Link Analysis methods applied in ranking IMDb movies

Silvia Mazzoleni

969789

1 Introduction

This report aims to implement a method to rank movies contained in Internet Movie Database (IMDb) using approaches based on Link Analysis. The dataset is one of the biggest archive for movies and other similar TV contents.

The graph the PageRank methods are implemented over has movies as nodes that are connected with bidirectional edge if the two films have an actor or actress in common.

During the analysis, I compared four different methods for the computation of movies rank. I considered implementations through two libraries, `GraphFrame` and `NetworkX`, and then I implemented the method from scratch implemented in two different ways.

Finally I used a scalable improvedmethod, starting from my form scratch methods. This last method is the most time consuming but is scalable and is comparable with all the other methods. The two methods implemented by the library are not scalable but results are comparable with the two implemented with the "from scratch" method, that are more time consuming and not scalable.

2 Dataset description

The dataset used for the analysis was downloaded from *kaggle.com*[1]. It contains all the information about movies, shorts and TV series and it is updated weekly.

The dataset is structured in different `.tsv` files that contains all the information about movies.

For the purpose of the analysis I considered only two files.

2.1 `title.basics`

The first one is the `title.basics.tsv` file that contains the main information about movies. Variables contained in the file are shown below:

- `tconst`: it is an alphanumeric unique identifier for the movie;
- `titleType`: it is the type/format of the object (e.g. movie, short, tvseries, tvepisode, video, etc);
- `primaryTitle`: it is the more popular title or the title used by the filmmakers on in promotional materials at the point of release of the movie;
- `originalTitle`: it is the original title, in the original language;
- `isAdult`: it is a boolean encoding if the movie is for adults (1) or not (0);
- `startYear`: it represents the release year of a title. In the case of TV Series, it is the series start year. It is in the form `YYYY`;
- `endYear`: it is the TV Series end year. For all other types of contents it is `NULL`;
- `runtimeMinutes` it is an integer for the run-time of the movie in minutes;
- `genres`: it is an array of strings with up to three genres associated with the title.

[1]https://www.kaggle.com/datasets/ashirwadsangwan/imdb-dataset

The `title.basics` dataset has 9372142 rows of which 627109 movies, that is the 6,7% of the dataset.

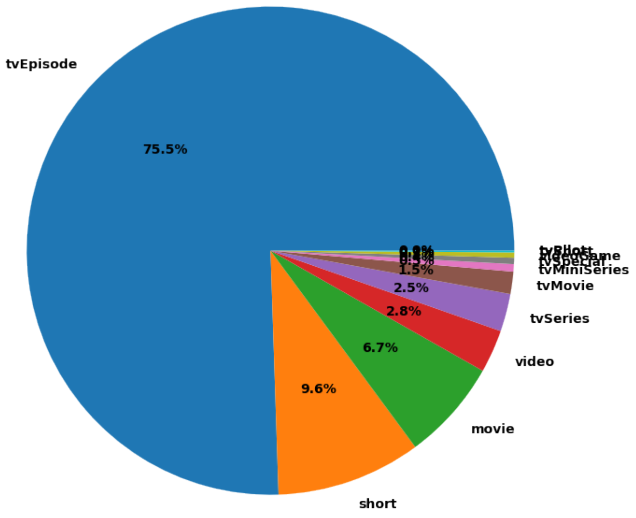


Figure 1: Distribution of `titleType` in `title.basics` dataset.

2.2 `title.principals`

The second `tsv` file used for the study was the `title.principals.tvs`. It contains the information about the cast and the crew of all movies.

It contains the following columns:

- `tconst`: it is an alphanumeric unique identifier for the movie;
- `ordering`; it is a number to uniquely identify rows for a given title Id (a variable contained in another file in the dataset);
- `nconst`; it is an alphanumeric unique identifier of the person the row is about;
- `category`; it is the category of job that person was in;
- `job`; it is the specific job title if applicable, otherwise `NULL`;
- `characters` (string) - the name of the character played if applicable, otherwise `NULL`.

