Item Based Collaborative Filtering using Hadoop MapReduce in Java

Data Analytics lab project report submitted

by

Aman Chopra

Reg. No. 140911358

Roll. No. 44(A2)

and

Suvimal Yashraj

Reg. No. 140911394

Roll. No. 51(A2)

Department of

Information and Comunication Technology,

MIT, Manipal

MANIPAL
INSTITUTE OF TECHNOLOGY
A Constituent Institute of Manipal University, Manipal

November 2017

ABSTRACT

Recommendation system is based on the historical records of user access, purchasing records and relation between items to construct interest model, but in this era of information explosion, in which we always get huge amounts of information, the efficiency of a single common computer will not satisfy the requirement and the super computer will cost too much [1]. In order to solve the problem, we have tried to use MapReduce to implement the recommendation system.

In this project, we have built a movie recommendation system based on **Item Collaborative Filtering** using Hadoop MapReduce in Java. Data comes from the training dataset of Netflix Prize Challenge.

The item-based collaborative filtering recommendation algorithm is the most widely used recommendation algorithm . Its principle is based on the users evaluation of items. The purpose is to find the similarity between users, and recommend items to the target user according to the records of the similar users. Based on the users ratings of the movies, our project recommends other movies that the user may be interested in.

Keywords: Item-based collaborative filtering; recommendation; MapReduce.

Contents

List of Figures

ABBREVIATIONS

UID	:	User Identification Number
MID	:	Movie Identification Number
nR	:	Normalized Relation

Chapter 1

Introduction

1.1 Recommendation system

In the era of information explosion, the recommendation system is one of most useful domain to help users get the interesting information in a short time. Recommendation system is based on the historical records of user access, purchasing records and relation between items to construct interest model, and with the user interest model of multifarious information, finally the system will recommend users the items they might be interested in. The collaborative filtering is one of the most popular techniques used in recommendation system. Item-based collaborative filtering recommendation systems uses the similarities of items to make recommendations to users.

1.2 Problem Statement

In this project, we are implementing a movie recommendation system. We are using Map Reduce model for high computation power and item based collaborative filtering algorithm which calculates similarity between items to recommend top movies to users according to their preference. This is an extremely important algorithm that is used in many domains like suggesting users places to go, products to buy etc., and if not implemented, users have

to manually search which is not feasible in today's world of humongous data.

1.2.1 Data set

Data comes from the training dataset of Netflix Prize Challenge. The file "training_set.tar" is a tar of a directory containing 17770 files, one per movie [2]. The first line of each file contains the movie id followed by a colon. Each subsequent line in the file corresponds to a rating from a customer and its date in the following format: CustomerID, Rating, Date.

- MovieIDs range from 1 to 17770 sequentially.
- CustomerIDs range from 1 to 2649429, with gaps. There are 480189 users.
- Ratings are on a five star scale from 1 to 5.
- Dates have the format YYYY-MM-DD.

1.3 Proposed Idea

Due to the continuous development of the global information industry, it becomes more and more difficult for network resource and data analysis technology to adapt to the demand in present era of the intensive data processing. In recent years, the cloud computing emerges. It is a kind of resource on-demand renting service mode, where users can access the data in the computer and storage system according to the requirement. The network of the computer resources, virtual as a resource pool, and the specific software are used to realize automatic and intelligent computing.

So it allows a variety of computing resources to work together. Based on this technology, Google Labs put forward MapReduce model as shown in figure 1.1 in the cloud computing, which is specialized in parallel computation

of large data sets. In our project, we realize a **pipeline item- based collaborative filtering by using MapReduce** as shown in figure 1.2 to make a **Movie Reccomender System**. The map operations on the input data can calculate an intermediate key/value pairs. Properly combining the data, we apply reduce operation in all the value which have the same key. The use of function model, combined with the user to specify the map and reduce operations, so we can easily realize the **large-scale parallel computing**.

