Report for the practical assignment

Part 1: USER-ITEM

INFDTA01-1 – Data Science (Recommender systems) – 2015/16

**Important**: you can work in groups of *two* students, but the final report is *individual*. The *code* can be the same for the two students working in pair, but *explanations/answers* must be different from each other. The code must be readable (by placing comments and naming well the variables).

**Delivery:** Fill this document with your answers (code, explanations, etc…) and upload a zipped folder (containing *this document* and *all your code*) on N@tschool *before* the deadline specified in the modulewijzer.

**Evaluation:** Each of the numbered sections below gives up to 1 point. The total number of points for this part of the assignment is, thus, 12. The final grade of the whole practical assignment is obtained combining the points of Part 1 and Part 2 (see formula in the modulewijzer).

Student name: Geddy Schellevis

Student number: 0883374

Class (3A/B/C): INF 3C

Language (Java, C#, …): JAVA

[ If code was developed in pairs, name of the other student: Gokhan Kacan 0858756 ]

# Importing data

public class UserPreference {  
 HashMap<Integer,Double> ratings = new HashMap<Integer,Double>();  
  
 public UserPreference(int id, double rating){  
 ratings.put(id,rating);  
 }  
  
 public double getRating(int id) {  
 return ratings.get(id);  
 }  
  
 public void setRating(int id, double rating){  
 ratings.put(id,rating);  
 }  
  
 public HashMap<Integer,Double> getRatings() {  
 return ratings;  
 }  
}

1. Which data structure did you use to store the ratings of items inside the UserPreference class? Why did you choose that in particular?

For store the items inside the userPreference I create a hashmap this is because a hashmap works with key value pairs so later on in the algorithms if I know the key ,I can easily get the right rating without looping through all the ratings of a user. So for much better performance. Read O(1) and write O(n) . I choose for a possible slower writing because we have to read the data much more times than we write the data.

*1 => {101=2.5, 102=3.5, 103=3.0, 104=3.5, 105=2.5, 106=3.0}*

*2 => {101=3.0, 102=3.5, 103=1.5, 104=5.0, 105=3.5, 106=3.0}*

*3 => {101=2.5, 102=3.0, 104=3.5, 106=4.0}*

*4 => {102=3.5, 103=3.0, 104=4.0, 105=2.5, 106=4.5}*

*5 => {101=3.0, 102=4.0, 103=2.0, 104=3.0, 105=2.0, 106=3.0}*

*6 => {101=3.0, 102=4.0, 104=5.0, 105=3.5, 106=3.0}*

*7 => {102=4.5, 104=4.0, 105=1.0}*

# Nearest neighbours

1. public void nearestNeighbour() {  
    UserPreference target = userRatings.get(targetUserKey);  
    Double lowestInList = 2.0;  
     
    Context context = CreateContext();  
     
    int count = 0;  
    for (Map.Entry<String, UserPreference> entry : userRatings.entrySet()) {  
    String key = entry.getKey();  
    UserPreference user = entry.getValue();  
     
    // if user in the list isn't the target user  
    if (!key.equals(targetUserKey)){  
    double distance = context.calculateDistance(target,entry.getValue());  
     
    // Check if user distance is greater then threshold and user have another item rated  
    if(distance > threshold && haveMoreRatings(target,user)){  
    Neighbour neighbour = new Neighbour(user,distance);  
    if(distance < lowestInList){  
    lowestInList = distance;  
    }  
     
    // Check if neighbours list isn't bigger then we want to.  
    // if it is bigger remove the lowest neighbour and change  
    // and change the threshold to lowest item.  
     
    if(count >= amountOfNeighbours){  
    if(distance >lowestInList){  
    // we add the neighbour before we check the lowest  
    neighbours.add(neighbour);  
     
    Neighbour lowestNeighbour = null;  
     
    // check for lowest neighbour  
    for (Neighbour n : neighbours){  
    if(lowestNeighbour == null){  
    lowestNeighbour = n;  
    }else if (lowestNeighbour.getDistance() > n.getDistance() ){  
    lowestNeighbour = n;  
    }  
    }  
    // remove the lowest  
    neighbours.remove(lowestNeighbour);  
    }  
    // change the threshold  
    threshold = lowestInList;  
    }else{  
    neighbours.add(neighbour);  
    count++;  
    }  
    }  
    }  
    }  
    neighbours.sort(new neighbourComparator());  
   }

# Relationship between algorithm and similarity measures

Is the algorithm generic with respect to the similarity/distance measure used (Pearson, Cosine, etc.)? If yes, explain how you achieved that.

I fixed this problem by using the sigmoid function. So I get a number between 0 and 1 back by the Euclidean. The other methods both return number between -1 and 1. And 0 and 1.

