Prop\_ronzonip\_IT.doc

**An empirical analysis of collaborative filtering’s algorithms**

Author: Paolo Ronzoni

Stipulation

The text of this project is my own, except where explicitly indicated. I give my permission for this essay to be submitted to the JISC Plagiarism Detection Service.

**Introduction**

This project test different collaborative filtering algorithms, compare the results obtained and analyze how these algorithms can be useful to improve the predictive accuracy of a collaborative recommendation system. I dati utilizzati per testare gli algoritmi di questo progetto provengono dal database Movilens. They consists of:

*100,000 ratings (1-5) from 943 users on 1682 movies.*

*Each user has rated at least 20 movies.*

*Simple demographic info for the users (age, gender, occupation, zip)*

*The data was collected through the MovieLens web site*

*(movielens.umn.edu) during the seven-month period from September 19th,*

*1997 through April 22nd, 1998. This data has been cleaned up - users*

*who had less than 20 ratings or did not have complete demographic*

*information were removed from this data set.*

Detailed descriptions of the data file can be found at the end of this file. (predict areas of the web site a user visited based on data on other areas the user visited).

**Technology Review**

***Recommender Systems***

A Recommendation System or Recommender System (RS) is software that provides users with personalized advice about a set of objects belonging to a specific domain (eg music, movies, books). A Recommender System uses the data, stored in its database, which relate to clients and objects, to provide purchasing advice.

The main idea of a recommender systems is to use the information about the behaviour or opinions of previous clients to predict what product the current customer will purchase or to predict which product, the current customer, will be interested in.

Nowadays, the recommender systems are widely adopted by enterprises, especially large retail firms.

***Types of recommendation systems***

Recommender systems can be divided into four different types:

* Collaborative recommendation
* Content-based recommendation
* Knowledge-based recommendation
* Hybrid recommendation approaches

The first type of recommendation systems is represented by collaborative recommendation. The main idea behind ​​collaborative recommender systems is to use information about the past behaviour of customers, or opinions of a community of users that already exists, in order to predict which products, the current user of the system might be interested in.

The second type of recommendation systems is represented by content-based recommendation. These RS use a series of discrete features of the product in order to recommend alternative products that possess similar characteristics. These systems use neither the past behaviour of customers nor a range of opinions (ratings). Rather, these systems rely on other sources of data to make correct predictions.

The third category of recommendation systems consists of knowledge-based recommendation. These RS are used in all those situations in which collaborative and content-based systems do not provide satisfactory results. That is, those cases in which the product is not purchased frequently: think, for example, buying a house or a car. In such situations, the collaborative and content-based systems do not work due to the limited number of available ratings and due to the wide interval of time between a purchase and the next: if there is a lot of time between the purchase of two similar products can also significantly change the preferences of the customer due to, for example, of changes in income or in the family composition. In such scenarios, knowledge-based systems are used. These recommendation systems use some explicit rules: the user must specify the requirements and the system tries to find the best solution. If it is not possible to find a solution, the user must change the list of requirements.

The last category of recommendation systems consists of "Hybrid recommendation approaches". The name of these recommendation systems comes from the latin word "hybrida": an object made by combining two different elements. The Hybrid recommendation systems try to mix, in a single system of recommendation, approaches and algorithms used by the previous three recommender systems that have been mentioned.

***Collaborative recommendation systems***

The collaborative recommender systems are the most common category of recommender systems. Thanks to the relative ease of implementation they are used both by the large sites of e-commerce and Internet companies of small and average size. The algorithms that use the collaborative recommendations systems are well-understood and applicable to many domains (movies, books, songs, ...). The basic assumption of these RS is that user provides, either implicitly or explicitly, a rating of the products in the catalog. The effectiveness of these algorithms is based on the idea that "Customers who had similar tastes in the past, will have similar tastes in the future"

In this project, I will limit my analysis to the collaborative filtering algorithms. This choice is dictated by a number of considerations. First, I can have access only to data that allow the analysis of collaborative recommendation systems. If I had to analyse, for example, a content-based recommender system, I should use a database containing a list of discrete characteristics of the product. The free access to these databases is precluded to me or in any case very difficult to achieve. Second, the wide range of recommender systems has generated a very large number of algorithms. Due to the limitations of time and resources, I must necessarily limit the number of algorithms that intend to analyze. I decided to focus on collaborative algorithms because the abundant literature developed on these algorithms will allow me to develop them, properly, in Java, and to test them using the metrics that have been developed in recent years (Gunawardana et al. 2009). Last but not least, the collaborative filtering algorithms are extremely popular and represent the type of algorithms more used in most recommender systems (Su et al. 2009, Herlocker et al. 2004).

The collaborative algorithms simulate, in a sense, a chat among friends, such as when a user does not know what movie to rent and so seeks the counsel of friends with similar taste or looks on the Internet a few reviews. Starting from films seen by a user, this type of algorithm adds the user to a cluster of users with similar tastes, so it recommends a movie that to other users (called neighbors) is liked. The Collaborative algorithms are therefore based on two concepts:

* Concept of closeness between users: users with similar tastes tend to vote for a film in a similar manner.
* Concept of closeness between item: movies related to each other are voted by the users in a similar way.

Unlike the algorithms Content-Based, the collaborative ones are able to recommend to a user a different movie from every movies that is in its profile, creating a sort of surprise effect: with the Content-Based algorithms, instead, the user will almost always able to understand why a movie is recommended, noting similarities between the contents of its interest and those of the films recommended, and therefore the user will be less inclined to be amazed in front of a recommendation. Even these types of algorithms have some limitations:

* For a new user it will not be possible to make any recommendation because he has not seen any movie and as a result he will not be included in any cluster of users.
* A new film will never be recommended, as it has not received any ratings and, as a result, this movie will not be comparable with any other film in the catalog.
* The quality of the recommendation depends greatly on the amount of movies rated by a user: if a user has voted only for a very few films, it will be difficult to identify a cluster suitable for him, because his interests may correspond with those of users that are present in multiple clusters.
* It is very difficult to compute recommendations for users with particular tastes for the same reason : it is difficult to associate them with a particular user group.

