Amazon US Customer Reviews - Link Analysis and Market Basket Analaysis

Luca Paoletti and Sofia Gervasoni

March 17, 2023

Contents

1 Introduction

The goal of this analysis is twofold: ranking of customers based on the linkage among them and ranking of products based on the linkage among them. In the first case, there will be a link between two customers if they have reviewed at least the same product, while in the second one two products will be linked if they have been reviewed at least by the same customer. To pursue these aims, we used the PageRank algorithm and in order to work on a large quantity of data has been introduced the MapReduce algorithm. In particular, we will work in a Spark context (using PySpark) where the files are stored as RDD.

To expand our analysis, we decided to implement the Market Basket Analysis through *A-priori* and *SON algorithm*. Thanks to this analysis, we were able to understand which pairs of products tend to be reviewed by the same customer.

2 Data

For this project, we used the dataset 'Amazon US Customer Reviews' from Kaggle. This dataset contains information regarding Amazon customer reviews for different product categories (one tsv each). To obtain these data we used Kaggle API.

2.1 Chosen dataset and the parts considered

We decided to run our PageRank algorithm on more than one category. In particular, we first focused our analysis on the smallest dataset in order to have results in a short time. After having proven our algorithm we validated it using larger datasets. As the smallest dataset, we chose the category *Digital Software* (53.86 MB), whereas regarding higher size we chose the category *Grocery* (956 MB).
On the other hand, regarding the Market Basket Analysis, we decided to run this second algorithm only on the *Digital Software* category.

Regardless of the products' category the dataset is referred to, the structure of the dataset is always the same. In each dataset, there are 15 columns containing information about products and customers details, text reviews description, rating and date of review, but we were interested only in *product id* and *customer id*. The variable *customer id* contains a code to identify the customer that rated one or more products. On the other hand, *product id* is a code that identifies univocally the products. As the main aim of this analysis is to find the linkages among different products, we assume that two products are linked if the same customer reviewed both of them. Vice versa, two customers are linked if they reviewed the same product.

2.2 Data organization

In order to manage large quantities of data, we implemented a PageRank algorithm using the PySpark library. As far as we are dealing with datasets we introduce the concept of RDD, which are immutable Distributed collections of objects of any type. As the name suggests is a Resilient (Fault-tolerant) record of data that resides on multiple nodes. RDDs are one of the main abstractions of Spark. They represent immutable elements distributed across different nodes. The main characteristics of RRD are that it is:

- **Resilient**, the system is able to recompute/recover missing or damaged partitions due to node failures;
- **Distributed**, data resides on multiple nodes in a cluster;
- **Dataset**, collection of data;
- **Immutable**, once created, they cannot change;
- **Lazy evaluated**, operations are performed only when necessary;

- **Parallel**, operations are performed parallelly.

In order to import the dataset as an RDD, we used the library PySpark from Python 3. We first initialize the Spark session and we then import the RDD file. In order to import the data, we used the command *sparkContext.textFile()*, which is a method used to read a text file from HDFS, S3 and any Hadoop-supported file system, this method takes the path as an argument and optionally takes a number of partitions as the second argument.

3 The algorithm

3.1 PageRank

The algorithm used to run our analysis is **PageRank**. PageRank is used to determine the importance of the web pages represented as nodes linked among them with edges. In our specific case, nodes are products/customers, and we aim to determine the relevance of each of them. From now on we will refer to products only for sake of clarity.

Each edge $i \to j$ means that the product i is linked to the product j. One first idea to determine the relevance of a product can be the number of other products referring to it. For example, in our case, product (*productcode*) is referred to the most, so one could assume that it is the most reviewed one. However, we do not account for the relevance of other products. For example, if a very relevant product, x, refers to another product y. We could also safely say that y is probably relevant although it may be referred to just a few times. Thus, to get the score of a product i, r_i, we can consider a score like this one:

$$r_i = \sum_{j \to i} \frac{r_j}{d_j}$$

This means that the relevance score of the page i, named r_i, is given as a weighted sum of the relevance scores of all the pages referring to i. Each page referring to i is weighted according to the out-degree d_j (the number of pages referred from j). Of course, without knowing the relevance of all pages referring to i, we cannot determine the relevance of i.

