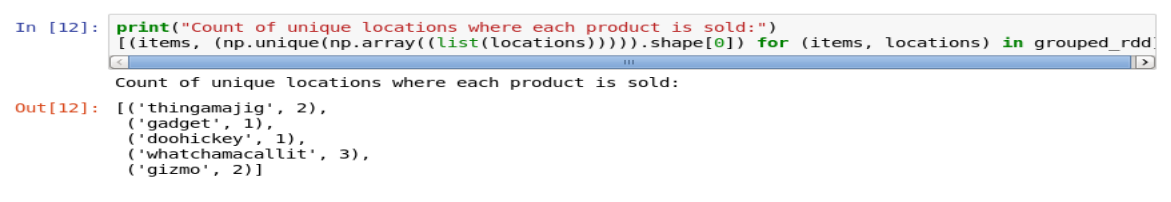
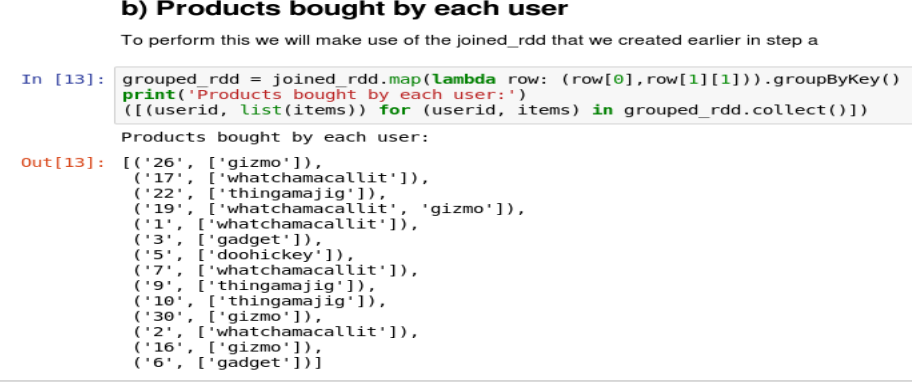
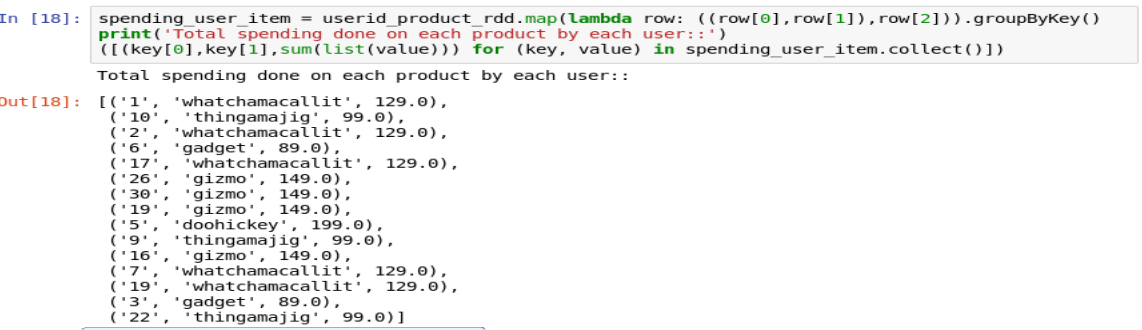
