BDM-1 Individual Assignment

Q1:

from pyspark import SparkContext,SparkConf

import sys

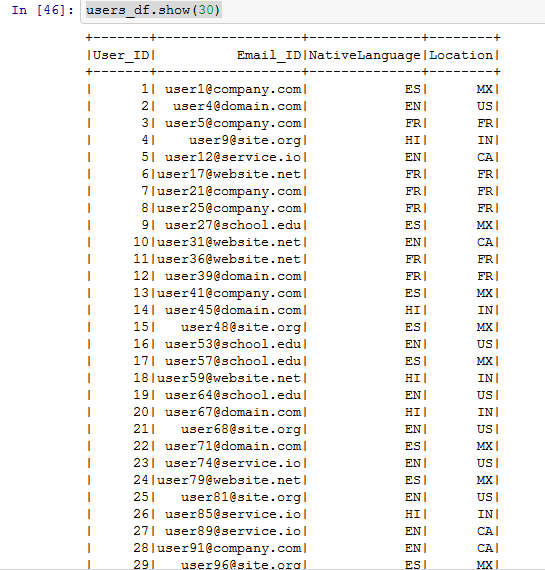
df = sc.textFile('file:///home/cloudera/users.csv').map(lambda l: l.split(","))

df.collect()



users\_df = df.toDF(['User\_ID','Email\_ID','NativeLanguage','Location'])

users\_df.show(30)



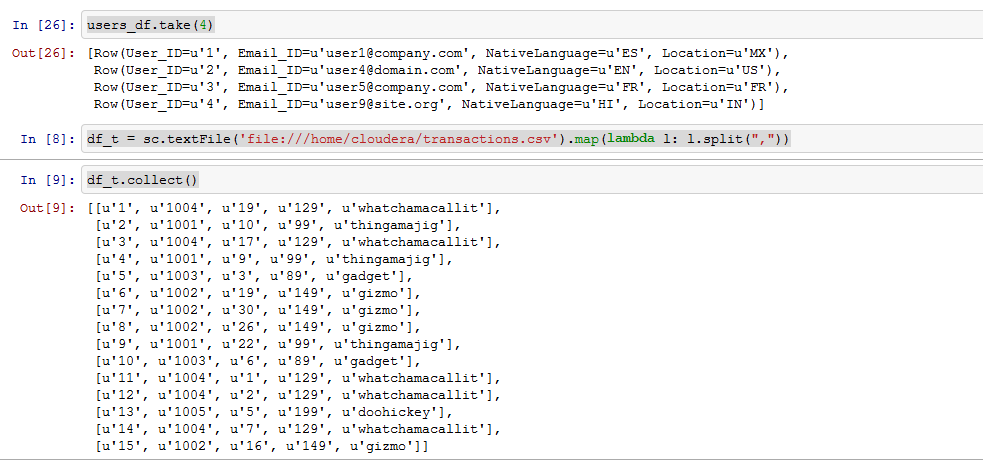
users\_df.count() – 30

df\_t.count() – 15

users\_df.take(4)

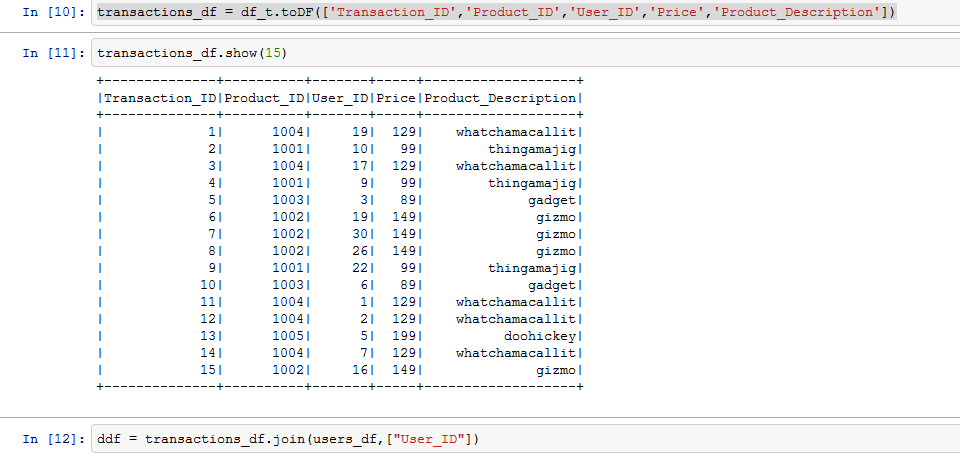
df\_t = sc.textFile('file:///home/cloudera/transactions.csv').map(lambda l: l.split(","))

df\_t.collect()