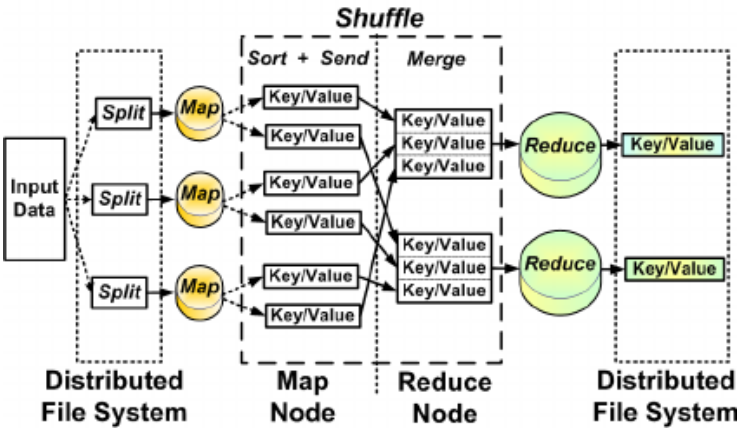

Figure 1.1: Map Reduce Model

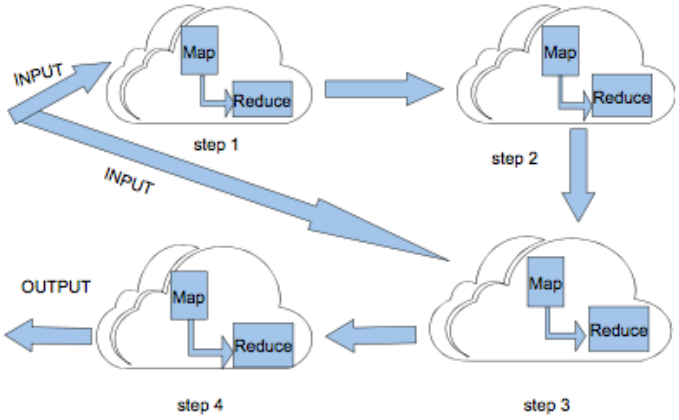

Figure 1.2: Pipeline Map Reduce model

At the same time we use the restart mechanism which can easily realize fault tolerance. Like MapReduce, this parallel computing model of analyzing large data sets has formed a cluster revolution in the industry. The current

software implementation is to specify a map function, using a set of keys for mapping into a new set of key/value pairs, specifying the concurrent reduce function, to ensure that all the keys of the map for each of the group share the same key.

1.4 Importance of Project

As we have seen that Map Reduce model provides high parallelism which is extremely important in today's world of big data. This model is important because apart from being used as a movie recommender system, this algorithm is/can be used in many other places like suggesting products to customer based on their and other customers ratings of other products and other frequent item set mining applications. Apart from that, the incorporation of Map Reduce makes recommendation extremely fast due to its parallel computing capabilities and it works well with huge amount of data which we actually have courtesy the increasing text and data mining techniques.

Chapter 2

Objectives

In this project, we want to recommend top k movies for all the users, where k is a predetermined value. The k movies are arranged in the ascending order with a rating on the scale of 1 to 5.

2.1 Data Pre-processing

We preprocessed the original dataset in two steps:

1. Change the data in each movie file into the following format: UserID, MovieID, Rating.

2. Merge 17770 movie files into one big input file since Hadoop is not good for dealing with lots of small files. And the big input file is the input of our recommender system.

Figure 2.1 shows the format of input file.

2.2 Modules

- Divide data by user id.

- Build co-occurrence matrix.

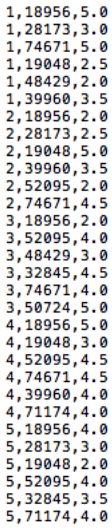

Figure 2.1: Input file

- Normalize the co-occurrence matrix.
- Build Rating Matrix.
- Multiply co-occurrence matrix and rating matrix.
- Generate Recommendation list.

2.3 How to run

hadoop jar recommender.jar Driver /input /dataDividedByUser /coOccurrenceMatrix /normalizedMatrix /multiplicationMapperJoin /multiplicationSum /recommender

- args0: original dataset.
- args1: output directory for DividerByUser job.
- args2: output directory for coOccurrenceMatrixBuilder job.
- args3: output directory for NormalizeCoOccurrenceMatrix job.
- output directory for MultiplicationMapperJoin job.