We can now define the transition matrix M, where each entry m_{ij} in row i and column j has value $1/d$ if product j has d arc out and one of them is to node i, otherwise it takes value 0. We can intend each entry of the matrix as the probability distribution of a random surfer that after various steps it reaches product j. Starting from a vector (v_0) equal to $1/n$ for each component, after one step the distribution of the surfer will be Mv_0, after two steps M^2v_0 and so on. The probability x_i that a random surfer will be at node i at the next step, is $\sum_j m_{ij}v_j$. Here m_{ij} is the probability that a surfer at node j will move to node i at the next step and v_j is the probability that the surfer was at node j at the previous step.
This process could lead back to the Markov chain, and consequentially we can assert that the surfer will approach the limit distribution v that satisfies $v = Mv$. This is true under two conditions: the nodes are strongly connected and there are no dead ends.
It is important to note that the transition matrix is column-wise stochastic, meaning that the sum of the elements for each column is 1. As it is a column-wise stochastic matrix the eigenvalue associated with the principal eigenvector is 1 (the principal eigenvector v_i is the probability vector that will not change at $i + 1$).

3.2 Market Basket Analysis

The Market Basket Analysis (MBA) is used to describe the relationship between two kinds of objects. On one hand, we have the *items*, and on the other hand, we have the *baskets* (or item sets). It is based on the association rule $I \to j$, where I is a set of items and j is an item. In order to understand this rule is important to define the *confidence* of the rule:

$$\frac{S(I \cup \{j\})}{S(I)}$$

where S represent the support.This measure indicates the probability of j being in the basket with I. Finally, to describe whether an item set is frequent or not, we must set a threshold. If $S(I) >$ threshold, then the item set is frequent.

3.2.1 A-priori

A-priori is an algorithm used to optimize the counting and the definition of the frequent item. Before all the first issue we can deal with is the definition of the size of subset k: a k too small means many frequent items, and vice versa a k high will drop the count of frequent item.
The A-priori algorithm is run in three steps:

- in the **first step**, we create two tables: the first is used to translate the items into integers (1 to n) if necessary and the second one is used to count the singleton stored before.
- in the **between step**, we examine the counts to determine which of them are frequent as a singleton. In this step, we also create a frequent-items table that is the array of n elements taking value 0 (if the singleton is not frequent) and 1...m (if it is frequent).
- in the **second step**, we count all the pairs composed of two frequent items. Thanks to this algorithm we are to reduce the space required from $2n^2$ to $2m^2$.

3.2.2 SON

As we deal with inputs coming from the *Simple Randomized Algorithm* , it could return false positives (item set frequent in the sample but not in the whole) or false negatives (item set frequent in the whole but not in the sample). With SON algorithm we avoid these mistakes.
We start by dividing the input files into chunks and setting the $threshold = ps$. After that, we store on a disk all frequent items found for each chunk and we take the union of all the item sets frequent in one or more chunks (that became the candidate item sets). In the end, we count all the candidate item sets and those which have support at least s in the frequent item sets.

4 Scalability

In order to implement PageRank on large quantities of data, there are two types of operations that we can perform on RDD: *transformation* and *action.*

- **Transformations** are applied on RDDs and produce other RDDs. Some common transformations are *map*, and *filter*
- **Actions** do not return RDDs anymore. Actions do set in motion the sequence of transformation required to produce the result. Once the computation is done you get the result as output. Some common actions are *collect*, *count*, *reduce*, and *take.*

In our specific case, we need to implement the distributed version of the matrix multiplication. We will have the matrix P represented as (i, j, m_{ij}) and the vector p represented as (j, v_j).
The MapReduce algorithm proceeds as follows: firstly, we map each $(i, j, m_{ij}) \rightarrow (i, m_{ij}v_j)$ and next, we reduce by key $(i, [m_{ij}v_j, \ldots, m_{it}v_t]) \rightarrow (i, m_{ij}v_j + \cdots + m_{it}v_t)$.

Iterating this algorithm n times, we will find out the PageRank result after n iterations.
This implementation is valid both for customers and products link analysis.