The `title.principals` dataset has 53049811 rows of which 11720940 actors (22, 1% of the whole dataset), 9037758 actresses (17, 1%) and 9220031 actors or actresses acting as themselves (17, 4%).

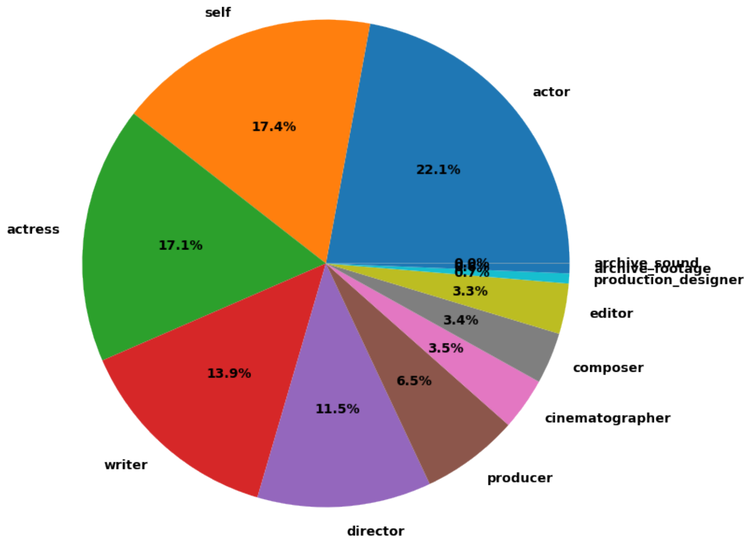


Figure 2: Distribution of `caterory` in `title.principals` dataset.

3 Preprocesing

The dataset was already usable in the analysis so there was no need to clean data, since the datasets do not contain raw data but well structured data.

As already seen, not all the information in the dataset is considered in this report since not all the variables are fundamental for the purpose of the analysis.

From `title.baiscs` dataset I used only the two columns `tconst` and `titleType`. I filtered the dataset on `titleType` to consider only movies during the analysis. I used `tconst` to join the two datasets and obtaining a single file with information about each element of the cast stored in file `title.principal` with the film in which they act, stored in `title.basics` dataset.

Regarding the dataset `title.principal`, I used only variables `tcount`, `nconst` and `category`. I filtered it over the variable `job`, considering only `actor`, `actress` and `self`. All the other people that worked at the movie are not needed for the purpose of the analysis.

At the beginning of the analysis, I filtered movies also on the base their genres. To start from a more manageable dataset I considered only movies that contains it the variable `genres` only the category *Drama* and *Romance*. In this way, the dataset reduced to 9500 movies that are 1% of the movies contained in the dataset, and the total number of edges between nodes are 153350.

3.1 Description of preprocessing implementation

Preprocessing consisted in joining the two datasets through an `inner join` command using variable `tcount`. It permits to obtain a new dataset containing all and only the useful information for the purpose of the report, that are `tconst`, `nconst`, `category` and `genres`. Then I filtered over *actors*, *actress* and *self* for the `category` variable and over *Drama,Romance* for `genres` variable.

After the join action with the SQL command I built a dataset with two columns containing all the couples of films with a common actor or actress. For each couple of movie there are two rows, one with the two movies and another with the same two movies but in the opposite order, because each row represent an edge and for each couple of two films with a common actor there will be two edges in both the directions.

In order to simplify methods implementation, once the dataset was reduced, I modified the content of the variable `tconst` containing the id of each movie so that the values ranged from 0 to the number of movies in the dataset minus one (9499). In this way the new movie ids are equal to the vector or matrix indexes used in the function implementing from scratch methods.

Starting from this new RDD dataset it has been easy to implement methods chosen for the analysis.

4 Algorithms and implementation

4.1 A Theoretical introduction to PageRank methods

During the analysis I used different PageRank methods. The fundamental elements of these methods are the vector $\mathbf{p}$ with pagerank values, the transition matrix M and the taxation factor β.

