COMP4121 Major Project
Recommendation Systems

Abstract

This report discusses the methods used to create recommendation systems. We take an in-depth look at how 2 particular systems work, and discuss their implementation and results when applied to a real dataset.

1 Introduction

In today's digital world, it can be a challenge to make sense of the vast amount of data available. When it comes to predicting user interests, whether in music, movies, or online shopping, it is crucial to have a reliable system of forming these predictions from the available data. Such systems are known as *recommendation systems*.

We have two main types of recommendation systems.

- A **content-based system** will recommend items based on their attributes. For example, if a user has previously listened to jazz music, the system will recommend other jazz music. The downside to this approach is that for every item, we need to keep track of all its attributes that may be useful for recommendation.
- A **collaborative filtering system** does not keep track of item attributes, but rather utilises the similarities between users and items.

In this report, we will focus on collaborative filtering systems. We will look at two such systems, and discuss their implementations and results when applied to a dataset.

2 Collaborative Filtering Systems

We will discuss in detail two commonly used collaborative filtering systems, in the context of recommending movies based on previous user ratings. It is important to note that no recommendation system is perfect. We will look at how we can create a sufficient system.

We can represent the ratings data in a matrix M, where the rows correspond to users, and columns correspond to movies. If we have a rating r made by user i for movie j, then we set $M_{ij} = r$. We leave all other cells blank.

	Movie A	Movie B	Movie C	Movie D	Movie E
User 1	4	5	4		
User 2		1		5	3
User 3	1			4	
User 4			1		
User 5		2			4

The goal of our recommendation system is to make predictions on how each user would rate each unrated movie. We want to fill in all the blank entries. Once we have filled the matrix, we can choose the movies not yet watched by the user that have the highest ratings as recommended movies.

2.1 Neighbourhood Model

The main idea behind the neighbourhood model is to find similar users / movies within the data. Then, for a particular user we can recommend movies that similar users liked. Alternatively, we can recommend similar movies to the ones liked by the user.

For example, if User A and User B both liked "Toy Story" and "The Lion King", then if User A liked "Inside Out" it would suggest that User B would also like it. Notice that although these particular movies are of the same *animation* genre, we do not have to provide any genre information to the system. By simply analysing the ratings of other users, we hope for the system to recommend movies of similar genre to those already like by a user.

2.1.1 Normalisation

We cannot simply assume that the same rating has the same meaning for all users. For instance, we might have a user that is highly critical, and tends to give movies a low score. So a 3/5 might be reasonably high on their scale. We might also have another user that is a very generous reviewer. A rating of 3/5 would have a very different meaning to them.

Similarly, some movies may be overhyped, and as a result the ratings may be negatively skewed. We would like to normalise the data to be independent of such biases.

To address this issue, we introduce biases b_i for each user i and b_j for each movie j. A critical reviewer would be expected to have a negative bias, and a generous reviewer a positive bias. As a baseline prediction for user i and movie j, we have $\hat{r}_{ij} = \bar{r} + b_i + b_j$, where $\bar{r}$ is the average rating of all ratings in our dataset.

In order to calculate the user biases for user i, we could just simply calculate the user's average rating $\bar{r_i}$, and set the user bias to the difference between the user's average rating and the average rating of the dataset $\bar{r_u} - \bar{r}$, and similarly for movie biases. However, this does not necessarily minimise the root-mean-square error (RMSE) between our predicted values and the dataset:

$$\sum_{(i,j)} (\hat{r}_{ij} - r_{ij})^2 = \sum_{(i,j)} (\bar{r} + b_i + b_j - r_{ij})^2$$

on pairs (i, j). We choose to instead calculate our biases to minimise the RMSE.

We have a convex, quadratic function that we wish to minimise. In order to minimise the RMSE, we set all partial derivatives to 0. Consider user u who has rated n movies, and has bias b_n. We have the partial derivative:

$$\begin{aligned}
0 &= \frac{\partial}{\partial b_1} \sum_{(i,j)} (\hat{r}_{ij} - r_{ij})^2 \\
&= 2 \sum_{j} (\bar{r} + b_u + b_j - r_{uj}) \\
&= \sum_{j} (\bar{r} + b_u + b_j - r_{uj}) \\
&= n(\bar{r} + b_u) + \sum_{j} (b_j - r_{uj}) \\
&= n\,\bar{r} + n\,b_u + \sum_{j} b_j - \sum_{j} r_{uj}
\end{aligned}$$

Similarly for movie k with m ratings we have:

$$\begin{aligned}
0 &= \frac{\partial}{\partial b_k} \sum_{(i,j)} (\hat{r}_{ij} - r_{ij})^2 \\
&= 2 \sum_{i} (\bar{r} + b_i + b_k - r_{ik}) \\
&= \sum_{i} (\bar{r} + b_i + b_k - r_{ik}) \\
&= m(\bar{r} + b_k) + \sum_{i} (b_i - r_{ik}) \\
&= m\,\bar{r} + m\,b_k + \sum_{i} b_i - \sum_{i} r_{ik}
\end{aligned}$$

Rearranging the above, we get

$$\begin{aligned}
n\,b_u + \sum_{j} b_j &= \sum_{j} r_{uj} - n\,\bar{r} \\
m\,b_k + \sum_{i} b_i &= \sum_{i} r_{ik} - m\,\bar{r}
\end{aligned}$$

Let us look closer at what we have. In the first equation, b_u is just the user's bias, $\sum_j b_j$ is the sum of all movie biases for movies rated by the user, $\sum_j r_{uj}$ is the

total of all the user's ratings, and $\bar{r}$ is the average rating in the whole dataset. Similarly, in the second equation, b_k is the movie's bias, $\sum_i b_i$ is the sum of all user biases for users that rated the movie, $\sum_i r_{ik}$ is the total of all the movie's ratings, and again $\bar{r}$ is the average rating in the whole dataset.

For a dataset containing x users and y movies, we have a system of linear equations in r_1, ..., r_x and r_a, ..., r_y. We can represent this in matrix form $A\boldsymbol{x} = \boldsymbol{b}$ where $\boldsymbol{x} = (b_1, ..., b_x, b_a, ..., b_y)^{\boldsymbol{T}}$. We can solve for $\boldsymbol{x}$ to find the biases.

Once the biases are found, we can subtract the average rating, along with the user / movie biases from our raw data to get our normalised ratings.

$$\tilde{r}_{ij} = r_{ij} - \bar{r} - b_i - b_j$$

2.1.2 Calculating Similarities

Now that we have normalised our ratings, we would like to find similar users / movies in order to perform our recommendations. We will discuss the method using similar users, but the method is exactly mirrored for similar movies.

We must first define a notion of similarity. For users A and B, we look at any movies that both users rated. If the ratings between the users tend to agree, we can say that users A and B have a strong *positive* similarity. On the other hand, users A and B may have completely opposite tastes, so where user A rates a movie highly, user B rates the movie poorly and vice versa. In this case, users A and B are said to have a strong *negative* similarity. We may also have that there is little correlation between the two users' ratings. In this case we can say that users A and B have a weak similarity, which may be positive or negative.

Users with strong similarities, whether positive or negative are the most useful for our recommendation system. We can model this notion of similarity by treating user ratings as vectors, only taking ratings for movies rated by both users. We can take the cosine similarity between the two vectors, which measures the cosine of the angle between the two vectors. From elementary linear algebra we have that

the angle α between two vectors is given by:

$$cos\ \alpha = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{|\boldsymbol{a}||\boldsymbol{b}|}$$

The range of cosine similarities is $[-1, 1]$. Vectors with similar orientations will have a cosine similarity close to 1, while opposite vectors will have a cosine similarity close to -1. Unoriented vectors will have a cosine similarity closer to 0.

This fits our notion of similarity. We can define the similarity between users A and B with the following equation:

$$d_{ab} = \frac{\sum_m \tilde{\boldsymbol{r}}_{am} \tilde{\boldsymbol{r}}_{bm}}{\sqrt{\sum_m (\tilde{\boldsymbol{r}}_{am})^2 \sum_u (\tilde{\boldsymbol{r}}_{bm})^2}}$$

summing over only the movie ratings common to both users.

For each user a, we compute user a's similarity d_{ab} to every other user b. When we want to predict user a's rating for a particular movie m, we choose the top n most similar users based on $|d_{ab}|$, and put them in a set $\boldsymbol{N}_a$ called the *neighbourhood*.

2.1.3 Predicting Ratings

Now that we have all the users' neighbourhoods, containing the top n most similar (or dissimilar) users, we can form a better prediction for a given user's rating on a particular movie.