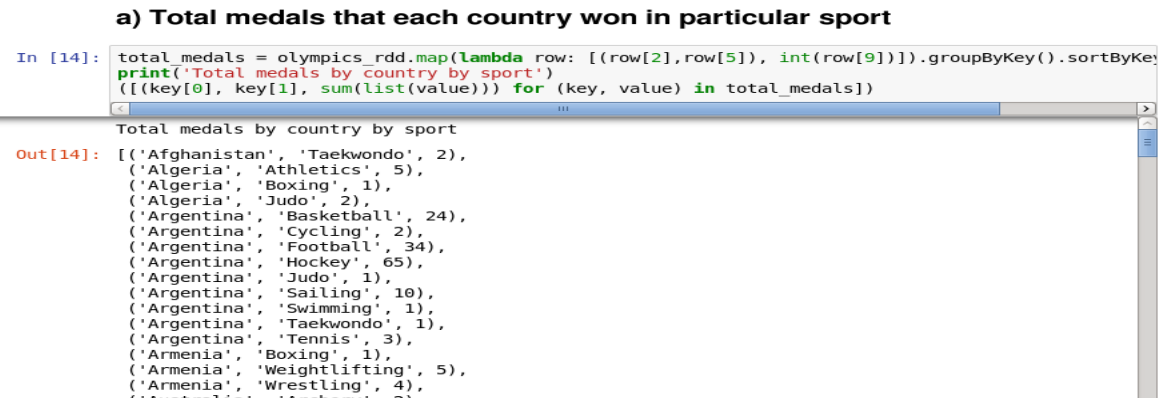
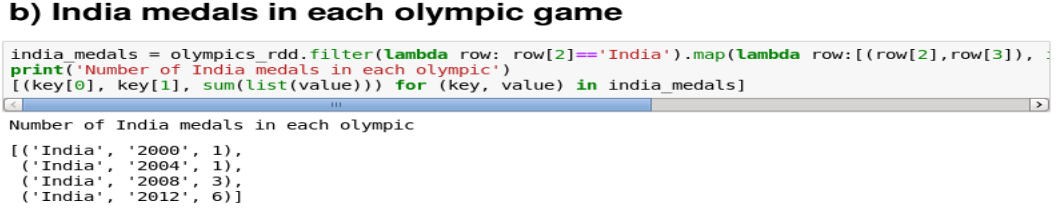
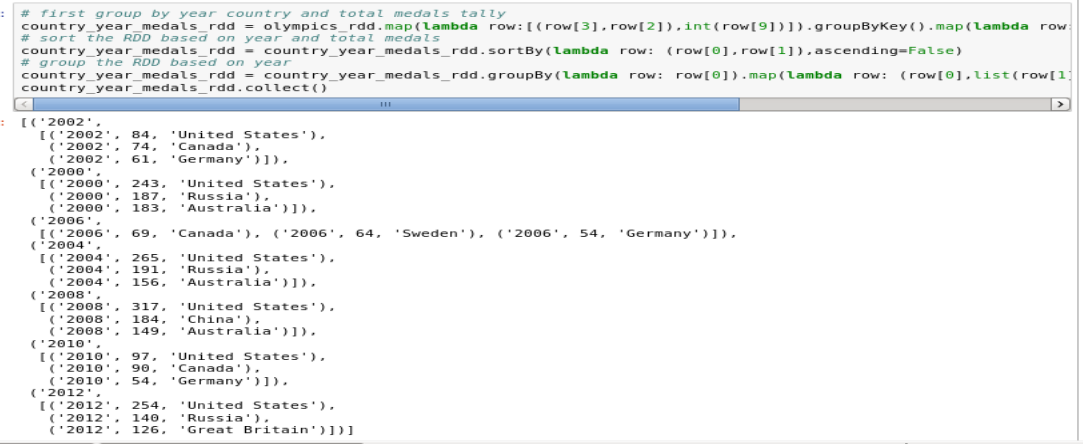
# Name: Sandipto Sanyal

