# Movie Recommendation System

# BIA 678B


- Mohit Ravi Ghatikar
- Vivek John Martins
- Xiangming Wang

# 1. Introduction

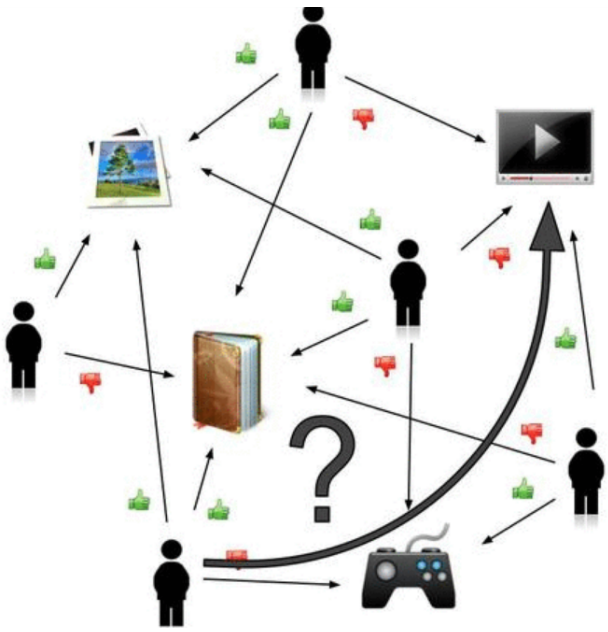
The initial goal of this project is to create a movie recommendation system using Spark MLlib and MapReduce.

We will work with the MovieLens dataset from the GroupLens Website. There are several datasets that consist of more than 1 million ratings, for more than 4000 movies, made by more than 6000 MovieLens users, who joined MovieLens from 2000 onwards. MovieLens is a recommender system and virtual community website that recommends movies for its users to watch, based on their film preferences using collaborative filtering. This benchmark dataset was released February 2003.

Our final goal is to test the Spark and MapReduce collaborative filtering algorithms based on processor time versus dataset size and observe how these algorithms adjust to scale.

## 1.1 What is Collaborative Filtering?

In Collaborative filtering, we make predictions (filtering) about the interests of a user by collecting preferences or taste information from many users (collaborating). The underlying assumption is that if a user A has the same opinion as a user B on an issue, A is more likely to have B's opinion on a different issue x than to have the opinion on x of a user chosen randomly.



The image to the side (from Wikipedia) shows an example of collaborative filtering. At first, people rate different items (like videos, images, games). Then, the system makes predictions about a user's rating for an item not rated yet. The new predictions are built upon the existing ratings of other users with similar ratings with the active user. In the image, the system predicts that the user will not like the video.

## 1.2 Why Spark?

Apache Spark is an open source project that has gained attention from analytics experts. In contrast to Hadoop MapReduce's two-stage, disk-based paradigm, Spark's multi-stage in-memory primitives provides performance up to 100 times faster for certain applications. Since it is possible to load data into a cluster's memory and query it repeatedly, Spark is commonly used for iterative machine learning algorithms at scale. Furthermore, Spark includes a library with common machine learning algorithms, MLlib, which can be easily leveraged in a Spark application. For an example, see the "Large-Scale Machine Learning with Spark on Amazon EMR" post on the AWS Big Data Blog.

Spark succeeds where traditional MapReduce approach fails, making it easy to develop and execute iterative algorithms. Many ML algorithms are based on iterative optimization, which makes Spark a great platform for implementing them.

Other open-source alternatives for building ML models are either relatively slow, such as Mahout using Hadoop MapReduce, or limited in their scale, such as Weka or R. Commercial managed services alternatives, which are taking most of the complexity out of the process, are also available, such as Dato or Amazon Machine Learning, a service that makes it easy for developers of all skill levels to use machine learning technology. In this post, we explore Spark's MLlib and show why it is a popular tool for data scientists using AWS who are looking for a DIY solution.

