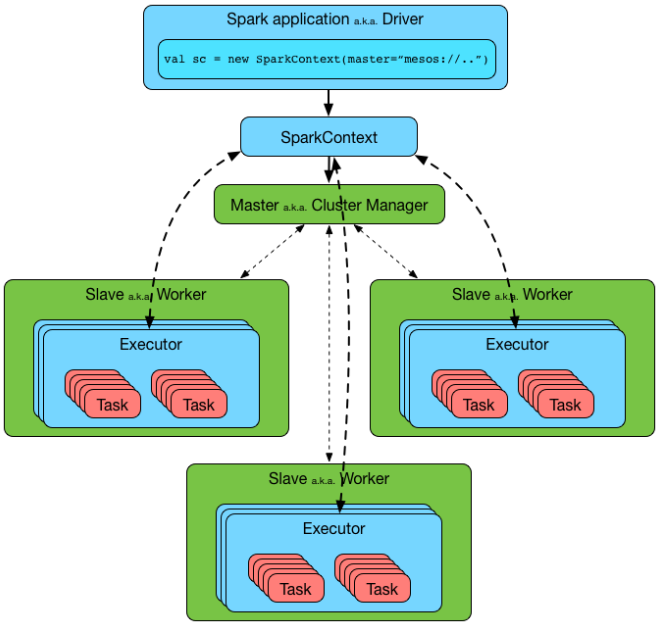

Fig. 2. Communication in Apache spark

Some of the components of the Apache Spark in cluster mode is described as follows:

- Driver program : It is the process that runs the main() function of the application and creates the SparkContext.
- Cluster manager : It is an external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
- Worker node : It is any node that can run application code in the cluster.
- Executor : It is a process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
- Stage : Each job gets divided into smaller sets of tasks called stages that depend on each other (similar to the map and reduce stages in MapReduce).This term is generally used in the driver's logs.

Some of the popular methods used in Pyspark are as follows:

- **Parallelize method**
  Parallelized collections are created by calling SparkContexts parallelize method on an existing iterable or collection in the driver program. The elements of the collection are copied to form a distributed dataset that can be operated in parallel.One important parameter for parallel collections is the number of partitions to cut the dataset into. Spark will run one task for each partition of the cluster. Generally,we want 2-4 partitions for each CPU in our cluster. Normally, Spark tries to set the number of partitions automatically based on our cluster. However, we can also set it manually by passing it as a second parameter to parallelize method.
- **RDD operations**
  Resilient Distributed Dataset (RDD) support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return a value to the driver program after performing a computation on the dataset. For example, map is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, reduce is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program.All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset. The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently.

III. EXPERIMENTS

For this project we have implemented 3 experiments: i) Singular Value Decomposition in a non-distributed manner on a single node on the local machine using Python ii.) Alternating Least Squares Method for Matrix Factorization on a cluster with 2 nodes using Apache Spark iii.) Singular Value Decomposition for Matrix Factorization on a cluster with 2 nodes using Apache Spark.
The dataset contain 17,700 Movies; 480,189 Users and 100,480,570 Ratings in total as seen in Figure 1 below. The dataset consists of user ratings where each user has rated the movies out of 5 but it also contains missing ratings. Thus, to build a recommendation system to predict those missing ratings.

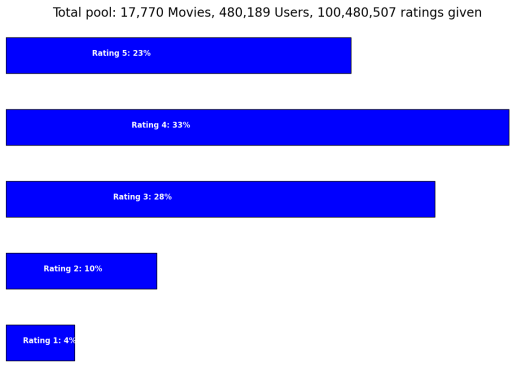

Fig. 3. Netflix Dataset

A. *Data Pre-Processing*

The Netflix Dataset was in a format such as that one row would be value of MovieId and all the consecutive lines after that ID were in a format "UserID,Rating". Thus, we had to first pre-process the data to bring it in "UserID,MovieID,Rating" format. Thus, we used Numpy and Pandas Dataframes to pre-process the data and saved the data to a csv file so that instead of pre-processing the data everytime we can just read the data whenever needed.