Within this category of algorithms is possible to distinguish two sub-categories:

* **User-Based**: The key concept is the user, we seek to build relationships of similarity between users: if in a system there are *m* users, the model created will be of dimension *m x m*, where both the rows and the columns represent users. Each user in the User Rating Matrix is then represented by a vector in a space of *n* dimensions, with *n* number of films in the Matrix User Rating: will be considered to be similar those users whose carriers will form a small angle size. This sub category of algorithms introduces two new problems: the inclusion of a new user in the system involves the restatement of the model, and the calculation of the model is computationally very heavy, because in any system, the number of users is very large (and normally much greater than the number of items, this, for example, is the case of recommendation systems for the movies )
* **Item-Based**: are based on the concept of similarity between items, for this reason, in a system with *n* movies, the created model will have dimension *n x n*, and in each cell of this array will be present the degree of similarity between the movie present on the line and the one present on the column. With this subcategory is easy to make recommendations, as simply multiply the user profile (a vector in the space of the movies) and the model created to have a list of ratings predicted for all films, for the given user. Conversely to what is going on for the User-Based algorithms, each film will be represented by a vector in the space of the users, and similar movies will have between their similar corresponding vectors a corner few wide.

**Main Recommender Systems' Algorithms**

My analysis will focus on the following methodologies adopted widely in recent years for the creation of collaborative filtering:

* User-based nearest neighbour recommendation;
* Item-based nearest neighbour recommendation;

.

# **Dataset**

The first problem I faced was to choose how to structure the data and possibly which database to use. If you want to create a system of recommendation "real" to be used to provide suggestions to real customers the choice of the data base is a strategic decision whose effects, positive or negative, will be reflected on the future performance of the recommendation system that you want to implement . The first decision to make is to choose between a classic relational database or opt for a non-relational database. The choice between these two alternatives is not simple. The choice of relying on the relational data model or no-relational should be carefully considered because the respective advantages and disadvantages derive from a variety of factors that at the beginning of the project are not always obvious and, worst, they can change over the course of the project.

As if that were not enough, their respective advantages and disadvantages are also linked to the particular type of data you want to analyze and to the structure of these data. For example, if these data can be stored efficiently in tabular form a relational database (Oracle, MySql, PostgreSQL, etc.) is the best alternative. Normally, the ranking of a recommender systems can be expressed in a more than appropriate, in tabular form. However, if, for example, you wish to collect and analyze, in real time, data from a social-network, a no-Sql database probably would be more appropriate.

In this case, a database such as Neo4j, described by its creators as "embedded, disk-based, fully transactional Java persistence engine that stores data structured in graphs rather than in table" (wikipedia.org/wiki/Neo4j), would handle better the data that can be expressed in the form of a graph and the "relations" between the various users of the social network. More generally, modern web applications are based on data that relational databases such as MySQL or Oracle, to name the most common, cannot manage very well. Because of this the major web companies are transferring the web data that they use daily on no-relational's databases.

**Data Structure**

For this project, considering the time constraints and the amount of data to be analyzed, relatively small, I opted for a simpler approach, abandoning completely the use of a database and choosing, instead, of uploading, whenever I wanted to perform my analysis, all data required in the ram memory on the computer. The only disadvantage to this approach is to have to wait a few seconds before running the algorithms in java, to allow loading of all data from the sheet, in .txt format, on the RAM of the computer. To transfer and transform data in .txt format into data usable by my java algorithms, I created three methods:

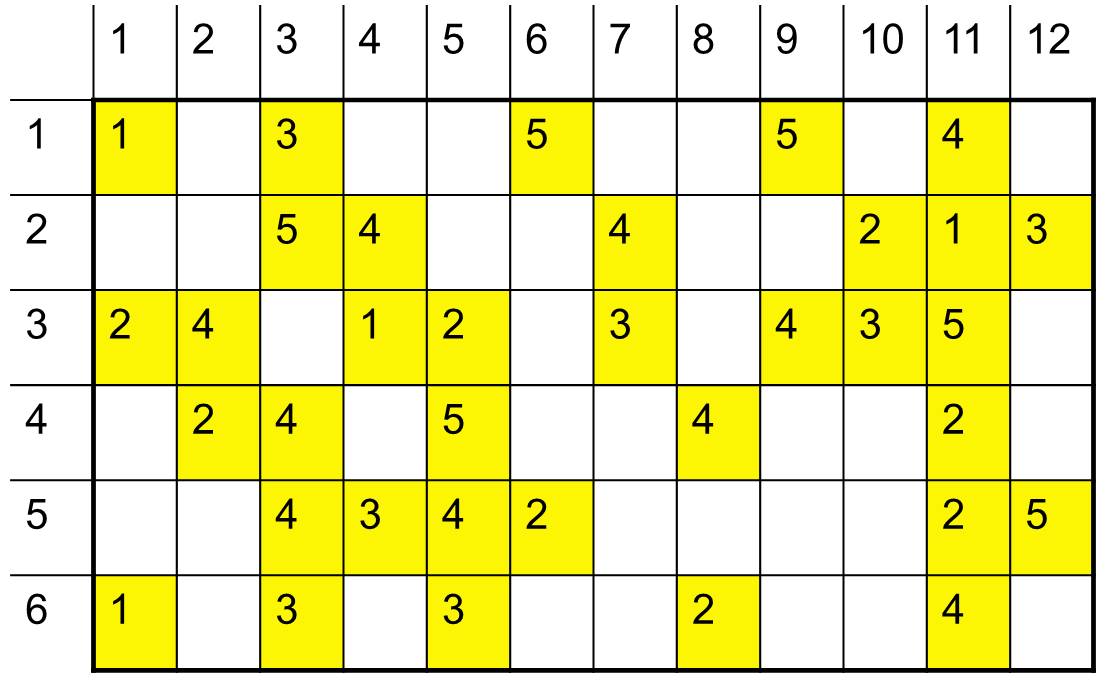
1. public static String[][] txtMatrix(String filePath, String splitExpression)
2. public static int[][] matrixConversion(String[][] inputMatrix)
3. public static double[][] matrixConversionDouble(String[][] inputMatrix)

