# CSE 427S Final Project - Collaborative Filtering using Netflix Data

Brad Hodkinson, Patrick Di Rita, Hongyu Meng
bradh@wustl.edu, pdirita@wustl.edu, menghongyu@wustl.edu

11 December, 2019

## 1 Motivation

Our project revolves around a problem originally presented by Netflix in 2009: analyze a subset of Netflix data comprised of users' ratings for different movies and implement a collaborative filtering algorithm to predict a user's enjoyment of a particular movie based on the user's ratings of other, similar movies. Such a problem may seem fairly insignificant to an outside observer, but on the contrary, Netflix offered $1 million to the group who developed the most accurate prediction algorithm back when the problem was originally presented. Large companies such as Netflix, Amazon, YouTube, etc. care so much about content recommendation due to the fact that accurately recommending a user products/content that they are likely to buy/watch increases revenue and keeps the user within the company's ecosystem, as the user is always seeing new content that they are likely to enjoy.

Problems such as these require very large datasets to both train the models and evaluate their effectiveness. We have a dataset with a huge volume of data points(over 3 million), each structured in a fixed pattern. This causes problem falls under the umbrella of "Big Data Analysis." The consequence of working with big data is that the recommendation algorithms must be run on the cloud using a service with a distributed file system such as Amazon EMR, as personal computers or small clusters are simply not intended for dealing with the amount of processing and memory required. Distributed file systems work by replicating and partitioning data across a cluster of worker nodes, with everything controlled by a master node. The data replication gives the system high failure tolerance, while the distribution across the worker nodes allows for computations to be executed in parallel.

Our Netflix recommendation problem is famous in the world of Big Data Analysis and Cloud Computing not only due to the large monetary reward associated with it, but also due to the fact that it is an issue that each and every large e-commerce platform and entertainment streaming service must face in order to most effectively compete with other companies in the same industry.

## 2 Documentation of Approach

The first step in tackling this problem was to determine our similarity measure, which is a metric that determines how similar two users/items, $x$ and $y$. It is based on the ratings for each item, $r_x$ and $r_y$. The three most commonly used similarity metrics are Jaccard similarity (Equation 1), cosine similarity (Equation 2) and Pearson correlation (Equation 3).

$$J(x,y) = \frac{|A \cap B|}{A \cup B} = \frac{|O_{xy}|}{|A| + |B| - |A \cap B|} \tag{1}$$

Where $A$ and $B$ are the sets of rated items of users $x$ and $y$, respectively, and $|O_{xy}|$ is the size of the overlap of $A$ and $B$.

$$C(x,y) = \frac{r_x^T \cdot r_y}{||r_x|| \cdot ||r_y||} = \frac{\sum_{i \in O_{xy}} r_{xi} r_{yi}}{\sqrt{\sum_{i \in O_{xy}} r_{xi}^2} \sqrt{\sum_{i \in O_{xy}} r_{yi}^2}} \tag{2}$$

Where $r_x$ and $r_y$ are vectors of overlapping ratings and $i$ are items rated by both users

$$P(x,y) = \frac{\sum_{i \in O_{xy}} (r_{xi} - \bar{r_x})(r_{yi} - \bar{r_y})}{\sqrt{\sum_{i \in O_{xy}} (r_{xi} - \bar{r_x})^2} \sqrt{\sum_{i \in O_{xy}} (r_{yi} - \bar{y_x})^2}} \tag{3}$$

Where $\bar{r_x}$ is the average rating of user $x$

In our case, we decided to use cosine similarity. Our reasoning for this was that Jaccard similarity does not take into account the value of the rating (meaning it is less accurate), and Pearson correlation requires extra computation of averages.

The collaborative filtering algorithm computes a top-N-list of the $k$ most similar users/items each specific user/item in the dataset. Choosing too low of a value for $k$ may cause this top-N-list to be overly biased towards popular items and completely ignore more niche items, while choosing too high a value for $k$ will damage efficiency and cause the final predictor to be misled by products that have few ratings assigned. After considering this trade-off, we decided to choose a $k$ value of 30, meaning an item that a

user has not yet rated has its rating predicted by comparing that item to the top 30 most similar items. To do this prediction, we use a weighted average where the weights are said similarity scores. This weighted average is calculated using Equation 4

$$r_{xi} = \frac{\sum_{y \in N_i} cos(x,y) \cdot r_{yi}}{\sum_{y \in N_i} cos(x,y)} \tag{4}$$

Where $y \in N_i$ is the set of the top-k most similar users/items to $x$, and $r_{yi}$ is the rating of $y$

The final decision we had to make was that of our model: user-user (which computes a similarity matrix comparing the similarity of each user to that of every other user) or item-item. We initially chose user-user due to our familiarity with the model. This choice was fairly shortsighted, as we did not consider the disparity between the number of users and the number of items. Attempting to compute the user-user similarity matrix proved to be extremely inefficient in both memory and computation due to the fact that the number of users in the Netflix dataset is substantially greater than the number of items (movies). We were able to drastically improve the efficiency of computing the similarities by switching to the item-item similarity model.