To predict user i's rating for movie j, we use the following:

$$\frac{\sum_{k \in \boldsymbol{N}_i} w_{kj} \tilde{r}_{kj}}{\sum_{k \in \boldsymbol{N}_i} |w_{kj}|}$$

where w_{kj} is a weight given to user k's rating of movie j. This gives a weighted average of user i's neighbours' ratings for movie j. A natural choice for the weight is simply the similarity j_{kj} which gives:

$$\frac{\sum_{k \in \boldsymbol{N}_i} d_{kj} \tilde{r}_{kj}}{\sum_{k \in \boldsymbol{N}_i} |d_{kj}|}$$

Once we have calculated this, we can add the initial average rating $\bar{r}$ and biases b_i and b_j to get our final predicted rating.

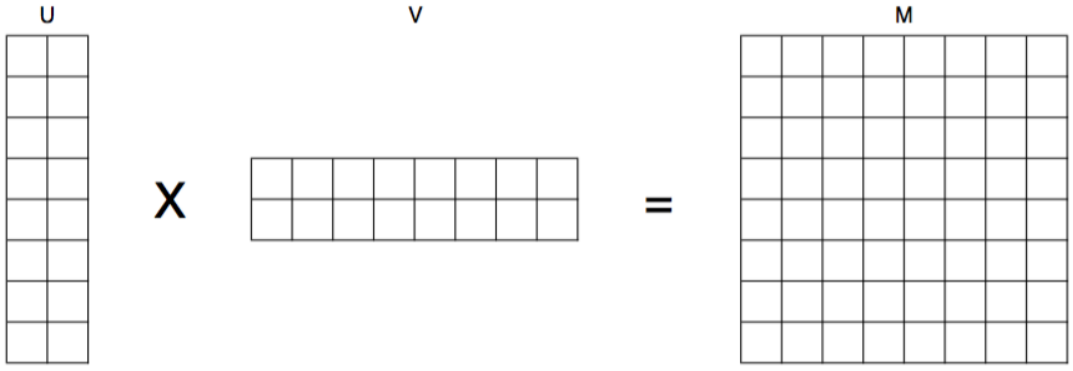
$$\hat{r}_{ui}^{\boldsymbol{N}} = \bar{r} + b_i + b_j + \frac{\sum_{k \in \boldsymbol{N}_i} d_{kj} \tilde{r}_{kj}}{\sum_{k \in \boldsymbol{N}_i} |d_{kj}|}$$

2.2 Latent-Factor Method

2.2.1 Intuition

The latent-factor method is less intuitive and not as straightforward as the neighbourhood model to understand. In this method, we assume that of all movies in our dataset, there are only a few factors which influence user ratings. For example these factors may be genre type, movie duration, presence of a famous actor etc. Every movie is given a score for each of these factors, and likewise every user is given a score for their "taste" of each factor.

We think of our matrix M as being a product of a matrix U containing the user tastes for the factors and a matrix V containing the value of the factors for each movie.



For example, if our factors are *romance*, and *long movie*. We could assign the movie "Gone With the Wind" a *romance* score of 2.5 and a *long movie* score of 2.

So in matrix V we would have the column

$$\begin{pmatrix} 2.5 \\ 2 \end{pmatrix}$$

Then for a particular user we could assign a *romance* taste of 1, and a *long movie* taste of 0.7. So in matrix U we would have the row

$$(1, 0.7)$$

In order to find the predicted user rating, we simply multiply the two vectors to get $2.5 \times 1 + 2 \times 0.7 = 3.9$.

At first, it appears as if we have the same problem as with a **content-based system**, where we must first figure out a suitable set of factors, and then assign each user and movie scores for each factor. However, we can perform an ingenious trick, and let the factors emerge from the data, without even having to understand what they are. We set a fixed number of factors (usually between 20 and 200), and simultaneously find the user tastes and movie factor values that best represent our data.

2.2.2 Normalisation

As with the neighbourhood model, we can first perform a normalisation of the data to remove any user / movie biases.

2.2.3 The Algorithm

We would like for UV to approximate the non-blank values of M as best as possible. We will again use the RMSE as a measure of how well UV approximates M. Minimising the RMSE is a non-convex optimisation problem, so that finding a local minima does not necessarily find the global minimum. Therefore it is necessary to randomise the process and perform repetitions.

We shall attempt to find U and V iteratively, by initially filling U and V with

arbitrary values. We then iteratively adjust the matrices to minimise the RMSE. For simplicity, we will only consider adjusting one element of U or V at a time.

We start off by choosing an arbitrary element from U or V, and vary it to minimise the RMSE. We then choose another element from U or V, and also vary it to minimise the RMSE. We continue this process until there is little improvement made to the RMSE.

We choose the order of elements randomly in order to try and find an arbitrary local minima. Ideally, we use a random permutation of elements so that all elements of U and V are updated in a round.

We would like to terminate the process when the RMSE converges to 0, however we cannot necessarily expect this, since we are only approximating. Instead, we terminate the process once the RMSE improvement in a round is below a certain threshold.

Pseudocode

```
do
    generate a permutation of elements in U and V
    for each element in the permutation:
        adjust the element to minimise RMSE
while improvement of RMSE >= THRESHOLD
```

To minimise the RMSE for an arbitrary element u_{rs} in U, we only need to consider the elements in $P = UV$ affected by u_{rs}, that is only the elements of P in row r.

$$p_{rj} = \sum_k u_{rk} v_{kj} = x v_{sj} + \sum_{k \neq s} u_{rk} v_{kj}$$

where x is the updated value of u_{rs}. We sum only over non-blank elements. We can write the sum of squares of the errors affected by x as

$$\sum_j (m_{rj} - \sum_{k \neq s} u_{rk} v_{kj} - x v_{sj})^2$$

To minimise the RMSE, we set the derivative with respect to x to 0.

$$-2\sum_{j} v_{sj}(m_{rj} - \sum_{k \neq s} u_{rk}v_{kj} - xv_{sj})$$

By rearranging, we get that

$$x = \frac{\sum_j v_{sj}(m_{rj} - \sum_{k \neq s} u_{rk}v_{kj})}{\sum_j v_{sj}^2}$$

Similarly to optimise the value of $y = v_{rs}$

$$y = \frac{\sum_i u_{ir}(m_{is} - \sum_{k \neq r} u_{ik}v_{ks})}{\sum_i u_{ir}^2}$$

3 Implementation

Both the Neighbourhood Model and the Latent-factor Method were implemented using Java 1.8. The external libraries used were Colt 1.2 for linear algebra, and Apache Commons Collections 4.1 used for a bidirectional tree map. Python's Numpy library was also used to approximate a solution to a linear system of equations.

3.1 The Dataset

It was decided to use the MovieLens datasets by GroupLens, a social computing research group at the University of Minnesota. MovieLens is an online movie recommendation system for research purposes run by the GroupLens group. The full dataset is available for download, along with a few smaller subsets. The dataset chosen for the final tests was the latest small dataset updated in October 2016, containing 100,004 ratings with 671 users and 9125 movies. The ratings are provided in a *csv* file with the format *userId, movieId, tag, timestamp*. The dataset also contains user *tags* for movies, and movie genres. The timestamp, tags, and genre data were not used in the algorithms.

In order to test the performance of the algorithms, we divide the dataset randomly into a **training** set and a **test** set. We go through all the ratings in the

dataset and with probability $\rho = 0.2$ assign it to the test set. Otherwise it is assigned to the training set. We run our algorithms on the training set, and compare the predictions for those in the test set, by measuring the RMSE.

3.2 Processing the data

The *csv* file containing the ratings data is parsed and stored in an *ArrayList* of *Ratings*. We keep track of the total number of ratings, and the total value of the ratings, so that the average rating can be computed later. We also keep track of how many ratings each user / movie has using a *TreeMap*. A *TreeMap* is chosen over a *HashMap* as it allows the keys to be listed in order without having to sort. We then filter the data to speed up the algorithms, whilst only removing a small amount of data. Any users that rated less than 20 movies are removed (though the dataset providers claim this is already the case). Similarly, any movies that received less than 20 ratings are removed. This retains the users / movies that contribute the most useful information.

Users and movies are assigned new IDs starting from 0 (as Colt uses 0 indexing on matrices). We use a bidirectional tree map to store the mapping between old and new user / movie IDs, this makes it easy to convert the IDs in both directions. The ratings are placed into a matrix as described previously (using Colt). All blank entries are represented by a constant. A sparse matrix is used in order to minimise memory usage.