# Similarity measures

1. public class Cosine implements Distance {  
    public float calculate(UserPreference targetUser ,UserPreference user){  
    float xDotY = 0;  
    float x = 0;  
    float y = 0;  
     
    HashMap<Integer,Double> items = new HashMap<Integer,Double>();  
     
    // put all articles in a list  
    for (Map.Entry<Integer,Double> entry : targetUser.getRatings().entrySet()) {  
    if(items.size() == 0 || !items.containsKey(entry.getKey())) {  
    items.put(entry.getKey(),entry.getValue());  
    }  
    }  
     
    for (Map.Entry<Integer,Double> entry : user.getRatings().entrySet()) {  
    if(items.size() == 0 || !items.containsKey(entry.getKey())) {  
    items.put(entry.getKey(),entry.getValue());  
    }  
    }  
     
    // the Cosine formule  
    for (Map.Entry<Integer,Double> entry : items.entrySet()) {  
    float tempX = 0;  
    float tempY = 0;  
    if(targetUser.getRatings().containsKey(entry.getKey())){  
    tempX = (float) targetUser.getRating(entry.getKey());  
    }  
    if(user.getRatings().containsKey(entry.getKey())){  
    tempY = (float) user.getRating(entry.getKey());  
    }  
    xDotY += tempX \* tempY;  
    x += Math.*pow*(tempX, 2);  
    y += Math.*pow*(tempY, 2);  
    }  
     
    float answer = xDotY / (float)(Math.*sqrt*(x) \* Math.*sqrt*(y));  
    return answer;  
    }  
   }

public class Euclidean implements Distance {  
 public float calculate(UserPreference targetUser ,UserPreference user){  
  
 Double sum = 0.0;  
 for (Map.Entry<Integer, Double> entry : targetUser.getRatings().entrySet()) {  
 Integer key = entry.getKey();  
 Double value = entry.getValue();  
 double userRating ;  
  
 // if the user we compare with rate the item calculate the sum to the power of 2.  
 if(user.getRatings().containsKey(key)) {  
 userRating = user.getRating(key);  
 sum += Math.*pow*((value - userRating), 2);  
 }  
 }  
  
 // sigmoid method returns a value between 0 and 1;  
 return 1/ (1 +((float)Math.*sqrt*(sum)));  
 }  
}

public class Pearson implements Distance {  
 public float calculate(UserPreference targetUser , UserPreference user){  
 int count = 0;  
 double sumX = 0;  
 double sumY = 0;  
 double sumXY = 0;  
 double sumXPower = 0;  
 double sumYPower = 0;  
  
 // Pearson Formule  
 for (Map.Entry<Integer, Double> entry : targetUser.getRatings().entrySet()) {  
 int key = entry.getKey();  
 double value = entry.getValue();  
 if (user.getRatings().containsKey(key)) {  
 count++;  
 double x = (double) value;  
 double y = (double) user.getRating(key);  
 sumX += x;  
 sumY += y;  
 sumXPower += Math.*pow*(x, 2);  
 sumYPower += Math.*pow*(y, 2);  
 sumXY += (x \* y);  
 }  
 }  
  
 double sumPartOne = sumXY - ((sumX \* sumY)/ count);  
 double domination = (Math.*sqrt*(sumXPower - (Math.*pow*(sumX,2)/ count))) \* (Math.*sqrt*(sumYPower - (Math.*pow*(sumY , 2) / count)));  
  
 if (domination == 0){  
 return 0;  
 }  
  
 return (float) (sumPartOne / domination);  
 }  
}

# Results of nearest neighbours

Consider the small dataset imported before (*userItem.data*). If we focus on user **7** (target user) and we compute its **3** nearest neighbours (with an initial similarity threshold of ), which are the results?

***PEARSON***

Nearest neighbour 1: User 1 with similarity 0,99

Nearest neighbour 2: User 4 with similarity 0,89

Nearest neighbour 3: User 2 with similarity 0,38

***COSINE***

Nearest neighbour 1: User 6 with similarity 0,81

Nearest neighbour 2: User 2 with similarity 0,77

Nearest neighbour 3: User 5 with similarity 0,73

***EUCLIDEAN***

Nearest neighbour 1: User 5 with similarity 0,4

Nearest neighbour 2: User 3 with similarity 0,39

Nearest neighbour 3: User 4 with similarity 0,36

# Comment on Pearson result

What is the **Pearson** coefficient of similarity between users **3** and **4**? *Why* does it have that value?

0,5 For all the ratings they give to the same articles are for user 4 0.5 points higher. But the lowest rating the give is by both 2.5 that’s why they are not completely similar.