We decided that Apache Spark with implementation in Python 2.7 best fit our needs for the project due to its speed, the ability to keep data in memory, high level of abstraction, and interactive shell. Spark utilizes data structures known as Resilient Distributed Datasets (RDDs) to store data. Operations on RDDs are categorized as either actions, which return a value by executing a job, or as transformations, which define a new RDD based on the current ones. Operations on RDDs are extremely efficient due to Spark's parallelization, lazy execution (data in RDDs is not processed until an action is performed), and pipelining (performing sequences of transformations by row so that no data is stored, only preforming operations on data records that are necessary to produce the return value).

Our implementation used two text files of data, TrainingRatings.txt and TestingRatings,txt, each with data points of the form (movieID,userID,rating). In order for our model to be valid, we need to avoid snooping on the data in the testing ratings file before making our model, so we must compute a similarity matrix using only the data in TrainingRatings.txt. We did this through a series of transformations on the training data that equate to calculating Equation 2 for each distinct item-item pair (ensuring that we have no duplicate similarities such as (**itemA**,**itemB**) and (**itemB**, **itemA**)).

After this item-item cosine similarities matrix had been computed, we iterated over each individual item, $x$, in our training set, to find the top $k$ items most similar to item $x$. Then we stored the results in an RDD. generated our model to compute predictions for the unknown ratings in the test set.

We then imported the test set from TestingRatings.txt into an RDD, but we omitted the rating value from the data (so we are left with (movie,user) pairs. These omitted ratings are the "gold standard" (i.e. true) ratings, and were only used to evaluate how well our model predicts the unknown ratings. This prediction was evaluated for each (**m**,**u**) pair in the test set as following:

- Filter the RDD containing the top-k similarities for each movie to obtain the top-k most similar movies to **m**
- Filter the training data to obtain all of the ratings that $u$ has already made
- Compute user $u$'s predicted rating for movie $m$ as a weighted average of previous ratings and similar movies using Equation 4 (only for the movies in the top-k-list that also been by rated by $u$)

After predictions are generated, we evaluated our model using Mean Absolute Error (Equation 5) and Root Mean Squared Error (Equation 6) by passing all the ratings generated so far and the "golden standard" ratings into our error function. This allows us to see how our model works at each iteration, and the total error will be calculated once all predicted ratings have been found.

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |y_i - x_i| \tag{5}$$

$$RMS = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - x_i)^2} \tag{6}$$

Where $n$ is total number of predictions, $y_i$ is the "golden standard" value for rating $i$, and $x_i$ is the prediction of rating $i$ [1], [2].

In addition to this cosine similarity based algorithm, we also implemented an algorithm using the Alternating Least Squares (ALS) utility from the pyspark.mllib.recommendation package. This algorithm is paramaterized by a training ratings file, testing ratings file, rank (number of features in the data) and number of iterations. The algorithm generates predictions and evaluation metrics as follows:

- Load training ratings file into an RDD

- Train model by calling **ALS.train()** with the training ratings RDD, rank, and number of iterations. The ratings matrix is approximated as the product of two lower-rank matrices of a given rank (number of features). To solve for these features, ALS is run iteratively with a level of parallelism automatically based on the number of partitions in ratings [3].
- Load testing ratings file into an RDD
- Call **model.predictAll()** function on our testing data to get rating predictions.
- Generate evaluation metrics (MAE, RMS) by calling our error function

We allowed for our program to easily switch between the two algorithms through the number of command line arguments passed into the system. If there were only two arguments (training file and testing file paths), then our program knew to run our initial implementation. If there were four arguments (training file, testing file, rank, iterations), then the ALS function was called.

# 3 Application

As is standard in the development timeline of a cloud computing application, first tested locally and on the pseudo-cluster with (small) toy data in order to find any errors within our program. Once we are sure our algorithm worked properly and efficiently, we moved to computing with big data on an Amazon EMR cluster.