3.3 Normalisation

We now have a matrix set up for the neighbourhood model. We first need to normalise the matrix by removing the user / movie biases. After consulting with the course lecturer Aleks, it was determined that it was not necessary to implement regularisation.

Recall that we want to solve the system of equations $Ax = b$ for x to find the biases.

The initial method of calculating the biases was to invert the matrix A, and thereby

solve the system of equations by calculating $x = A^{-1}b$. However this proved to be numerically unstable, and resulted in extremely incorrect results.

Instead, the final approach was to use a least squares approximate. We find an $\hat{x}$ that minimises $||Ax - b||$, and use this as an approximation to x. There was no built in implementation of this method in Colt, and no Java implementation could be found. We pass our system of equations to Python's *numpy.linalg.lstsq* to compute the approximation. The resulting vector is read back into our Java program. Initially the older method *scipy.linalg.lstsq* was used instead as it handles larger datasets where *numpy.linalg.lstsq* fails. However it was far to slow to be usable.

3.4 Neighbourhood Model

Once the biases are computed, we subtract them along with the average rating to proceed with the neighbourhood model.

To calculate similarity scores, a threshold was used. Neighbouring users / movies that did not have a certain number of ratings in common were not included as part of the neighbourhood. This was done to produce more accurate similarity scores. To calculate a predicted rating from similar users, we sorted the neighbouring users by $|similarity|$, and chose the top n most similar (or dissimilar) users. The weight for each neighbour was simply chosen to be the similarity score. A minimum threshold of neighbours was also used, so that if a user had less than m neighbours, the neighbourhood model was not used, and a default value of 0 was used instead. Such ratings are predicted purely by adding user / movie biases to the average rating. An identical approach using similar movies was also implemented.

Once we have predicted ratings, we add the biases and average rating that were previously subtracted. Once capped to a range of $[0.5, 5]$ we are left with our final predictions.

3.5 Latent-Factor Method

We test the latent-factor method with various number of factors. We make use of the normalisation implementation from the neighbourhood model. In our tests, we see what effect performing normalisation has on the results.

Matrices U and V are represented using a dense matrix in Colt. We initially fill U and V with a value of $\sqrt{\frac{a}{d}}$, where a is the average rating after normalisation (if performed). This gives all the elements of $P = UV$ a value of a.

We generate a random permutation of elements as a *List*, and optimise each element one by one using the equations discussed earlier

$$x = \frac{\sum_j v_{sj}(m_{rj} - \sum_{k \neq s} u_{rk} v_{kj})}{\sum_j v_{sj}^2}$$
$$y = \frac{\sum_i u_{ir}(m_{is} - \sum_{k \neq r} u_{ik} v_{ks})}{\sum_i u_{ir}^2}$$

Once we have gone through the permutation, we check the RMSE improvement from the previous round. If the improvement is less than a threshold of 0.005 we terminate the process. If not, we generate a new permutation and repeat the process. Finally we add the biases removed during normalisation (if performed).

Since this method does not guarantee the global minimum RMSE, we perform a number of repetitions, and average the results.

4 Results

We first removed all users with less than 20 movies rated. As the dataset provider claimed, no such users were found. We then removed all movies with less than 20 ratings. 7763 out of the 9066 movies were removed, leaving only 1303 movies. 69104 ratings were left in the dataset.

When dividing the dataset into training and test sets, we always used a seed

of 4121. This ensures that the training and test sets were the same across all tests. The training set contained 55247 ratings, and the test set contained 13857.

4.1 Neighbourhood Model

We ran tests using user similarities, and movie similarities. We tried varying the neighbourhood size. We also tried running the tests with normalisation and without (only subtracting the average rating).

neighbourhood size	normalisation	test set RMSE
5	No	0.9181
10	No	0.8969
15	No	0.8933
20	No	0.8921
25	No	0.8913
5	Yes	0.9611
10	Yes	0.9335
15	Yes	0.9289
20	Yes	0.9295
25	Yes	0.9319

Table 1: Neighbourhood model using movie similarities

neighbourhood size	normalisation	test set RMSE
5	No	0.9244
10	No	0.9078
15	No	0.9034
20	No	0.9004
25	No	0.8996
5	Yes	1.0611
10	Yes	1.0331
15	Yes	1.0249
20	Yes	1.0215
25	Yes	1.0204

Table 2: Neighbourhood model using user similarities

We found that the RMSE was increased when applying normalisation. We noted that in general a larger neighbourhood size contributed to a better result. This is a possible indication of overfitting.

4.2 Latent-Factor Method

We ran several different tests using the latent-factor method. Initially, tests were performed on the training set without doing any sort of normalisation. We varied the number of factors, testing the set $\{5, 10, 15, 20, 30, 40, 60, 80, 100\}$. For each test, we ran as many rounds of iteration as required until the improvement in RMSE of the training set was below 0.005. It was found that if this threshold was set too low, it would lead to overfitting, and therefore lead to suboptimal results for the test set. A threshold of 0.005 was found to be a good balance between reducing the RMSE and We repeated each test 10 times and averaged the results. Finally we capped the results to a range of $[0.5, 5]$ before calculating the RMSE of the test set.

We then ran the same tests but applied normalisation by removing user / movie biases. We did not however, subtract the average rating, as this was found to an excessive amount of predictions outside of the range $[0.5, 5]$.

factors	iterations required (mode)	average training set RMSE	test set RMSE
5	13	0.7296	0.8525
10	15	0.5941	0.8533
15	15	0.4882	0.8623
20	16	0.3985	0.8773
30	17	0.2666	0.8898
40	17	0.1768	0.8941
60	15	0.0815	0.8924
80	13	0.0424	0.8900
100	11	0.0269	0.8997

Table 3: Latent-Factor without normalisation

factors	iterations required (mode)	average training set RMSE	test set RMSE
5	13	0.7813	0.8603
10	13.5	0.6754	0.8653
15	14	0.5967	0.8830
20	13.5	0.5353	0.8914
30	13	0.4605	0.9208
40	12	0.4171	0.9372
60	9	0.3826	0.9731
80	7	0.3703	0.9944
100	6	0.3638	1.0063

Table 4: Latent-Factor with normalisation

It was found that performing normalisation actually increased the RMSE in every case. Again, this could be an indication of overfitting. The RMSE was found to be lower with a smaller number of factors. We also found that normalisation caused the algorithm to converge faster.

Overall, we can see that the latent-factor method performs considerably better than the neighbourhood model.

References

[1] Chiang, M. (2012). *Networked life*. 1st ed. Cambridge, U.K.: Cambridge University Press.

[2] Harper, F. and Konstan, J. (2017). *The MovieLens Datasets*.

[3] Mining of Massive Datasets. (2014). 2nd ed. [ebook] Available at: http://infolab.stanford.edu/~ullman/mmds/ch9.pdf [Accessed 20 Nov. 2017].

5 Appendix

The code use in the implementation is listed below.

Main.java

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;

import cern.colt.matrix.DoubleFactory2D;
import cern.colt.matrix.DoubleMatrix2D;
import cern.jet.math.Functions;

public class Main {

   public static final int USERS = 1;
   public static final int MOVIES = 2;
   public static final int NEIGHBOURHOOD = 1;
   public static final int LATENT_FACTOR = 2;
   public static final int MATRIX_EMPTY_VAL = 999;

   private static String timestamp;

   public static void main(String[] args) {
      Main m = new Main();
      m.beginTest();
   }

   private void beginTest() {

      timestamp = new SimpleDateFormat("yyyy.MM.dd.HH.mm.ss").format(new
```

```
    Date());

// Global Parameters
int nUsersFilter = 20;
int nMoviesFilter = 20;
boolean normalisationEnabled = false;
boolean normalisationSubtractMean = true;
int algorithmType = NEIGHBOURHOOD;

// Neighbourhood Parameters
int neighbourhoodType = MOVIES;
int minNeighbourhoodSize = 5;
int maxNeighbourhoodSize = 25;

// Latent Factor Parameters
int nTests = 10;
int factors = 20;
int maxIterations = 200;
double rmseImprovementThreshold = 0.005;

// ======================
// Preprocess the data
// ======================

// Make a new directory to hold all output files
new File("tests/" + timestamp).mkdir();

// Specify the ratings file
String ratingsFile = "ratings.csv";

// Create the RatingsData object to hold ratings data
RatingsData ratingsData = new RatingsData();