Spark MLlib library for Machine Learning provides a Collaborative Filtering implementation by using Alternating Least Squares. The implementation in MLlib has the following parameters:

- numBlocks is the number of blocks used to parallelize computation.
- Rank is the number of latent factors in the model.
- Iterations is the number of iterations to run.
- Lambda specifies the regularization parameter in ALS.
- Alpha is a parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations.

## 1.3 Why MapReduce?

MapReduce, which is inspired by functional programming has become a popular paradigm for data-intensive parallel processing on shared-nothing clusters. Since we had studied how to implement mappers and reducers for any big data problem in class, we wanted to the same for recommendation system too.

# 2. The story of the data

## 2.1 Data Source & Usage License:

GroupLens makes the MovieLens data sets publicly available for download at http://grouplens.org/datasets/. Initially we downloaded the ml-20m Data set. It contains 20,000,263 ratings by 138,997 unique Users for 26,812 unique Movies, over a time period stretching from January,1995 to March 2015. The Usage License agreement can be found at http://files.grouplens.org/datasets/movielens/ml-20m-README.html . Our projects wishes to abide by this license.

## 2.2 Exploratory Data Analysis:

User Ids : MovieLens users were selected at random for inclusion. Their ids have been anonymized. No demographic information is included. Each user is represented by an id, and no other information is provided. User ids are consistent between ratings.csv and tags.csv (i.e., the same id refers to the same user across the two files).[1] There were 138,997 Unique Users found within the dataset.

The table below displays the Top 5 Most Active Users

| **UserId** | 118205 | 8405 | 82418 | 121535 | 74142 |
|---|---|---|---|---|---|
| **Frequency** | 9,322 | 6,961 | 5,708 | 5,520 | 5,480 |

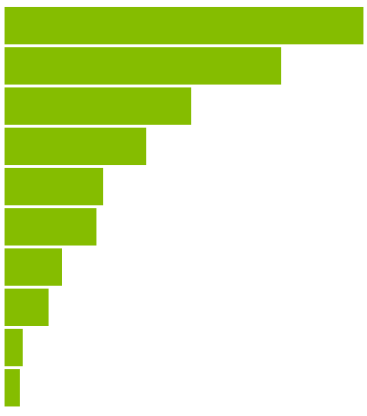
Movie Ids : The data cotained a total of 26,812 unique Movies. Only movies with at least one rating or tag are included in the dataset. These movie ids are consistent with those used on the MovieLens web site (e.g., id 1 corresponds to the URL https://movielens.org/movies/1). Movie ids are consistent between ratings.csv, tags.csv, movies.csv, and links.csv (i.e., the same id refers to the same movie across these four data files). [1] .

| **Movie** | **# of Rating** |
|---|---|
| Pulp Fiction (1994) | 67,310 |
| Forrest Gump (1994) | 66,394 |
| Shawshank Redemption, The (1994) | 63,366 |
| Silence of the Lambs, The (1991) | 63,299 |
| Jurassic Park (1993) | 59,715 |

The table to the right descirbes the Top 5 Most Rated Movies

Ratings : All ratings are contained in the file ratings.csv. Each line of this file after the header row represents one rating of one movie by one user, and has the following format: userId,movieId,rating,timestamp.

| Value | Count | Percent |
|---|---|---|
| 4 | 5,561,926 | 27.809% |
| 3 | 4,291,193 | 21.456% |
| 5 | 2,898,660 | 14.493% |
| 3.5 | 2,200,156 | 11.001% |
| 4.5 | 1,534,824 | 7.674% |
| 2 | 1,430,997 | 7.155% |
| 2.5 | 883,398 | 4.417% |
| 1 | 680,732 | 3.404% |
| 1.5 | 279,252 | 1.396% |
| 0.5 | 239,125 | 1.196% |



Ratings are made on a 5-star scale, with half-star increments (0.5 stars - 5.0 stars).

