PageRank: an index of Musical Instruments

Marta Campagnoli
Data Science and Economics
Algorithms for Massive Data
ID 928635
marta.campagnoli@studenti.unimi.it

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

1 Introduction

PageRank is an algorithm generally used to compute an index representing the importance of a node in a network, based on the concept of "random surfers" navigating through the network itself and their probability of finding themselves in a specific node at the end of the "surfing".

In this report I describe the process carried out to implement the PageRank algorithm in such a way that could be replicated on big data, by analyzing the Amazon US Customer Reviews Dataset. In such dataset, each record contains a review for a product in the USA market along with unique identifiers for the product and the reviewer associated with the record. For the purpose of this work, I used the subcategory of Musical Instruments reviews to create an index of products, where two nodes are connected if they share at least one reviewer. Code for this project has been written and compiled in Python 3; the process in order to create edges and the PageRank algorithm has been run using Spark on Google Colab.

While the majority of the results and code presented refer to experiments run on the single file of Musical Instruments, the process has been tested on a larger set of data to ensure that they could be replicable on a bigger scale. Results for this ulterior tests and the choice of additional files to use alongside the Musical Instruments one, along with the reason for presenting results for a single file, are later described.

2 Methodology

The project has been carried out in three main steps: preprocessing the data, creating the edge files to use in the algorithm, and running the algorithm on the data. The PageRank algorithm is a topic discussed during both class in the course and tutoring lessons; in the project, I tried to adapt the proposed code to work on the data obtained through the processing phase.

2.1 Dataset: chosen data and preprocessing

The Amazon US Customer Reviews Dataset is composed of 37 files, one for multilingual reviews and 36 for different categories of phisical and digital products, collected from the Amazon marketplace from 1995 to August 31st 2015. As previously stated, this project has been carried out on the Musical Instruments Category.

Each record contains, along with the corpus of the review itself and additional information such as star rating and helpful votes, two main variables that have been used during preprocessing, of which I provide the official description:

- "customer_id", a random identifier that can be used to aggregate reviews written by a single author;
- "product_parent", a random identifier that can

be used to aggregate reviews for the same product.

Before the brief preprocessing phase, I created two simple functions to help the process:

- "indexesprod" has been created to reindex products in the dataset for cleaner results when creating edges; it needs the dataset to be ordered by values in "product_parent" column. Takes the first value of the product_parent column after ordering the dataset by its value and creates a list of indexes by scanning the column. Guarantees the number of nodes are correctly counted in the PageRank algorithm. The list can be added to the dataframe, and the function can be potentially adapted to create indexes for any column if needed;

- "tuple_list" takes the "customer_id" column and "indexes" column created with the first function to output the list of tuples to be transformed into a resilient distributed dataset simply by zipping the two. It's been created for a cleaner result in the corpus of the project.

Both function only need to receive the dataframe as input.

The choice of using the Music categories was mainly based on personal preference, while the one of performing the experiments on a single file was based on the fact that, while the preprocessing phase and edges creation work using more than one file, as will be later explained in more detail, the actual PageRank algorithm, while working itself, proved often too computationally intensive for the resources at hand. Details on this matter are also provided later on, but for this reason the main results provided here are based on the unique file of Musical Instruments reviews.

The file is of such a size (475,22 MB) that can be preprocessed locally; while working on all the files would most probably require to distribute the data or working directly in the Spark context, we will see later that because the data actually required is based on unique codes, the preprocessing phase can work on multiple files in order to create and parallelize exclusively the list of tuples needed to run the entire MapReduce processes used to obtain the edges and run the PageRank algorithms.

After uploading the dataset by adapting proposed code found on Kaggle, I choose reduce it to the "customer_id","review_id", "product_id","product_parent","product_title" and "product_category" columns.

The dataset was originally composed of 904.765 records. After creating a new column containing the number of times each reviewer appears in the data and removing all the reviewers whose frequency is 1, 463.713 records remain. This was done because reviewers appearing only once have reviewed only one product, and thus would not form any edge between any two products; since customers IDs are used as key in the MapReduce process used to form edges, such keys would produce no output during the Reduce phase of the process. As for the products removed along with the review, if a product only appeared in one review, it would constitute an isolated node, while if it appears in other records, the one removed would not create any additional edges that would change the results of the experiments.