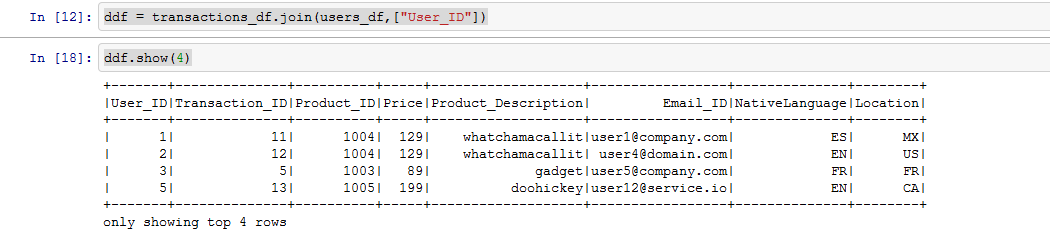
transactions\_df = df\_t.toDF(['Transaction\_ID','Product\_ID','User\_ID','Price','Product\_Description'])

transactions\_df.show(15)



ddf = transactions\_df.join(users\_df,["User\_ID"])

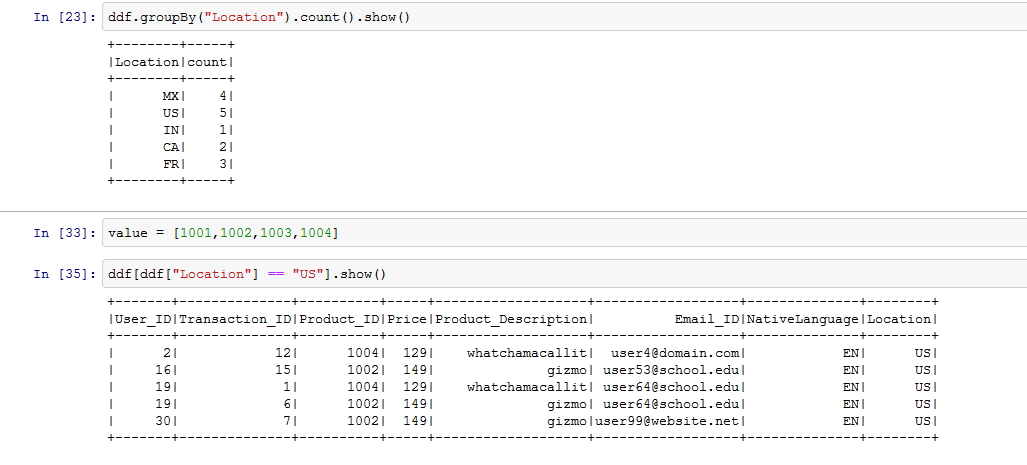
ddf.show(4)



1. Count of unique locations where each product is sold.

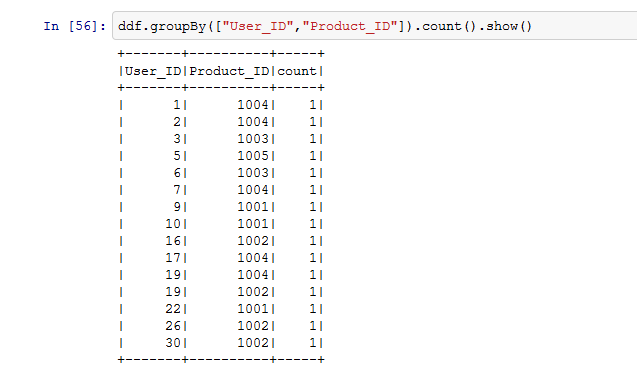
ddf.groupBy("Location").count().show()

ddf[ddf["Location"] == "US"].show()



