LINK ANALYSIS

Amazon US Customer Review Dataset

LAST NAME: RASHAD
NAME: AMR MOHAMED NAZIH MOHAMED

Abstract

The aim of this project is to perform a link analysis on the given Amazon US customer review dataset. I have decided to focus on the customers and how they are linked together if they have reviewed a common product. I believe this would be more interesting so that we can visualize who are the most/least influential customers in our dataset. I have tried different taxation scores to observe the time it takes to reach convergence, time here refers to iterations, and to observe the effect of it on the actual PageRank scores. Three different csv files have been processed to calculate the PageRank scores.

1. Introduction:

PageRank is an algorithm that is designed to observe and measure the influence or relevancy of nodes in a network graph. Nodes can be described as web pages, customers, products… A network graph is composed of nodes and edges, where an edge exists between two nodes if there's an interaction or a link between them. This project is an experimentation with the aim of calculating the PageRank scores on a large dataset. A well-known dataset has been used in this project, which is Amazon US customer reviews, and in this dataset, we consider a link exists when two customers have reviewed the same product.

This project is composed of three parts which are the following:

- Performing DEA (Data Exploration Analysis)
- Network/graph & adjacency list creation
- Calculating the PageRank scores for each customer

The three CSV files that have been used to implement the PageRank algorithm are the following:

- Major Appliances (amazon_reviews_us_Major_Appliances_v1_00.tsv)
- Mobile Electronics (amazon_reviews_us_Mobile_Electronics_v1_00.tsv)
- Personal Care Appliance (amazon_reviews_us_Personal_Care_Appliances_v1_00.tsv)

Columns considered in each file:

- product_id
- customer_id
- star-rating

2. Data Exploration Analysis:

A basic DEA has been developed for the three csv files, where I observe the schema of each dataframe, check if null variables exist, observe the top 10 products with 5 star-rating, by top I mean the most products with 5 star-rating that have been reviewed. Same observation was made for the top 10 products with 1 star-rating review.

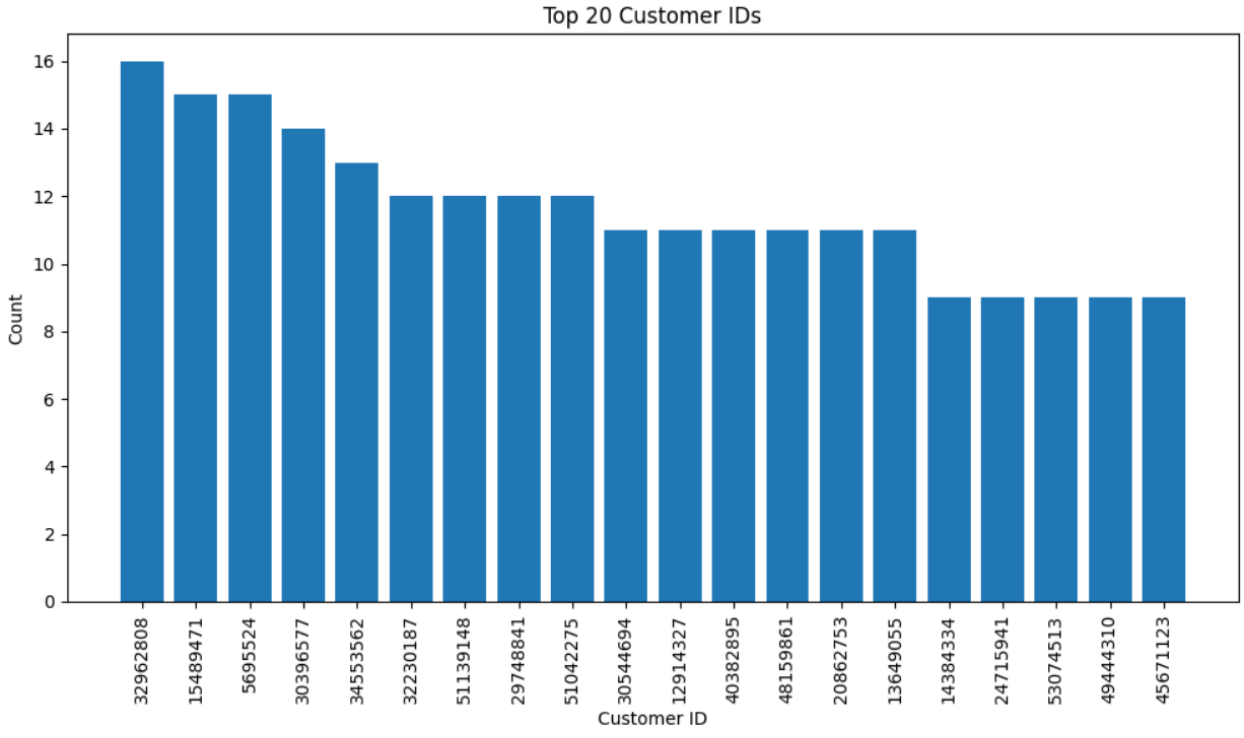
Below is an example of my DEA on the Personal Care Appliances csv file:

Top 10 products with 5-star rating:

product_title	count
GE MWF SmartWater Compatible Water Filter Cartridge - Refrigerator	926
SPT Countertop Dishwasher	698
NewAir AI-100R 28-Pound Portable Icemaker, Red	639
Avalon Bay Portable Ice Maker	483
Broan 412402 ADA Capable Non-Ducted Under-Cabinet Range Hood	475
Whirlpool Washer Lid Switch 3949238	450
Koolatron Coke Personal Mini Fridge	422
Danby 0.7 cu.ft. Countertop Microwave	381
Danby DAR195BL 1.8 cu.ft. All Refrigerator - Black	379
Haier HLP21N Pulsator 1-Cubic-Foot Portable Washer	367

Top 10 products with 1-star rating:

product_title	count
NewAir AI-100R 28-Pound Portable Icemaker, Red	151
Whirlpool WTW8800YW Cabrio 4.6 Cu. Ft. White Top Load Washer - Energy Star	140
IMCG Gas Range Protectors Set of 4	106
Koolatron Coke Personal Mini Fridge	105
Samsung SMH1816S 1.8 Cu. Ft. Stainless Steel Over-the-Range Microwave	101
Danby 0.7 cu.ft. Countertop Microwave	92
Whirlpool Duet WFW94HEXW 27 Front-Load Washer 5.0 cu. ft. Capacity - White	85
Danby 120 Can Beverage Center, Stainless Steel DBC120BLS	81
GE MWF SmartWater Compatible Water Filter Cartridge - Refrigerator	80
SPT Countertop Dishwasher	79



3. PageRank Theoretical Definition & Concept:

PageRank is algorithm developed by the co-founders of Google, with the aim of measuring the importance of web pages across the entire web network. Each web page is assigned a PageRank score, and this score can be used as an indication of the importance of the web page within the entire network. It's worth noting that this algorithm can also be applied to other types of networks, not specifically the web network. For instance, it can be applied to social networks or in our case, it can be applied to a review dataset.

The algorithm can be broken down into 6 steps:

i. **Graph Representation:**

   A graph needs to be initialized to represent the network, the entities, and the relations between them. Each entity is assigned an initial PageRank score that would be updated later when performing PageRank calculation.

ii. **Random Walking/Surfing:**

   The PageRank algorithm begins with a simulation where a random surfer or walker navigates through the network. Navigating here means going from one node to another by following the links between them. It's worth noting that the random surfer chooses to follow a random edge from the node s/he resides on to another node.

iii. **Taxation/Teleport:**

   A problem that the random surfer might face is reaching a dead-end node, which is a node that doesn't have any outgoing links, or being stuck in an endless loop, i.e., spider trap. To prevent such issues the algorithm uses a damping factor or a taxation variant (β), which is the probability that the surfer will continue its random walking. The reason is, when facing such situation, the surfer might stop, in other words leave the network, therefore asymptotically speaking, this would lead to the number of surfers going to be 0 at some point. So, when introducing the damping factor, a portion of the users will be distributed back across the other nodes ($1- \beta$). The value of the value of the damping factor is between [0,1], however it needs to be high, a typical value for example would be around 0.85.

iv. **PageRank Calculation:**

As we have said, all the nodes are initially assigned equal PageRank scores and in each iteration the score is updated until the algorithm reaches convergence or achieved maximum number of iterations. The PageRank score is based on the damping factor and the accumulated PageRank scores of the neighboring nodes. Below is the mathematical formula of how exactly the score is calculated:

$$\vec{V}(t+1) = \beta\, M\, \vec{V}(t) + (1-\beta)/n$$

Where V is the PageRank score, t is the iteration, n is the total number of nodes in the network, β is the damping factor and M is the transition matrix.