With the first method I create a rectangular double array of type String, starting from data sheets in .txt format. The first method takes as its second argument a regular expression that provides great flexibility because it allows you to divide in rows and columns, data contained in the .txt file, regardless of whether the data are separated by a space, a comma, a semicolon, etc. The next two methods, matrixConversion and matrixConversionDouble, allow to transform the matrix of type String into an double array of type int or double, respectively. The choice of the double passage, first from txt format to String and then from String format into int or double is desired and is not redundant. Transforming data directly from .txt format into int or double would make it impossible processing no-numeric data, expressed, in the .txt file, in String format.

In some databases, for example, the ranking of products / services is not expressed by a numerical rating (a score of 1 to 5, where five is the highest score) but with a "qualitative" judgment ": bad, poor, moderate, good excellent . In these cases, using an array of type String [] [] it is possible to load in memory all the data and analyze the column containing alphanumeric data with other ad hoc methods. In practice, for this project, given the time constraints, I analyzed only the database Movilens, and it was not necessary to transform the "qualitative" data in numerical ones. It should be said that the Movilens database contains also alphanumeric files, for example, it contains the list (u.occupation) of users' occupations: administrator, artist, doctor, etc. The no-numeric data such as the profession, the city, the gender allow for further analysis and probably to increase the accuracy of forecasts.

For example you can imagine to use the list of cities and occupations and the gender to create a geospatial map of the ratings of users broken down by occupation and gender. For consumer products such a map would be of great help in guiding marketing decisions. This type of analysis, however, leads to a considerable increase in the computational complexity of the algorithms and it would need to implement in Java statistical tools rather complex. For these reasons, I am limited to use only the three types of data in the main file of Movilens: user, movie, rating.

Even using only three types of data (represented by integers) you have to face several problems.

First of all, in any recommender system, you have to handle tables or matrices which, by their nature, are sparse. The following figure (Ullman: Mining of Massive Datasets, http://www.mmds.org) clearly illustrates this structure.

If to each column we associate a user and, for each row, we associate a product, and at the intersection between the row and column we insert the corresponding rating, it is clear that most of the intersections of the row with the column will be empty

To better understand this concept, we imagine that the recommendation system created by Amazon is organized with a table of this type. If in the first column we inserted the first customer this column should be composed of a number of rows equal to those of all products sold by Amazon (every customer, in fact, can buy any product sold by Amazon). Obviously, the typical customer will buy a few dozen, or at most a few hundred products a year. But if Amazon sells millions of products, which means that the typical client will provide only a few tens / hundreds of explicit or implicit rating. The rest of the cells of the column associated with the millions of products that the customer has not bought, it will be empty, that is, it will be unrated.

Consequently, the table's customers/products/rating will have the appearance of a sparse matrix: a few cells contain a rating, the others, the majority, will be empty. A sparse matrix does not facilitate the writing of algorithms and creates two types of problems. First you need to write algorithms that can handle empty cells. If you want to create tables of type int [] [] or double [] [] I can not insert a "null" for each empty cell because the compiler will return 'an error message. For an array of type double [] [], for example, the error message is: "Type mismatch: can not convert from null to double". In theory you can overcome this problem by using an array of type Object [] [] where can be inserted integers, floating points number and null values. In this case, however, for each matrix (of not negligible size) it will need to create hundreds of thousands, or millions, of java objects with all that this entails in terms of both memory consumption and in terms of execution speed of the algorithms.

Even the strategy of using specific numbers to represent the empty cells do not effectively solves the problem: If you use, for example, negative numbers to represent the empty cells, the algorithms will not know how to manage data in tables where the negative numbers represent useful information to the recommender system (eg. temperature, negative judgments, etc.) and then the algorithms will have to discard these values. ​​The second problem, caused by a sparse matrix relates to the execution times of the algorithms, which increase significantly due to the use of a data structure not efficient. If we rethink to Amazon's example, in which the first customer makes 10 assessments that are scattered about, for example, six million products, the algorithm should search in all six millions of rows that represent the products sold, hypothetically, from Amazon.

In this project, I have not had to deal with the problem of managing sparse matrices because I used data from Movilens. These data were normalized and are structured in a tabular form:

|  |
| --- |
| utente movie rating timestamp |
| 1 6 5 887431973 |
| 1 10 3 875693118 |
| 1 12 5 878542960 |
| 1 14 5 874965706 |

The first column contains the user code (an integer from 1 to 943), the second column the code of the movie (an integer from 1 to 1682), the third column the rating of the movie (an integer from 1 to 5), the fourth column is the timestamp (Unix second since 1/1/1970 UTC). In the course of this project I did not use the timestamp data in the fourth column. In the main file I used to test the algorithms, called u.data, users are not distributed in chronological order, but were stored in an absolutely random manner. Also, each user has rated at least 20 movies. The normalization of the data in the file u.data and other test files (u1.base, u1.test, u2.base, etc ...) has helped me to avoid the problems that I have mentioned and that are typical of a sparse matrix.

**Project Structure**

As regards the structure of my project I divided it into four packages. In package "recommendationSystem" I inserted the java classes containing the algorithms that I created. To create the algorithms of this project was not necessary to use advanced concepts of object-oriented programming and it was not necessary to use interfaces. All algorithms that I wrote are static and can be used easily without resorting to a sophisticated objects programming.