5 Description of the experiments

5.1 PageRank

We started our analysis by importing the dataset as an RDD, and we set the number of partitions to 8 (customizable). For this purpose we used the following code line:

```
df = spark.sparkContext.textFile('file.tsv', minPartitions=8)
```

From now on the goal of the project is to not save in main memory (*.collect()* and *.take()*), but to work with RDD objects.

5.1.1 Pre-processing

We now introduce the pre-processing steps with the aim of converting our raw data into something more structured that could be given as input to the PageRank algorithm.

The script proceeds with a custom function that helps us keep only customers and products id columns.

```
def parse_data(line):
    fields = line.split("\t")
    return fields[1], fields[3]
```

Then, we apply this function to the RDD in order to import only the useful data and, with the same code line we produce a (k, v) where k is the customer code and v is a list of products that the single customer reviewed. What does that mean? We are keeping as value all the products that are linked with each other.

```
df = df.map(parse_data).groupByKey().mapValues(list)
```

The code above regards the products linkage (group by customer id). In order to switch to customer linkage (grouping by product id) we use the following pipeline with which we are simply changing the order to (k, v) pairs.

```
df = df.map(parse_data).map(lambda x: (x[1], x[0])).groupByKey().mapValues(list)
```

This way, we are using products id as k and customers id as a list of values v.
At this point, we decide to drop every (k, v)-pair where we have only one element in the value's list. This decision came from the intuition stating that a customer who reviews only one product will not be the customer who lets two products link to each other. This does not mean that the reviewed product will not appear among our network's nodes, in fact, this product can appear in someone else's value list with other products.
The only case where we are losing that product (node) is when this item is reviewed only by a customer who reviews that product only. Below, is the code regarding this intuition.

```
df = df.filter(lambda x: len(x[1])>1).map(lambda x: x)
```

The next step is to transform the list of products in (k, v) pairs where k and v are existing links in the network (e.g., we start from $(k, [v_1, v_2, v_3])$ and we end up with $(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_2), (v_3, v_1), (v_2, v_1)$). To do that, we define a new custom function that returns each possible combination of the value list given as input.

```
import itertools
def combination(row):
    l = row[1]
    k = row[0]
    res1 = [(v[0], v[1]) for v in itertools.combinations(l, 2)]
    to_add = []
    for x in res1:
        to_add.append(tuple(reversed(x)))
    return (res1+to_add)
```

Once we have defined this function, we can use it to get the desired results in the form of (k, v) pairs.

```
df = df.map(lambda x: combination(x)).flatMap(lambda l: l)
```

As it is possible to notice from the above cell, we are using the *flatMap* function to pass from a list of lists to a single list of tuples.

This is the form of data that we were trying to reach, actually we were able to pass from a list of lists to a plain list. These steps were necessary to prepare the data to be given as input to our PageRank algorithm.

5.1.2 PageRank Implementation

We need now to define two important variables: the number of total nodes and the out-degree of each node.
As far as the number of nodes is concerned, it is pretty easy to count the number of tuples grouped by key and consequently get the cardinality of nodes.

```
total_nodes = df.groupByKey().count()
```

For the latter, we need to count by key the tuples. The result will be a (k, v) pair where k is the product id and v is the out-degree for each k.

```
id2id = df
id2degree = id2id.countByKey()
```

As far as the probability that a random surfer starts from a random node is $\frac{1}{\text{tot. nodes}}$, we set the vector of probability equal to $\frac{1}{\text{tot. nodes}}$ for each entry. In this case, we decide to use a dictionary where the key is the product id and the value is the probability in order to have the chance to update, at each iteration of the PageRank algorithm, the probability vector v.

```
prods = list(id2degree.keys())
p = 1/(total_nodes)
p2diz = {}
for prod in prods:
    p2diz[prod] = p
```

We now need to compute the sparse transition matrix as a list of tuples of the form (i, j, m_{ij}).

```
P = id2id.map(lambda x:(x[0],x[1],1/id2degree[x[0]]))
PT = P.map(lambda x: (x[1],x[0],x[2]))
```