- output directory for MultiplicationSum job.
- output directory for RecommenderListGenerator job.

Chapter 3

Methodology

3.1 Algorithm

Item-item collaborative filtering, or item-based, or item-to-item, is a form of collaborative filtering for recommender systems based on the similarity between items calculated using people's ratings of those items as shown in figure figure 3.1. First, the system executes a model-building stage by finding the similarity between all pairs of items. This similarity function can take many forms, such as correlation between ratings, weighted mean or cosine of those rating vectors. As in user-user systems, similarity functions can use normalized ratings (correcting, for instance, for each user's average rating).

Second, the system executes a recommendation stage. It uses the most similar items to a user's already-rated items to generate a list of recommendations. Usually this calculation is a weighted sum or linear regression. This form of recommendation is analogous to "people who rate item X highly, like you, also tend to rate item Y highly, and you haven't rated item Y yet, so you should try it" [3].

	Item 1	Item 2	...	Item j	...	Item n
User 1	1.0	0	...	3.0	...	2.0
User 2	2.0	3.0	...	4.0	...	5.0
...	...	...	...	...	...	...
User i	4.0	0	...		...	0
...	...	...	...	...	...	...
User m	0	1.0	...	4.0	...	1.0

Figure 3.1: User Rating matrix

3.2 Explanation

If we have user's preference for some items then we can predict the preference in other items using weighted average or other similarity indexes to find similarity between item that the user has rated and items that has not been rated by the user. Figure **??** shows the item-item similarity notion.

```
for every item i that u has no preference for yet
  for every item j that u has a preference for
    compute a similarity s between i and j
    add u's preference for j, weighted by s,
          to a running average
return top items, ranked by weighted average
```

Figure 3.2: Item-item similarity notion

Item-item similarity could be based on content but in collaborative filtering, it is based only on preference/numbers [4]. To calculate the similarity, different methods can be used like pearson correlation and log likelihood ration but in our project, we have used co-occurrence i.e., items similar, when appearing often in the same user's set of preferences as shown in figure 3.3.

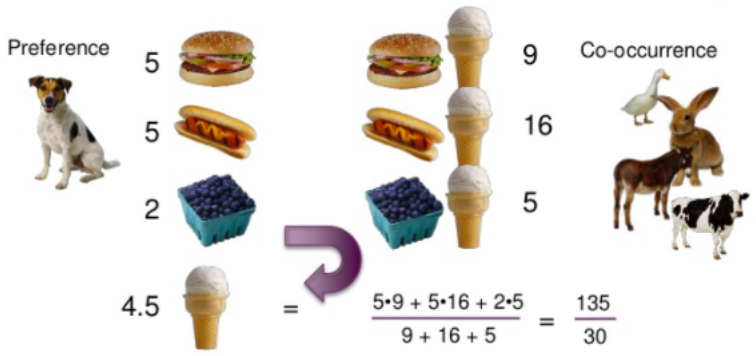

Figure 3.3: Co-occurrence of items

User's preferences are a vector where each dimension corresponds to one item and the dimension value is the preference value . Item-item co-occurrence matrix is a matrix where row i and column j is the count of item i/j co-occurrence [5]. The value of the preference is:

$co - occurrencematrix * preferencecolumnvector$

Figure 3.4 shows the approach.

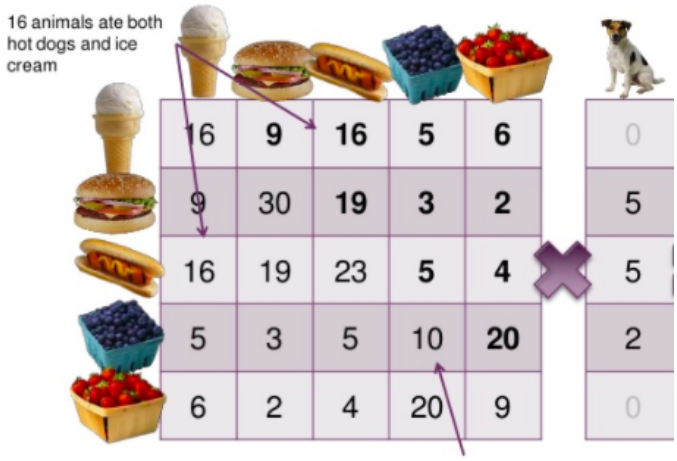

Figure 3.4: Generation of final preferences

Since the preference vector is a column vector, the matrix multiplication becomes easier using the process as shown in figure 3.5. Inside out multiplication:

1. Multiply scalar with corresponding matrix column.

2. Yield column vector.

3. Sum these to get result for a particular user.

4. Can skip for zero vector elements.

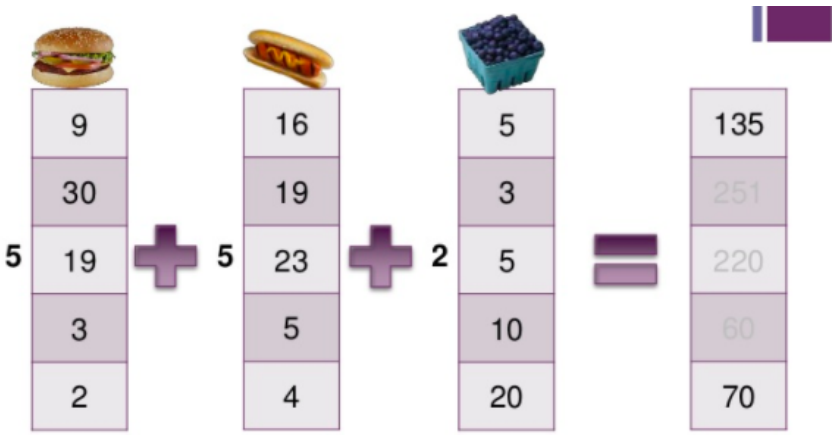

Figure 3.5: Matrix multiplication

3.3 Implementation in map-reduce

In this project we have made seven mapper and six reducers and use the concept of pipe lining to send the output of one reducer to the mapper of the next mapper in order to implement all the modules listed in section 2.2. The detailed implementation of all the mappers and reducers along with the key-value pairs is shown below:

Mapper 1:

Input: $UID, MID, Rating$

Output: $UID < MID : Rating >$

Reducer 1:

Input: $UID < MID : Rating >< MID : Rating > - - - - -$

Output: $UID < MID : Rating, MID : Rating, - - - - -$

Mapper 2:

Function: $MID_1 : MID_2 < 1 >$

Reducer 2:

Input: $MID_1 : MID_2 < 1 >< 1 >< 1 > - - - - -$

Output: $MID_1 : MID_2 < sum >$

Mapper 3:

Function: $MID_2 < MID_1 : sum >$

Reducer 3:

Input: $MID_2 < MID_1 : sum_1 >< MID_3 : sum_3 > - - - - -$

Function: Makes a hash map of movie id and sum to calculate Normalized relation.

$$\begin{bmatrix} MID_1 & sum_1 \\ MID_2 & sum_2 \\ \vdots & \vdots \\ MID_n & sum_n \end{bmatrix} => \begin{bmatrix} MID_1 & sum_1/total \\ MID_2 & sum_2/total \\ \vdots & \vdots \\ MID_n & sum_n/total \end{bmatrix}$$

where $total = sum_1 + sum_2 + - - - - - + sum_n$

Output: $MID1 : MID2 < nR >$

Mapper 4:

Function: $MID_1 <$ "$coOccurrence$" $: MID_2 : nR > - - - - -$

Mapper 5:

Function: $MID <$ "$rate$" $: UID : rating > - - - - -$

Reducer 4:

Input : $MID_1 <$ "$coOccurrence$" $: MID_2 : nR >< $ "$rate$", $UID : rating >$

Function: Makes two hash maps, one of movie id and normalized relation and the other of UID and rating to calculate score of movie.

$$\begin{bmatrix} MID_2 & nR_2 \\ MID_3 & nR_3 \\ \vdots & \vdots \\ MID_n & nR_n \end{bmatrix} and \begin{bmatrix} UID_1 & rating_1 \\ UID_2 & rating_2 \\ \vdots & \vdots \\ UID_n & rating_3 \end{bmatrix}$$

where scores are $< rating_1 * nr_2 >, < rating_1 * nr_3 >, - - - - -$

Output: For a particular movie id, we get,

$UID_1 : MID_2 < score >, UID_1 : MID_3 < score > - - - - -$

Mapper 6:

Input: $UID_1 : MID_2 < score >$

Output: $UID_1 : MID_2 < score >$

This output is same as input but we need the mapper for reducer.