**User-based nearest neighbor recommendation**

This methodology uses a database containing the ratings of the products; while, as input, uses the choices made by the current user of the system by identifying other users, called peer users or nearest neighbors, characterized by preferences similar to those that the current user has shown in the past (Jannach et al. 2011).

At the basis of this methodology there are two main assumptions:

* if the users of the system had certain tastes in the past they will have the same tastes in the future;
* the preferences of users remain stable and consistent over time.

We can express these concepts in mathematical terms.  to denote the set of users,  to denote the set of products (items) and R as a *n x m* matrix of ratings  with .

The possible rating values ​​can be defined, for example, on a numerical scale from 1 (strongly dislike) to 5 (strongly like). If a particular user *i* did not assess an item *j*, the corresponding value in the array  is empty.

A most widely used measure in practice, by the nearest neighbor recommender systems, to represent the set of similar users is the Pearson's correlation coefficient. The similarity sim (*a, b*​​) of users a and b, given a rating matrix R is defined by the following formula (where corresponds to the average rating of the user a):



The Pearson's correlation coefficient takes values ​​between +1 (strong positive correlation) to -1 (strong negative correlation) (Jannach et al. 2011).

In my project to build this similarity index I used the following algorithm:

/\*\*

\* @param inputMatrix an input matrix of type int

\* @param columnNumber the position of the column where to search the user

\* @param firstElementSearched the first user searched

\* @param secondElementSearched the second user searched

\* @return the simililarity between the first element searched and the second

\* @author Paolo Ronzoni

\*/

public static double userNearestNeighborValue(int[][] inputMatrix, int columnNumber, int firstElementSearched, int secondElementSearched) {

// create a matrix of three column: items, user1scores, user2scores

int[][] intermediateMatrix = MatrixBuilder.matchIDchoices(inputMatrix, columnNumber,firstElementSearched, secondElementSearched);

int user1columnScores = 1;

int user2columnScores = 2;

int numRows = intermediateMatrix.length;

double numerator = 0;

double denominator1 = 0;

double denominator2 = 0;

// calculates the averages of the columns' users scores

double user1Average = StatAndMathTools.matrixColumnAverage(intermediateMatrix, user1columnScores);

double user2Average = StatAndMathTools.matrixColumnAverage(intermediateMatrix, user2columnScores);

for (int row = 0; row < numRows ; row++)

{

numerator += ((intermediateMatrix[row][user1columnScores] - user1Average) \* (intermediateMatrix[row][user2columnScores] - user2Average));

denominator1 += Math.pow((intermediateMatrix[row][user1columnScores] - user1Average),2);

denominator2 += Math.pow((intermediateMatrix[row][user2columnScores] - user2Average),2);

} // end for loop

return (numerator / (Math.sqrt(denominator1 \* denominator2)));

} // end method userNearestNeighborValue

On the contrary for computing a prediction for the rating of user a and item p I have used this algorithm:

/\*\*

\* @param inputMatrix an input matrix of type int

\* @param usercolumnNumber the position of the column where to search the user

\* @param firstElementSearched the user searched

\* @param itemcolumnNumber the position of the column where to search the item

\* @param itemSearched the item searched

\* @param numOfNearestNeighbor the number of similarity to use to calculate the predictionUserBasedValue

\* @return double the prediction of the rating for the item searched for the specific user

\* @author Paolo Ronzoni

\*/

public static double predictionUserBasedValue(int[][] inputMatrix, int userColumnNumber, int userSearched, int itemcolumnNumber, int itemSearched, int numOfNearestNeighbor) {

// find all users, without duplication, in the inputMatrix

int[] usersVector = MatrixBuilder.findAllUsers(inputMatrix, userColumnNumber);

// swap the userSearched position at the beginning of the usersVector

for ( int i = 0; i < usersVector.length; i++) {

if ( usersVector[i] == userSearched) {

int tmp = usersVector[0];

usersVector[0] = userSearched;

usersVector[i] = tmp;

break;

}

} // end for s

// creates a matrix of three column: userSearched, allOtherUsers, userNearestNeighborValue

double[][] intermediateMatrix = new double[usersVector.length -1][3];

for (int row = 0; row < usersVector.length -1 ; row++) {

intermediateMatrix[row][0] = userSearched;

intermediateMatrix[row][1] = usersVector[row + 1];

intermediateMatrix[row][2] = userNearestNeighborValue( inputMatrix, userColumnNumber, userSearched, usersVector[row + 1]);

}

// orders the intermediateMatrix respect to userNearestNeighborValue

int lastMatrixColumn = 5;

double[][] finalMatrix = StatAndMathTools.sortMultidimensionArray(intermediateMatrix, 2);

double[][] lastMatrix = new double[numOfNearestNeighbor][lastMatrixColumn];

int lastRow = finalMatrix.length -1;

double[][] forMatrix;

outerloop:

for ( int i = 0; i < numOfNearestNeighbor; i++) {

forMatrix = MatrixBuilder.userIDchoicesDouble(inputMatrix, userColumnNumber,(int) finalMatrix[lastRow][1]);

while( Double.isNaN(finalMatrix[lastRow][2]) ||

!MatrixBuilder.isThereItem(forMatrix, 1, itemSearched)) {

lastRow -= 1;

if(lastRow < 0 ) break outerloop;

forMatrix = MatrixBuilder.userIDchoicesDouble(inputMatrix, userColumnNumber,(int) finalMatrix[lastRow][1]);

} // end while

// if(lastRow < 0 ) break outerloop;

lastMatrix[i][0] = finalMatrix[lastRow][0]; // the user searched

lastMatrix[i][1] = finalMatrix[lastRow][1]; // an other users

lastMatrix[i][4] = finalMatrix[lastRow][2];

lastMatrix[i][2] = MatrixBuilder.getItem(forMatrix, 1, itemSearched, 2);

lastMatrix[i][3] = StatAndMathTools.matrixColumnAverage(forMatrix,2 );

lastRow--;