// Read in the ratings file and add to data
String delimiter = ",";
boolean skipFirstLine = true;
```

```
processRatingsFile(ratingsFile, delimiter, ratingsData,
    skipFirstLine);

// Filter out any users and movies with less than a certain number
    of ratings
ratingsData.filterUsers(nUsersFilter);
ratingsData.filterMovies(nMoviesFilter);

// Once we are done filtering, we need to assign new IDs for users
    and
// movies
ratingsData.createNewIds();

// Generate the training and test sets
DatasetSubsets ds = new DatasetSubsets(ratingsData, 0.2f, 4121);
RatingsData trainingSet = ds.getTrainingSet();
RatingsData testSet = ds.getTestSet();

// ======================
// Normalisation
// ======================

Normalisation norm = new Normalisation(trainingSet);
if (normalisationEnabled) {
   System.out.println("Normalising");
   long start = System.currentTimeMillis();
   norm.normalise(normalisationSubtractMean);
   long stop = System.currentTimeMillis();
   System.out.println("Done Normalising (" + (stop - start) +
       "ms)");
   writeMatrixToFile(norm.getNormalised(), "normalised");
} else {
   norm.normaliseNoBiases();
}

// ======================
```

```
// Perform recommendation algorithm
// ======================

DoubleMatrix2D finalPredictionMatrix = null;
if (algorithmType == NEIGHBOURHOOD) {
   Neighbourhood neighbourhood = new
       Neighbourhood(norm.getNormalised());
   if (neighbourhoodType == USERS) {
      neighbourhood.calculateUserSimilarityScores();
      DoubleMatrix2D predictedFromUsers =
          neighbourhood.calculatePredictionsFromUserSimilarities(minNeighbourhoodSiz
          maxNeighbourhoodSize);
      writeMatrixToFile(predictedFromUsers, "predUsers");
      if (normalisationEnabled) {
         DoubleMatrix2D unNormalisedFromUsers =
             norm.unNormaliseMatrix(predictedFromUsers,
               normalisationSubtractMean);
         writeMatrixToFile(unNormalisedFromUsers,
             "predUsers_unNorm");
         finalPredictionMatrix = unNormalisedFromUsers;
      } else {
         finalPredictionMatrix =
             norm.unNormaliseNoBiases(predictedFromUsers);
      }
   } else if (neighbourhoodType == MOVIES) {
      neighbourhood.calculateMovieSimilarityScores();
      DoubleMatrix2D predictedFromMovies =
          neighbourhood.calculatePredictionsFromMovieSimilarities(minNeighbourhoodSi
          maxNeighbourhoodSize);
      writeMatrixToFile(predictedFromMovies, "predMovies");
      if (normalisationEnabled) {
         DoubleMatrix2D unNormalisedFromMovies =
             norm.unNormaliseMatrix(predictedFromMovies,
               normalisationSubtractMean);
         writeMatrixToFile(unNormalisedFromMovies,
             "predMovies_unNorm");
```

```
            finalPredictionMatrix = unNormalisedFromMovies;
        } else {
            finalPredictionMatrix =
                norm.unNormaliseNoBiases(predictedFromMovies);
        }
    }
} else if (algorithmType == LATENT_FACTOR) {
    DoubleMatrix2D averageResults =
        DoubleFactory2D.dense.make(trainingSet.getNUsers(),
        trainingSet.getNMovies(), 0);
    for (int testId = 1; testId <= nTests; testId++) {
        if (normalisationEnabled) {
            double initValue = Math.sqrt(norm.getNormalisedMean() /
                (double) factors);
            LatentFactor latentFactor = new LatentFactor(testId,
                trainingSet, norm.getNormalised(), factors,
                  maxIterations, initValue, rmseImprovementThreshold);
            latentFactor.runAllIterations();
            DoubleMatrix2D product = latentFactor.getProductUV();
            DoubleMatrix2D unNormalised =
                norm.unNormaliseMatrix(product,
                normalisationSubtractMean);
            averageResults.assign(unNormalised, Functions.plus);
            writeMatrixToFile(product,
                String.format("%02d_prodUV_Final", testId));
            writeMatrixToFile(unNormalised,
                String.format("%02d_prodAB_UnNorm", testId));
        } else {
            double initValue = Math.sqrt(trainingSet.getMeanRating() /
                (double) factors);
            LatentFactor latentFactor = new LatentFactor(testId,
                trainingSet, trainingSet.getMatrix(), factors,
                  maxIterations, initValue, rmseImprovementThreshold);
            latentFactor.runAllIterations();
            DoubleMatrix2D product = latentFactor.getProductUV();
            averageResults.assign(product, Functions.plus);
```

```
            writeMatrixToFile(product,
                String.format("%02d_prodUV_Final", testId));
        }
        writeMatrixToFile(averageResults,
            String.format("%02d_sumOfResults", testId));
      }
      averageResults.assign(Functions.div(nTests));
      writeMatrixToFile(averageResults, "Average");
      finalPredictionMatrix = averageResults;
   }

   // ======================
   // Compare against test set
   // ======================
   double rmse = RMSE.getRMSE(testSet.getMatrix(),
       finalPredictionMatrix, testSet);
   System.out.println("Test set rmse = "+rmse);


}

/**
 * Read the ratings file and add to ratings data
 *
 * @param ratingsFile
 * @param delimiter
 * @param ratingsData
 */
private void processRatingsFile(String ratingsFile, String delimiter,
    RatingsData ratingsData,
      boolean skipFirstLine) {
   try (BufferedReader br = new BufferedReader(new FileReader(new
       File(ratingsFile)))) {
      String line;
      if (skipFirstLine) {
         br.readLine();
```

```
        }
        while ((line = br.readLine()) != null) {
            String[] lineParts = line.split(delimiter);

            int userId = Integer.parseInt(lineParts[0]);
            int movieId = Integer.parseInt(lineParts[1]);
            double rating = Double.parseDouble(lineParts[2]);

            ratingsData.addRating(userId, movieId, rating);

        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void writeMatrixToFile(DoubleMatrix2D matrix, String
    filename) {
    try {
        BufferedWriter out = new BufferedWriter(
                new FileWriter("tests/" + timestamp + "/" + timestamp +
                    "_" + filename + ".txt"));
        out.write(matrix.toString());
        out.close();
    } catch (IOException e) {
    }
}

public static String writeStringToFile(String s, String filename,
    String extension) {
    try {
        BufferedWriter out = new BufferedWriter(
                new FileWriter("tests/" + timestamp + "/" + timestamp +
                    "_" + filename + extension));
        out.write(s);
        out.close();
```

```
            return "tests/" + timestamp + "/" + timestamp + "_" + filename
                + extension;
        } catch (IOException e) {
            return null;
        }
    }

}
```

RatingsData.java

```
import java.util.ArrayList;
import java.util.TreeMap;

import org.apache.commons.collections4.bidimap.TreeBidiMap;

import cern.colt.matrix.DoubleFactory2D;
import cern.colt.matrix.DoubleMatrix2D;

// A class used to hold a dataset of user / movie ratings

public class RatingsData {

    private int ratingsTotal = 0;
    private int ratingsCountAll = 0;

    // Mappings between old and new IDs
    private TreeBidiMap<Integer, Integer> oldToNewUserIds;
    private TreeBidiMap<Integer, Integer> oldToNewMovieIds;

    private TreeMap<Integer, Integer> ratingsCountPerOldUserId;
    private TreeMap<Integer, Integer> ratingsCountPerOldMovieId;

    private ArrayList<Rating> allRatings;
    private DoubleMatrix2D matrix;
```

```
public ArrayList<Rating> getAllRatings() {
   return allRatings;
}

public RatingsData() {
   this.ratingsTotal = 0;
   this.ratingsCountAll = 0;

   allRatings = new ArrayList<Rating>();
   ratingsCountPerOldUserId = new TreeMap<Integer, Integer>();
   ratingsCountPerOldMovieId = new TreeMap<Integer, Integer>();

   oldToNewUserIds = new TreeBidiMap<Integer, Integer>();
   oldToNewMovieIds = new TreeBidiMap<Integer, Integer>();
}

// Add a new rating to the system
public void addRating(int oldUserId, int oldMovieId, double rating) {

   allRatings.add(new Rating(oldUserId, oldMovieId, rating));

