Code Description

1. Recommend System

Because I thought I need to output the recommendation score of all the users by mistake, the output data type and scheme in few places may be slightly different from the given code. The co-occurrence matrix could be more efficient because we only need information related with one user. Overall, there are still five steps to get the sorted recommendation score of a user whose number is 935 (the last three number of my student number).

The steps are described as following:

- Step1: To get score matrix, group the users (same as the given code). The mapper output `<user.id, item.id:score>`. The reducer congregate `item.id:score` with the same key (`user.id`).
- Step2: To build co-occurrence matrix. The mapper read in the output of Step1 and output `<item.id_item.id, 1>`. The reducer aggregates the values of the same key. The output of this step is `< item.id_item.id,# of co-occurrence>.`

```
//Step2
// Step2_UserVectorToCooccurrenceMapper
for (String i : tokens){
                for (String j : tokens){
                    if (!(j.equals(i))) {
                        context.write(new Text(i.split(":")[0]+'_'+j.split(":")[0]),v);
                    }
                }
            }                    //count each co-occurrence of two paired items
            for (String i :tokens){
                context.write(new Text(i.split(":")[0]+'_'+i.split(":")[0]),v);
            }                    //count co-occurrence of two paired items
// Step2_UserVectorToConoccurrenceReducer
Integer N = 0;
            for (IntWritable v:values){
                N += v.get();
            }
            context.write(key,new IntWritable(N));
        }            //aggregate the co-occurrence number
```

- Step3:
  - Step3.1: To get the score information of user with the target user id "935". The mapper read the output of Step1 and output `<item.id, (target_user.id:score)_us>`. Note that "_us" is the label the mapper added to tell if it is the data for scores. There is no self-defined reducer.

```
//Step31_UserVectorSplitterMapper
String[] tokens = Recommend.DELIMITER.split(values.toString());
                                for (String i : tokens){
                                            if(key.toString().equals("935")){
                                                        context.write(new Text(i.split(":")[0]),new
Text(key.toString()+":"+i.split(":")[1]+"_us"));
                                            }
                                }                   // find user "935"'s information and add label "_us"
```

  - Step3.2: To transfer the co-occurrence matrix `< item.id_item.id,# of co-occurrence>` into another formation `< item.id, (item.id:# of co-occurrence)_it>`. The mapper read the output of Step2. Note that "_it" is the label the mapper added to tell if it is the data for scores. There is no self-defined reducer.

```
//Step32
String[] iid = key.toString().split("_");
context.write(new Text(iid[0]),new Text(iid[1]+":"+value.toString()+"_it"));          //transfer into anther formation
```

- Step4
  - Step4.1: To get co-occurrence information and user score information together. The mapper read the output of Step3.1 and Step3.2 and output <item.id, (target_user.id:score)_us> or `<item.id, (item.id:# of co-occurrence)_it >`. The reducer join the `(target_user.id:score)_us` and `(item.id:# of co-occurrence)_it` with the same

item.id together. The reducer output `< item.id, (target_user.id:score)/ (item.id:# of co-occurrence)>.`

```
//Step4_PartialMultiplyMappe
context.write(key,values);
//Step4_AggregateReducer
String it = "";
String us = "";
String k;
for(Text v : values){
  k = v.toString();
  if(k.substring(k.length()-3,k.length()).equals("_it")){
     it = it+"_"+k.substring(0,k.length()-3);
   } else{
     us = us + "_"+k.substring(0,k.length()-3);
   }
}                    //join the co- occurrence and score information together
it.replaceFirst("_","");
us.replaceFirst("_","");
context.write(key, new Text(it+"/"+us));
```

- Step4.2: To calculate the multiple of co- occurrence matrix and score vector. The mapper make the # of pair (`item.i and item.j`) multiple the score of user.935 on item.i which is the impact of user's score on item i on the recommendation score of item j for user.935. Then, the reducer aggregates all the impacts on the recommendation score of the item. j. The reducer outputs `<item.id, recommendation score>`. Please note we do the item filtering which removes the scored item in this step. Specifically, we use mapper to send a indictor (a large negative value actually) to indict the scored item. and do the elimination in the reducer.

```
//Step4_RecommendMapper
String[] iu = value.toString().split("/");
            if(iu.length!=1){
                String[] it = iu[0].split("_");
                String[] us = iu[1].split("_");
                for (int i= 1;i<it.length;i++){
                    String item = it[i];
                    for (int j =1;j<us.length; j++){
                        String user = us[j];
                        String[] items = item.split(":");
                        String [] users = user.split(":");
                        context.write(new Text(items[0]),new DoubleWritable((Double.valueOf(items[1])*Double.valueOf(users[1]))));
//multiplication
                    }
                }
                context.write(key,new DoubleWritable(-100000.0)); //send an indictor show the item has been score, set a large value
            }
//Step4_RecommendReducer
Double rs = 0.0;
            for(DoubleWritable v : values){
                rs += v.get();
            }                    //aggregate scores
            if(rs>=0){
                context.write(key,new DoubleWritable(rs));
            }                    // if the item has been score, the score will be negative and the item is filtered in this step
```

- Step5: To sort the recommendation sore in descending order. The main technique used here is the same as that used in Task1. Just by switching the key and value and reverse the return results of the comparator of Hadoop.io.