# Predicting ratings

1. public List<Recommendation> recommend(){  
    ArrayList<Recommendation> recommendations = new ArrayList<Recommendation>();  
     
    // get articles from the neighbours that the target user didn't rate  
    ArrayList<Integer> articles = new ArrayList<Integer>();  
    for (Neighbour n : neighbours){  
    for(Map.Entry<Integer, Double> entry : n.getUser().getRatings().entrySet()) {  
    if (!targetUser.getRatings().containsKey(entry.getKey()) && !articles.contains(entry.getKey())) {  
    articles.add(entry.getKey());  
    }  
    }  
    }  
     
    // Loop the list of articles for calculate the rate for the target user.  
    for (Integer article : articles ){  
    HashMap<Double,Double> tempRecommendation = new HashMap<Double, Double>();  
     
    // get the ratings of the neighbours and put them in the hash map.  
    // The key is the rating and value is the distance  
    for (Neighbour n : neighbours){  
    if(n.getUser().getRatings().containsKey(article)){  
    tempRecommendation.put(n.getUser().getRating(article),n.getDistance());  
    }  
    }  
     
    Double ratingSum = 0.0;  
    Double weightSum = 0.0;  
     
    // calculate the rating for a article. and add the article if there a more then one rating  
    int amountOrates = 0;  
    for(Map.Entry<Double, Double> entry : tempRecommendation.entrySet()){  
    ratingSum += entry.getKey() \* entry.getValue();  
    weightSum += entry.getValue();  
    amountOrates++;  
     
    }  
     
    // if there a 3 or more rates add to recommendation  
    if (amountOrates >= 3) {  
    Recommendation rec = new Recommendation(article, (ratingSum / weightSum));  
    recommendations.add(rec);  
    }  
    }  
     
    // sort the list  
    recommendations.sort(new recommendationComparator());  
    return recommendations.subList(0,numberOfRecommendations);  
   }

# Results of predicting ratings

Given the 3 nearest neighbours of user **7** computed with the **Pearson** similarity measure, predict the ratings that user 7 would give to items **101**, **103** and **106**:

Predicted rating for item 101: 2.67

Predicted rating for item 103: 2.45

Predicted rating for item 106: 3.95

Given the 3 nearest neighbours of user **4** computed with the **Pearson** similarity measure, predict the rating that user 4 would give to item **101**:

Predicted rating for item 101: 2.73

# Adding and updating ratings

Suppose that user **7** rates the item **106** with **2.8**. Add this rating into your dataset and update the nearest neighbours of user 7 (using **Pearson**). Compute again the predicted ratings of the other items (**101** and **103**):

Predicted rating for item 101: 2.64

Predicted rating for item 103: 2.40

Suppose that user **7** changes idea and rates the item **106** with **5**. Update the rating into your dataset and the nearest neighbours of user 7 (using **Pearson**). Compute again the predicted ratings of other items (**101** and **103**):

Predicted rating for item 101: 2.5

Predicted rating for item 103: 3.0

# Importing another dataset

I created 3 hashmaps for efficentie. 1 for users, 1 for movies and last one for the genre’s  
the users hashmap contains key (user id) and the value a map with movie id and rating value.  
this is enough for get the prediction done. After the prodection function is done I can get the value of a movie by id.

The movie contains a map with id and item object. In the item object is a id,name,imdb link and list of genre’s

*USERS:*

*90 => {512=4.0, 514=3.0, 515=5.0, 516=5.0, 517=3.0, 6=4.0, 518=2.0, 519=5.0, 0 ect …}  
91 => {515=5.0, 264=4.0, 520=4.0, 265=5.0, 526=4.0, 527=4.0, 529=4.0, 22=5.0, 1050=3.0, ect …}  
92 => {1=4.0, 2=3.0, 1028=2.0, 4=4.0, 5=4.0, 7=4.0, 8=5.0, 1033=2.0, 9=4.0, ect …}  
93 => {1=5.0, 866=2.0, 934=3.0, 235=4.0, 845=4.0, 14=4.0, 15=5.0, 815=4.0, 275=4.0, 0 ect …}  
94 => {1=4.0, 1028=2.0, 4=4.0, 7=4.0, 1032=2.0, 8=5.0, 9=5.0, 11=5.0, 12=4.0, 0 ect …}  
95 => {1=5.0, 514=2.0, 2=2.0, 3=1.0, 515=5.0, 518=4.0, 7=5.0, 520=4.0, 8=5.0 ect …}  
96 => {1=5.0, 1154=5.0, 514=4.0, 645=5.0, 519=4.0, 7=5.0, 8=5.0, 265=5.0, 525=2.0, ect …}  
97 => {1=4.0, 132=5.0, 133=1.0, 7=5.0, 135=5.0, 526=3.0, 655=5.0, 661=5.0, 919=5.0, 0 ect …}  
98 => {321=3.0, 514=5.0, 322=3.0, 194=5.0, 517=5.0, 70=3.0, 523=5.0, 655=3.0, ect …}  
99 => {1=4.0, 258=5.0, 3=3.0, 4=5.0, 7=4.0, 265=3.0, 11=5.0, 268=3.0, 12=5.0, 780=5.00 ect …}*