# PGID: 12010004

# Question 1

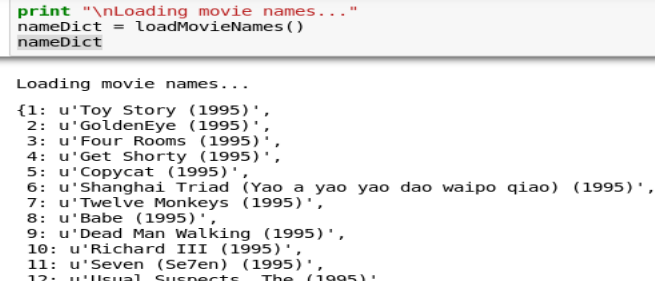
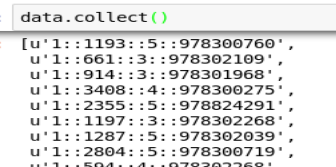
1. Count of unique locations where each product is sold: 
2. Products bought by each user: 
3. Total spending done by each user on each product: 

# Question 2

1. Total medals that each country won in a particular sport (such as Gymnastics).: 
2. In each Olympic games, how many medals has India won: 
3. Compute top 3 countries in terms of total medals by each Olympic games year. 

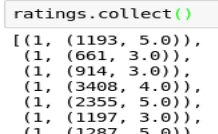
# Question 3

Below we discuss each lines of the code and mention which part of our plan the corresponding code(s) refer to.

1. **conf = SparkConf().setMaster("local[\*]").setAppName("MovieSimilarities"):** Spark configuration to run spark codes
2. **sc = SparkContext(conf = conf):** create an object of SparkContext with configuration in step 1 passed as parameter
3. **nameDict = loadMovieNames():** Calls loadMovieNames() which does the following:
   1. **movieNames = {}:** Declare an empty dictionary movieNames.
   2. **with open("/home/cloudera/moviedata/itemfile.txt") as f:** declare file handler f to open itemfile.txt
   3. **for line in f:** Read each line for itemfile.txt
   4. **fields = line.split('|'):** Split each line on basis of ‘|’ character. This creates a list from string of characters split at each ‘|’.
   5. **movieNames[int(fields[0])] = fields[1].decode('ascii', 'ignore'):** Append to the movieNames dictionary where key is int(fields[0]) and value is fields[1].decode('ascii', 'ignore'). We will see the content of this dictionary after the method call ends to understand the contents
   6. **return movieNames:** Control returns from function loadMovieNames() to the main program and returns the dictionary that we created after reading the itemfile.txt
4. **nameDict:** Let’s view the contents of the dictionary returned by loadMovieNames() function. So the dictionary as expected is containing movie ID as key and movie name as value.
5. **data = sc.textFile("file:///home/cloudera/moviedata/ratings.dat"):** Load the ratings.dat as textFile in spark and creates an RDD named data.
6. **ratings = data.map(lambda l: l.split(‘::’)).map(lambda l: (int(l[0]), (int(l[1]), float(l[2])))):** Here data RDD is mapped to ratings RDD in the following manner:
   1. **map(lambda l: l.split(‘::’)):** The contents of data RDD were as follows: 

So this first mapping is splitting each line on the basis of ‘::’

* 1. **map(lambda l: (int(l[0]), (int(l[1]), float(l[2])))):** This second map function is responsible for transforming as follows:
     1. **int(l[0]):** Type cast and convert the characters in first column of data RDD to integer as key.
     2. **(int(l[1]), float(l[2])):** Creates a tuple out of 2nd and 3rd columns of data RDD by type casting them to int and float respectively

Let’s view the result of ratings RDD what it received after the above transformations: 

So this completes our first plan: **Read the data and map it to (user, (movie,rating))**

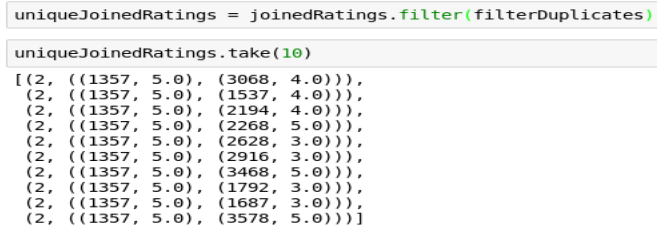
1. **joinedRatings = ratings.join(ratings):** This completes our second plan: **Self-join for each user to get all combinations of movie ratings by the user:**