Timestamps represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970. [1]

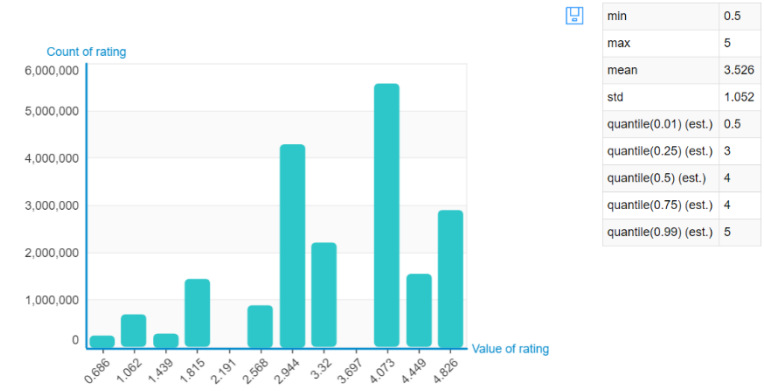
For our analysis timestamps were not utilized and hence deleted from the datasets to help processes run faster.

## 2.3 Data Preparation:

Reading the data : Spark reads the data as textfiles for quick and easy file reading. On working with the individual datasets we became aware that the delimiters in each dataset varied: Some utilized ',' , '::' and others utilized tabs to delimit.

Singular Value Decomposition: We did not run SVD to prepare the data as the Spark Mllib recommender ALS function runs the SVD within the function itself.

Data Splitting Issues : On trying to split the data into Training and Test Datasets , we found a unique issue. After running the model for both 100,000 and 1,000,000 ratings we obtained similar unusually high root mean squared error values. We could only conclude that the algorithm did not have suffiecient data to learn from. Therefore the problem must lie with the data split.




| | |
|---|---|
| min | 0.5 |
| max | 5 |
| mean | 3.526 |
| std | 1.052 |
| quantile(0.01) (est.) | 0.5 |
| quantile(0.25) (est.) | 3 |
| quantile(0.5) (est.) | 4 |
| quantile(0.75) (est.) | 4 |
| quantile(0.99) (est.) | 5 |

Unlike other datasets this dataset cannot be split , because for the recommendation engine to work, we must have a minimum number of ratings from each user and a minimum number of ratings for each movie. Grouplens provides users which were selected at random for inclusion. However, all selected users had rated at least 20 movies and only movies with at least one rating are included in the dataset. Therefore inorder to continue our analysis we utilized the ml-100k, ml-1m and ml-10m datasets. (*Please note: The ml-20m dataset was left out as we encountered errors due to the large volume of data.)*

# 3. Recommender Systems with Spark

## 3.1 The Model:

Alternating Least Squares Method :

When using a Matrix Factorization approach to implement a recommendation algorithm you decompose your large user/item matrix into lower dimensional user factors and item factors. In the most simple approach you can then estimate the user rating (or in general preference) by multiplying those factors according to the following equation:

$$r'_{ui} = p_u^T q_i \ (1)$$

In order to learn those factors you need to minimize the following error function:

$$min_{q,p} \sum_{u,i} (r_{ui} - p_u^T q_i)^2 \ (2)$$

Alternating Least Squares (ALS) represents a different approach to optimizing the error function compared to gradient descent. While gradient desccent moves in the direction opposite to the gradient inorder to reach the minima, ALS uses the following key insight: The non-convex optimization problem in Equation (2) can be converted into an "easy" quadratic problem if you fix either pu or qi. ALS fixes each one of those alternatively. When one is fixed, the other one is computed, and vice versa.[2]

What makes ALS so special? :

1. Easy to Parallelize: With the data distributed Gradient Discent cannot function as it requires all the data to find the minima. However ALS benefits with parallelization due to the easy technique mentioned before.
2. Practicality: When dealing with sparse, specially of large sized datasets (order of Billions) ,Gradient Discent is simply not practical.