The transition matrix M is a matrix with dimension nxn with N number of nodes, that are the movies in the dataset. Each movie is associated to an index to identify its row and column in the transition matrix. Each entry m_{jk} of matrix M represent the probability to go from node k to node j and it is computed as 1 over the number of outgoing edges from node k.

The PageRank of each node is computed considering a random surfer moving between nodes following the edges path and taking the probability to be in that node after an high number of steps of the surfer.

We start from a vector $\mathbf{p}$ of N equal entries of value $1/N$. It represent the starting probability to be in a given node. We multiply the matrix M and vector $\mathbf{p}$ in each step to obtain a new vector that contain the new probability of the surfer to be in each node.

For an increasing number of steps, the methods converge to the vector $\mathbf{p}$ containing the PageRank of each movie.

To avoid the problem of spider traps and dead ends, we add a taxation, so we consider a factor β usually between $0,8$ and $0,9$, such that $1-\beta$ represents the probability for the surfer to be teleported in another random node instead of following the path drew by edges. We call $1-\beta$ dumping factor.

In the equation below is reported the PageRank function of a generic node i.

$$PageRank(i) = (1 - \beta) + \beta \cdot \sum_{j \in Nodes} PageRank(j) \cdot m_{ij} \tag{1}$$

This method permits to obtain a PageRank value for each node. The sum of all the pagerank values is the number of the nodes N. It is possible to normalize the PageRank sum to 1 modifying the equation as follows.

$$PageRank(i) = \frac{(1 - \beta)}{N} + \beta \cdot \sum_{j \in Nodes} PageRank(j) \cdot m_{ji} \tag{2}$$

Using both the equations, all pagerank values are always greater than 0.

In the implementation of PageRank methods it is important to consider two parameters to monitor the convergence: the number of iterations and the tolerance.

During all the analysis I always considered the parameter $\beta = 0,85$ because it is the most generally used value in the literature.

In the following section, I present the three method I used for the analysis. The firsts two methods are implemented using two different packages: `GraphFrame` and `NetworkX`, while the third one is implemented from scratch.

4.2 GraphFrame method

As first method I implemented the Equation 1 through `pagerank` function in `GraphFrame` library.

The `pagerank` algorithm runs on a graph and has as default dumping factor $0,15$. I could run the algorithm for a fixed number of iteration using the variable `maxIter` or run it until the tolerance reaches a given value `tol`. Using the tolerance allows to control the convergence of the method in a more direct way.

To define the graph over which the algorithm run two dataframes are needed. First, I defined a `Row` object to create a dataframe composed by only one variable and containing all the nodes in the graph. The second dataframe contains the edges of the graph, so I only needed to convert the already existing RDD dataset in a dataframe where variable names are fixed: `src` (i.e. source) is the variable containing all the starting node of each edge, `dst` (i.e. destination) is the variable containing the destination node of the edge. With the two dataframe containing nodes and edges, it was possible to implement the `GraphFrame` method.

The algorithm returns a new dataframe containing nodes and pagerank values.

4.3 NetwokX method

With the second method I implemented the Equation 2 through `pagerank` function in `NetworkX` library.

The algorithm runs over a `NetworkX` graph that I obtained starting from the same dataframe used for the implementation of pagerank through library `GraphFrame` and containing all edges.

As before, I needed to choose how to control the convergence of the method. I could fix the number of iteration with variable `max_iter` or the tolerance to reach with variable `tol`. Again I set the tolerance.

Also in this case, the default value of the taxation factor is set to $0,85$.

This method returns a dictionary with nodes as keys and pagerank as values.

4.4 Implementation from scratch

This method could implement both the PageRank equations seen before from scratch.

This method also permits to select when to stop implementation due to a maximum value of iteration knowing the reached tolerance or to stop the process once a level of tolerance was reached also returning the number of iteration used in the process.

In Appendix A are reported the functions used for the implementations of the methods and how they works. In Appendix C are reported the improved functions that are scalable. Both the couple of functions have the same functioning and principle behind them.