Reducer 5:

Input: $UID_1 : MID_2 < score_1 >< score_2 > - - - - -$

$sum = score_1 + score_2 + - - - - -$

Output: $UID < MID : sum >$

Mapper 7 and Reducer 6 are used to generate a heap data structure to show the top k movies in ascending order for all the users.

Chapter 4

Results

We have ran our code in Hadoop cluster in a single node and as the size of the input increased, the performance efficiency was not deteriorated owning to the high parallel computing efficiency of Map Reduce.

The result is shown in figure 4.1 and we have seen that this recommender system is effective because it gave high rating to movies of same genre that a particular user liked according to his previous ratings.

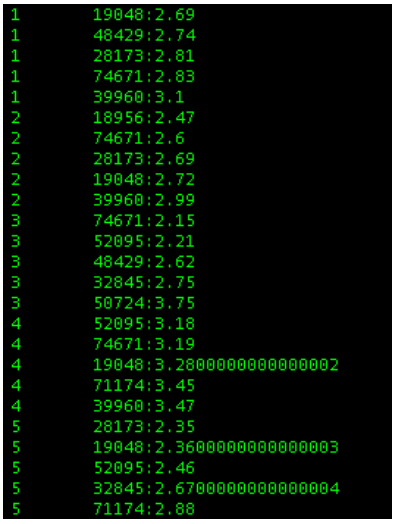

Figure 4.1: Result

Chapter 5

Conclusion and Future Works

Recommendation system is a very popular topic and it has brought great convenience to our daily life. With the development of science and technology, recommendation system was applied in more and more situations. But with the growth of data, recommendation system needs more time and power to meet the requirements. There should be a new scheme to solve the problem.

n this project, we execute the traditional item-based collaborative filtering recommendation algorithm on MapReduce by splitting the whole big dataset into some smaller datasets. In this way, we can build a powerful computing cluster by collecting the computing ability of many common PCs to process a large scale dataset efficiently.

Results show that the proposed method shortens the execution time of recommendation system. But there are still many work we should do in the future. In this project, we just propose a method to execute the traditional item-based collaborative filtering on Hadoop. As the development of the recommendation system, the quality of the recommendation algorithm will be improved but the algorithm may be more complex.

References

[1] [Online]. Available: https://en.wikipedia.org/wiki/MapReduce

[2] [Online]. Available: https://www.kaggle.com/netflix-inc/netflix-prize-data

[3] [Online]. Available: https://en.wikipedia.org/wiki/Item-item_collaborative_filtering

[4] [Online]. Available: http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=7310709

[5] [Online]. Available: https://www.slideshare.net/huguk/collaborative-filtering-at-scale-2-7671787

Appendices

Appendix A

Code

A.1 Driver

```
public class Driver {
   public static void main(String[] args) throws Exception {

       /*
           args0: original dataset
           args1: output directory for DividerByUser job
           args2: output directory for coOccurrenceMatrixBuilder job
           args3: output directory for NormalizeCoOccurrenceMatrix
               job
           args4: output directory for MultiplicationMapperJoin job
           args5: output directory for MultiplicationSum job
           args6: output directory for RecommenderListGenerator job
       */

     DataDividerByUser dataDividerByUser = new DataDividerByUser();
     CoOccurrenceMatrixGenerator coOccurrenceMatrixGenerator = new
         CoOccurrenceMatrixGenerator();
```

```
        NormalizeCoOccurrenceMatrix normalizeCoOccurrenceMatrix = new
            NormalizeCoOccurrenceMatrix();
        MultiplicationMapperJoin mapperJoin = new
            MultiplicationMapperJoin();
        MultiplicationSum multiplicationSum = new MultiplicationSum();
        RecommenderListGenerator generator = new
            RecommenderListGenerator();

        String[] path1 = {args[0], args[1]};
        String[] path2 = {args[1], args[2]};
        String[] path3 = {args[2], args[3]};
        String[] path4 = {args[3], args[0], args[4]};
        String[] path5 = {args[4], args[5]};
        String[] path6 = {args[5], args[6]};

        dataDividerByUser.main(path1);
        coOccurrenceMatrixGenerator.main(path2);
        normalizeCoOccurrenceMatrix.main(path3);
        mapperJoin.main(path4);
        multiplicationSum.main(path5);
        generator.main(path6);
    }

}
```