Afterwards, I ordered the dataset by value of the "product_parent" column and applied the "indexesprod" function in order to obtain an ordered index of products starting from 0, both to obtain a cleaner version of the edges and to obtain a first count of the nodes (number of product) by simply taking the maximum of the list of indexes and adding 1. We seed that the algorithm will be run on 78.063 products. The choice of using "product_parent" values instead of "product_id", which are both unique in the dataset, was simply based on the fact that the former are strictly numerical and the latter are of alphanumerical nature.

Finally, I applied the tuple_list function to the dataset to obtain a list of tuples ("rp") where in each touple the first element is the customer ID of the reviewer and the second element is the index of the reviewed product. As by description in previous paragraph, the function uses the "customer_id" column and the now created"indexes" column.

This file, which in such a limited setting has a size of only 4,16 MB, will be parallelized in Spark and will be used to produce edges.

2.2 Edges: Spark and MapReduce

From this point on, the process is carried out through Spark and MapReduce, to ensure that the

process could be realistically carried out on big data. While the time taken to perform an operation is dependent on the hardware on which a process is run, we can see that creating edges with such a process takes in this case less than a minute; performing the same operation with a nested loop could take hours (or days), as estimated through a function previously used to obtain edges and check for the correctness of the MapReduce job as proposed, because the number of operations to perform increases exponentially with the number of nodes.

First I parallelize the list of reviewer-product index tuples ("rp") to obtain a resilient distributed dataset; afterwards, I use a MapReduce process on the rdd, with reviewers IDs as keys and product indexes as values. By grouping by key, I find for each key (reviewer) a list of all its associated values (products), and by applying in the Reduce phase a permutation function on the values grouped for each key I create all possible tuples between the values, which correspond to the edges connecting two products reviewed by the same customer. The use of the FlatMap transformation makes sure the obtained rdd containing the list of tuples, now named "edges", is flattened (not nested) and ready to be used for the following operations.

The process outputs 3.377.442 edges to be given as initial input for the PageRank algorithms. While the file is an rdd, we can check that the it has a size of 27.43 MB.

2.3 PageRank: Theory, Spark and MapReduce

Before describing the following steps in the process, I briefly describe the theory behind the PageRank algorithm and its implementation. The problem of link analysis through the computation of the PageRank index is one of the classical settings where the number of nodes and links involved in the algorithm creates a problem of storage and RAM memory usage. The algorithm, which I will explain in a more detailed way later, simulates through a stochastic process the navigation of random surfers assigned to nodes uniformly at random, and is based on the repeated multiplication of the column wise stochastic transition matrix M of a network, where each element is

$$m_{ij} = 1/outdegreeofnodej \tag{1}$$

if the node j has a link going to node i, or 0 if j has no link to i, and the v_t vector of the distribution of probabilities of random surfers being in node i and time t. m_{ij} corresponds to the probability of a random surfer going from j to i, while the vector v, of length n corresponding to the number of nodes, has all values initially set to $1/n$ corresponding to $1/numberofnodes$.

Since we are considering the probability of random surfers being in a given node, each initial probability value will be the same, since surfers are, as we said, randomly and uniformly assigned to the nodes at the start of the process. Starting from

$$v_1 = M * v_0 \tag{2}$$

and given that the matrix M representing the structure of the network doesn't change, we repeat the process and update the vector of probabilities:

$$v_2 = M * v_1 \tag{3}$$

and in general, $v_{t+1} = M * v_t$. The mathematical properties of M which is column wise stochastic, i.e. its columns sum to 1, ensure that its principal eigenvalue λ_1 = 1, and the power method ensures the iterated multiplication of a matrix and a vector of a good number of components will converge to a vector with the same direction as its principal eigenvector, the one associated with the highest eigenvalue. So the repeated multiplication will ensure that the result converges to the principle eigenvector, and since λ_1 = 1 we are therefore sure $M * v = \lambda * v$ is equal to $M * v = 1 * v$ and $M * v = v$. The vector v_t therefore converges to v and will correspond to the vector representing the distributions of final probabilities, where the highest value corresponds to the most "popular node", or the node with the highest probability of random surfers being there at the end of the surfing. In other words, it can be seen as the node where the highest fractions of users will be when the algorithm stops. Practically, at each step, the algorithm mimics the "navigation" of users through the network: each of the surfers will start the navigation by following the path of an outgoing link chosen uniformly at random, and will continue its navigation at each step; the stochastic process will continue until convergence, and we can think that the

highest values of PageRank are assigned to nodes which are better connected, therefore reached by more surfers and more popular or important.