Finally, we implement PageRank iteration to get the rank of each product related to the others. In this section, we have to define a number of iterations after which we have to stop. The only parameter tuning we could implement refers to the decision of the correct number of iterations to get to the limit distribution where $v = Mv$. Unfortunately, due to the huge amount of data, we cannot find this number and we iterate the algorithm 70 times (customizable parameter). Of course, in the case of a lower amount of data, it will be possible to understand after how many iterations v_i remains the same as v_{i-1} (limit distribution).

```
for i in range(70):
new_p = PT.map(lambda x:(x[0],(x[2]*p2diz[x[1]])))\
         .reduceByKey(lambda x,y: x+y)\
         .collect()
for idx,prb in new_p:
    p2diz[idx] = prb
```

As we can see from the above code, we are updating our dictionary of probabilities at each iteration. After 70 rounds, we get the position of the random surfer in our network of products.

We run this algorithm on both the categories chosen obtaining the results presented in the following section.

5.2 Market Basket Analysis

We started our analysis by importing the dataset as an RDD. For this purpose we used the following code line:

```
df = sqlContext.read.csv('amazon_reviews_us_Digital_Software_v1_00.tsv', header=True, sep =
    '\t')
df = df.select(col("customer_id"),col("product_id"))
```

We defined two functions that we will use further in the script: the first one is a simple sum function, and the other one will help us to check whether an item set is a subset or not of a basket.

```
def sum_actors(x,y):
    return x+y

def filtering(rddlist, filt):
  for item in filt:
    if set(list(item)).issubset(set(rddlist)):
      return ((item, 1))
```

At this point, we defined the A-priori function, which we will call after. This function takes two parameters as input: an RDD object (baskets' list) and an integer representing the threshold. As far as we are interested in finding frequent pairs of products, we will start by converting a list of lists (baskets) into a list of elements (products). We start now the counting of singleton, firstly mapping each item in a tuple where keys are equal to product code and value is equal to 1; we then reduce by key summing the value of each (k, v)-pair, keeping only frequent items (with count greater or equal than the threshold). Now that we have frequent singleton, we create every possible pairs of products, and we check whether or not the pair created is a subset of the existent baskets. The pairs that are subsets of the market baskets are then grouped by key (*product id*) and counted. The ones with a count greater than the threshold are the candidate pairs.

```
def apriori(rdd, threshold):

  flat_list = rdd.flatMap(list)

  singleton = flat_list.map(lambda item: (item , 1))
  singleton_summed = singleton.reduceByKey(sum_prod)
  singleton_filtered = singleton_summed.filter(lambda item: item[1] >= threshold )

  frequent_prod = singleton_filtered.map(lambda item: (item[0]))

  pairs_list = list(itertools.combinations(frequent_prod.toLocalIterator(),2))

  support_table_pairs = rdd.map(lambda x : filtering(x, pairs_list)).filter(lambda x: x is
      not None).cache()
  support_table_pairs_summed = support_table_pairs.reduceByKey(sum_prod)
  support_table_pairs_filtered = support_table_pairs_summed.filter(lambda item: item[1] >=
      threshold)

  return (support_table_pairs_filtered)
```

The function defined above will be used within the SON algorithm. So we start implementing this second algorithm by defining a list of lists representing a list of baskets containing their products. We then parallelize this baskets' list in different chunks (we defined 5 as the number of chunks) and we are now ready to run the algorithm in each chunk.

```
baskets = df.groupBy("customer_id").agg(collect_set("product_id").alias("product_id"))
basket_list = baskets.select('product_id').rdd.flatMap(list)
basket_list = sc.parallelize(basket_list.collect(), 5)
```

So we set the threshold equal to 35 (the mean of the counts) and as far as we are working with chunks, we set the adjusted threshold for the chunks, too.

```
minSupport = 35
numPartitions = basket_list.getNumPartitions()
adjSupport = minSupport/numPartitions
```