B. *Singular Value Decomposition using a single node on local machine*

We first implemented a code for running the matrix factorization on a single node. The code was written in Python and we used Singular Value Decomposition to conduct the matrix factorization. We implemented the non-distributed code in Python with 5 cross-validations. The accuracies, Root Mean Squared Error (RMSE), Mean Absolute Error(MAE), Fit Time , Test Time for each cross-validation.
The results of the Matrix Factorization on the Netflix Dataset on a single node are as follows:

As seen from Table 1, the average RMSE is about 83 percent on the test sets which were split randomly for each cross-validation. This gives us a good idea about the model

TABLE I
MATRIX FACTORIZATION RESULTS FOR SINGLE NODE(NON-DISTRIBUTED)

	Fold 1	**Fold 2**	**Fold 3**	**Fold 4**	**Fold 5**	**Mean**	**SD**
MAE (Test Set)	0.6404	0.6404	0.6403	0.6405	0.6403	0.6404	0.0001
RMSE (Test Set)	0.8359	0.8357	0.8356	0.8361	0.8356	0.8358	0.0002
Fit Time	4855.01	5047.06	4971.42	4940.95	6018.94	5166.68	430.55
Test Time	318.96	286.00	302.34	235.28	273.26	283.17	28.45

and seems to perform good. But the main disadvantage of this method is the execution time. The execution time for this method was about 29,077 seconds i.e. approximately 8 hrs. This is not at all feasible in the real world where movie recommender systems are used. Thus, the non-distributed implemented solution is definitely not the solution. Thus, our next step would be to parallelize the implementation using Apache Spark.

C. Alternating least Squares Method on Juliet Server with 2 nodes using Apache Spark

For implementing the Alternating Least Squares method using Apache Spark, we used the Alternating Least Squares package available in the Pyspark MLlib library. We first had to configure the parameters such as number of executors, number of cores per worker and executor memory. Thus, we to find the right number of executors we did a comparative study wherein we ran the ALS implementation for different number of executors and compared their performance. But, for implementing the ALS method we had to give the number of factors we wanted. But as the number of executors and cores were not dependent on the number of factors, we randomly gave 3 as the number of factors.

Number of Executors	Number of Cores/ worker	Time
10	2	8 min 13 sec
	4	5 min 37 sec
	6	3 min 50 sec
30	2	8 min 15 sec
	4	5 min 42 sec
	6	3 min 53 sec
60	2	8 min 23 sec
	4	5 min 46 sec
	6	3 min 57 sec

Thus, the best combination based on time performance was number of executors = 10 and cores per worker = 6. Now, that we had decided on the optimal parameters, we needed to decide the optimal number of factors needed for the best accuracy performance. Thus, we conducted a grid search for value of factors as 3,4,8,12. Thus, the accuracies obtained from them are given below:

Number of Factors	Accuracy
3	88.32 percent
4	87.89 percent
8	86.51 percent
12	86.19 percent

Thus, as we can see the best accuracy is obtained for 3 factors thus, we select 3 as the number of factors.

Thus, we execute our final model with executors = 10, cores per worker = 6 and factors = 3 and obtain an accuracy of 88.32 percent which performs in 3 minutes and 50 seconds.

D. Singular Value Decomposition Method on Juliet Server with 2 nodes using Apache Spark

In this approach, we have implemented our own implementation of SVD with minimal use of a ready to use package. Apache Spark does not have its own start to finish implementation of SVD but rather a package called svd available in the Pyspark MLlib library we nothing but just calculates the U, S and V matrix need for further prediction of the ratings.