} // end for

int user1ColumnScores = 2;

int numRows = lastMatrix.length;

int lastMatrixSimColumn = 4;

int lastMatrixScoresColumn = 2;

int lastMatrixAveragesColumn = 3;

double numerator = 0;

double denominator = 0;

// calculates the averages of the firt users

double user1Average = StatAndMathTools.matrixColumnAverage(MatrixBuilder.userIDchoicesDouble(inputMatrix, userColumnNumber, userSearched),user1ColumnScores);

for (int row = 0; row < numRows ; row++)

{

numerator += lastMatrix[row][lastMatrixSimColumn] \* (lastMatrix[row][lastMatrixScoresColumn] - lastMatrix[row][lastMatrixAveragesColumn]);

denominator += lastMatrix[row][lastMatrixSimColumn];

} // end for loop

return user1Average + (numerator / denominator);

// return lastMatrix;

} // end method predictionUserBasedValue

**Item-Based nearest neighbor recommendation**

The approach "User-based nearest neighbor recommendation" is used successfully in several Internet companies, but in certain scenarios when you need to handle millions of users and you need to manage the ratings from a catalog composed of millions of products, this approach has limitations. For this reason the e-commerce sites of greater size prefer to adopt another technique called "item-based nearest neighbor recommendation" (Jannach et al. 2011).

The main idea of this approach is to make predictions using the similarity between the products rather than the similarity between users. To find similar products, there must be defined a measure of similarity. The item-based approach uses, as a standard metric, the cosine similarity. This metric measures the similarity between two *n*-dimensional vectors based on the corner between them.

The similarity between two items *a* and *b* is calculated with the following formula:



In my project to build this similarity index I used the following algorithm:

/\*\*

\* @param inputMatrix an input matrix of type int

\* @param columnNumber the position of the column where to search

\* @param firstElementSearched the first user searched

\* @param secondElementSearched the second user searched

\* @return

\* @author Paolo Ronzoni

\*/

public static double cosineSimilarity(int[][] inputMatrix, int columnNumber, int firstElementSearched, int secondElementSearched) {

// create a matrix of three column: items, user1scores, user2scores

double[][] intermediateMatrix = MatrixBuilder.simpleDoubleItemRatingMatrix(inputMatrix, columnNumber,firstElementSearched, secondElementSearched);

int numRows = intermediateMatrix.length;

int item1columnScores = 1;

int item2columnScores = 2;

double numerator = 0;

double denominator1 = 0;

double denominator2 = 0;

for (int row = 0; row < numRows ; row++)

{

numerator += (intermediateMatrix[row][item1columnScores] \* intermediateMatrix[row][item2columnScores]);

denominator1 += Math.pow(intermediateMatrix[row][item1columnScores],2);

denominator2 += Math.pow(intermediateMatrix[row][item2columnScores],2);

} // end for loop

return (numerator / (Math.sqrt(denominator1 \* denominator2)));

} // end method adjustedCosineMeasure

Let U be the set of all users who value the items *a* and *b*. The adjusted cosine measure is calculated in this way:



To build this adjusted similarity index I have written the following algorithm:

/\*\*

\* @param inputMatrix an input matrix of type int

\* @param columnNumber the position of the column where to search

\* @param firstElementSearched the first element searched

\* @param secondElementSearched the second element searched

\* @return

\* @author Paolo Ronzoni

\*/

public static double adjustedCosineSimilarity(int[][] inputMatrix, int columnNumber, int firstElementSearched, int secondElementSearched) {

// create a matrix of three column: items, user1scores, user2scores

double[][] intermediateMatrix = MatrixBuilder.doubleItemRatingMatrix(inputMatrix, columnNumber,firstElementSearched, secondElementSearched);

int numRows = intermediateMatrix.length;

int item1columnScores = 1;

int item2columnScores = 2;

int averagecolumnScores = 3;

double numerator = 0;

double denominator1 = 0;

double denominator2 = 0;

for (int row = 0; row < numRows ; row++)

{

numerator += ((intermediateMatrix[row][item1columnScores] - intermediateMatrix[row][averagecolumnScores]) \* (intermediateMatrix[row][item2columnScores] - intermediateMatrix[row][averagecolumnScores]));

denominator1 += Math.pow((intermediateMatrix[row][item1columnScores] - intermediateMatrix[row][averagecolumnScores]),2);

denominator2 += Math.pow((intermediateMatrix[row][item2columnScores] - intermediateMatrix[row][averagecolumnScores]),2);

} // end for loop

return (numerator / (Math.sqrt(denominator1 \* denominator2)));

} // end method adjustedCosineMeasure

It is possible to predict the rating for the user *u* of the product *p* in this way:



This is the algorithm that I have used to implement this formula:

/\*\*

\* @param inputMatrix an input matrix of type int

\* @param usercolumnNumber the position of the column where to search the user

\* @param firstElementSearched the user searched

\* @param itemcolumnNumber the position of the column where to search the item

\* @param itemSearched the item searched

\* @param numOfNearestNeighbor the number of similarity to use to calculate the predictionUserBasedValue

\* @return double the prediction of the rating for the item searched for the specific user

\* @author Paolo Ronzoni

\*/

public static double predictionItemBasedValue(int[][] inputMatrix, int userColumnNumber, int userSearched, int itemcolumnNumber, int itemSearched, int numOfNearestNeighbor) {

// find all users, without duplication, in the inputMatrix

int[] usersVector = MatrixBuilder.findAllUsers(inputMatrix, userColumnNumber);

// swap the userSearched position at the beginning of the usersVector

for ( int i = 0; i < usersVector.length; i++) {

if ( usersVector[i] == userSearched) {

int tmp = usersVector[0];

usersVector[0] = userSearched;

usersVector[i] = tmp;

break;