**(user, ((movie,rating),(movie,rating))**

Below is some of the output of the self join

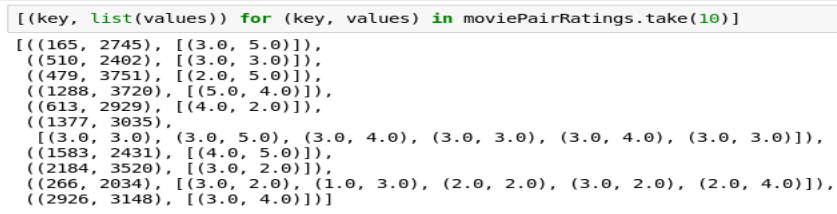
****

1. **uniqueJoinedRatings = joinedRatings.filter(filterDuplicates):** This line performs the following to filter out data as per the rules written in function filterDuplicates:
   1. **def filterDuplicates( (userID, ratings) ):** takes in 1 tuple argument as (userID, ratings).For e.g. from the **joinedRatings** screenshot from above it will take userID as 2 and ratings as ((1357, 5.0), (1357,5.0)) and so on.
   2. **(movie1, rating1) = ratings[0]:** take the first element from ratings and put the tuple (1357, 5.0) as (movie1, rating1)
   3. **(movie2, rating2) = ratings[1]:** take the 2nd element from ratings and put the tuple (1357, 5.0) as (movie2, rating2)
   4. **return movie1 < movie2:** returns a Boolean expression if movie1 < movie2. In other words it will return true when movie1 ID < movie2 ID.

**Implication of this method:** Since we did a self join of ratings to create joinedRatings it created a matrix with all possible cartesian mapping of (movieID, ratings) for each user. filterDuplicates method removes the rows for which movieID1 >= movieID2 and keeps only those where movieID1 < movieID2. Thus from our screenshot above rows like (2, (1357, 5.0), (1357, 5.0)) or (2, (1357, 5.0), (647, 3.0)) will be removed and rows like (2, ((1357, 5.0), (3068, 4.0))), (2, ((1357, 5.0), (1537, 4.0))) will be kept. Let’s see the output below: 

1. **moviePairs = uniqueJoinedRatings.map(makePairs):** Map **uniqueJoinedRatings** as per rules written in method makePairs:
   1. **def makePairs((user, ratings)):** takes in 1 tuple argument as(userID, ratings).For e.g. from the **joinedRatings** screenshot from above it will take userID as 2 and ratings as ((1357, 5.0), (3068, 4.0)) and so on.
   2. **(movie1, rating1) = ratings[0]:** take the first element from ratings and put the tuple (1357, 5.0) as (movie1, rating1)
   3. **(movie2, rating2) = ratings[1]:** take the 2nd element from ratings and put the tuple (3068, 4.0) as (movie2, rating2)
   4. **return ((movie1, movie2), (rating1, rating2)):** Returns a tuple containing two tuples ((movie1, movie2), (rating1, rating2))

**Implications:** This method is just a rearrangement from (2, ((1357, 5.0), (3068, 4.0))) to ((1357, 3068), (5.0, 4.0)) so that it creates tuples mapped as ((movie1, movie2), (rating1, rating2))

1. **moviePairRatings = moviePairs.groupByKey():** Here key is (movie1, movie2) tuple and after grouping the **moviePairRatings** will contain list of ratings for each movie pairs. Let’s view the sample output of the transformation. 
2. **moviePairSimilarities = moviePairRatings.mapValues(computeCosineSimilarity).cache():** Calls the method **computeCosineSimilarity** and changes the values as per rules written in **computeCosineSimilarity:**
   1. **def computeCosineSimilarity(ratingPairs):** Takes the rating pairs generated as part of above transformations.
   2. **numPairs = 0:** Initialize a variable numPairs to 0
   3. **sum\_xx = sum\_yy = sum\_xy = 0:** Initialize 3 variables sum\_xx, sum\_yy, sum\_xy to 0
   4. **for ratingX, ratingY in ratingPairs:** Travel each rating pairs for the given (movieID1, movieID2) and store them in two variables ratingX and ratingY respectively. Thus for movieID1 the rating will be stored in ratingX and movieID2 the rating will be stored in ratingY.
   5. **sum\_xx += ratingX \* ratingX:** sum\_xx calculated as cumulative sum of ratingX2
   6. **sum\_yy += ratingY \* ratingY:** sum\_yy calculated as cumulative sum of ratingY2
   7. **sum\_xy += ratingX \* ratingY:** sum\_xy calculated as cumulative sum of ratingX\*ratingY
   8. **numPairs += 1:** increase the numPairs by 1
   9. **numerator = sum\_xy:** declare a variable numerator as sum\_xy
   10. **denominator =** **sqrt(sum\_xx) \* sqrt(sum\_yy):** declare a variable denominator as sqrt(sum\_xx) \* sqrt(sum\_yy).
   11. **score = 0:** initialize a score variable as 0
   12. **if (denominator):** checks if denominator is not 0
   13. **score = (numerator / (float(denominator))):** computes the score as (numerator / denominator – type cast to float)
   14. **return (score, numPairs):** returns from **computeCosineSimilarity** to main function with a tuple (score, numPairs).

