# Final Project Report

# Mining of Massive Datasets

Brian Carr – 7120309055, Elias Geisseler – 7120309030
Charlie Duan 段粲超- 5113709185, Tan Jia [illegible]嘉- 5113709081

## I. Introduction

For this project the team was required to construct a recommender system which, using the provided movie rating data, would generate predictions for the unknown ratings provided

## II. Problem Description

The team was provided with rating data in two files, training.txt, and test.txt. The provided rating data in the training.txt file contained only user IDs, movie IDs, date IDs, and a rating value for each rating. The ratings set contained 100,000 unique user IDs, 17770 movie IDs, and5115 date IDs (which represent consecutive days, e.g. x+1 is the day after x). The ratings provided were on a scale from 1 through 5, inclusive, and all integers. The data in the test.txt file was similar to this except the ratings were removed. All IDs remained intact in these ratings though. The recommender system was to generate predictions for these rating values, rounded to the nearest tenth.

## III. Solution and Discussion

### Collaborative Filtering Model

The team constructed a multi-model solution employing several different well known data mining strategies. These models included several different forms of collaborative filtering, as well as gradient descent. Each model was trained on the training data independently, then fed the test data to generate predictions for. Finally, all of these predictions were weighted and aggregated together, to generate the final predictions for submission.

The collaborative filtering model was constructed to be highly modular, thus allowing easy modification for different types of collaborative filtering. As a result, once the first collaborative filtering model, a user-based system, had been constructed, it was straightforward to generate additional collaborative filtering models, which included movie-based, as well as inverse user-based, and inverse movie-based systems.

The user-based and movie-based systems function largely as one would expect. They create rating vectors for each user or movie, respectively, find the N most similar vectors using locality-sensitive hashing to perform a kNN search, and then generate ratings based on those of their kNN results. The results are generated using a baseline prediction with temporal biases, , multiplied by the similarity factor for the two vectors. The sum of each of these results is then divided by the total of all the similarity scores used. The baseline is essentially a measure of the deviation from the mean both for the user, and for the item, and it also takes into account drift in those deviations over time. The similarity factor used was a weighted aggregation of the cosine distance and the pearson correlation coefficient.

A couple further optimizations are made, including ignoring users below a certain similarity threshold, even if they are part of the k nearest neighbors, and also ‘hedging our bets’ so to speak by pulling the predicted rating towards the baseline by a factor dependent on the number of useful vectors we are actually able to utilize when generating the rating (i.e. if there are not many useful vectors, we will pull the prediction back towards the baseline in a manner inversely proportional to the number of useful vectors found). These optimizations are meant to prevent minimize the impact of making bad predictions when little useful information is available, while still allowing predictions to be made normally when there is enough information to have higher confidence.

The inverse collaborative filtering systems operate in much the same way, but instead they try to exploit dissimilar users: essentially making positive predictions for one user when another dissimilar user gives a negative rating and vice versa. To accomplish this the similarity factor (which had a range from -1 to 1) was multiplied by negative one so that similar users now had a very low similarity score, and dissimilar users a very high score. Then the locality-sensitive hashing kNN was applied exactly as above. Finally, ratings were generated as above, but once they had been generated, their offset from the mean was taken and they were flipped to the opposite side of the mean. This approach was found to be slightly less accurate (generally a RMSE of a few tenths behind) than the standard collaborative filtering models, but since it provided strong predictions for different ratings than the standard collaborative filtering models, it proved useful when aggregated with the results of the other models.

## Latent Factor Model

The Latent Factor Model with time based biases was the other model constructed by the team. Initially we had bad results caused by the fact that all feature vectors started out with 0 and therefore never corrected or converged. To solve this problem we projected the result onto a Sigmoid scale (excluding 0 – 1 excluding), and also causing a random value to be added to the feature vectors. The Sigmoid approach was taken since a different model of a neural network was using a working method to solve the 0 problem. This other model, however, was not used in the final system and was thus removed. The Latent Factor model was able to reach a local split RMSE of 0.91 with 50 features. Adding time dependence and parallelizing the model training contributed significantly to the efficacy for training.

### Aggregation of Model Results

After the predicted ratings of each individual model had been generated they were multiplied by a weight, and then aggregated together to compute the final result. First, the four collaborative filtering models were aggregated together. Each predicted rating was logged, along with the sum of the similarity scores of all vectors used to generate that rating. This data was taken from each of the four models, and the total similarity scores for each rating from each model were then summed together. The rating was multiplied by its own similarity total and then divided by the sum of all four totals. This produced a result weighted by the amount of confidence (judged by the 'amount of similarity' used when predicting that rating') in the rating so that ratings from a model with high confidence on that particular prediction would factor into the final prediction more heavily than one that had low confidence on that prediction. After this aggregation was complete, the results were weighted again and aggregated with the results of the gradient descent model to produce the final output.