   // Initialise the maps if not done previously
   if (!ratingsCountPerOldUserId.containsKey(oldUserId)) {
      ratingsCountPerOldUserId.put(oldUserId, 0);
   }
   if (!ratingsCountPerOldMovieId.containsKey(oldMovieId)) {
      ratingsCountPerOldMovieId.put(oldMovieId, 0);
   }

   ratingsCountPerOldUserId.put(oldUserId,
       ratingsCountPerOldUserId.get(oldUserId) + 1);
   ratingsCountPerOldMovieId.put(oldMovieId,
       ratingsCountPerOldMovieId.get(oldMovieId) + 1);
   ratingsCountAll++;

   ratingsTotal += rating;
```

```
}

// Create new IDs (contiguous starting from 0)
public void createNewIds() {
   oldToNewUserIds.clear();
   oldToNewMovieIds.clear();
   int newUserId = 0;
   for (int oldUserId : ratingsCountPerOldUserId.keySet()) {
      oldToNewUserIds.put(oldUserId, newUserId);
      newUserId++;
   }
   int newMovieId = 0;
   for (int oldMovieId : ratingsCountPerOldMovieId.keySet()) {
      oldToNewMovieIds.put(oldMovieId, newMovieId);
      newMovieId++;
   }
}

// Remove any users that rated < min movies
public boolean filterUsers(int minNumRatings) {

   System.out.println("Filtering Users");
   System.out.println("nUsers: " + getNUsers());
   int usersRemoved = 0;

   boolean removedAUser = false;

   // Find all oldUserIds that have not rated enough movies
   ArrayList<Integer> oldUserIdsToRemove = new ArrayList<Integer>();
   for (Integer oldUserId : ratingsCountPerOldUserId.keySet()) {
      if (ratingsCountPerOldUserId.get(oldUserId) < minNumRatings) {
         oldUserIdsToRemove.add(oldUserId);
         usersRemoved++;
      }
   }
```

```
    // Go through all ratings, and remove all ratings rated
    // by oldUserIds in the list
    for (int i = allRatings.size() - 1; i >= 0; i--) {
       Rating currentRating = allRatings.get(i);
       // Check if user is contained in the list to be removed
       if (oldUserIdsToRemove.contains(currentRating.getUserId())) {
          removedAUser = true;

          // Decrement the count for oldMovieId
          int oldMovieId = currentRating.getMovieId();
          ratingsCountPerOldMovieId.put(oldMovieId,
              ratingsCountPerOldMovieId.get(oldMovieId) - 1);

          // Decrement the global ratings count
          ratingsCountAll--;

          // Subtract from total
          ratingsTotal -= currentRating.getRatingVal();

          // Finally, remove the rating from the list
          allRatings.remove(i);
       }
    }

    // Remove any other references of oldUserIds
    for (int oldUserId : oldUserIdsToRemove) {
       ratingsCountPerOldUserId.remove(oldUserId);
       System.out.println("removing user [" + oldUserId + "]");
    }

    System.out.println("removed: " + usersRemoved);
    System.out.println("nUsers: " + getNUsers() + "\n");
    return removedAUser;

}
```

```
// Remove any movies with < min ratings
public boolean filterMovies(int minNumRatings) {

   System.out.println("Filtering Movies");
   System.out.println("nMovies: " + getNMovies());
   int moviesRemoved = 0;

   boolean removedAMovie = false;

   // Find all oldMovieIds that have not received enough ratings
   ArrayList<Integer> oldMovieIdsToRemove = new ArrayList<Integer>();
   for (Integer oldMovieId : ratingsCountPerOldMovieId.keySet()) {
      if (ratingsCountPerOldMovieId.get(oldMovieId) < minNumRatings) {
         oldMovieIdsToRemove.add(oldMovieId);
         moviesRemoved++;
      }
   }

   // Go through all ratings, and remove all ratings for
   // oldMovieIds in the list
   for (int i = allRatings.size() - 1; i >= 0; i--) {
      Rating currentRating = allRatings.get(i);
      // Check if movie is contained in the list to be removed
      if (oldMovieIdsToRemove.contains(currentRating.getMovieId())) {

         removedAMovie = true;

         // Decrement the count for oldUserId
         int oldUserId = currentRating.getUserId();
         ratingsCountPerOldUserId.put(oldUserId,
             ratingsCountPerOldUserId.get(oldUserId) - 1);

         // Decrement the global ratings count
         ratingsCountAll--;

         // Subtract from total
```

```
            ratingsTotal -= currentRating.getRatingVal();

            // Finally, remove the rating from the list
            allRatings.remove(i);

        }
    }

    // Remove any other references of oldMovieIds
    for (int oldMovieId : oldMovieIdsToRemove) {
        ratingsCountPerOldMovieId.remove(oldMovieId);
    }

    System.out.println("removed: " + moviesRemoved);
    System.out.println("nMovies: " + getNMovies() + "\n");

    return removedAMovie;

}

public int getNewUserId(int oldUserId) {
    return oldToNewUserIds.get(oldUserId);
}

public int getNewMovieId(int oldMovieId) {
    return oldToNewMovieIds.get(oldMovieId);
}

public int getOldUserId(int newUserId) {
    return oldToNewUserIds.inverseBidiMap().get(newUserId);
}

public int getOldMovieId(int newMovieId) {
    return oldToNewMovieIds.inverseBidiMap().get(newMovieId);
}
```

```
public double getMeanRating() {
   return (double) ratingsTotal / (double) ratingsCountAll;
}

public int getNUsers() {
   return ratingsCountPerOldUserId.size();
}

public int getNMovies() {
   return ratingsCountPerOldMovieId.size();
}

// Form a matrix from the ratings data
private void createMatrix() {

   // Initialise a new matrix
   int nUsers = oldToNewUserIds.inverseBidiMap().lastKey() + 1;
   int nMovies = oldToNewMovieIds.inverseBidiMap().lastKey() + 1;
   matrix = DoubleFactory2D.sparse.make(nUsers, nMovies,
       Main.MATRIX_EMPTY_VAL);

   for (Rating r : allRatings) {
      int newUserId = getNewUserId(r.getUserId());
      int newMovieId = getNewMovieId(r.getMovieId());
      double ratingVal = r.getRatingVal();
      matrix.set(newUserId, newMovieId, ratingVal);
   }

}

public DoubleMatrix2D getMatrix() {

   if (matrix == null) {
      createMatrix();
   }
   return matrix;
```

```
    }

    public TreeBidiMap<Integer, Integer> getOldToNewUserIds() {
        return oldToNewUserIds;
    }

    public void setOldToNewUserIds(TreeBidiMap<Integer, Integer>
        oldToNewUserIds) {
        this.oldToNewUserIds = oldToNewUserIds;
    }

    public TreeBidiMap<Integer, Integer> getOldToNewMovieIds() {
        return oldToNewMovieIds;
    }

    public void setOldToNewMovieIds(TreeBidiMap<Integer, Integer>
        oldToNewMovieIds) {
        this.oldToNewMovieIds = oldToNewMovieIds;
    }

}
```

Rating.java

```
public class Rating {

    private int userId;
    private int movieId;
    private double ratingVal;

    public Rating(int userId, int movieId, double ratingVal) {
        this.userId = userId;
        this.movieId = movieId;
        this.ratingVal = ratingVal;
```

```
    }

    public int getUserId() {
        return userId;
    }

    public int getMovieId() {
        return movieId;
    }

    public double getRatingVal() {
        return ratingVal;
    }

    @Override
    public String toString() {
        return "[User: "+userId+",\tMovie: "+movieId+",\tRatingVal:
            "+ratingVal+"]";
    }

}
```

RMSE.java

```
import cern.colt.matrix.DoubleMatrix2D;

public class RMSE {

    // Calculates the RMSE between the reference matrix M and the working
        matrix P
    public static double getRMSE(DoubleMatrix2D M, DoubleMatrix2D P,
        RatingsData rData) {

        double sumErrSquared = 0;
        int nRatings = rData.getAllRatings().size();
```

```
      for (Rating r : rData.getAllRatings()) {
         int newUserId = rData.getNewUserId(r.getUserId());
         int newMovieId = rData.getNewMovieId(r.getMovieId());
         double M_Val = M.get(newUserId, newMovieId);
         double P_Val = Math.max(0.5, Math.min(5, P.get(newUserId,
             newMovieId)));
         sumErrSquared += Math.pow(M_Val - P_Val, 2);
      }

      return Math.sqrt(sumErrSquared / nRatings);
   }

}
```

DatasetSubsets.java

```
import java.util.Random;

public class DatasetSubsets {

   private RatingsData trainingSet;
   private RatingsData testSet;

   public DatasetSubsets (RatingsData rData, float pTestSet, long seed) {

      trainingSet = new RatingsData();
      testSet = new RatingsData();