5 Analysis

In the analysis I implemented each one of the methods showed before for a different level of tolerance. I chose to use a fixed tolerance instead of a fixed number of iterations to control the level of convergence of each method.

After calculating the pagerank with different levels of decreasing tolerance, I compared each result with that obtained with one level of tolerance less, to verify the level of convergence. If the pagerank values converge, then the similarity between the two goes to zero. In Appendix A.2 I report the function I used to calculate the similarity and how it works. The similarity function take as inputs the top q nodes with higher pagerank values obtained with the same method and two different levels of tolerance and checks if each node is ranked the same way in the two lists with a tolerance of 2 positions. In this way I check if the nodes reached they final position in the ranking or the method is still converging to final pagerank vector.

I looked at different levels of tolerance if the methods implementing Equation 1 or 2. In fact, pagerank values have different order of magnitude in the two cases: if all pageranks sum to 1, their greatest values are around 10^{-4}, if instead all pageranks sum to N, greatest pageranks are between 2 and 3. So the same level of convergence was reached at different level of tolerance.

Each method was implemented for each level of tolerance. Then I computed the similarity as explained before.

In the following table I reported the used levels of tolerance for the two different Equations.

Eq 1	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}				
Eq 2	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}	10^{-10}	10^{-11}	10^{-12}	10^{-13}	10^{-14}

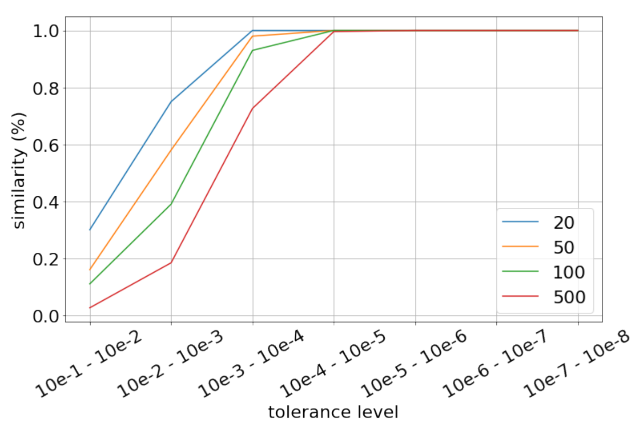
Table 1

For `GraphFrame` method and from scratch method that sum to N the tolerance goes from 10^{-3} to 10^{-14}, while for `NetworkX` method and the from scratch method that sum to 1 goes from 10^{-1} to 10^{-8}.

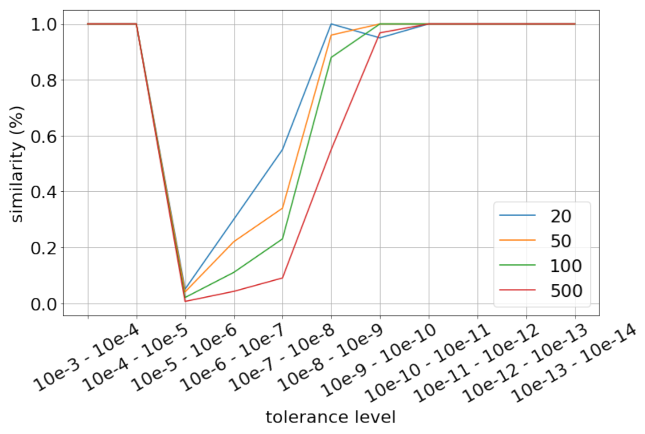
6 Results

In this section I report the result of the analysis.[2] Adding an order of magnitude to the tolerance at any iteration of the method and comparing the firsts 20, 50, 100 and 500 nodes with highest pagerank at each iteration, I computed the similarity of each couple of results.