A transition matrix represents the probabilities of transitioning from one node to another through the outgoing links. For example, the transition probability from node 'a' to node 'b' is determined by the fraction of outgoing links from node 'a' to node 'b' over the total number of outgoing links existing from node 'a', in other words 1/outdegree. The sum of transition probabilities of each node needs to be equal to 1. The size of the transition matrix is NxN, where N is the number of nodes in the network.

v. **Iterative Process & Reaching Convergence:**

Convergence relies on the power method, which is multiplying the transition matrix by the initial PageRank vector several times, after each multiplication the PageRank vector is updated with the new scores. This multiplication is done until convergence is achieved, and convergence can be determined by measuring the Euclidean distance between the current PageRank vector and the previous one, if the distance is below a given threshold, then convergence is achieved.

vi. **Importance Interpretation:**

The final PageRank scores represent the influence of each node within the network. As we discussed earlier, nodes with higher scores are to be considered more influential than nodes with lower scores. The reason is that they are linked to other influential or important nodes.

4. PageRank Practical Application:

The application of the PageRank algorithm is divided into three steps:

- Creating the network graph & adjacency list
- Computing the PageRank scores
- Visualizing the output

i. Creating The Network Graph & Adjacency List:

PageRank algorithm was implemented using Apache Spark, which is a distributed computing framework. The reason behind using Spark is to efficiently leverage its distributed processing capabilities and data partitioning. Few problems were encountered such as the large size of the data, which results in a large graph.

However, it's worth noting that in real-world graphs, such as this example, nodes have relatively small number of edges compared to the total number of nodes, which leads to a sparse graph / matrix. Hence storing a sparse matrix can be very intensive process for the memory. A good approach to deal with such challenge is using an adjacency list representation instead of a transition matrix as this would only store the non-zero connections, this can be more memory-efficient especially when dealing with large graphs. Hence instead of using the transition matrix to calculate the PageRank scores, we just traverse the graph and compute the contributions from the neighboring nodes, it's worth noting that traversing the adjacency list is done by iterating over the neighboring nodes, this is much faster than just iterating over all the nodes.

One key point regarding the use of adjacency list representation is that it actually goes hand in hand with the Spark framework. The reason is that it can be partitioned and processed in a distributed manner. In our case, we have indeed partitioned our adjacency list through the use of 'repartition ()' method. I have set number of partitions to 100, however, due to working on Google Collaboratory environment with limited resources, it's challenging to have as many numbers of partitions as we want. The reason we would want to increase

number of partitions is if we're dealing with skewed data, meaning if we have some nodes having significant number of edges compared to other nodes, then one way to handle such issue is by increasing the number of partitions as the data will be distributed more evenly. Hence, we can observe that there is a trade-off between the resources we have and the number of partitions we want.

```
num_partitions = 100
partitioned_edge_df = edge_df.repartition(num_partitions)
partitioned_edge_df.cache()
```

```
DataFrame[src: int, dest: int]
```

```
vertices = partitioned_edge_df.select("src", "dest").distinct()
partitioned_edge_df.unpersist()
vertices.cache()
vertices.unpersist()
adjacency_list = vertices.rdd.groupByKey().mapValues(list)
adjacency_list.cache()
```

```
PythonRDD[212] at RDD at PythonRDD.scala:53
```

The adjacency list created is an RDD (Resilient Distributed Dataset), which is actually more memory efficient than a dataframe especially if we're dealing with large graphs and if there exist memory constraints due to the limited resources. One of the main reasons why that's the case is because an RDD offers lower memory overhead than a dataframe, that's due to the data being stored in a more compact and serialized format. Also, RDD can maintain references to the same objects across several partitions which again results in lower memory usage.

Finally, before computing the PageRank scores the adjacency list was cached in memory in order to avoid recalculation of costly transformations and to access the data inside it much faster. This is a beneficial process to apply when we are going to implement an iterative algorithm, in our case PageRank. It's worth noting that more advanced caching techniques can be implemented in a more advanced Spark environment such as Databricks, for instance persistent RDD caching can be an improvement or a more advanced approach. Hence, working in a more scalable environment can be a significant improvement when dealing with large graphs as the limitations of the resources wouldn't be a relative concern.

ii. Computing The PageRank Scores:

The computation of the PageRank is divided into 5 points:

- Functions created
- Calculating the contributions from each node
- Damping factor
- Calculating the new PageRank Values
- Number of iterations
- Threshold

First function is 'calculate_distance', which calculates the Euclidean distance between the two PageRank vectors.

```
[ ]  def calculate_distance(PR_values, OLD_PR_values):
         dist = math.sqrt(PR_values.join(OLD_PR_values).mapValues(
             lambda ranks: (ranks[0] - ranks[1])**2
         ).values().sum())
         return dist
```