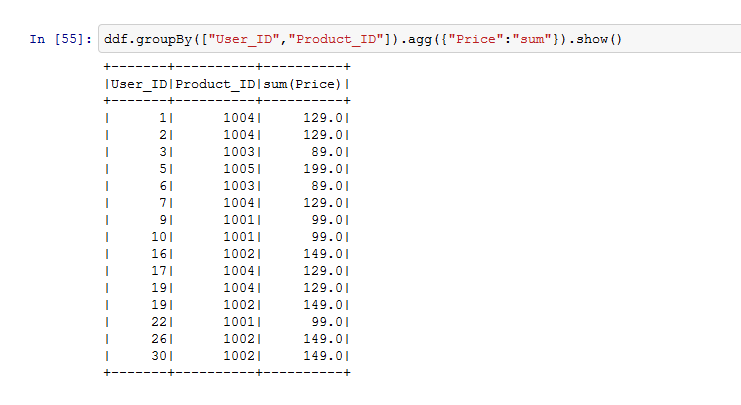
1. Find out products bought by each user.

ddf.groupBy(["User\_ID","Product\_ID"]).count().show()



1. Total spending done by each user on each product.

ddf.groupBy(["User\_ID","Product\_ID"]).agg({"Price":"sum"}).show()



Q2: SPARK-SQL:

1. **Save the dataset as a DataFrame, and print the schema.**

1. tw = sqlContext.jsonFile("file:///home/cloudera/tweets.json")

tw.show()

A screenshot of a cell phone

Description generated with very high confidence

tw.printSchema()

A screenshot of a cell phone

Description generated with very high confidence

tw.registerTempTable("tw")

tw.cache()

A close up of a newspaper

Description generated with high confidence

1. **Get all the tweets made by a user (any user would work. We should be able to replace user names to get tweets by that user).**

2. sqlContext.sql("SELECT user,text as tweets FROM tw WHERE user = 'Nene Kiameso' ").show(10)

A screenshot of a cell phone

Description generated with very high confidence

1. **Find count of all tweets by each user user.**

3. sqlContext.sql("SELECT user, count(text) as Count\_of\_tweets FROM tw GROUP BY user").show(50)

A screenshot of a cell phone

Description generated with very high confidenceA screenshot of a cell phone

Description generated with very high confidence

**d) Get a list of all of the people who are mentioned in tweets.**

4. sqlContext.sql("SELECT SUBSTRING(text,LOCATE('@',text),INSTR(text,' ')) as people\_in\_tweets FROM tw WHERE text LIKE '@%'").show()

A screenshot of a cell phone

Description generated with very high confidence

**e) Count the number of time each person is mentioned in the entire dataset of tweets.**

5. people = sqlContext.sql("SELECT id,SUBSTRING(text,1,INSTR(text,' ')) as people\_in\_tweets FROM tw WHERE text LIKE '@%'")

people.registerTempTable("people")

5. sqlContext.sql("SELECT count(id), people\_in\_tweets FROM people GROUP BY people\_in\_tweets").show(10)

A screenshot of a cell phone

Description generated with very high confidence

**f) Give top 50 users who are mentioned the most.**

6. sqlContext.sql("SELECT user, count(text) as Count\_of\_tweets FROM tw GROUP BY user ORDER BY Count\_of\_tweets DESC ").show(50)

A screenshot of text

Description generated with very high confidence

**g) Get a list of all hashtags mentioned in the dataset.**

7. sqlContext.sql("SELECT id,SUBSTRING(text,LOCATE('#',text),INSTR(text,' ')) as hastags FROM tw WHERE text LIKE '%#%'").show(20)

A screenshot of a cell phone

Description generated with very high confidence

hashtag = sqlContext.sql("SELECT id, SUBSTRING(text,LOCATE('#',text),INSTR(text,' ')) as hastags FROM tw WHERE text LIKE '%#%'")

hashtag.registerTempTable("hashtag")

h) Find how many times each hashtag is mentioned in the dataset.