MapReduce solves the problem of computing the matrix-vector multiplication of a matrix of possibly millions or trillions of pages by only storing in a parallelized dataset triplets of the form (i, j, m_ij), representing two nodes with a link going from j to i, and the m_ij value as previously described. The triplets are stored only if a link between the two nodes exists and thus represent the connection matrix in a way that highly reduces the sparsity of M.

The matrix-vector multiplication is computed through a MapReduce job which operates a transformation according to they key i, mapping the triplets into the form $(i, m_ij * v_j)$, while the reduce phase sums all the values $m_ij * v_j$ computed for each key i. The result is then sorted by key and collected, and the vector v_t is updated with the new values; the procedure is repeated until the algorithm stops according to a set rule, although in general only 50 or 100 iterations are needed to obtain a good result.

The index j is necessary to retrieve the v_j component of the vector v_t. The vector v, of length corresponding to the number of nodes in the network, needs to be stored in main memory; when storing it becomes unfeasible for the RAM memory of a computer because of its size - as we said, there can be million or trillion of nodes - this issue can be solved by splitting the matrix and vector until the result becomes of such a size that can be store in main memory and the process is parallelized. The size of the vector will not constitute a problem in our case, since it is of approximately 78.000 elements for a size of 0.64 MB.

The teleport or taxation variation of the algorithm solves the problem of dead ends and spider traps in networks, which cause a "surfers leakage" and the artificial assignment of an high value of the index to nodes which are not important, because surfers are trapped in a node or loop with no way to continue navigation. The algorithm is modified by adding a variable β, which corresponds to the probability of a surfer to continue on its path, while $1 - \beta$ will correspond to the probability of each surfer being "teleported" back to the network, so that at least a portion of the "leaked" surfers go back to the navigation.

After this theoretical introduction, it's easy to describe the elements and variables created to run the PageRank algorithm, which have been adapted from the code proposed during the course:

- "total_prod", a variables representing the number of nodes in the edges list, obtained computing the maximum value of the indexes in the "edges" list and adding 1;
- "id2degree", the list of outdegrees for each node obtained by applying a countByKey operation on the "edges" rdd;
- "connection_matrix", maps "edges" into the connection matrix, i.e. the list of (i, j, m_ij) triplets as previously described;
- "page_rank", the vector of lenght "total_prod" containing the distribution of starting probabilities where each value is set to $1/numberofnodes$, which will be kept in main memory and will contain the final result of the algorithm;
- "old_page_rank", a vector of lenght "total_prod" with all values set to 1, which ensures the loop inside the algorithm is started.

While there are easier ways to find the number of nodes, the ones I found in this context all needed to refer to the original dataset - to generalize the implementation for a "real life" setting where our data could possibly be already stored as edges or as the connection matrix form in a distributed environment, I chose to use a solution that could be obtained by using exclusively the "edges" rdd.

The elements described in the previous section are used in three different functions which in fact only need to receive the "edges" rdd to perform all the steps previously illustrated. A more detailed description will be provided in the next section, but as a first introduction:

- "get_page_rank_iteration", based exclusively on a set number of iterations, was used as a first test for the functioning of the code. While the iterations are set to 50, the value can be changed for a faster test or a more accurate result;
- "get_page_rank_distance", is PageRank in its basic form, where the algorithm stops when the Euclidean distance between the vector of old values of the page rank and the updated vector is lower than a given threshold;

- "get_page_rank_taxation", is an implementation of the taxation (or teleport) variation of the algorithm;

- "l2distance", is a function computing the Euclidean distance of two vectors.

As previously noted, the core code for the algorithms and distance function has been adapted for this project from the solutions proposed during the curse and tutoring lessons. The functions could also be modified to receive the parallelized tuples list as an rdd and perform the task of creating the edges inside the function itself.

3 PageRank: experiments and results

The practical aim of this project is to build a PageRank index of products by connecting them through a link if they were reviewed by the same reviewer at least once. We can hypothesize that more popular products will be bought more frequently, and thus will more often be reviewed, and have an higher chance of being reviewed by a customer who has already left a review for another product. In other words, in such a setting, the more popular a product, the more it's bought and reviewed on, and if the chance of having links connecting it to other reviewers is higher, the more connected it will be in the network, increasing its PageRank value, which will constitute an index of its importance. It is reasonable to think that the PageRank value will be higher for common or well known products in the category, for generic products or items that are frequently bought with other ones.