A.2 Movie

```
public class Movie implements Comparable<Movie>{

    private int movieId;

    private double rating;


    public Movie(int movieId, double rating) {

        this.movieId = movieId;

        this.rating = rating;

    }


    public int getMovieId() {

        return movieId;

    }

    public void setMovieId(int movieId) {

        this.movieId = movieId;

    }

    public double getRating() {

        return rating;

    }

    public void setRating(double rating) {

        this.rating = rating;

    }


    public int compareTo(Movie m) {

        double diff = rating - m.getRating();

        if(diff < 0) {

            return -1;

        } else if(diff > 0) {

            return 1;

        } else {
```

```
            return 0;
        }
    }
}
```

A.3 DataDividedByUser

```
public class DataDividerByUser {
   public static class DataDividerMapper extends
       Mapper<LongWritable, Text, IntWritable, Text> {

      // map method
      @Override
      public void map(LongWritable key, Text value, Context context)
          throws IOException, InterruptedException {

         //input user,movie,rating
         String[] user_movie_rating =
             value.toString().trim().split(",");
         int userID = Integer.parseInt(user_movie_rating[0]);
         String movieID = user_movie_rating[1];
         String rating = user_movie_rating[2];

         context.write(new IntWritable(userID), new Text(movieID +
             ":" + rating));
      }
   }

   public static class DataDividerReducer extends
       Reducer<IntWritable, Text, IntWritable, Text> {
      // reduce method
      @Override
      public void reduce(IntWritable key, Iterable<Text> values,
          Context context)
            throws IOException, InterruptedException {
```

```
        StringBuilder sb = new StringBuilder();
        while (values.iterator().hasNext()) {
           sb.append("," + values.iterator().next());
        }
        //key = user value=movie1:rating, movie2:rating...
        context.write(key, new Text(sb.toString().replaceFirst(",",
            "")));
    }
}
```

A.4 CoOccurrenceMatrixGenerator

```
public class CoOccurrenceMatrixGenerator {

   public static class MatrixGeneratorMapper extends
       Mapper<LongWritable, Text, Text, IntWritable> {


      // map method
      @Override
      public void map(LongWritable key, Text value, Context context)
          throws IOException, InterruptedException {
         //value = userid \t movie1: rating, movie2: rating...
         //key = movie1: movie2 value = 1
         String line = value.toString().trim();
         String[] user_movieRatings = line.split("\t");
         String user = user_movieRatings[0];
         String[] movie_ratings = user_movieRatings[1].split(",");
         //{movie1:rating, movie2:rating..}
         for(int i = 0; i < movie_ratings.length; i++) {
            String movie1 = movie_ratings[i].trim().split(":")[0];


            for(int j = 0; j < movie_ratings.length; j++) {
               String movie2 = movie_ratings[j].trim().split(":")[0];
               context.write(new Text(movie1 + ":" + movie2), new
                   IntWritable(1));
            }
         }

      }
   }
```

```
public static class MatrixGeneratorReducer extends Reducer<Text,
    IntWritable, Text, IntWritable> {
   // reduce method
   @Override
   public void reduce(Text key, Iterable<IntWritable> values,
       Context context)
         throws IOException, InterruptedException {
      //key movie1:movie2 value = iterable<1, 1, 1>

      // if the calculated score is smaller than the threshold, do
          not write this result out in order to optimize
      // the speed of the program

      int threshold = 10;
      int sum = 0;
      while(values.iterator().hasNext()) {
         sum += values.iterator().next().get();
      }

      if(sum > threshold) {
         context.write(key, new IntWritable(sum));
      }
   }
}
```