      Random random = new Random(seed);
      for (Rating rating : rData.getAllRatings()) {
         if (random.nextFloat() > pTestSet) {
            // Add to training set
            trainingSet.addRating(rating.getUserId(),
                rating.getMovieId(), rating.getRatingVal());
         } else {
            // Add to test set
```

```
            testSet.addRating(rating.getUserId(), rating.getMovieId(),
                rating.getRatingVal());
         }
      }

      trainingSet.setOldToNewUserIds(rData.getOldToNewUserIds());
      trainingSet.setOldToNewMovieIds(rData.getOldToNewMovieIds());
      testSet.setOldToNewUserIds(rData.getOldToNewUserIds());
      testSet.setOldToNewMovieIds(rData.getOldToNewMovieIds());

   }

   public RatingsData getTrainingSet() {
      return trainingSet;
   }

   public RatingsData getTestSet() {
      return testSet;
   }

}
```

Normalisation.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;

import cern.colt.matrix.DoubleFactory2D;
import cern.colt.matrix.DoubleMatrix2D;
import cern.colt.matrix.impl.DenseDoubleMatrix2D;
import cern.colt.matrix.linalg.Algebra;

public class Normalisation {
```

```
public static Algebra algebra = new Algebra();

private RatingsData rData;

private DoubleMatrix2D normalised;

private ArrayList<Double> userBiases;
private ArrayList<Double> movieBiases;

public Normalisation(RatingsData rData) {
   this.rData = rData;
   userBiases = new ArrayList<Double>();
   movieBiases = new ArrayList<Double>();
}

// Calculate the user / movie biases and remove them from all
    non-empty
// cells
public void normalise(boolean subtractMean) {
   // Create a blank matrix
   DoubleMatrix2D optimisation =
       DoubleFactory2D.sparse.make(rData.getNUsers() +
       rData.getNMovies(),
         rData.getNUsers() + rData.getNMovies(), 0);

   DenseDoubleMatrix2D rhs = new
       DenseDoubleMatrix2D(rData.getNUsers() + rData.getNMovies(), 1);

   int[] userRatingCount = new int[rData.getNUsers()];
   int[] userRatingTotal = new int[rData.getNUsers()];
   int[] movieRatingCount = new int[rData.getNMovies()];
   double[] movieRatingTotal = new double[rData.getNMovies()];

   for (Rating rating : rData.getAllRatings()) {
```

```
    int userId = rData.getNewUserId(rating.getUserId());
    int movieId = rData.getNewMovieId(rating.getMovieId());
    double ratingVal = rating.getRatingVal();

    // Set the optimisation matrix (user row)
    optimisation.set(userId, userId, optimisation.get(userId,
        userId) + 1);
    optimisation.set(userId, rData.getNUsers() + movieId, 1);

    // Set the optimisation matrix (movie row)
    optimisation.set(rData.getNUsers() + movieId, rData.getNUsers()
        + movieId,
          optimisation.get(rData.getNUsers() + movieId,
              rData.getNUsers() + movieId) + 1);
    optimisation.set(rData.getNUsers() + movieId, userId, 1);

    // Update values for optimisation
    userRatingCount[userId]++;
    userRatingTotal[userId] += ratingVal;
    movieRatingCount[movieId]++;
    movieRatingTotal[movieId] += ratingVal;

}

for (int userId = 0; userId < rData.getNUsers(); userId++) {
    double finalValue = userRatingCount[userId] *
        rData.getMeanRating() - userRatingTotal[userId];
    rhs.set(userId, 0, finalValue);
}
for (int movieId = 0; movieId < rData.getNMovies(); movieId++) {
    double finalValue = movieRatingCount[movieId] *
        rData.getMeanRating() - movieRatingTotal[movieId];
    rhs.set(rData.getNUsers() + movieId, 0, finalValue);
}

Main.writeMatrixToFile(optimisation, "opt");
```

```
Main.writeMatrixToFile(rhs, "rhs");

String optimiseMatrixString = "[";
optimiseMatrixString +=
    Arrays.toString(optimisation.viewRow(0).toArray()).replaceAll("\\.\\d+",
    "");
for (int row = 1; row < optimisation.rows(); row++) {
   optimiseMatrixString += ", "
        +
            Arrays.toString(optimisation.viewRow(row).toArray()).replaceAll("\\.\\d+
            "");
}
optimiseMatrixString += "]";
String rhsVectorString =
    Arrays.toString(rhs.viewColumn(0).toArray());

Double[] biases = new Double[optimisation.rows()];
try {
   // Write a python script to solve for the biases to a file
   String fileName = Main.writeStringToFile(
        "import numpy\na=" + optimiseMatrixString + "\nb=" +
            rhsVectorString
              + "\nresult=numpy.linalg.lstsq(a,b)[0]\nfor item in
                  result:\n\tprint item",
        "calculateBiases", ".py");
   // Run the python script and capture the result
   Process p = Runtime.getRuntime().exec("python " + fileName);
   BufferedReader stdInput = new BufferedReader(new
       InputStreamReader(p.getInputStream()));
   String s = null;
   int i = 0;
   while ((s = stdInput.readLine()) != null) {
      biases[i] = Double.parseDouble(s);
      i++;
   }
   Main.writeStringToFile(Arrays.toString(biases), "biases",
```

```
            ".txt");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Set the ArrayLists holding the biases
        for (int userId = 0; userId < rData.getNUsers(); userId++) {
            double userBias = biases[userId];
            userBiases.add(userBias);
        }

        for (int movieId = 0; movieId < rData.getNMovies(); movieId++) {
            double movieBias = biases[rData.getNUsers() + movieId];
            movieBiases.add(movieBias);
        }

        // Create the normalised matrix
        DoubleMatrix2D ratingsMatrix = rData.getMatrix();
        normalised = DoubleFactory2D.sparse.make(rData.getNUsers(),
            rData.getNMovies(), Main.MATRIX_EMPTY_VAL);
        for (int row = 0; row < normalised.rows(); row++) {
            for (int col = 0; col < normalised.columns(); col++) {
                if (ratingsMatrix.get(row, col) == Main.MATRIX_EMPTY_VAL) {
                    normalised.set(row, col, Main.MATRIX_EMPTY_VAL);
                } else {
                    if (subtractMean) {
                        normalised.set(row, col, ratingsMatrix.get(row, col) -
                            rData.getMeanRating()
                              - userBiases.get(row) - movieBiases.get(col));
                    } else {
                        normalised.set(row, col,
                              ratingsMatrix.get(row, col) - userBiases.get(row)
                                  - movieBiases.get(col));
                    }
                }
            }
```

```
    }

}

// Subtract only the average rating from all non-empty cells
public void normaliseNoBiases() {
    DoubleMatrix2D ratingsMatrix = rData.getMatrix();
    normalised = DoubleFactory2D.sparse.make(rData.getNUsers(),
        rData.getNMovies(), Main.MATRIX_EMPTY_VAL);
    for (int row = 0; row < normalised.rows(); row++) {
        for (int col = 0; col < normalised.columns(); col++) {
            if (ratingsMatrix.get(row, col) == Main.MATRIX_EMPTY_VAL) {
                normalised.set(row, col, Main.MATRIX_EMPTY_VAL);
            } else {
                normalised.set(row, col, ratingsMatrix.get(row, col) -
                    rData.getMeanRating());
            }
        }
    }
}

// Add the previously removed biases and average rating (if previously
// removed) to the entire matrix
public DoubleMatrix2D unNormaliseMatrix(DoubleMatrix2D
    normalisedMatrix, boolean didSubtractMean) {
    DoubleMatrix2D unNormalisedMatrix = normalisedMatrix.copy();
    for (int row = 0; row < normalisedMatrix.rows(); row++) {
        for (int col = 0; col < normalisedMatrix.columns(); col++) {
            double unNormalisedValue = unNormalise(row, col,
                normalisedMatrix.get(row, col), didSubtractMean);
            unNormalisedMatrix.set(row, col, unNormalisedValue);
        }
    }
    return unNormalisedMatrix;
}
```

```
// Add the previously removed biases and average rating (if previously
// removed) to a single cell
public double unNormalise(int row, int col, double normalisedValue,
    boolean didSubtractMean) {
   return (didSubtractMean ? rData.getMeanRating() : 0) +
       userBiases.get(row) + movieBiases.get(col)
         + normalisedValue;
}

// Subtract only the average rating from all non-empty cells
public DoubleMatrix2D unNormaliseNoBiases(DoubleMatrix2D
    normalisedMatrix) {
   DoubleMatrix2D unNormalisedMatrix = normalisedMatrix.copy();
   for (int row = 0; row < normalisedMatrix.rows(); row++) {
      for (int col = 0; col < normalisedMatrix.columns(); col++) {
         double unNormalisedValue = normalisedMatrix.get(row, col) +
             rData.getMeanRating();
         unNormalisedMatrix.set(row, col, unNormalisedValue);
      }
   }
   return unNormalisedMatrix;
}

// Get the mean rating of the normalised matrix
public double getNormalisedMean() {
   double total = 0;
   int count = 0;
   for (int row = 0; row < normalised.rows(); row++) {
      for (int col = 0; col < normalised.columns(); col++) {
         double cellValue = normalised.get(row, col);
         if (cellValue != Main.MATRIX_EMPTY_VAL) {
            total += cellValue;
            count++;
         }
      }
   }
```

```
        double normalisedMean = total / (double) count;
        return normalisedMean;
    }

    public DoubleMatrix2D getNormalised() {
        return normalised;
    }

    public ArrayList<Double> getUserBiases() {
        return userBiases;
    }

    public ArrayList<Double> getMovieBiases() {
        return movieBiases;
    }

}
```