In the next paragraphs, I comment the results of three different variations of the algorithm. While the order of the values may change, all three algorithms detect the same ten products as the ones with highest rank.

3.1 PageRank: Iterations

The first experiment has been run mainly as a check for the functioning of the algorithm with the elements created in the project, by setting a set number of 50 iterations inside the function. As previously stated, this value can be changed for a faster result or a more accurate one; this value has been chosen based on the knowledge that in general 50-100 iterations are needed to obtain a satisfying result. The core code of the function follows the procedure as described in the previous paragraph with no stopping rule. The highest rank is of 0.00297, closely followed by the second value of 0.00273 and the third of 0.00256. Remembering that data on musical instruments was used, it can be noticed that the values of the three highest values correspond to products connected to very popular products in the musical field and of very common use:

- index 26012 corresponds to pedals for electric guitars acoustic effects of the Joyo brand;

- index 15398 corresponds to the SN-1 tuner for guitars of the Snark brand;

- index 34129 corresponds to the Ernie Ball Slinky Nickel, which is the name of a brand and type of electric guitar strings.

The next two products are also related to guitar playing, in particular a string cutter and another kind of guitar strings; the guitar is in fact a notoriously famous instrument, commonly played for a hobby outside the professional field by an high number of people. They are in fact popular products, and it is also reasonable that they could be frequently bought along with other useful musical products and perhaps reviewed in the context of the same purchase, so that the reviewer creates a link between two products. While this is true, the values of the index are still low; no product seems to be particularly central to the network.

3.2 PageRank: Euclidean Distance

In order to introduce a stopping rule in the algorithm, it is now modified by computing the distance between the updated and old value of the page rank after each iteration of the MapReduce process; the algorithm stops when the distance is lower than a set value. In order for the algorithm to perform a reasonable - but still quite low - number of iterations, the tolerance has been lowered to 10^{-8}.

The algorithm stops after 21 iterations, with similar results to the previous algorithm: the first product is 26012 with rank of 0.00299, the second is product 15398 with index value of 0.00274, and the third is product 34129 with PageRank value of 0.00257. As we see, the product order is the same and the index values are almost the same.

3.3 PageRank: Taxation Variation

While the way the edges file was build excludes the existence of dead ends or spider traps (all the edges exist in both directions, so an edge can always have at least one path to follow back or to exit a loop) we cannot exclude the existence of isolated subcomponents, so I chose to also run the taxation variation of the algorithm. However, this needed the value of the tolerance for the distance to be lowered to 10^{-10}, and even then the algorithm stops after 16 iterations.

The variation works by assigning a value to the β probability of a surfer continuing on its path - in our case, $\beta = 0.85$ - so that the probability of being teleported back to the network is $1 - \beta$, and modifying the new values after each iteration, before updating the old vectors, translating the following formula:

$$v_{t+1} = \beta * M * v_t + (1 - \beta) * 1/n * \underline{1} \qquad (4)$$

as needed inside the MapReduce job, by applying it component by component after the reduce and collection phase. In the general formula, $\underline{1}$ is a vector of ones of size n, and n is the size of the vector v, i.e. the number of components.

While the ten highest values are the same as in the previous cases, the ordering of the first elements is changed, suggesting there could indeed be one or more isolate components, with the first and second value switched with respect to the previuos results; however, the first product, with index 15398, now has rank of 0.00217, and the second product, 26012, has a very similar value of 0.00212.

This again suggests that no product is particularly central to the network, and that this version of the algorithm would need a very low value of tolerance to perform a satisfactory number of iterations, but this could contradict the aim of stopping the algorithm at convergence.

3.4 Additional Results

When checking the complete list of the ten highest ranking products, we see that all 10 items are actually related to the field of guitar playing, as of the remaining not described until now, other two are Snark tuners, one is an unnamed Snark product, one is a guitar stand, and the last code represents guitar strings. Once again, this is reasonable: musical instruments are costly and personal items, and most professional musicians refer to specialized stores, digital or phisical, to buy products for their instruments, because of specific needs or preferences. Guitar playing is instead a very common hobby for all ages, and professional and semi-professional products have become commodified, mass produced and sold outside specialized stores. It is indeed reasonable that a network built connecting products reviewed by the same customers would be such that products of this subgroup would have high importance and an high index.