## IV. Implementation Details

All the code for this project was implemented in Java 1.8, with the exception of the shell script for automated runs. A simple framework was constructed to facilitate interaction between the models and the data. This allowed the team to easily perform splits of data, run training sessions, test the RMSE and publish results.
To handle the Math we used Java internal packages as well as external ones. The Latent Factor Model for example uses Apache commons linear algebra. Matrix Toolkit for Java, which is a wrapper around BLAS was also used for efficient matrix computations. Additionally, we used a customized version of an open source locality-sensitive hashing library, TarsosLSH. This modified version has certain functionality which is not necessary for our purposes removed in order to drastically decrease memory usage. Without these modifications the system would have required hardware with significantly more RAM than most consumer grade systems have in order to run.

## V. Challenges

The team faced a number of challenges over the course of this project. The data set was large enough that it forced programmers to be conscious of both space and computational efficiency. Otherwise, there would not be enough RAM to run the system, or predictions would take an unacceptably long time. There were often tradeoffs that had to be made in favor of efficient space usage at the expense of computation time due to limited memory.

Additionally, the team struggled greatly with inconsistencies when testing locally on a subset of the training data, and when submitting actual test predictions to the test site. Our system was consistently able to achieve RMSEs of around 0.90 – 0.92 on subsets of the training data, and as low as 0.86 with one subset, while struggling to even break below 1.03 on the test site. We are unsure if this is an overfitting problem(as suggested by the TA), or if something with our submission format is somehow incorrect.

***RMSE gap***

For testing purposes, a subset of the given training.txt file was used to perform local tests. This subset was not included in the training data for these local tests. The local training set was sampled randomly over the whole training set. However, the team noticed that this method did not provide a reasonable approximation of the data on the testing site as our scores were consistently much higher on the local tests with various subsets of training.txt. It was noticed that training.txt did share the same distribution of ratings with respect to dateId as testing.txt. See Illustration 1, a histogram of the training.txt in 20 day bucket size. (y = numberOfRatings ; x = day /20)

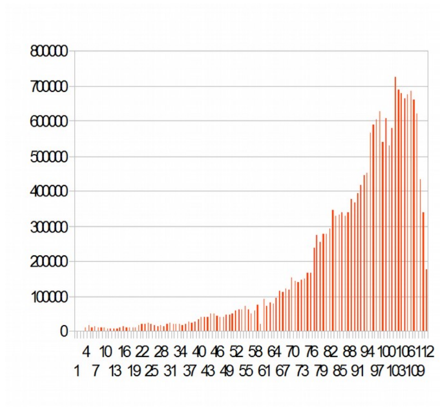

*Illustration 1: training.txt*

The test set is different, as seen in the Illustration 2 with the same 20 Day bucket Size.

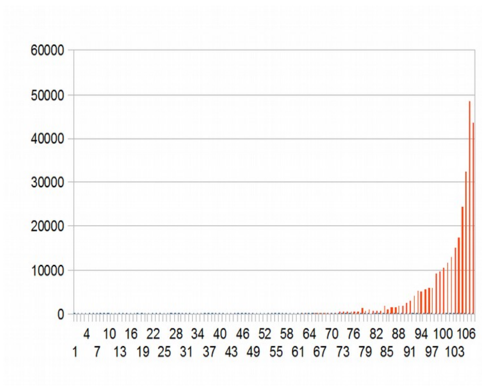

*Illustration 2: test.txt*

After seeing the different distributions we changed our polishing local training sets to have a similar dateId distribution to that of testing.txt. However, this brought no improvements in local or remote test results.
This situation brings us to only two conclusions:
1. Our Training is wrong (Wrong model, wrong training set)
2. The data in testing.txt is significantly different than that of training.txt in some way that we failed to properly account for.

## VI. Conclusion

The project did produce a system that is indeed capable of making movie predictions on the given data. However, the predictions are not as accurate as the team would have liked, especially on the test site data. Given more time, further work on this project would involve a more in depth investigation into why the system scores so well on subsets of the training data but performs poorly on the testing data. This would likely lead to discovery of overfitting issues to be solved. Even so, this project contributed greatly to the team members’ understanding of the concepts involved in data mining, especially in a practical and applied sense.