While implementing this approach we faced a number of hurdles which we overcame as we went forward. The first hurdle was that the svd package takes a sparse pivot matrix as input but our data being huge creating and storing a huge sparse matrix was not feasible and resulted in a memory error. Thus, to overcome this, we first created the pivot table and converted each row into a Dense vector wherein it only stores the indices and values of non-zero elements of the row. Thus, after creating a matrix of Dense Vectors we were able to overcome the memory error.

The next step was to parallelize the matrix into RDD's which is the core concept of Apache Spark with which it achieves fast performance on huge data. Thus, we parallelized the Row Matrix and passed it to the svd package which resulted in the u, S and V matrix. To get the final ratings predicted by this method we implemented a distributed matrix multiplication to get the final matrix of ratings.

The final step was to conduct the testing of the predicted ratings. While conducting the testing, we had to extract each row from the dataframe created above using the take() method and compare it with the original row from the pivot table. But with the use use of the take() and collect() it brings all data from each RDD into the drivers memory first and then conducts the search. Thus after implementing various methods and searching the right optimal method and also playing with configurations we were not able to conduct the testing of the predictions that we have already calculated above. But to show that are testing approach is working we have tested it on a small dataset and it gives an accuracy of 64 percent which will definitely be less as it is not the full dataset but this just to show that our code for testing works but we have only this one hurdle for the entire dataset.

But the entire code upto the testing part has a time performance of 5 minutes and 23 seconds which compared to the non-distributed method is significantly very fast.

As we know, that we were executing the spark jobs on a remote server we were not able to access the User Interface which gives a pretty good overall picture of the spark jobs. But after a lot of efforts I was able to get a little part of the UI working by implementing port forwarding by forwarding the Spark UI port from Juliet server to my local machine and below are some images of the same.

These pictures show the line by line time taken and and number of tasks needed to complete each stage.

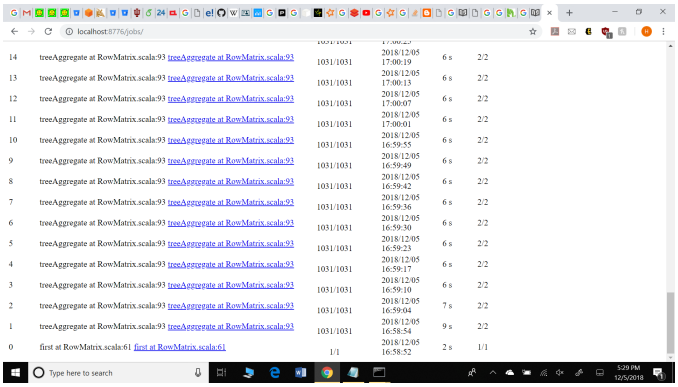

Fig. 4. Spark User Interface

Fig. 5. Spark User Interface

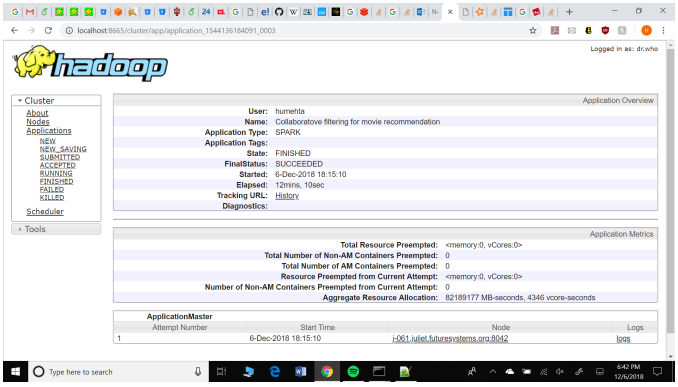

Fig. 7. Spark User Interface

Figure 5 is the User Interface for the Cluster Metrics for Hadoop which shows the Aggregate resource allocation and the average time a vcore is used.