8. sqlContext.sql("SELECT COUNT(id), hashtags FROM hashtag GROUP BY hashtags").show()

A screenshot of a cell phone

Description generated with very high confidence

i) Get a list of all of the people who are located in a particular city (e.g. Paris)

9. place1 = sqlContext.sql("SELECT id,place,SUBSTRING(text,1,INSTR(text,' ')) as people\_in\_tweets FROM tw WHERE text LIKE '@%'")

place1.registerTempTable("place1")

9. sqlContext.sql("SELECT place, people\_in\_tweets FROM place1 WHERE place = 'Boston'").show()

A close up of a logo

Description generated with high confidence

j) Get country wise distribution of users, and find out which country ranks highest in terms of number of tweets, and number of users.

10. sqlContext.sql("SELECT COUNT(user) as total\_users, country FROM tw GROUP BY country ORDER BY total\_users DESC").show()

A screenshot of a cell phone

Description generated with high confidence

10. sqlContext.sql("SELECT COUNT(text) as total\_tweets, country FROM tw GROUP BY country ORDER BY total\_tweets DESC").show()

A close up of a piece of paper

Description generated with very high confidence

10. sqlContext.sql("SELECT COUNT(text) as total\_tweets,COUNT(user) as total\_users, country FROM tw GROUP BY country ORDER BY total\_tweets DESC ,total\_users DESC").show()

A close up of a piece of paper

Description generated with very high confidence

k) Find out number of tweets where a user is from France and mentions Paris in their tweets.

11. sqlContext.sql("SELECT COUNT(\*) FROM sqlContext.sql("SELECT user,text, place FROM tw WHERE place = 'France' AND text LIKE '%Paris%'").show()

A close up of a logo

Description generated with high confidenceA screenshot of a cell phone

Description generated with high confidence

Q3:-

**Import the data and create a graph using GraphFrames (Hint: Your graph will have nodes and edges. Nodes here would be individual stations so id field would be name field in station.csv file. Edges would have src and dst so it would Start Station and End Station fields in trip.csv file respectively. You can make use of other fields as properties of nodes and edges).**

from graphframes import \*

trip = sqlContext.read.format("com.databricks.spark.csv").options(header='true', inferschema='true').load("file:///home/cloudera/trip.csv")

station = sqlContext.read.format("com.databricks.spark.csv").options(header='true', inferschema='true').load("file:///home/cloudera/station.csv")

trip.registerTempTable("trip")

trip.cache()

A screenshot of a social media post

Description generated with very high confidence

station.registerTempTable("station")

station.cache()

A screenshot of a social media post

Description generated with very high confidence

st\_unique=sqlContext.sql("select distinct name,lat,long from station")

st\_unique.registerTempTable("st\_unique")

from pyspark.sql.functions import \*

from graphframes import \*

vertices = st\_unique.withColumnRenamed("name", "id").distinct()

trip = trip.withColumnRenamed("Start Station","src")

trip = trip.withColumnRenamed("End Station","dst")

tripEdges = trip.select('Trip ID', 'Duration','src', 'dst','Start Terminal','End Terminal','Start Date','End Date','BIke #')

# Only include airports with atleast one trip from the departureDelays dataset

#airports = sqlContext.sql("select f.IATA, f.City, f.State, f.Country from airports\_na f join tripIATA t on t.IATA = f.IATA")

#airports.registerTempTable("airports")

#airports.cache()

tripEdges.cache()

vertices.cache()

vertices.show(4)

tripEdges.show(4)

g = GraphFrame(vertices,tripEdges)

print g

g.vertices.show()

g.edges.show()

inDg = g.inDegrees

inDg.show(10)

out\_Dg =g.outDegrees

out\_Dg.show(10)

g.degrees.show()

g.inDegrees.count()

g.outDegrees.count()

**b) Find out number of incoming connections and outgoing connections for each node and print the top 10 nodes.**

g.inDegrees.sort(desc("inDegree")).show(10)

A screenshot of a cell phone

Description generated with very high confidence

g.outDegrees.sort(desc("outDegree")).show(10)

A screenshot of a social media post

Description generated with very high confidence

**c) Find out which are the most common direct routes that people take and print top 10.**

motifs = g.find("(a)-[e]->(b)")

motifs.show()

A close up of text on a white background

Description generated with very high confidence

motifs.e.count()

**e) Find all such patterns where any station a is connected to station b, b is connected to c, but c is not directly connected to a.**

motifs = g.find("(a)-[ab]->(b); (b)-[bc]->(c)")

motifs.show()