Second function is ‘calculate_page_rank’ which calculates the PageRank score for each node. The following is how the algorithm is processed, I initialize two PageRank vectors, one has its values equal to 1/n, where n is the total number of nodes and the other one has its values equal to 1. I then join these PageRank scores to the adjacency list representation created, so now each node has its links and its initial PageRank score. This is stored in an RDD called joinedRDD. Each element in the RDD goes through a ‘computation’ applied by the flatMap () transformation and the results are stored into a new RDD, in our case the function actually takes an element from the joinedRDD and produces several key-value pairs for each destination node (node in the list of neighboring nodes). The overall result of this computation is calculating the contribution from the source node to the destination node and the output is in the form of key-value pairs stored in contributionsRDD. This is a main transformation in the PageRank algorithm because as we mentioned earlier, it calculates the contributions from each node to its neighbors.

ContributionsRDD is then used to calculate the PageRank score for each node, firstly the values per each key are summed up, in other words we sum up the contributions for each destination node. Then each destination node has its PageRank score calculated, this is done by applying the formula that we discussed earlier:

$$(1 - damping\_factor) / n + damping\_factor * rank$$

Hence, the overall result is a new RDD which contains the updated PageRank scores (PR_values_new). This RDD contains key-value pairs, where the key is the destination node, and the value is the PageRank score. This process is iterated 100 times or if convergence is reached then the iteration stops.

```python
def calculate_page_rank(damping_factor, threshold, max_iterations, adjacency_list):
    n = adjacency_list.count()
    distance = 1
    initial_PR = 1.0 / n
    PR_values = adjacency_list.mapValues(lambda neighbors: initial_PR)
    OLD_PR_values = adjacency_list.mapValues(lambda neighbors: 1.0)
    for i in range(max_iterations):
        OLD_PR_values = PR_values
        joinedRDD = PR_values.join(adjacency_list)

        contributionsRDD = joinedRDD.flatMap(
            lambda entry: [(dest, entry[1][0] / len(entry[1][1])) for dest in entry[1][1]]
        )

        PR_values_new = contributionsRDD.reduceByKey(lambda x, y: x + y).mapValues(
            lambda rank: (1 - damping_factor) / n + damping_factor * rank
        )

        PR_values = PR_values_new
        print("Iteration:", i)

        if distance < threshold:
            break

        distance = calculate_distance(PR_values, OLD_PR_values)
        print("Distance:", distance)

    return PR_values.collect()
```

The convergence is checked through the call of the 'calculate_distance' function, where the Euclidean distance is calculated between the two PageRank vectors current and old. If the result is lower than the threshold specified, then we declare convergence has been reached and the algorithm concludes. The threshold value selected is 10e-7, however this can be another hyperparameter that can be tuned in order to observe its effect on the actual scores and the time needed to reach convergence. Damping factor is also another hyperparameter that was tuned, where three values have been experimented in order to see the implications of each value on the PageRank scores and number of iterations performed. The three values are 0.8, 0.85 and 0.9.

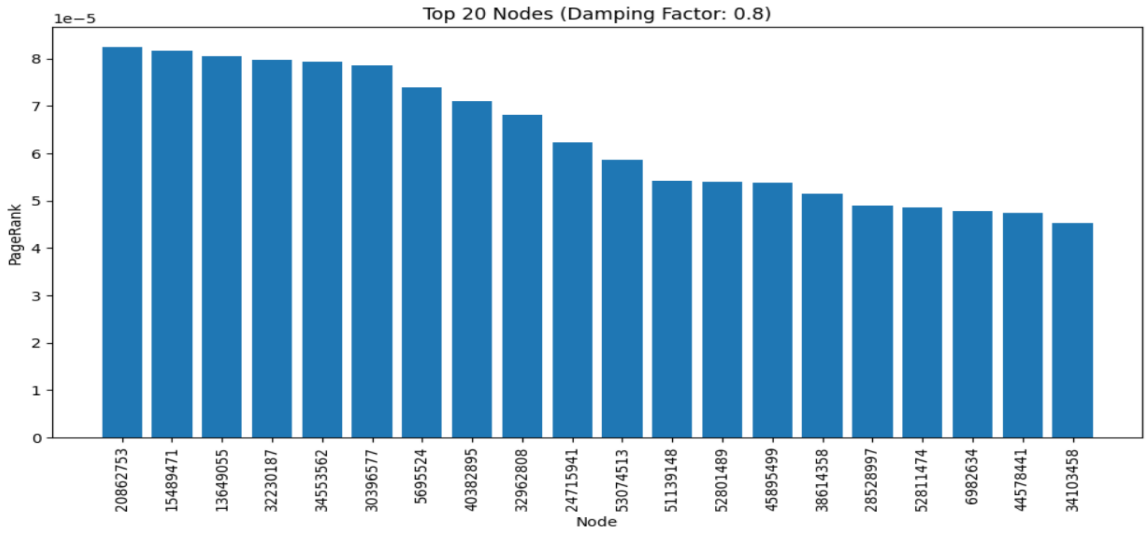
```
damping_factors = [0.8, 0.85, 0.9]
threshold = 10e-7
max_iterations = 100
final_PR = []
for damping_factor in damping_factors:
    page_rank_scores = calculate_page_rank(damping_factor, threshold, max_iterations, adjacency_list)
    final_PR.append(page_rank_scores)
```