*Genre:*

*0 => unknown  
1 => Action  
2 => Adventure  
3 => Animation  
4 => Children's*

*Movie*

*Note that after the => all properties are in the Item object. I only print name and genre but the object contains the name. imdb links and genre*

*1 => Toy Story (1995) / [Animation, Children's, Comedy]  
2 => GoldenEye (1995) / [Action, Adventure, Thriller]  
3 => Four Rooms (1995) / [Thriller]  
4 => Get Shorty (1995) / [Action, Comedy, Drama]  
5 => Copycat (1995) / [Crime, Drama, Thriller]  
6 => Shanghai Triad (Yao a yao yao dao waipo qiao) (1995) / [Drama]*

# ) Computing the top recommendations

1. public List<Recommendation> recommend(){  
    ArrayList<Recommendation> recommendations = new ArrayList<Recommendation>();  
     
    // get articles from the neighbours that the target user didn't rate  
    ArrayList<Integer> articles = new ArrayList<Integer>();  
    for (Neighbour n : neighbours){  
    for(Map.Entry<Integer, Double> entry : n.getUser().getRatings().entrySet()) {  
    if (!targetUser.getRatings().containsKey(entry.getKey()) && !articles.contains(entry.getKey())) {  
    articles.add(entry.getKey());  
    }  
    }  
    }  
     
    // Loop the list of articles for calculate the rate for the target user.  
    for (Integer article : articles ){  
    HashMap<Double,Double> tempRecommendation = new HashMap<Double, Double>();  
     
    // get the ratings of the neighbours and put them in the hash map.  
    // The key is the rating and value is the distance  
    for (Neighbour n : neighbours){  
    if(n.getUser().getRatings().containsKey(article)){  
    tempRecommendation.put(n.getUser().getRating(article),n.getDistance());  
    }  
    }  
     
    Double ratingSum = 0.0;  
    Double weightSum = 0.0;  
     
    // calculate the rating for a article. and add the article if there a more then one rating  
    int amountOrates = 0;  
    for(Map.Entry<Double, Double> entry : tempRecommendation.entrySet()){  
    ratingSum += entry.getKey() \* entry.getValue();  
    weightSum += entry.getValue();  
    amountOrates++;  
     
    }  
     
    // if there a 3 or more rates add to recommendation  
    if (amountOrates >= 3) {  
    Recommendation rec = new Recommendation(article, (ratingSum / weightSum));  
    recommendations.add(rec);  
    }  
    }  
     
    // sort the list  
    recommendations.sort(new recommendationComparator());  
    return recommendations.subList(0,numberOfRecommendations);  
   }

# ) Results and analysis of top recommendations

Consider user **186** and use your algorithm (with the following parameters: **25** nearest neighbours, **0.35** threshold, **Pearson** similarity) to compute the **8** top recommendations for him:

Recommendation 1: Hoop Dreams (1994) with predicted rating: 5

Recommendation 2: Antonia's Line (1995) with predicted rating: 5

Recommendation 3: It Happened One Night (1934) with predicted rating: 5

Recommendation 4: Swept from the Sea (1997) with predicted rating: 5

Recommendation 5: Cold Comfort Farm (1995) with predicted rating: 5

Recommendation 6: Angels and Insects (1995) with predicted rating: 5

Recommendation 7: Pillow Book, The (1995) with predicted rating: 5

Recommendation 8: Freeway (1996) with predicted rating: 5

Based on the results above, do you think it could be better to compute the predicted rating only for movies which were rated by *more than one* nearest neighbour (i.e., at least two or three)? Why?

I think it would be the best to use at least 3 ratings because the height rating is now based on 1 user who rate a item. This is not really reliable for a good rating. No other user has now a influence on the ratings expected for the target user.

# 12) New results of top recommendations

If yes, modify your algorithm to compute the predicted ratings considering only products rated by *at least* **3** neighbours. Execute again the program and put here the updated results:

Recommendation 1: Cool Hand Luke with predicted rating: 4.05

Recommendation 2: People vs. Larry Flynt, The (1996) with predicted rating 4.04

Recommendation 3: Butch Cassidy and the Sundance Kid (1969) with predicted rating: 4.03

Recommendation 4: Wizard of Oz, The (1939) with predicted rating: 4.02

Recommendation 5: Saint, The (1997) with predicted rating: 4.02

Recommendation 6: Sting, The (1973) with predicted rating: 4.02

Recommendation 7: Manchurian Candidate, The (1962) with predicted rating: 4.02

Recommendation 8: To Kill a Mockingbird (1962) with predicted rating: 4.02