IV. CONCLUSION

This project for High Performance Big Data has given us a lot of experience as to how do we implement big data problems in a distributed manner. It has led us to learn about Apache Spark which is one of the booming frameworks for Big data. By the experiments, conducted above we have seen that distributed implementation learns much faster than non-distributed methods. By this project we have learned how

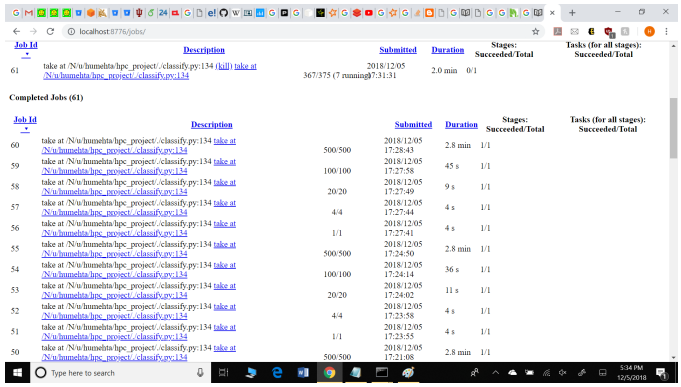

Fig. 6. Spark User Interface

to code in python for a distributed implementation which is different that usual python implementation. The importance of parameters like executors, cores per worker etc. We were able to conduct a comparative study between 3 things i) Singular Value Decomposition in a non-distributed manner on a single node on the local machine using Python ii.) Alternating Least Squares Method for Matrix Factorization on a cluster with 2 nodes using Apache Spark iii.) Singular Value Decomposition for Matrix Factorization on a cluster with 2 nodes using Apache Spark which helped us glean more into the aspect of distributed versus non-distributed computation.

We have also been able to learn that as we know Python only uses off-heap memory even though there space reserved in RAM for heap memory. Thus we can see that when we run a spark job both java and a python processes are started where java uses the uses the heap memory while python uses the off-heap memory. Also as we know that parallelism is about communication. Big partitions will under utilize resources and small partitions will over utilize the resources. Thus it is important to find a perfect balance between the two. One of the main observations during this project was that development and documentation of python based implementation of spark jobs is very less developed as compared to scala or java. Thus, at this moment it is better to implement spark in scala.

Synchronization in distributed systems comes with its own pros and cons. Data in distributed systems is distributed thus improper communication and synchronization may lead to huge losses in businesses. The processes running on different nodes implement and make decisions only based on data available with them thus proper aggregation and collection and communication is important. Also the main theorem in distributed computing is the CAP theorem which stands for Consistency, Availability and Partition Tolerance. All distributed systems should first be able to achieve these 3 principles and everything else falls in later.

V. ACKNOWLEDGEMENTS

We thank Professor Judy for giving us this opportunity and also the freedom to choose our own framework and projects and providing such a great learning experience.We

also would like to thank Langshi Chen & Selahattin Akkas for their continued assistance with the technical details of Spark & Hadoop Framework & providing valuable insights that helped us in our project.

REFERENCES

[1] Stephen Reddle, "Factorization Machines", Department of Reasoning for Intelligence, Osaka University, Japan, 1994

[2] S. Rendle and L. Schmidt-Thieme, Pairwise interaction tensor factoriza- tion for personalized tag recommendation, in WSDM 10: Proceedings of the third ACM international conference on Web search and data mining. New York, NY, USA: ACM, 2010, pp. 8190.

[3] S. Rendle, C. Freudenthaler, and L. Schmidt-Thieme, Factorizing personalized markov chains for next-basket recommendation, in WWW 10: Proceedings of the 19th international conference on World wide web. New York, NY, USA: ACM, 2010, pp. 811820.

[4] Spark architecture. Retrieved from https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-architecture.html

[5] Spark programming guide. Retrieved from https://spark.apache.org/docs/2.1.1/programming-guide.html

[6] Alternating Least Squares Method for Collaborative Filtering. Machine Learning Newsletter.(2014, April 19). Retrieved from https://bugra.github.io/work/notes/2014-04-19/alternating-least-squares-method-for-collaborative-filtering/

[7] Y. Koren, Factorization meets the neighborhood: a multifaceted collabo- rative filtering model, in KDD 08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining. New York, NY, USA: ACM, 2008, pp. 426434.

[8] Cluster Mode overview. Retrieved from https://spark.apache.org/docs/latest/cluster-overview.html