}

} // end for s

// creates a matrix of three column: userSearched, allOtherUsers, userNearestNeighborValue

double[][] intermediateMatrix = new double[usersVector.length -1][3];

for (int row = 0; row < usersVector.length -1 ; row++) {

intermediateMatrix[row][0] = userSearched;

intermediateMatrix[row][1] = usersVector[row + 1];

intermediateMatrix[row][2] = cosineSimilarity(inputMatrix, userColumnNumber, userSearched, usersVector[row + 1]);

}

// orders the intermediateMatrix respect to userNearestNeighborValue

int lastMatrixColumn = 5;

double[][] finalMatrix = StatAndMathTools.sortMultidimensionArray(intermediateMatrix, 2);

double[][] lastMatrix = new double[numOfNearestNeighbor][lastMatrixColumn];

int lastRow = finalMatrix.length -1;

double[][] forMatrix;

outerloop:

for ( int i = 0; i < numOfNearestNeighbor; i++) {

forMatrix = MatrixBuilder.userIDchoicesDouble(inputMatrix, userColumnNumber,(int) finalMatrix[lastRow][1]);

while( Double.isNaN(finalMatrix[lastRow][2]) ||

!MatrixBuilder.isThereItem(forMatrix, 1, itemSearched)) {

lastRow -= 1;

if(lastRow < 0 ) break outerloop;

forMatrix = MatrixBuilder.userIDchoicesDouble(inputMatrix, userColumnNumber,(int) finalMatrix[lastRow][1]);

} // end while

// if(lastRow < 0 ) break outerloop;

lastMatrix[i][0] = finalMatrix[lastRow][0]; // the user searched

lastMatrix[i][1] = finalMatrix[lastRow][1]; // an other users

lastMatrix[i][4] = finalMatrix[lastRow][2];

lastMatrix[i][2] = MatrixBuilder.getItem(forMatrix, 1, itemSearched, 2);

lastMatrix[i][3] = StatAndMathTools.matrixColumnAverage(forMatrix,2 );

lastRow--;

} // end for

int numRows = lastMatrix.length;

int lastMatrixSimColumn = 4;

int lastMatrixScoresColumn = 2;

double numerator = 0;

double denominator = 0;

for (int row = 0; row < numRows ; row++)

{

numerator += (lastMatrix[row][lastMatrixSimColumn] \* lastMatrix[row][lastMatrixScoresColumn]);

denominator += lastMatrix[row][lastMatrixSimColumn];

} // end for loop

return (numerator / denominator);

// return lastMatrix;

} // end method predictionUserBasedValue

**Solution Description**

There are two types of metrics to evaluate a Recommender System: measures to assess the quality and measures to assess performance. The first analyzes the ratings, the second analyzes the RS as system.

In my project, I will confine my analysis to the metrics used to evaluate the quality.

Among the measures to evaluate the quality of ratings of a RS we can distinguish the accuracy (Su et al., 2009), and the coverage (Vozalis et al 2003)

***Accuracy Evaluation Metrics of Recommendation Systems***

Accuracy: There are many techniques designed to quantify the accuracy. The first category of techniques includes the mean absolute error (MAE) and root mean squared error (RMSE), the second category includes the ROC sensitivity.

Mean Absolute Error (MAE): one of the methods to determine the accuracy (or vice versa, the classification error), is the Mean Absolute Error, which calculates the average of the absolute difference between the ratings provided by the RS and the assessment that subsequently the user expresses.



where *n* is the total number of rating of all users, is the expected rating for the user *i* on the item *j*, and is the actual rating. The lower the value of the MAE, the better the prediction of the rating.

These are the algorithms that I have used to evaluate the MAE for User Based:

/\*\*

\* @param inputMatrix an input matrix of type int with columns for users, products and ratings

\* @param usercolumnNumber the position of the column where to search the user

\* @param userSearched the user searched

\* @param itemcolumnNumber the position of the column where to search the item

\* @param numOfNearestNeighbor the number of similarity to use to calculate the predictionUserBasedValue

\* @return double the weighted average of the differences between the actual scores and the predicted scores

\* @author Paolo Ronzoni

\*/

public static double meanAbsoluteErrorUserBasedOneUser(int[][] inputMatrix, int userColumnNumber, int userSearched, int itemColumnNumber, int numOfNearestNeighbor) {

// a three column matrix which corresponds to the userSearched with userID, ItemId, rating

int[][] userMatrix = MatrixBuilder.userIDchoices(inputMatrix, userColumnNumber, userSearched);

int totalNumOfRating = userMatrix.length;

int ratingColumnNumber = 2;

double numerator = 0;

for ( int row = 0; row <totalNumOfRating; row++) {

numerator += Math.abs((UserBasedNearestNeighbor.predictionUserBasedValue(inputMatrix, userColumnNumber, userSearched, itemColumnNumber, userMatrix[row][itemColumnNumber], numOfNearestNeighbor) - userMatrix[row][ratingColumnNumber]) );

}// end for meanAbsoluteErrorUserBasedOneUser

return numerator / totalNumOfRating;

} // end method meanAbsoluteErrorUserBasedOneUser

And this is the Item-based MAE:

/\*\*

\* @param inputMatrix an input matrix of type int with columns for users, products and ratings

\* @param usercolumnNumber the position of the column where to search the user

\* @param userSearched the user searched

\* @param itemcolumnNumber the position of the column where to search the item

\* @param numOfNearestNeighbor the number of similarity to use to calculate the predictionUserBasedValue

\* @return double the weighted average of the differences between the actual scores and the predicted scores

\* @author Paolo Ronzoni

\*/

public static double meanAbsoluteErrorItemBasedOneUser(int[][] inputMatrix, int userColumnNumber, int userSearched, int itemColumnNumber, int numOfNearestNeighbor) {