## 3.1 Small Data/Pseudo Cluster

Testing with a small, representative sample of toy data both locally and on the pseudo cluster helped to expose both logical errors and efficiency bottlenecks. Many of these bottlenecks were caused by the use of the **collect()** action (which aggregates an RDD into a Python list) at points when it was not necessary, using **lookup()** on large RDDs, and not persisting RDDs that were referenced many times into memory.

To solve those problems, we replaced many of the **collect()** and **lookup()** calls with filter transformations in order to more efficiently find specific elements within our RDDs, and we calculated the cosine similarity matrix fully using RDD transformations instead of an iterative process. Once our algorithm was properly optimized, we obtained the following final error measures:

**MAE** = 1.8947
**RMS** = 1.5323

These error measures are sub-optimal, but this is not an issue. The sub-optimality of the predicted ratings on our toy data stems from the small volume of toy data available for the algorithm, and the fact that our algorithm worked at all and output evaluation metrics properly means that we are ready to run on our big dataset.

As an aside, we were unable to test the ALS solver on our toy data locally due to an issue installing NumPy on the local machine. This was not an issue, however, as we tested the ALS implementation on a different machine on some randomly generated data in order to ensure it preformed as desired.

Once we were confident in our algorithm's ability to produce proper predictions, we moved on to testing the entire Netflix subset data on the Amazon EMR cluster.

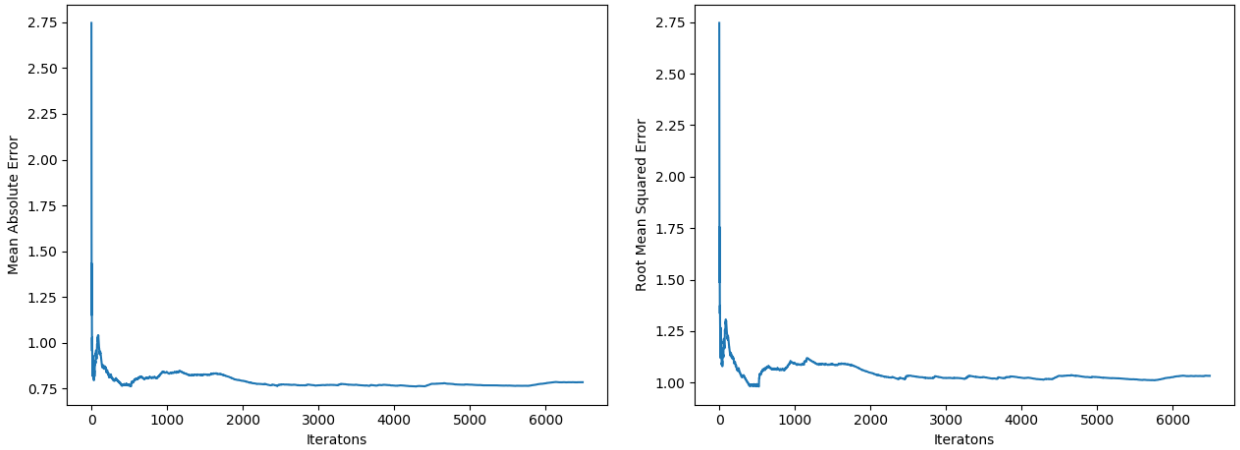
## 3.2 Big Data

We set up our Amazon EMR cluster to have a master node with a m5.xlarge instance (4 cores, 16GB memory, 64GB storage) and four core nodes with the same m5.xlarge instances. To submit a job, we added a Spark Application step to the cluster and passed in the locations of our training and text files (along with rank and number of iterations when running the ALS solver) as arguments. We were then able to monitor the Spark application through either the EMR applications tab or the YARN Resource Manager.

Running our item-item cosine similarity recommender system on the entire dataset resulted in a much longer execution time than originally expected, so we decided to terminate the program early and analyze the intermediate results. We graphed the data received from each intermediate result, which can be seen in Figure 1

We obtained a minimum MAE of 0.76095 and a minimum RMS of 0.9807, with a final (iteration 6490) MAE of 0.784556 and final RMS of 1.03263. It is clear that both error values seem to level out around 2500 iterations, so we believe that, even though we were unable to predict ratings for all datapoints in TestingRatings.txt, our error would not have improved much more than this.

When running the ALS implementation on our whole data, however, we did get a final evaluation metric output in about a minute. This is to be expected, as the ALS module is dependant on NumPy, which does an incredible job of parallelizing matrix operations. Due to the speed of the execution, we ran the ALS function on multiple different combinations of rank and number of iterations to find an optimal combination. The outputs for all such trials can be found in Table 1.