A.5 NormalizeCoOccurrenceMatrix

```
public class NormalizeCoOccurrenceMatrix {


    public static class NormalizeMatrixMapper extends
        Mapper<LongWritable, Text, IntWritable, Text> {


        // map method
        @Override
        public void map(LongWritable key, Text value, Context
            context) throws IOException, InterruptedException {
            //input movieA:movieB \t relation
            //output movieB \t movieA:relation


            String[] tokens = value.toString().trim().split("\t");
            String[] movies = tokens[0].split(":");


            int movie1 = Integer.parseInt(movies[0]);
            int movie2 = Integer.parseInt(movies[1]);
            int relation = Integer.parseInt(tokens[1]);


            context.write(new IntWritable(movie2), new Text(movie1 +
                ":" + relation));
        }
    }


    public static class NormalizeMatrixReducer extends
        Reducer<IntWritable, Text, Text, DoubleWritable> {
        // reduce method
        @Override
```

```
    public void reduce(IntWritable key, Iterable<Text> values,
        Context context)
            throws IOException, InterruptedException {

        Map<Integer, Double> movieRelationMap = new
            HashMap<Integer, Double>();

        double sum = 0;
        //movie1:relation
        while(values.iterator().hasNext()) {
            String[] tokens =
                values.iterator().next().toString().trim().split(":");
            int movie1 = Integer.parseInt(tokens[0]);
            double relation = Double.parseDouble(tokens[1]);
            sum += relation;
            movieRelationMap.put(movie1, relation);
        }

        for(Map.Entry<Integer, Double> entry:
            movieRelationMap.entrySet()) {
            double normalizedRelation = entry.getValue() / sum;
            DecimalFormat df = new DecimalFormat("#.0000");
            normalizedRelation =
                Double.valueOf(df.format(normalizedRelation));
            context.write(new Text(entry.getKey() + ":" + key),
                new DoubleWritable(normalizedRelation));
        }
    }
}
```

A.6 MultiplicationMapperJoin

```
public class MultiplicationMapperJoin {

    public static class CoOccurrenceMapper extends
        Mapper<LongWritable, Text, IntWritable, Text> {

        // map method
        @Override
        public void map(LongWritable key, Text value, Context
            context) throws IOException, InterruptedException {
            //input movieA: movieB \t normalizedRelation
            //output movieA \t coOccurrence:movieB:normalizedRelation

            String[] tokens = value.toString().trim().split("\t");
            String[] movies = tokens[0].split(":");

            int movie1 = Integer.parseInt(movies[0]);
            int movie2 = Integer.parseInt(movies[1]);
            double relation = Double.parseDouble(tokens[1]);

            context.write(new IntWritable(movie1), new
                Text("coOccurrence:" + movie2 + ":" + relation));
        }
    }

    public static class RateMapper extends Mapper<LongWritable,
        Text, IntWritable, Text> {

        // map method
        @Override
```

```
    public void map(LongWritable key, Text value, Context
        context) throws IOException, InterruptedException {
        //input user,movie,rating
        //output movie \t rate:user:rating
        String[] tokens = value.toString().trim().split(",");
        int user = Integer.parseInt(tokens[0]);
        int movie = Integer.parseInt(tokens[1]);
        double rating = Double.parseDouble(tokens[2]);

        context.write(new IntWritable(movie), new Text("rate:" +
            user + ":" + rating));
    }
}

public static class MapperJoinReducer extends
    Reducer<IntWritable, Text, Text, DoubleWritable> {
    // reduce method
    @Override
    public void reduce(IntWritable key, Iterable<Text> values,
        Context context)
            throws IOException, InterruptedException {

        // if the calculated score is smaller than the threshold,
            do not write this result out in order to optimize
        // the file size of the intermediate jobs
        double threshold = 0.1;

        Map<Integer, Double> movieRelationMap = new
            HashMap<Integer, Double>();
```

```
Map<Integer, Double> ratingMap = new HashMap<Integer,
    Double>();

//user:movie score
while(values.iterator().hasNext()) {
    String[] tokens =
        values.iterator().next().toString().trim().split(":");
    if(tokens[0].equals("coOccurrence")) {
        movieRelationMap.put(Integer.valueOf(tokens[1]),
            Double.valueOf(tokens[2]));
    } else {
        ratingMap.put(Integer.valueOf(tokens[1]),
            Double.valueOf(tokens[2]));
    }
}

for(Map.Entry<Integer, Double> ratingEntry:
    ratingMap.entrySet()) {

    for(Map.Entry<Integer, Double> relationEntry:
        movieRelationMap.entrySet()) {

        double score = relationEntry.getValue() *
            ratingEntry.getValue(); //5 * 8 = 40
        if(score >= threshold) {
            DecimalFormat df = new DecimalFormat("#.00");
            score = Double.valueOf(df.format(score));
            context.write(new Text(ratingEntry.getKey() +
                ":" + relationEntry.getKey()), new
                DoubleWritable(score));
```

```
                }
                // user:movie score
            }


        }
    }
}
```