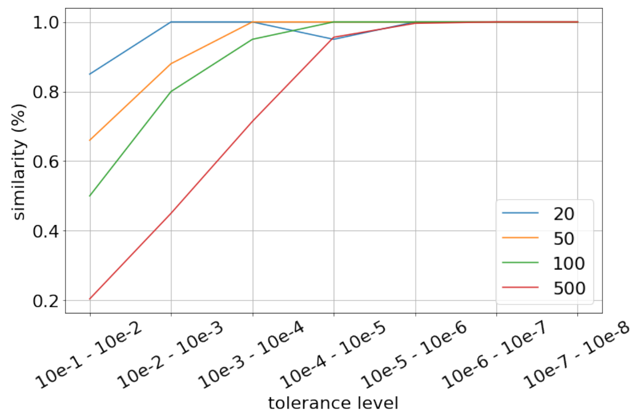
For each of the four methods I reported the graphs of the similarity of the firsts 20, 50, 100 and 500 nodes for decreasing value of tolerance.



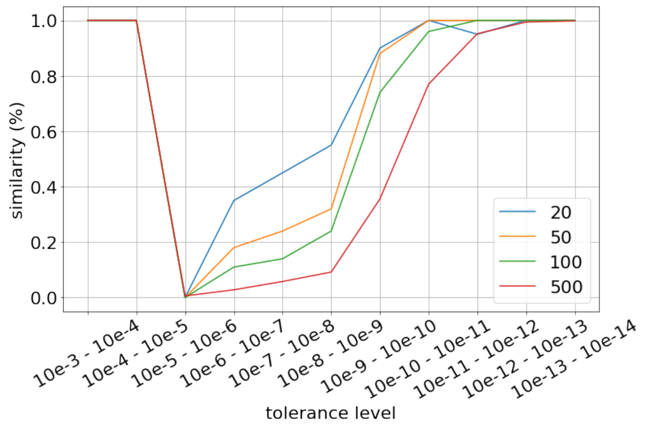
(a) GraphFrame method



(b) NetworkX method



(c) Method from scratch normalized to N



(d) Method from scratch normalized to 1

Figure 3: Similarity between decreasing levels of tolerance for each method and for different numbers of nodes.

Decreasing the tolerance, the methods converged to the final pagerank values and the ranking of movies did not change anymore. It is easy to notice that looking to the similarity of more nodes,

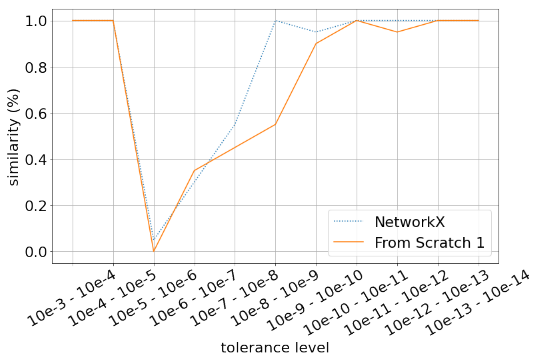
[2]All the tables with the values used for the following graphs are reported in Appendix B.

a lowe tolerance is needed to converge with respect to the case of only 20 nodes. This is true for each method.

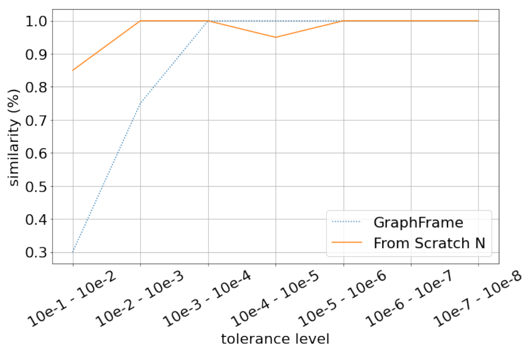
Methods implemented through libraries in general need an order of magnitude less to reach convergence with respect to those those methods implemented from scratch.

Looking at the methods using Equation 2, their pagerank values are of the order of 10^{-4}. For this reason I need at least tolerance of 10^{-5} to start to have an improvement in the result of the algorithm. In fact for higher levels of tolerance the similarity is 1 since the methods do not improve the pagerank values, ranking all nodes with the same pagerank.