Neighbourhood.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

import cern.colt.matrix.DoubleMatrix2D;

public class Neighbourhood {

    // Similarity scores are stored as a 2D list.
    // First dimension is ordered by user / movie
    // Second dimension is ordered by |similarityScore|
    private List<List<SimilarityScore>> userSimilarityScores;
    private List<List<SimilarityScore>> movieSimilarityScores;

    private DoubleMatrix2D M;
```

```
private int nUsers;
private int nMovies;

public Neighbourhood(DoubleMatrix2D M) {

   this.M = M;
   this.nUsers = M.rows();
   this.nMovies = M.columns();

}

// Calculate pairwise movie similarity scores
public void calculateMovieSimilarityScores() {

   movieSimilarityScores = new ArrayList<List<SimilarityScore>>();
   for (int i = 0; i < nMovies; i++) {
      movieSimilarityScores.add(i, new ArrayList<SimilarityScore>());
   }

   double sumOfProductOfRatings = 0;
   double sumOfSquareOfRatingsI = 0;
   double sumOfSquareOfRatingsJ = 0;

   // For each movie pair
   for (int movieI = 0; movieI < nMovies; movieI++) {
      System.out.println("Similarity [" + movieI + "]");
      for (int movieJ = movieI + 1; movieJ < nMovies; movieJ++) {
         sumOfProductOfRatings = 0;
         sumOfSquareOfRatingsI = 0;
         sumOfSquareOfRatingsJ = 0;
         int numMatches = 0;
         // Calculate the similarity score
         for (int user = 0; user < nUsers; user++) {
            double ratingForI = M.get(user, movieI);
            double ratingForJ = M.get(user, movieJ);
```

```
            // Find all users that rated both movies
            if (ratingForI != Main.MATRIX_EMPTY_VAL && ratingForJ !=
                Main.MATRIX_EMPTY_VAL) {
               numMatches ++;
               sumOfProductOfRatings += ratingForI * ratingForJ;
               sumOfSquareOfRatingsI += ratingForI * ratingForI;
               sumOfSquareOfRatingsJ += ratingForJ * ratingForJ;
            }
         }
         int THRESHOLD = 5;
         if (numMatches > THRESHOLD) {
            double similarityScore = sumOfProductOfRatings
                  / Math.sqrt(sumOfSquareOfRatingsI *
                      sumOfSquareOfRatingsJ);
            movieSimilarityScores.get(movieI).add(new
                SimilarityScore(movieJ, similarityScore));
            movieSimilarityScores.get(movieJ).add(new
                SimilarityScore(movieI, similarityScore));
         }
      }
      // Sort by |similarityScore|
      Collections.sort(movieSimilarityScores.get(movieI), new
          Comparator<SimilarityScore>() {
         public int compare(SimilarityScore left, SimilarityScore
             right) {
            double difference = Math.abs(left.getSimilarityScore()) -
                Math.abs(right.getSimilarityScore());
            if (difference < 0)
               return 1;
            if (difference > 0)
               return -1;
            return 0;
         }
      });
   }
```

```
}

// Calculate pairwise user similarity scores
public void calculateUserSimilarityScores() {

   userSimilarityScores = new ArrayList<List<SimilarityScore>>();
   for (int i = 0; i < nUsers; i++) {
      userSimilarityScores.add(i, new ArrayList<SimilarityScore>());
   }

   double sumOfProductOfRatings = 0;
   double sumOfSquareOfRatingsI = 0;
   double sumOfSquareOfRatingsJ = 0;

   // For each user pair
   for (int userI = 0; userI < nUsers; userI++) {
      System.out.println("Similarity [" + userI + "]");
      for (int userJ = userI + 1; userJ < nUsers; userJ++) {
         sumOfProductOfRatings = 0;
         sumOfSquareOfRatingsI = 0;
         sumOfSquareOfRatingsJ = 0;
         int numMatches = 0;
         // Calculate the similarity score
         for (int movie = 0; movie < nMovies; movie++) {
            double ratingForI = M.get(userI, movie);
            double ratingForJ = M.get(userJ, movie);
            // Find all movies rated by both users
            if (ratingForI != Main.MATRIX_EMPTY_VAL && ratingForJ !=
                Main.MATRIX_EMPTY_VAL) {
               numMatches ++;
               sumOfProductOfRatings += ratingForI * ratingForJ;
               sumOfSquareOfRatingsI += ratingForI * ratingForI;
               sumOfSquareOfRatingsJ += ratingForJ * ratingForJ;
            }
         }
         int THRESHOLD = 5;
```

```
            if (numMatches > THRESHOLD) {
               double similarityScore = sumOfProductOfRatings
                     / Math.sqrt(sumOfSquareOfRatingsI *
                         sumOfSquareOfRatingsJ);
               userSimilarityScores.get(userI).add(new
                   SimilarityScore(userJ, similarityScore));
               userSimilarityScores.get(userJ).add(new
                   SimilarityScore(userI, similarityScore));
            }
         }
         // Sort by |similarityScore|
         Collections.sort(userSimilarityScores.get(userI), new
             Comparator<SimilarityScore>() {
            public int compare(SimilarityScore left, SimilarityScore
                right) {
               double difference = Math.abs(left.getSimilarityScore()) -
                   Math.abs(right.getSimilarityScore());
               if (difference < 0)
                  return 1;
               if (difference > 0)
                  return -1;
               return 0;
            }
         });
      }

   }

   // Calculate predictions from user similarities
   public DoubleMatrix2D calculatePredictionsFromUserSimilarities(int
       minNeighbourhoodSize, int maxNeighbourhoodSize) {

      DoubleMatrix2D predictions = M.copy();

      for (int user = 0; user < predictions.rows(); user++) {
         for (int movie = 0; movie < predictions.columns(); movie++) {
```

```
        if (predictions.get(user, movie) == Main.MATRIX_EMPTY_VAL) {

            double topSum = 0;
            double bottomSum = 0;
            int neighbourCount = 0;
            for (int i = 0; i < userSimilarityScores.get(user).size();
                i++) {
               SimilarityScore s =
                   userSimilarityScores.get(user).get(i);
               int neighbourUserId = s.getId();
               double weight = s.getSimilarityScore();
               double neighbourRating =
                   predictions.get(neighbourUserId, movie);
               if (neighbourRating != Main.MATRIX_EMPTY_VAL) {
                  neighbourCount++;
                  topSum += weight * neighbourRating;
                  bottomSum += Math.abs(weight);
               }
               if (neighbourCount == maxNeighbourhoodSize) {
                  break;
               }
            }

            if (neighbourCount >= minNeighbourhoodSize) {
               double finalPrediction = topSum / bottomSum;
               predictions.set(user, movie, finalPrediction);
            } else {
               predictions.set(user, movie, 0);
            }

         }
      }
   }

   return predictions;
}
```

```
// Calculate predictions from movie similarities
public DoubleMatrix2D calculatePredictionsFromMovieSimilarities(int
    minNeighbourhoodSize,
      int maxNeighbourhoodSize) {