A.7 MultiplicationSum

```
public class MultiplicationSum {


    public static class MultiplicationSumMapper extends
        Mapper<LongWritable, Text, Text, DoubleWritable> {


        // map method
        @Override
        public void map(LongWritable key, Text value, Context
            context) throws IOException, InterruptedException {
            //input user:movie \t score
            //output user:movie score
            String[] tokens = value.toString().trim().split("\t");
            double score = Double.parseDouble(tokens[1]);
            context.write(new Text(tokens[0]), new
                DoubleWritable(score));
        }
    }


    public static class MultiplicationSumReducer extends
        Reducer<Text, DoubleWritable, IntWritable, Text> {
        // reduce method
        @Override
        public void reduce(Text key, Iterable<DoubleWritable>
            values, Context context)
                throws IOException, InterruptedException {


            // if the calculated score is smaller than the threshold,
                do not write this result out in order to optimize
            // the file size of the intermediate jobs
```

```
        double threshold = 1;

        //user:movie score
        double sum = 0;
        while(values.iterator().hasNext()) {
            sum += values.iterator().next().get();
        }

        if(sum >= threshold) {
            String[] tokens = key.toString().split(":");
            int user = Integer.parseInt(tokens[0]);
            context.write(new IntWritable(user), new
                Text(tokens[1] + ":" + sum));
        }
    }
}
```

A.8 RecommenderListGenerator

```
public class RecommenderListGenerator {
   public static class RecommenderListGeneratorMapper extends
       Mapper<LongWritable, Text, IntWritable, Text> {

      @Override
      public void map(LongWritable key, Text value, Context context)
          throws IOException, InterruptedException {
         //recommender user \t movie:rating
         String[] tokens = value.toString().split("\t");
         int user = Integer.parseInt(tokens[0]);
         int movie = Integer.parseInt(tokens[1].split(":")[0]);
         context.write(new IntWritable(user), new Text(movie + ":" +
             tokens[1].split(":")[1]));
      }
   }

   public static class RecommenderListGeneratorReducer extends
       Reducer<IntWritable, Text, IntWritable, Text> {

      // reduce method
      @Override
      public void reduce(IntWritable key, Iterable<Text> values,
          Context context)
            throws IOException, InterruptedException {

         // Top K: recommend top k movies with highest calculated
             ratings for each user
         int K = 5;
         PriorityQueue<Movie> heap = new PriorityQueue<Movie>();
```

```
        //movie_id:rating
        while(values.iterator().hasNext()) {
           String[] tokens =
               values.iterator().next().toString().trim().split(":");
           int movie_id = Integer.parseInt(tokens[0]);
           double rating = Double.parseDouble(tokens[1]);
           if(heap.size() < K) {
              heap.offer(new Movie(movie_id, rating));
           } else {
              if(heap.peek().getRating() < rating) {
                 heap.poll();
                 heap.offer(new Movie(movie_id, rating));
              }
           }
        }

        while(!heap.isEmpty()) {
           Movie movie = heap.poll();
           context.write(key, new Text(movie.getMovieId() + ":" +
               movie.getRating()));
        }
     }
  }
```