As already mentioned before, we are going to apply the A-priori algorithm to each chunk using as a threshold the adjusted one. The output of this phase is the collection of all the pairs frequent in the chunks.

```
candidates = sc.parallelize([])

for i in range(0, numPartitions-1):
  partition = sc.parallelize(basket_list.glom().collect()[i])
  support_table_pairs_filtered = apriori(partition, adjSupport)
  candidate_chunk = support_table_pairs_filtered.map(lambda item: (item[0],1))
  candidates = candidates.union(candidate_chunk)
```

The next step regards the elimination of false positive pairs. There could be false positive (FP) itemsets as far as at the step before we extract the candidates from each chunk but despite of that, a pair could be frequent in the sample but not in the whole dataset (definition of false positive).

```
candidates_prods = candidates.map(lambda item : item[0])
candidates_list = candidates_prods.collect()

candidates_check = basket_list.map(lambda x : filtering(x, candidates_list)).filter(lambda
    x: x is not None).cache()
candidates_summed = candidates_check.reduceByKey(sum_prod)
candidates_filtered = candidates_summed.filter(lambda item: item[1] >= minSupport)
```

This final object *candidate filtered* contains the frequent pairs obtained with the SON algorithm.

6 Comments and discussion on the experimental results

6.1 PageRank

We explore below some results obtained from the PageRank Algorithm we have just explained. In particular, we decide to run the algorithm on two different datasets: Digital Software and Grocery. Of course, as it is easy to imagine, the Grocery one is the biggest and, consequently, also the more time-consuming one. We found some issues when we were dealing with datasets with sizes greater than 2.5GB because of 'Run out of Memory' errors. Despite that, the algorithm is set to work with any size given as input as long as the available resources give this possibility.
The last important thing to underline is that, as far as the unique values of customers are higher compared with the unique values of products, the algorithm should take more time to run during the link analysis for customers (this will generate issues during our analysis due to the computer power).

We start with the Digital Software dataset. Here we list the result of the probabilities of the first 20 nodes of the network (top 20 products listed).

```
With prob: 0.024342511268255998, you surf on the product with code: B00NG7JVSQ
With prob: 0.018551667741591504, you surf on the product with code: B00FGDDTSQ
With prob: 0.01844041094023381, you surf on the product with code: B00FFINOWS
With prob: 0.016983349588758894, you surf on the product with code: B00H9A60O4
With prob: 0.016978121648307087, you surf on the product with code: B00PG8FOSY
With prob: 0.014882673115643886, you surf on the product with code: B009HBCU9W
With prob: 0.01475200597221529, you surf on the product with code: B00E7X9RUK
With prob: 0.014498762878422146, you surf on the product with code: B00MHZ6Z64
With prob: 0.014464889757401806, you surf on the product with code: B00M9GTHS4
With prob: 0.01423682760521059, you surf on the product with code: B008SCNLEY
With prob: 0.013944802201358616, you surf on the product with code: B008S0IMCC
With prob: 0.012905541769826409, you surf on the product with code: B00FGDEPDY
With prob: 0.012105505786555887, you surf on the product with code: B008SCMUUA
With prob: 0.012040529931077237, you surf on the product with code: B00NG7K2RA
With prob: 0.011246372827008893, you surf on the product with code: B00G0DXA9Y
With prob: 0.008601851234576421, you surf on the product with code: B00A42LWH0
With prob: 0.008360088854246987, you surf on the product with code: B00B1TGUMG
With prob: 0.007931474375427008, you surf on the product with code: B00NG7JYYM
With prob: 0.007437104213699128, you surf on the product with code: B008XAXAC4
With prob: 0.007311014949880904, you surf on the product with code: B00E7X9WZU
```