**Implication of the above function:** The above function is very important as acts as main algorithm for our recommendation engine. It first projects n rating pairs (where n is the number of users who rated both movie1 and movie2) into n-dimensional vectors with movie1 as vector\_X and movie2 rating as vector\_Y. Let’s dry run the algorithm to understand it deeper:

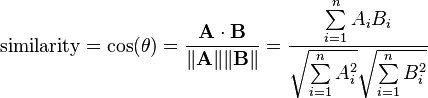
* + 1. Let us suppose (movie1, movie2) has 2 rating pairs as follows:

|  |  |
| --- | --- |
| **movie1** | **movie2** |
| r1m1 | r1m2 |
| r2m1 | r2m2 |

* + 1. So the formula for calculations of Sum\_XX, Sum\_YY, Sum\_XY, Numerator, Denominator and Cosine similarity score are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **movie1** | **movie2** | **sum\_xxi** | **sum\_yyi** | **sum\_xyi** |
| r1m1 | r1m2 | r1m1^2 | r1m2^2 | r1m1xr1m2 |
| r2m1 | r2m2 | r2m1^2 | r2m2^2 | r2m1xr2m2 |
| **Final variable values** | | | | |
|  | **Sum\_XX** | (r1m1^2 + r2m1^2) |  |  |
|  | **Sum\_YY** |  | (r1m2^2 + r2m2^2) |  |
|  | **Sum\_XY** |  |  | r1m1xr1m2 + r2m1xr2m2 |
|  | **Numerator** |  |  | r1m1xr1m2 + r2m1xr2m2 |
|  | **Denominator** |  |  | sqrt(Sum\_XX) \* sqrt(Sum\_YY) |
|  | **Cosine similarity score** |  |  | Numerator/Denominator |

* + 1. In vector space the vectors will look as below where we have to find the value of cos(**θXY**) which is given as below:



In our case A is represented as X vector, B is represented as Y vector.

X

Y

X(r1m1, r2m1)

Y(r1m2, r2m2)

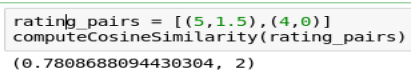
**θXY**

Vector representation of cosine similarity logic

Figure Not drawn to scale

* + 1. Let’s look at 2 sample calculations depicting the above idea in an excel sheet

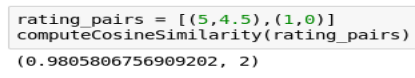
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **movie1** | **movie2** | **sum\_xxi** | **sum\_yyi** | **sum\_xyi** |
| 5 | 1.5 | 25 | 2.25 | 7.5 |
| 4 | 0 | 16 | 0 | 0 |
| **Final variable values** | | | | |
|  | **Sum\_XX** | 41 |  |  |
|  | **Sum\_YY** |  | 2.25 |  |
|  | **Sum\_XY** |  |  | 7.5 |
|  | **Numerator** |  |  | 7.5 |
|  | **Denominator** |  |  | 9.604686 |
|  | **cosine similarity score** |  |  | 0.780869 |

* + 1. So given the following input to our function it should be able to give the same similarity score alongwith the dimension of the vector space (i.e. **numPairs** variable as 2 in our case). Let’s validate that too: 

So our custom similarity scoring function is working as per expecation

* + 1. Let’s see another example as below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **movie1** | **movie2** | **sum\_xxi** | **sum\_yyi** | **sum\_xyi** |
| 5 | 4.5 | 25 | 20.25 | 22.5 |
| 1 | 0 | 1 | 0 | 0 |
| **Final variable values** | | | | |
|  | **Sum\_XX** | 26 |  |  |
|  | **Sum\_YY** |  | 20.25 |  |
|  | **Sum\_XY** |  |  | 22.5 |
|  | **Numerator** |  |  | 22.5 |
|  | **Denominator** |  |  | 22.94559 |
|  | **cosine similarity score** |  |  | 0.980581 |

* + 1. Validation: 
    2. **Inference:** Thus from our above validations we can conclude that movie pairs with ratings as in example (iv) has lower cosine similarities than the movie pairs as in (vi). This means that if a user is liking movie1 in example (iv) they are not liking movie2 and hence they have a lower similarity score.