A close up of text on a white background

Description generated with very high confidence

**#PageRank algorithm**

results = g.pageRank(resetProbability=0.15, maxIter = 2)

results.vertices.orderBy(results.vertices.pagerank.desc()).limit(20).show()

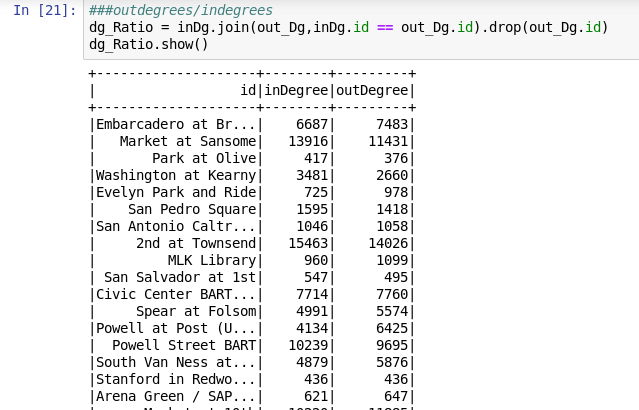
A screenshot of a cell phone

Description generated with very high confidence

**###outdegrees/indegrees**

dg\_Ratio = inDg.join(out\_Dg,inDg.id == out\_Dg.id).drop(out\_Dg.id)

dg\_Ratio.show()



degree\_ratio = dg\_Ratio.select(“id”, dg\_Ratio.outDegree/dg\_Ratio.inDegree)

degree\_ratio.sort(desc(“(outdegree / inDegree)”)).show(10)

A screenshot of a social media post

Description generated with very high confidence

**Q4 : - Problem - We need to make recommendation engine as similar to IMDB movies website using the data.**

**We have data related to user, movies and ratings. We have to write an algorithm for recommending the movies to users based on principle "people who like these movies also like"**

Solution : Data of users, movies and ratings need to be used for it. Few crucial points that can be considered are:

1. How many recommendations you can provide on a single page(10 are recommended)

2. Past user data can be used.

3. Complete Sample Based recommendation can be used.

4. Collaborative Filtering - a Machine learning technique can be applied.

5. Find correlation between user ratings in different movie pairs for each user and

set Co-occurrence threshold - At least 50 people , who have watched both movies.

and Score threshold - Correlation between two movies.

The algorithm procedure:

1. Read the data and map it to user.

2. Self-join for each user to get all combinations of movie ratings by the user:  
(user, ((movie,rating),(movie,rating))

3. Every user rated at least 20 movies, so this blows up the data to huge scale

4. Remove duplicates from the self-join

5. Key by every combination of movies that were rated together:  
((movie1,movie2),(rating1,rating2))

6. Group pairs of ratings by pairs of movies: ((m1,m2),((r1,r2),(r1,r2),…,(r1,r2)))

7. Calculate correlation for each pair: ((m1,m2),(score,numPairs))

8. Filter and sort the top ten similarities to a movie requested by the user.

**Code and comments:**

import sys

**#Import spark classes into your program**

from pyspark import SparkConf, SparkContext

from math import sqrt **#Import square root function from Math package**

def loadMovieNames(): **### a user defined function- LoadMovieNames**

movieNames = {} **### movienames is an empty dictionary**

with open("/home/cloudera/moviedata/itemfile.txt") as f: **###open itemfile using with function and put it in variable 'f'**

for line in f: **###for loop takes each line of file 'f'**

fields = line.split('|') **### split the line and assign it to variable "fields"**

movieNames[int(fields[0])] = fields[1].decode('ascii', 'ignore') **###making a key: value pair, using decode to make it human readable movie name**

return movieNames **###returning a dictionary**

def makePairs((user, ratings)): **##user-defined function having two arguments - user and ratings**

(movie1, rating1) = ratings[0] **###making every combination of movie and ratings**

(movie2, rating2) = ratings[1] **###making every combination of movie and ratings**

return ((movie1, movie2), (rating1, rating2)) **## returning movie and ratings pairs**

def filterDuplicates( (userID, ratings) **): ## user defined function**

(movie1, rating1) = ratings[0]