## 3.2 Results:

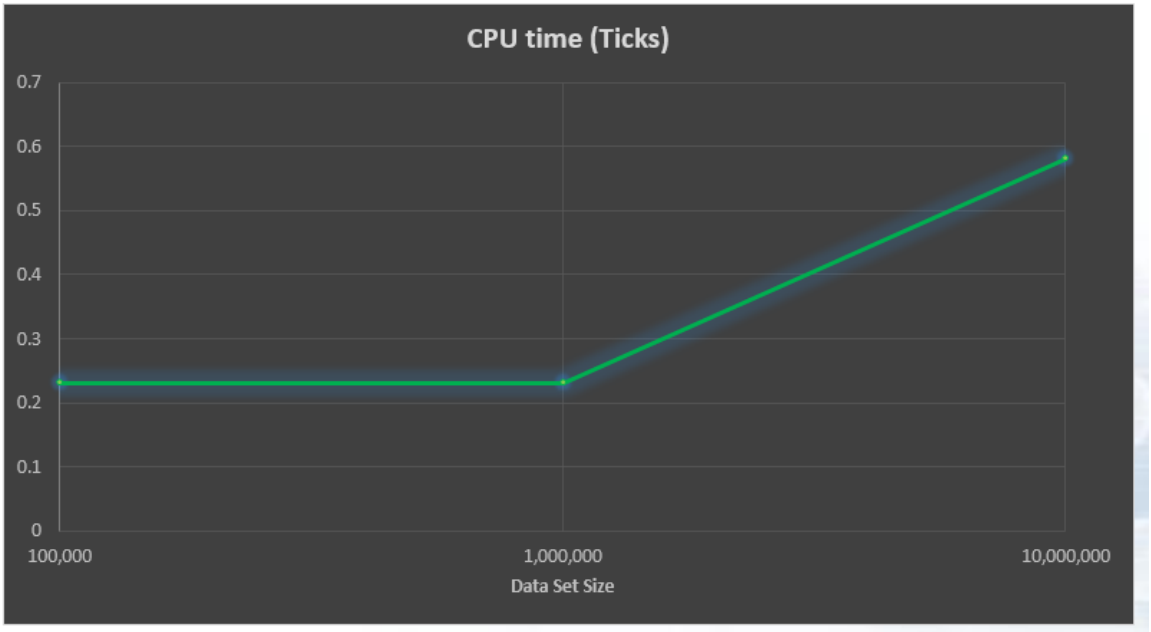
Factorized Matrix Rank : In Spark's MlLib, the options to vary parameters like the Rank of the singular value decomposed matrix end up being very useful. We ran our model for the 3 different dataset sizes and allowed the rank to vary between 4,8,12. It is visible that with the increasing size of

| Size of Data (# records) | CPU time (Ticks) | Rank | RMSE |
|---|---|---|---|
| 100,000 | 0.23 | 4 | 0.934 |
| 1000000 | 0.23 | 8 | 0.8685 |
| 10,000,000 | 0.58 | 12 | 0.8196 |

data the best rank utilized increased. This is because larger datasets would have higer variance and a higher rank would explain more of that variance.

RMSE : The analysis also showed that with more data there was an improvement with the Root Mean Squared Error (RMSE). The explanation is simple: there is more data to learn from and hence the error of the ALS model reduces.

CPU Time : Lastly the table and graph in this section show the relation of dataset size with CPU time. Here we see that the model runs equally fast for both 100K and 1M




datasets. This shows the algorithm cannot get any faster.

*(The CPU time was calculated for training , optimizing the parameters and predicting the ratings)*

This can be explained by the fact that, what Spark cannot fit in memory, it writes to disc. Later it reads the data onto memory and performs the task. Writing to and Reading from disc is slower than processing in memory (despite the machine utilizing a solid state drive). Since the Virtual Machine we utilized to run Spark was limited to 4GB of allocated RAM, therefore the bigger 10M data set took longer to process.

# 4. Recommender Systems with MapReduce

## 4.1 The Design:

In our project, we used item based collaborative filtering in MapReduce. Mrjob library in python was used to define mappers and reducers and later Amazon Web Service (EC2) was utilized to measure the performance for large data sets. Our objective was to find every pair of movie that were watched by the same person and measure the similarity of ratings across all users who watched both sets of movies. In the end, the movies were sorted using the similarity strength. We measured similarity between movies to be the cosine similarity.