These products are the most linked among the ones in our network. In this case, being linked means for a product to be reviewed by the same customer as other products (not necessarily at the same time). Of course, these probabilities are really low because of the number of possible nodes in the network. Despite that, when we pass to the customer link analysis, we will see how these probabilities will decrease more and more with respect to the ones of the products. Here is an example with the same dataset.

```
With prob: 9.594895217431937e-05, you surf on the customer with code: 50861391
With prob: 9.192075506823401e-05, you surf on the customer with code: 12975480
With prob: 9.168479592475086e-05, you surf on the customer with code: 6881693
With prob: 8.60753475164656e-05, you surf on the customer with code: 13067700
With prob: 8.228625213632127e-05, you surf on the customer with code: 53055158
With prob: 7.865996464508964e-05, you surf on the customer with code: 16279212
With prob: 6.988512552313613e-05, you surf on the customer with code: 45875818
With prob: 6.740853858699272e-05, you surf on the customer with code: 33491881
With prob: 6.652876189804205e-05, you surf on the customer with code: 52318215
With prob: 6.479164581488242e-05, you surf on the customer with code: 53049444
With prob: 6.379748481409276e-05, you surf on the customer with code: 47176379
With prob: 6.372362664959905e-05, you surf on the customer with code: 50773234
With prob: 6.329907860114872e-05, you surf on the customer with code: 45912422
With prob: 6.293214840069963e-05, you surf on the customer with code: 38075126
With prob: 6.266694515122427e-05, you surf on the customer with code: 40981988
With prob: 6.266694515122427e-05, you surf on the customer with code: 39591562
With prob: 6.199068592364789e-05, you surf on the customer with code: 43861020
With prob: 6.191425349834097e-05, you surf on the customer with code: 47116728
With prob: 6.167965947353301e-05, you surf on the customer with code: 45895351
With prob: 6.139004985082112e-05, you surf on the customer with code: 53082499
```

As we mentioned before, the probabilities in these examples are low. Comparing these results with the product ones, the main difference is due to the number of nodes. In fact, looking at the customer analysis, we are dealing with many unique *customer id* and consequently, the number of nodes in the graph is higher. Moreover, because of the number of customers in the dataset, the time the algorithm takes to run is higher than the products' link analysis.

These two results listed above are related to the smallest files in the Kaggle dataset we are working with. When working with bigger files (e.g., grocery) we will see differences in terms of probabilities and run-time. Why? Simply because of the number of unique values (products or customers) we are dealing with.
Here below it is possible to notice what we have just underlined, the value of probabilities of link analysis among Grocery products are lower than the one of the previous analysis.

```
With prob: 0.0007596998245068759, you surf on the product with code: B00DS842HS
With prob: 0.0005766619579232616, you surf on the product with code: B007PE7ANY
With prob: 0.0005043842422496992, you surf on the product with code: B007Y59HVM
With prob: 0.0005024502929769325, you surf on the product with code: B001E05Q64
With prob: 0.00047671385395609917, you surf on the product with code: B0029XDZIK
With prob: 0.0004420240743141137, you surf on the product with code: B007TGDXMU
With prob: 0.0003967991564293243, you surf on the product with code: B008I1XPKA
With prob: 0.0003809488580372655, you surf on the product with code: B000LL0R8I
With prob: 0.0003723393869325713, you surf on the product with code: B005K4Q1VI
With prob: 0.0003429612933543691, you surf on the product with code: B000H7LVKY
With prob: 0.0003388321635835888, you surf on the product with code: B005K4Q1YA
With prob: 0.0003307721245324353, you surf on the product with code: B00MGW81YM
With prob: 0.0003277183871566991, you surf on the product with code: B000Z93FQC
With prob: 0.0003132522823542001, you surf on the product with code: B00EKLPLU4
With prob: 0.00030930332860118855, you surf on the product with code: B00H889MGK
With prob: 0.00030722732138225737, you surf on the product with code: B0051SU0OW
With prob: 0.0003002906980469096, you surf on the product with code: B00DDT116M
With prob: 0.00027784936039107293, you surf on the product with code: B000EV0SE4
With prob: 0.00026302511201713896, you surf on the product with code: B00856TSCC
With prob: 0.00026181205920464416, you surf on the product with code: B007TGDXN0
```

When we run the same analysis on the customer, to find customer linkage, we had some issues with computing power and the algorithm crushed during the page iteration.