(movie2, rating2) = ratings[1]

return movie1 < movie2 **###return TRUE or FALSE**

def computeCosineSimilarity(ratingPairs): **##function for computing cosine similarity and takes only one argument**

numPairs = 0 **###assigning numpairs variable to 0**

sum\_xx = sum\_yy = sum\_xy = 0 **###assigning value 0 to three variables**

for ratingX, ratingY in ratingPairs: **##FOR Loop taking two temporary variables in ratingpairs variable**

sum\_xx += ratingX \* ratingX **## two same taings are multipled and added on to sum\_xx**

sum\_yy += ratingY \* ratingY **## two same taings are multipled and added on to sum\_yy**

sum\_xy += ratingX \* ratingY **## Acombination of X and Y rating is multiplied and added on to sum\_xy**

numPairs += 1 **##incrementing it by 1**

numerator = sum\_xy **##numerator will be sum\_xy- multiplication od 2 ratings**

denominator = sqrt(sum\_xx) \* sqrt(sum\_yy) **##denominator will be multiplication of magnitude of 2 ratings**

score = 0 **# a variable score is assigned to 0.**

if (denominator): **## If loop is used and an argument -denominator is used**.

score = (numerator / (float(denominator))) **##calculate score- cosine similarity**

return (score, numPairs) **###return score – cosine similarity between two ratings and numpairs is equal to total number of movies.**

### Crating an application "MovieSimilarities" , creating a spark context, local[\*] is argument, it maens that we are going to use Spark's built - in cluster manager and treat each core on desktop as a node on a cluster

conf = SparkConf().setMaster("local[\*]").setAppName("MovieSimilarities")

sc = SparkContext(conf = conf) **##creating sparkcontext sc**

## It will return a dictionary of movies that maps movie IDs to human readable names

print "\nLoading movie names..." **###printing a line**

nameDict = loadMovieNames() #**calling a function and creating a dictionary - nameDict**

data = sc.textFile("file:///home/cloudera/moviedata/datafile2.txt**") ###using sparkcontext to load a textfile**

# Map ratings to key / value pairs: user ID => movie ID, rating

ratings = data.map(lambda l: l.split()).map(lambda l: (int(l[0]), (int(l[1]), float(l[2])))) **###It will split columns and map it into a key-value pair, ratings can have float datatype**

**# Emit every movie rated together by the same user.**

**# Self-join to find every combination.**

joinedRatings = ratings.join(ratings)

**# At this point our RDD consists of** userID => ((movieID, rating), (movieID, rating))

**# Filter out duplicate pairs**

uniqueJoinedRatings = joinedRatings.filter(filterDuplicates)

# Now key by (movie1, movie2) pairs.

moviePairs = uniqueJoinedRatings.map(makePairs)

# We now have (movie1, movie2) => (rating1, rating2)

# Now collect all ratings for each movie pair and compute similarity

moviePairRatings = moviePairs.groupByKey()

# We now have (movie1, movie2) = > (rating1, rating2), (rating1, rating2) ...

# Can now compute similarities.

moviePairSimilarities = moviePairRatings.mapValues(computeCosineSimilarity).cache()

# Save the results if desired

#moviePairSimilarities.sortByKey()

#moviePairSimilarities.saveAsTextFile("movie-sims")

# Extract similarities for the movie we care about that are "good".

if (len(sys.argv) > 1):

scoreThreshold = 0.10 ##Score Threshold is minimum cosine similarity value between 2 movies to identify good movies

coOccurenceThreshold = 2 ## Co-occurence threshold is the no of times two movies are rated together

movieID = int(sys.argv[1])

# Filter for movies with this sim that are "good" as defined by

# our quality thresholds above

filteredResults = moviePairSimilarities.filter(lambda((pair,sim)): \

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

and sim[0] > scoreThreshold and sim[1] > coOccurenceThreshold)

# Sort by quality score.

results = filteredResults.map(lambda((pair,sim)): (sim, pair)).sortByKey(ascending = False).take(10)

print "Top 10 similar movies for " + nameDict[movieID]

for result in results:

(sim, pair) = result

**# Display the similarity result that isn't the movie we're looking at**

similarMovieID = pair[0]

if (similarMovieID == movieID):

similarMovieID = pair[1]

print nameDict[similarMovieID] + "\tscore: " + str(sim[0]) + "\tstrength: " + str(sim[1])