Whereas the movie pairs in (vi) where users liked both movie1 and movie2, those who disliked movie1 also disliked movie2. Thus they have a higher similarity scores hence this pair should be recommended given that user has liked either movie1 or movie2. The ratings given in both the cases also convey the same idea.

Thus we can conclude that our logic is established well and working as per expectation.

Our vector space representation in cases (iv) and (vi) looks like below:

X

Y

X(5, 4)

Y(1.5,0)

**θXY\_4**

Vector representation of case (iv)

Figure Not drawn to scale

X

Y

X(5, 1)

Y(4.5,0)

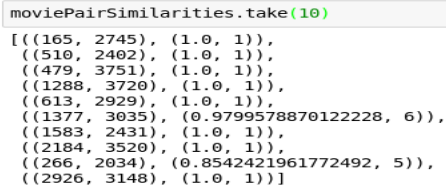
**θXY\_6**

Vector representation of case (vi)

Figure Not drawn to scale

Thus from the above figure clearly since **θXY\_4** > **θXY\_6** thus **cos(θXY\_4) < cos(θXY\_6),** hence movie pair in case (iv) is less similar than movie pair in case (vi) in our example.

1. Now moviePairSimilarities is an RDD which is created and it contains a map of similarity scores and **numPairs** in the format [(movie1, movie2), (similarity, numPairs)]. Below are some of the examples:



1. **if (len(sys.argv) > 1):** If user gives an argument in command line while invoking the code then perform the following:
   1. **scoreThreshold = 0.10:** Declare a score threshold variable
   2. **coOccurenceThreshold = 2:** Declare a co-Occurrence threshold as 2
   3. **movieID = int(sys.argv[1]):** Take the movieID passed as 2nd argument
   4. **filteredResults = moviePairSimilarities.filter(lambda((pair,sim)): \**

**(pair[0] == movieID or pair[1] == movieID) \**

**and sim[0] > scoreThreshold and sim[1] > coOccurenceThreshold):** Filter moviePairSimilarities RDD on the below conditions:

* + 1. if the key having movie pairs as tuple as (movie1, movie2) format then check whether movie1 or movie2 contains the ID passed by user.
    2. And Cosine similarity > scoreThreshold
    3. And numPairs > coOccurenceThreshold (i.e. coOccurenceThreshold decides the minimum number of ratings per movie pair should posses to come as recommendation pair.
  1. **results = filteredResults.map(lambda((pair,sim)): (sim, pair)).sortByKey(ascending = False).take(10):** Map the (key,value) pair as (value as key and key as value) then sortByKey in descending order. Thus it will be sorted in descending order of similarity score and numPairs i.e. number of ratings a particular movie pair posseses.
  2. **print "Top 10 similar movies for " + nameDict[movieID]:** User prompt to do a lookup from nameDict to extract the movie name from movieID which prints the message **"Top 10 similar movies for " <Movie name for MovieID passed as user input>**
  3. **for result in results:** traverse each item in our result RDD
  4. **(sim, pair) = result:** Extract out the information (similarity, numpairs) as sim and (movieID1, movieID2) as pair
  5. **similarMovieID = pair[0]:** take movieID1 and store in similarMovieID
  6. **if (similarMovieID == movieID):** check whether the movieID1 is equal to what user has given.
  7. **similarMovieID = pair[1]:** if condition in j is true make the movieID2 as the most similar movie.
  8. **print nameDict[similarMovieID] + "\tscore: " + str(sim[0]) + "\tstrength: " + str(sim[1]):** Finally print the message containing the following information:

**Top 10 similar movies for <User input movie>**

**<Moviename corresponding to most similar ID> score: (similarity from step h) strength: (numPairs i.e. total number of ratings this movie pair had from step h)**