// a three column matrix which corresponds to the userSearched with userID, ItemId, rating

int[][] userMatrix = MatrixBuilder.userIDchoices(inputMatrix, userColumnNumber, userSearched);

int totalNumOfRating = userMatrix.length;

int ratingColumnNumber = 2;

double numerator = 0;

// original code: for ( int row = 0; row <totalNumOfRating; row++) {

for ( int row = 0; row < 5; row++) {

numerator += Math.abs((ItemBasedNearestNeighbor.predictionItemBasedValue(inputMatrix, userColumnNumber, userSearched, itemColumnNumber, userMatrix[row][itemColumnNumber], numOfNearestNeighbor) - userMatrix[row][ratingColumnNumber]) );

}// end for meanAbsoluteErrorUserBasedOneUser

return numerator / totalNumOfRating;

} // end method meanAbsoluteErrorItemBasedOneUser

Sometimes it is useful to normalize this value relative to the scale used in the RS specific, so you can then compare systems with minimum rating () and maximum () different. In this case one calculates the normalized MAE (NMAE):



This are the NMAE algorithms. User-based:

/\*\*

\* @param inputMatrix an input matrix of type int with columns for users, products and ratings

\* @param usercolumnNumber the position of the column where to search the user

\* @param userSearched the user searched

\* @param itemcolumnNumber the position of the column where to search the item

\* @param numOfNearestNeighbor the number of similarity to use to calculate the predictionUserBasedValue

\* @return double the weighted average of the differences between the actual scores and the predicted scores normalized dividing by (maxRating - minRating)

\* @author Paolo Ronzoni

\*/

public static double normalizedMeanAbsoluteErrorUserBasedOneUser(int[][] inputMatrix, int userColumnNumber, int userSearched, int itemColumnNumber, int numOfNearestNeighbor) {

// a three column matrix which corresponds to the userSearched with userID, ItemId, rating

int[][] userMatrix = MatrixBuilder.userIDchoices(inputMatrix, userColumnNumber, userSearched);

// the column of the ratings

int totalNumOfRating = userMatrix.length;

int ratingColumnNumber = 2;

double numerator = 0;

for ( int row = 0; row <totalNumOfRating; row++)

{

numerator += Math.abs(UserBasedNearestNeighbor.predictionUserBasedValue(inputMatrix, userColumnNumber, userSearched, itemColumnNumber, userMatrix[row][itemColumnNumber], numOfNearestNeighbor) - userMatrix[row][ratingColumnNumber] );

}// end for meanAbsoluteErrorUserBasedOneUser

// the column of rating the rating

double[] ratingColumn = MatrixBuilder.columnExtractor(userMatrix, ratingColumnNumber);

double rMax = StatAndMathTools.maxValue(ratingColumn);

double rMin = StatAndMathTools.minValue(ratingColumn);

return (numerator / totalNumOfRating)/(rMax - rMin);

} // end method normalizedMeanAbsoluteErrorUserBasedOneUser

And NMAE item-based:

/\*\*

\* @param inputMatrix an input matrix of type int with columns for users, products and ratings

\* @param usercolumnNumber the position of the column where to search the user

\* @param userSearched the user searched

\* @param itemcolumnNumber the position of the column where to search the item

\* @param numOfNearestNeighbor the number of similarity to use to calculate the predictionUserBasedValue

\* @return double the weighted average of the differences between the actual scores and the predicted scores normalized dividing by (maxRating - minRating)

\* @author Paolo Ronzoni

\*/

public static double normalizedMeanAbsoluteErrorItemBasedOneUser(int[][] inputMatrix, int userColumnNumber, int userSearched, int itemColumnNumber, int numOfNearestNeighbor) {

// a three column matrix which corresponds to the userSearched with userID, ItemId, rating

int[][] userMatrix = MatrixBuilder.userIDchoices(inputMatrix, userColumnNumber, userSearched);

// the column of the ratings

int totalNumOfRating = userMatrix.length;

int ratingColumnNumber = 2;

double numerator = 0;

// original code: for ( int row = 0; row <totalNumOfRating; row++)

for ( int row = 0; row < 1; row++)

{

numerator += Math.abs(ItemBasedNearestNeighbor.predictionItemBasedValue(inputMatrix, userColumnNumber, userSearched, itemColumnNumber, userMatrix[row][itemColumnNumber], numOfNearestNeighbor) - userMatrix[row][ratingColumnNumber] );

}// end for meanAbsoluteErrorUserBasedOneUser

// the column of rating the rating

double[] ratingColumn = MatrixBuilder.columnExtractor(userMatrix, ratingColumnNumber);

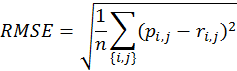
double rMax = StatAndMathTools.maxValue(ratingColumn);

double rMin = StatAndMathTools.minValue(ratingColumn);

return (numerator / totalNumOfRating)/(rMax - rMin);

} // end method normalizedMeanAbsoluteErrorItemBasedOneUser

Root Mean Squared Error (RMSE):



where *n* is the total number of rating of all user,  is the expected rating for the user *i* on the item *j* and  is the actual rating.

This is the RMSE algorithms User-based:

/\*\*

\* @param inputMatrix an input matrix of type int with columns for users, products and ratings

\* @param usercolumnNumber the position of the column where to search the user

\* @param userSearched the user searched

\* @param itemcolumnNumber the position of the column where to search the item

\* @param numOfNearestNeighbor the number of similarity to use to calculate the predictionUserBasedValue

\* @return double the root weighted average of the squares between the difference among actual scores and the predicted scores

\* @author Paolo Ronzoni

\*/

public static double rootMeanSquaredErrorUserBasedOneUser(int[][] inputMatrix, int userColumnNumber, int userSearched, int itemColumnNumber, int numOfNearestNeighbor) {