   DoubleMatrix2D predictions = M.copy();

   for (int user = 0; user < predictions.rows(); user++) {
      for (int movie = 0; movie < predictions.columns(); movie++) {
         if (predictions.get(user, movie) == Main.MATRIX_EMPTY_VAL) {

            double topSum = 0;
            double bottomSum = 0;
            int neighbourCount = 0;
            for (int i = 0; i <
                movieSimilarityScores.get(movie).size(); i++) {
               SimilarityScore s =
                   movieSimilarityScores.get(movie).get(i);
               int neighbourMovieId = s.getId();
               double weight = s.getSimilarityScore();
               double neighbourRating = predictions.get(user,
                   neighbourMovieId);
               if (neighbourRating != Main.MATRIX_EMPTY_VAL) {
                  neighbourCount++;
                  topSum += weight * neighbourRating;
                  bottomSum += Math.abs(weight);
               }
               if (neighbourCount == maxNeighbourhoodSize) {
                  break;
               }
            }

            if (neighbourCount >= minNeighbourhoodSize) {
               double finalPrediction = topSum / bottomSum;
               predictions.set(user, movie, finalPrediction);
```

```
            } else {
               predictions.set(user, movie, 0);
            }

         }
      }
   }

   return predictions;
}

}
```

SimilarityScore.java

```
public class SimilarityScore {

   private int id;
   private double similarityScore;

   public SimilarityScore(int id, double similarityScore) {
      this.id = id;
      this.similarityScore = similarityScore;
   }

   public int getId() {
      return id;
   }

   public double getSimilarityScore() {
      return similarityScore;
   }

}
```

LatentFactor.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import cern.colt.matrix.DoubleFactory2D;
import cern.colt.matrix.DoubleMatrix2D;
import cern.colt.matrix.linalg.Algebra;

public class LatentFactor {

   public static final int MATRIX_U = 1;
   public static final int MATRIX_V = 2;

   private Algebra algebra = new Algebra();

   private int matrixURows;
   private int matrixUCols;
   private int matrixVRows;
   private int matrixVCols;

   private int factors;
   private int maxIterations;

   private int nUsers;
   private int nMovies;

   private double rmseImprovementThreshold;

   private DoubleMatrix2D M;
   private DoubleMatrix2D U;
   private DoubleMatrix2D V;

   private RatingsData rData;

   private int testId;
```

```
public LatentFactor(int testId, RatingsData rData, DoubleMatrix2D M,
    int factors, int maxIterations, double initValue, double
    rmseImprovementThreshold) {

   this.testId = testId;

   this.factors = factors;
   this.maxIterations = maxIterations;
   this.nUsers = M.rows();
   this.nMovies = M.columns();

   this.M = M;
   this.U = DoubleFactory2D.dense.make(nUsers, factors, initValue);
   this.V = DoubleFactory2D.dense.make(factors, nMovies, initValue);

   this.rData = rData;
   this.rmseImprovementThreshold = rmseImprovementThreshold;

   this.matrixURows = nUsers;
   this.matrixUCols = factors;
   this.matrixVRows = factors;
   this.matrixVCols = nMovies;
}

// Get a list of all cells in U and V, so that we can form
    permutations.
private List<int[]> getAllCells() {
   List<int[]> allMatrixUVEntries = new ArrayList<int[]>();
   for (int row = 0; row < nUsers; row++) {
      for (int col = 0; col < factors; col++) {
         allMatrixUVEntries.add(new int[] { MATRIX_U, row, col });
      }
   }
   for (int row = 0; row < factors; row++) {
      for (int col = 0; col < nMovies; col++) {
```

```
            allMatrixUVEntries.add(new int[] { MATRIX_V, row, col });
        }
    }
    return allMatrixUVEntries;
}

// Perform as many rounds of optimisation as necessary until the rmse
    improvement threshold is met
public void runAllIterations() {

    double previousRMSE = 9999;

    List<int[]> allCellEntries = getAllCells();
    System.out.println("====================");
    for (int iteration = 1; iteration <= maxIterations; iteration++) {
        System.out.println("Iteration: "+iteration);
        Collections.shuffle(allCellEntries);
        runIteration(allCellEntries);
        Main.writeMatrixToFile(U, String.format("%02d_matrixU_%03d",
            testId, iteration));
        Main.writeMatrixToFile(V, String.format("%02d_matrixV_%03d",
            testId, iteration));

        DoubleMatrix2D product = getProductUV();
        if (iteration % 20 == 0) {
            Main.writeMatrixToFile(product,
                String.format("%02d_prodUV_%03d", testId, iteration));
        }

        double rmse = RMSE.getRMSE(M, product, rData);
        double rmseDiff = previousRMSE - rmse;
        previousRMSE = rmse;
        if (rmseDiff < rmseImprovementThreshold) {
            System.out.println("rmse = "+rmse);
            System.out.println("rmseDiff = "+rmseDiff);
            break;
```

```
        }
    }
}

// Perform a round of optimising all the elements of U and V.
private void runIteration(List<int[]> cellPermutation) {
    long start = System.currentTimeMillis();
    for (int[] cell : cellPermutation) {
        int matrixId = cell[0];
        int row = cell[1];
        int col = cell[2];
        optimiseCell(matrixId, row, col);
    }
    System.out.println("Iteration: " + (System.currentTimeMillis() -
        start) + "ms");
}

// Optimise the cell using the optimisation equations for an
    arbitrary element
public void optimiseCell(int matrixId, int chosenRow, int chosenCol) {
    if (matrixId == MATRIX_U) {
        double topSum = 0;
        double bottomSum = 0;
        for (int j = 0; j < matrixVCols; j++) {
            // Check if original matrix is non-empty
            if (M.get(chosenRow, j) != Main.MATRIX_EMPTY_VAL) {
                double v_sj = V.get(chosenCol, j);
                double m_rj = M.get(chosenRow, j);
                double multFromMat = 0;
                for (int k = 0; k < matrixUCols; k++) {
                    if (k != chosenCol) {
                        double u = U.get(chosenRow, k);
                        double v = V.get(k, j);
                        multFromMat += u * v;
                    }
                }
```

```
            topSum += v_sj * (m_rj - multFromMat);
        }
    }

    for (int j = 0; j < matrixVCols; j++) {
        // Check if original matrix is non-empty
        if (M.get(chosenRow, j) != Main.MATRIX_EMPTY_VAL) {
            double v_ir = V.get(chosenCol, j);
            bottomSum += v_ir * v_ir;
        }
    }

    double optimised = topSum / bottomSum;
    // This should never happen
    // Since we eliminate any 0 rows / columns
    if (Double.isNaN(optimised)) {
        System.out.println(String.format("row: %d, col: %d",
            chosenRow, chosenCol));
        System.out.println(String.format("top: %f, bottom: %f",
            topSum, bottomSum));
        System.exit(0);
    }
    U.set(chosenRow, chosenCol, optimised);

} else if (matrixId == MATRIX_V) {

    double topSum = 0;
    double bottomSum = 0;
    for (int i = 0; i < matrixURows; i++) {
        // Check if original matrix is non-empty
        if (M.get(i, chosenCol) != Main.MATRIX_EMPTY_VAL) {
            double u_ir = U.get(i, chosenRow);
            double m_is = M.get(i, chosenCol);
            double multFromMat = 0;
            for (int k = 0; k < matrixVRows; k++) {
                if (k != chosenRow) {
```

```
                double u = U.get(i, k);
                double v = V.get(k, chosenCol);
                multFromMat += u * v;
            }
         }
         topSum += u_ir * (m_is - multFromMat);
      }
   }

   for (int i = 0; i < matrixURows; i++) {
      // Check if original matrix is non-empty
      if (M.get(i, chosenCol) != Main.MATRIX_EMPTY_VAL) {
         double v_ir = U.get(i, chosenRow);
         bottomSum += v_ir * v_ir;
      }
   }

   double optimised = topSum / bottomSum;
   if (Double.isNaN(optimised)) {
      // This should never happen
      // Since we eliminate any 0 rows / columns
      System.out.println(String.format("row: %d, col: %d",
          chosenRow, chosenCol));
      System.out.println(String.format("top: %f, bottom: %f",
          topSum, bottomSum));
      System.exit(0);
   }
   V.set(chosenRow, chosenCol, optimised);

  }
}

public DoubleMatrix2D getProductUV() {
   DoubleMatrix2D prod = algebra.mult(U, V);
   return prod;
}
```

```
   public DoubleMatrix2D getMatrixM() {
      return M;
   }

   public DoubleMatrix2D getMatrixU() {
      return U;
   }

   public DoubleMatrix2D getMatrixV() {
      return V;
   }

}
```