Because of this issue, we decide to run the last product's analysis on two different datasets merged together (Digital Software and Grocery). The results are linked below

```
With prob: 0.00075956969701099, you take the product with code: B00DS842HS
With prob: 0.0005751947356096408, you take the product with code: B007PE7ANY
With prob: 0.0005067025722491099, you take the product with code: B007Y59HVM
With prob: 0.0005002456006044353, you take the product with code: B001EO5Q64
With prob: 0.0004775940127311519, you take the product with code: B0029XDZIK
With prob: 0.00044267583345439546, you take the product with code: B007TGDXMU
With prob: 0.0003966127428385575, you take the product with code: B008I1XPKA
With prob: 0.00038029449395837967, you take the product with code: B000LL0R8I
With prob: 0.0003728020085753976, you take the product with code: B005K4Q1VI
With prob: 0.00034173402665837516, you take the product with code: B000H7LVKY
With prob: 0.0003389001240417885, you take the product with code: B005K4Q1YA
With prob: 0.00032905274906427527, you take the product with code: B00MGW81YM
With prob: 0.0003270737775977991, you take the product with code: B000Z93FQC
With prob: 0.0003121829074269907, you take the product with code: B00EKLPLU4
With prob: 0.00030994656843850597, you take the product with code: B0051SU0OW
With prob: 0.0003084753805524429, you take the product with code: B00H889MGK
With prob: 0.0002998416360069919, you take the product with code: B00DDT116M
With prob: 0.0002776659782343587, you take the product with code: B000EV0SE4
With prob: 0.00026406322581829574, you take the product with code: B007TGDXN0
With prob: 0.00026236641642822186, you take the product with code: B00856TSCC
```

As we expected, the results underline the decrease in probabilities vector values and, regarding the products rank, the first results are about Grocery and not Digital Software. Of course, the same customer can 'create' a link between products of both categories, but in this case, the differences between the products of the two categories and the difference in terms of the number of nodes per category, let the Grocery ones be more likely to be ranked as firsts. Despite that, comparing these results with the ones of only Grocery products, we can see how similar the probabilities are (and the rank too) because of the ratio between the number of Software products and Grocery products.

Running the algorithm on different files, we notice how the most time-consuming operations have been the ones regarding the counts of total nodes (because of the *group by* operation) and the ones regarding the calculation of PageRank for different iterations. Of course the higher the number of records in the dataset, the higher the run time will be.

6.2 Market Basket Analysis

We now focus on the results obtained with the Market Basket Analysis. In particular, we decided to calculate the confidence of the top 3 association rules (most frequent pairs according to MBA). Before doing that we need the support of each singleton in the dataset, the code below explains how we do that.

```
flat_list = basket_list.flatMap(list)
singleton = flat_list.map(lambda item: (item , 1))
singleton_summed = singleton.reduceByKey(sum_prod)
singleton_filtered = singleton_summed.filter(lambda item: item[1] >= minSupport)
```

We decided to display the first ten most frequent pairs using the following code:

```
candidates_filtered.sortBy(lambda x: x[1], ascending=False).take(10)
```

Obtaining the following output:

```
[(('B008SCNLEY', 'B00FGDDTSQ'), 84),
 (('B009HBCU9W', 'B00FFINOWS'), 83),
 (('B00FFINOWS', 'B00NG7JVSQ'), 78),
 (('B00G0DXA9Y', 'B00PG8FOSY'), 63),
 (('B008SCMUUA', 'B00FGDEPDY'), 60),
 (('B00A42LWHO', 'B00G0DXA9Y'), 49),
 (('B00M9GTHS4', 'B00E7X9RUK'), 49),
 (('B00NG7JVSQ', 'B00PG8FOSY'), 43),
 (('B00NG7JVSQ', 'B00NG7K2RA'), 37)]
```

Furthermore, we conclude by summarizing the association rules and the corresponding confidence values in the following table.

Association rule	**Confidence**
B008SCNLEY $\rightarrow$ B00FGDDTSQ	0,046
B008SCNLEY $\leftarrow$ B00FGDDTSQ	0,066
B009HBCU9W $\rightarrow$ B00FFINOWS	0,06
B009HBCU9W $\leftarrow$ B00FFINOWSQ	0,065
B00FFINOWS $\rightarrow$ B00NG7JVSQ	0,012
B00FFINOWS $\leftarrow$ B00NG7JVSQ	0,056

On average, the probability that item i appears in a basket given the presence of j (where i and j are the items of the association rule) is 6%.