// a three column matrix which corresponds to the userSearched with userID, ItemId, rating

int[][] userMatrix = MatrixBuilder.userIDchoices(inputMatrix, userColumnNumber, userSearched);

int totalNumOfRating = userMatrix.length;

int ratingColumnNumber = 2;

double numerator = 0;

for ( int row = 0; row <totalNumOfRating; row++) {

numerator += Math.pow((UserBasedNearestNeighbor.predictionUserBasedValue(inputMatrix, userColumnNumber, userSearched, itemColumnNumber, userMatrix[row][itemColumnNumber], numOfNearestNeighbor) - userMatrix[row][ratingColumnNumber]), 2);

}// end for rootMeanSquaredErrorUserBasedOneUser

return Math.sqrt(numerator / totalNumOfRating);

} // end method rootMeanSquaredErrorUserBasedOneUser

This is the RMSE algorithms item-based:

/\*\*

\* @param inputMatrix an input matrix of type int with columns for users, products and ratings

\* @param usercolumnNumber the position of the column where to search the user

\* @param userSearched the user searched

\* @param itemcolumnNumber the position of the column where to search the item

\* @param numOfNearestNeighbor the number of similarity to use to calculate the predictionUserBasedValue

\* @return double the root weighted average of the squares between the difference among actual scores and the predicted scores

\* @author Paolo Ronzoni

\*/

public static double rootMeanSquaredErrorItemBasedOneUser(int[][] inputMatrix, int userColumnNumber, int userSearched, int itemColumnNumber, int numOfNearestNeighbor) {

// a three column matrix which corresponds to the userSearched with userID, ItemId, rating

int[][] userMatrix = MatrixBuilder.userIDchoices(inputMatrix, userColumnNumber, userSearched);

int totalNumOfRating = userMatrix.length;

int ratingColumnNumber = 2;

double numerator = 0;

// original code: for ( int row = 0; row <totalNumOfRating; row++) {

for ( int row = 0; row < 1; row++) {

numerator += Math.pow((ItemBasedNearestNeighbor.predictionItemBasedValue(inputMatrix, userColumnNumber, userSearched, itemColumnNumber, userMatrix[row][itemColumnNumber], numOfNearestNeighbor) - userMatrix[row][ratingColumnNumber]), 2);

}// end for rootMeanSquaredErrorUserBasedOneUser

return Math.sqrt(numerator / totalNumOfRating);

} // end method rootMeanSquaredErrorItemBasedOneUser

**Conclusions**

In this project will test different collaborative filtering algorithms, compare the results obtained and analyze how these algorithms can be useful to improve the predictive accuracy of a collaborative recommendation system. I will limit my analysis to the collaborative filtering algorithms. I decided to focus on collaborative algorithms because the abundant literature developed on these algorithms will allow me to develop them, properly, in Java, and to test them using the metrics that have been developed in recent years. In this project, I will confine my analysis to the metrics used to evaluate the quality of e Recommender Systems.

**References**

The following books and papers were reviewed during the compilation of this project proposal:

John S. Breese, David Heckerman, Carl Kadie, “Empirical Analysis of Predictive Algorithms for Collaborative Filtering,” in *UAI’98 Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence.*

http://research.microsoft.com/apps/pubs/default.aspx?id=69656

Robin Burke, “Knowledge-based recommender systems,” www.cs.odu.edu/~mukka/cs795sum10dm/Lecturenotes/Day6/burke-elis00.pdf

Derek Hamilton-Gibbs, Don Esslemont, and Dalton McGuinness, “Predicting the Demand for Frequently Purchased Items,”in *Marketing Bulletin*, 1992, 3, 18-23, Article 3. http://marketing-bulletin.massey.ac.nz/V3/MB\_V3\_A3\_Esslemont.pdf.

Asela Gunawardana and Guy Shani, “A Survey of Accuracy Evaluation Metrics of Recommendation Tasks,” in *Journal of Machine Learning Research* 10, 2009, 2935-2962.

http://dl.acm.org/citation.cfm?id=1755883

Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl, “Evaluating Collaborative Filtering Recommender Systems,” in *ACM Transactions on Information Systems*, Vol. 22, No. 1, January 2004, Pages 5–53.

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.78.8384

Jonathan L. Herlocker, Joseph A. Konstan, and John T. Riedl, “Explaining Collaborative Filtering Recommendations,” in *CSCW '00 Proceedings of the 2000 ACM conference on Computer supported cooperative work*, Pages 241-250.

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.252

Dietmar Jannach , Markus Zanker, Alexander Felfering, Gerhard Friedrich, “Recommender Systems. An Introduction,” *Cambridge University Press,* 1st edition 2011, ISBN. 978-0521493369.

Daniel Lemire, Anna Maclachlan, “Slope One Predictors for Online Rating-Based Collaborative Filtering,” in *SIAM Data Mining (SDM'05)*, Newport Beach, California, April 21-23, 2005

http://arxiv.org/abs/cs/0702144.

Prem Melville, Raymond J. Mooney and Ramadass Nagarajan, “Content-Boosted Collaborative Filtering,” in *Proceedings of the SIGIR-2001 Workshop on Recommender Systems*, New Orleans, LA, September 2001.

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.6429

Prem Melville and Vikas Sindhwani, ”Recommender Systems,” *in Encyclopedia of Machine Learning*, Claude Sammut and Geoffrey Webb (Eds), Springer, 2010. http://www.prem-melville.com/.

Xiaoyuan Su and Taghi M. Khoshgoftaar, “Survey of Collaborative Filtering Techniques,” in *In Advances in Artificial Intelligence archive* Volume 2009; Hindawi

Publishing Corp. New York, NY, United States.

http://portal.acm.org/citation.cfm?id=1722966

Emmanouil Vozalis, Konstantinos G. Margaritis, “Analysis of Recommender Systems’ Algorithms” presented at the *Sixth Hellenic-European Conference on Computer Mathematics and its Applications* (HERCMA), Athens, Greece, 2003, pp. 732-745.

http://www.mendeley.com/research/analysis-of-recommender-systems-algorithms-1/