3.5 Popular Items: a check

Lastly, I briefly comment on a last part of the project; here I computed for each product the number of times it appeared in the dataset already filtered out of reviewers that appeared only once, to see if highest values of ranks correspond to the most reviewed products. This hypothesis is only partly confirmed; of the 5 products with highest value of the index, only two are conversely in the list of the 5 most reviewed products: the Snark Tuner, which is the second highest ranking product according to the algorithm computed with no taxation and the highest ranking with taxation, and the guitar strings with fifth highest value of PageRank (index 58553). However, all the 5 products most frequently reviewed by at least two people do in fact appear in the list of the 10 highest ranked items by PageRank index.

4 Scalability and Issues

While the project results have been presented working on a dataset obtained from a single file, the main steps illustrated here have also been applied on a larger portion of the dataset to make sure it could scale up to a larger setting. While an alternative solution would be to directly upload the data in a SparkContext, for these experiments it is still possible to work on the data locally, and it is to be noticed that the process to create the dataframe from the original files, adapted from Kaggle and used in the notebooks, solves the problem of bad lines that need to be dropped if the file is automatically uploaded, thus losing some of the records.

The experiments have been replicated by simply joining the Musical Instruments file and the Digital Music file, since I deemed appropriate to work on

a subsets of products in the same macrocategory. A specific characteristic of the data ensures that every subsequent step in the process can simply be repeated by following the process as described: the values for the product_parent and customer_id are unique over the entire dataset, so there is no risk of finding more than one reviewer or product associated to the same IDs. Once the files have been joined, filtered by frequency after the merging, reindexed and once the two needed columns have been retrieved, provided the ability to run the algorithm in an actually distributed computational environment, the algorithm can be replicated on big data.

In fact, the choice of presenting the results for a single file is a result of the computational problem given by limited resources; while all experiments have by been run initializing a SparkContext and creating a Spark object in Google Colab, the resources provided by the free version of Colab made it so that, while all the process worked on the joined files, the algorithm could run for a very limited time before incurring in RAM memory errors and stopping. Because of this reason, as to avoid presenting unreliable results for the algorithm itself, I preferred using the Musical Instruments data exclusively. Once again, while the time needed to perform an operation depends on the machine used for the computation, it is to be noticed that on such limited resources the time to compute even as little as 5 iterations highly increased.

Moreover, as seen before, the utilization of a MapReduce job for the creation of edges largely reduces the complexity of this task, and the utilization of a MapReduce process to implement the process makes it suitable for big data replication. Lastly, as previously stated, the issue of the size of the vector v can be addressed by splitting the matrix and the vector until the results are reduced to a size such that can be stored in main memory.

5 Conclusions

The presented work had the aim of creating an index of products inside of a dataset of reviews; on a merely technical level, the goal has been reached, as all three versions of the PageRank algorithm work on the data as processed from the original file. An index of products has been obtained, with results that, even without a deep knowledge of the musical field, can been seen as having basis in reality: the highest ranking values pertain to the subgroup of guitar products, the instrument most commonly played outside of the professional music field.

The techniques used in the project make it so that they can scale up to larger dataset; however, there are aspects that could be improved, such as the fact that the algorithms stop after very few iterations and the PageRank values obtained are low.

This could be due to the nature of the data, where no node is particularly well connected with respect to the others, and the category of musical instruments contains an high number of products unrelated to each other; people rarely play more than one instrument, so it is reasonable to assume not many people often review products for more than one of them, and as a consequence edges will tend to connect products related to the same instrument. However, a possible modification of the algorithm that could impact the results could be implemented by finding a way to weight the edges to take into account those which may appear more than once, to signal those products that are more frequently reviewed together.

Finally, as a curiosity, I report that while the results when using both the Musical Instrument and Digital Music files, not present in the corpus of the project, are limited to the output of 5 iterations to avoid RAM memory errors, I found that the second and third highest ranked products were once again the Snark SN-1 Tuner with an extremely low value of 0.00048 of the index and the Joyo Pedals with a value of 0.00045. The highest ranked product, with and index of 0.00065, corresponded to the digital purchase of the song “Happy” From the soundtrack of the“Despicable Me 2” Oscar nominated kids movie; once again, this makes sense as a starting result. The data stops at 2015, and the movie, released in June 2013, was extremely successful and the song became a world hit, because of the combined popularity of the actors in the movie, of the cartoon itself, of the mainstream popularity in pop culture of Minions, monster characters in the movie, and the fact that the song is by Pharrell Williams, an extremely famous singer in the R&B and Rap USA scene, so that the song was not only purchased by parents for their kids but by the general public, making it for sure one of the most purchase songs of that year and the next.