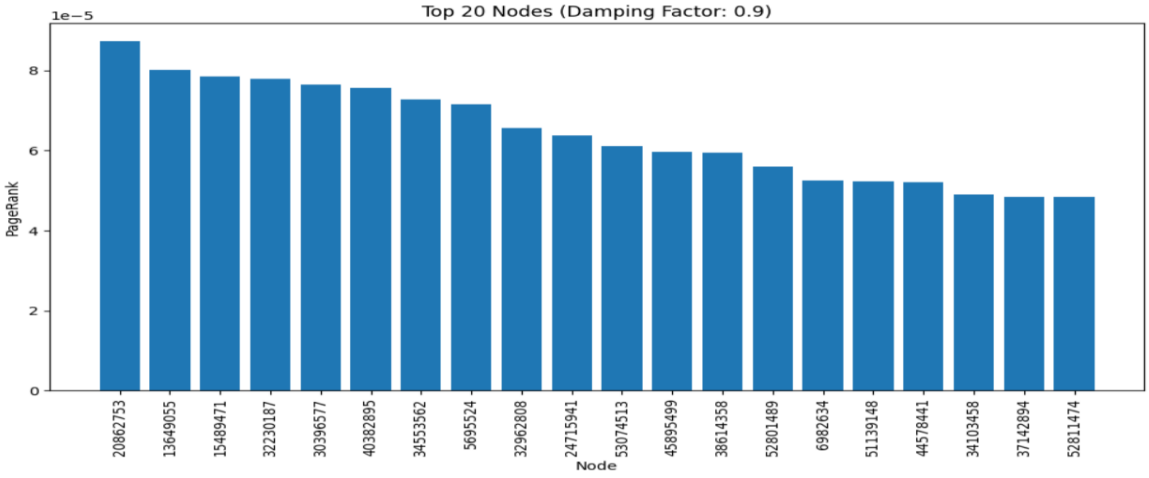
```
Iteration: 0
Distance: 0.0007678845277688204
Iteration: 1
Distance: 0.00021317563021490304
Iteration: 2
Distance: 7.715848352323021e-05
Iteration: 3
Distance: 5.0892303740495564e-05
Iteration: 4
Distance: 3.376428622269167e-05
Iteration: 5
Distance: 2.416003049288616e-05
Iteration: 6
Distance: 1.753876695090322e-05
Iteration: 7
Distance: 1.3067977166705747e-05
Iteration: 8
Distance: 9.827808123882932e-06
Iteration: 9
Distance: 7.48274915747416e-06
Iteration: 10
```

We can observe that in our case a lower damping factor leads to faster convergence, i.e., a smaller number of iterations and as expected, the scores decrease with each iteration, which makes sense because of the redistribution and the taxation process that takes place.

iii. Visualizing The Output:

Finally, some visualizations have been performed in order to see top influential and least influential nodes. Below is an example of the output plots.





One thing we can clearly observe is the order of the top 20 is changed when the damping factor is increased, this might be because of the decrease in the influence of random jumps. Looking at the plots, one might apply future research in order to determine if increasing the damping factor leads to more accurate rankings or would it introduce bias towards the nodes with strong local connectivity. These might be interesting concepts to approach while working on this dataset.

5. Conclusion & Final Thoughts:

Calculating the PageRank scores for a large graph can be demanding if the resources are limited. By leveraging the spark distributed framework, I was able to apply the PageRank algorithm efficiently and I can conclude from calculating the PageRank scores of three different csv files that increasing the taxation variant has an effect on the number of iterations needed in order to reach convergence and the flow of the PageRank scores.

Final thought: Doing this project was actually beneficial as after doing some research I was able to conclude that part of my thesis project could be integrating the page rank scores in a dataset in order to compute an implicit score that can be used to create a recommendation system.