(a) Graph of Mean Absolute Error with respect to iterations

(b) Graph of Root Mean Squared Error with respect to iterations

Figure 1: Graphs of MAE and RMS with respect to iterations. 6490 iterations completed over 2.5 hours. Minimum MAE: 0.760953285, Minimum RMS: 0.980700737

| Number of features | 10 iterations | 15 iterations | 20 iterations | 25 itetations |
|---|---|---|---|---|
| 9 | MAE: 0.6508<br>RMS: 0.7024 | MAE: 0.6513<br>RMS: 0.7036 | MAE: 0.6490<br>RMS: 0.6981 | MAE: 0.6495<br>RMS: 0.6995 |
| 20 | MAE: 0.6675<br>RMS: 0.7508 | N/A | N/A | N/A |

Table 1: Table of ALS feature and iteration combination errors

We quickly discovered that 9 features was superior to 20, which accounts for the lack of output for 20 features after the 10 iteration trial. We determined that our optimal number of features was 9 and the optimal number of iterations was 20 by generating the graph found in Figure 2.

In addition to testing these methods on the ratings found in TestingRatings.txt, we also generated a set of personalized ratings for movies we had seen, and after training the model on TrainingRatings.txt with some personal ratings appended, ran our models to predict our own personal for movies our personal userID had not rated in the training set. Interestingly, after generating personal ratings for over 50 movies, both of our models output MSE and RMS of 0. This seemed strange, and upon further investigation, we discovered that none of the movies we had rated appeared in the original training dataset. This brought to light the issue of the cold-start problem, which occurs when items added to the catalog have either none or very little interactions. Collaborative filtering algorithms such as our item-item model and ALS rely on interactions between items to make recommendations, and when an item is brand new, there is no overlap.

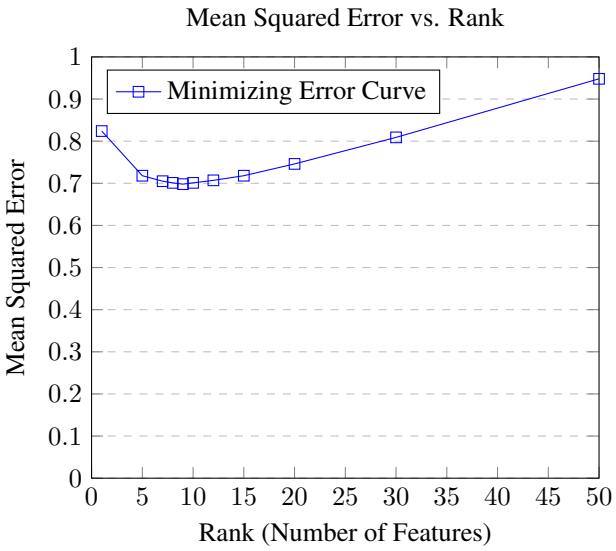



Figure 2: Mean squared error at different ranks for 20 iterations. Minimum MSE found at rank 9

# 4 Conclusion and Future Work

We worked on the Netflix recommendation list task by building model, such as choosing item-item measure method and cosine similarity; selecting tools, such as python and SPARK; and testing code with training set and testing set. Graphs of MAE and RMS respect to iterations, and MSE to ranks are shown in results part. We also compared how different algorithms will affect the final result, with testing cosine similarity versus ALS algorithm.

When comparing the two algorithms, it is clear that ALS beats our cosine similarity implementation in not

only in efficiency, which was to be expected, but also in evaluation metrics. Because of this, it is safe to conclude that item-item cosine similarity recommendation is not the most optimal algorithm for predicting user ratings. In lieu of this fact, however, we do not consider our project a failure, as we did obtain error metrics that were within an acceptable range. Future work on our algorithm would mainly consist of improving the efficiency even further, as there is clearly a bottleneck somewhere within the portion of our algorithm where the predictions are being made (evident by observing where in the code YARN Resource Manager shows jobs are being instantiated).

Differences aside, both methods are under the umbrella of collaborative filtering algorithms, which means they both suffer when new items are added due to the cold start problem. This issue can be overcome by using metadeta (genre, actors, director, etc.) of newly added movies when generating rating predictions, and implementing this metadata into our model is something that could be focused on in the future.

# References


[1] *Mean absolute error*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Mean_absolute_error.

[2] *Root-mean-square deviation*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Root-mean-square_deviation.

[3] *Spark mllib documentation*, 2019. [Online]. Available: https://spark.apache.org/docs/2.2.0/api/scala/index.html#org.apache.spark.mllib.recommendation.ALS.