The design of mappers and reducers is described below. It can be divided into 3 stages.
In Stage-1:
The mapper extracts the User as the key and the (Movie, Rating) pair as the values. In the reducer, it groups all the (Movie, Rating) pair by users. So at this stage, we will have for every user, every movie they have watched and the rating they gave for every movie.

In Stage-2:
The output of the reducer in stage-1 is fed into the mapper of stage-2. This is where the bulk of the work happens. The mapper outputs every pair of movie that each user watched. From a list of movies each user watched, the mapper constructs every permutation of movie pairs.
(Movie1, Movie2) ---> (Rating1, Rating2)
Therefore, the key is the movie pair and the values are its corresponding rating pair.

In the reducer part, it computes the similarity between the movies using cosine similarity for each movie pair.
(Movie1, Movie2) ---> (Similarity, the number of users who watched both movies)

In Stage-3:
In the final stage, we manipulate the mapper and reducer to sort the results. We can make the movie name and the similarity score the sorting key. The reducer displays the final sorted output.

The algorithm was implemented on 100k ratings dataset. A sample output is shown below.

```
"Star Wars (1977)"      ["Night Falls on Manhattan (1997)", 0.9795749380425283, 28]
"Star Wars (1977)"      ["Princess Caraboo (1994)", 0.9805675486863789, 15]
"Star Wars (1977)"      ["Raiders of the Lost Ark (1981)", 0.981760098872619, 380]
"Star Wars (1977)"      ["Some Folks Call It a Sling Blade (1993)", 0.9823449614960231, 40]
"Star Wars (1977)"      ["Blue Sky (1994)", 0.983771402943414, 11]
"Star Wars (1977)"      ["Primary Colors (1998)", 0.9839386850044775, 11]
"Star Wars (1977)"      ["Meet John Doe (1941)", 0.9851718095561594, 24]
"Star Wars (1977)"      ["Return of the Jedi (1983)", 0.9857230861253026, 480]
"Star Wars (1977)"      ["Empire Strikes Back, The (1980)", 0.9895522078385338, 345]
"Stargate (1994)"       ["Extreme Measures (1996)", 0.9502307311756256, 18]
"Stargate (1994)"       ["Sneakers (1992)", 0.9503133273383328, 70]
"Stargate (1994)"       ["Hackers (1995)", 0.9503383521783552, 22]
"Stargate (1994)"       ["Juror, The (1996)", 0.950381926622983, 16]
```

We can see that for the movie Star Wars, Empire strikes Back and Return of the Jedi are the most similair movies. Also, a fairly high number of users have watched both movies (345 and 480 respectively).

### 4.2 MapReduce Scaled Up:

The 100k ratings was run on an individual machine and the elapsed time was around 16 minutes. We decided to test how MapReduce would perform on large datasets. To accomplish this, we used Amazon Web Service (EC2) to find out the taken taken.

| Dataset | AWS EC2 | Time Taken |
|---|---|---|
| 100k | 1 machine | 50 mins |
| 100k | 4 machines | 26 mins |
| 1 million | 8 machines | 2.5 hrs |
| 1 million | 16 machines | 1.1 hrs |

Initially, we ran the same 100k ratings dataset using a single machine on EC2. As we can see, the elapsed time of 50 minutes is much greater than 16 minutes. This is because, there is a lot of overhead costs involved in transferring the data from the computer to the cloud and again, the results back from the cloud back to the computer. By using 4 machines, the time taken is cut into half. Similarly, the elapsed time is cut into half for a 1 million ratings dataset, when the nodes are increased from 8 to 16.

## 5. Conclusion

We recommend using Spark rather than MapReduce, as Spark's code is already optimized to run recommender systems. We found the recommendations in most cases

to be sound. In future we would like to run the Spark code on a cluster and measure its performance with varying processing cores. We also recommed using the cloud services like AWS for large datasets. If our dataset is small, then a single computer is usually powerful enough to handle it.

## 6. References


[1] http://files.grouplens.org/datasets/movielens/ml-20m-README.html

[2] https://www.quora.com/What-is-the-Alternating-Least-Squares-method-in-recommendation-systems

https://en.wikipedia.org/wiki/Recommender_system