In the following graphs I compared in a more specific way `NetworkX` method with the method implemented from scratch with Equation 2 looking at 20, 50, 100 or 500 nodes. I do the same for the `GraphFrame` method with the method implemented from scratch with Equation 1.

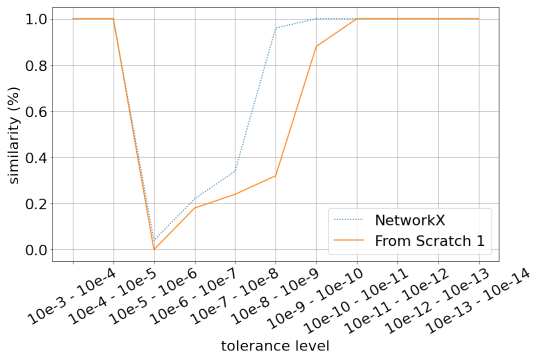


(a)

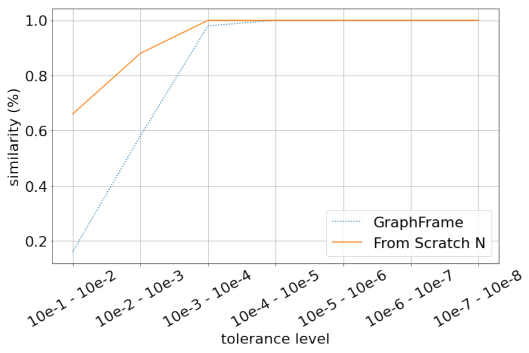


(b)

Figure 4: Similarity of firsts 20 nodes with `NetworkX` method and the method from scratch that sum to 1 in Figure 4a, and with `GraphFrame` method and the method from scratch that sum to N in Figure 4b.

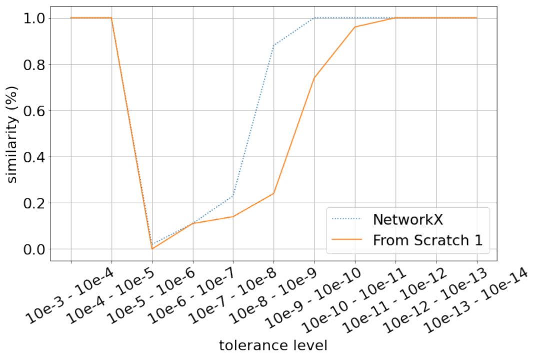


(a)

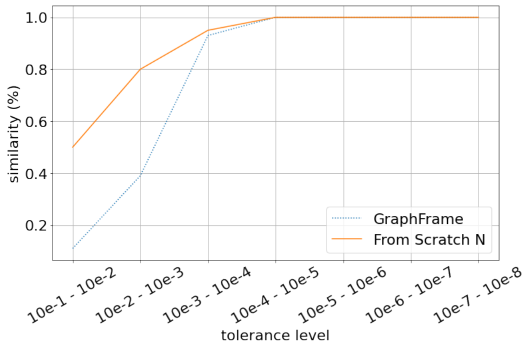


(b)

Figure 5: Similarity of firsts 50 nodes with `NetworkX` method and the method from scratch that sum to 1 in Figure 5a, and with `GraphFrame` method and the method from scratch that sum to N in Figure 5b.

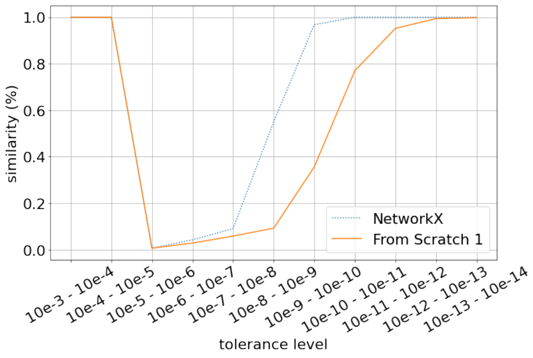


(a)

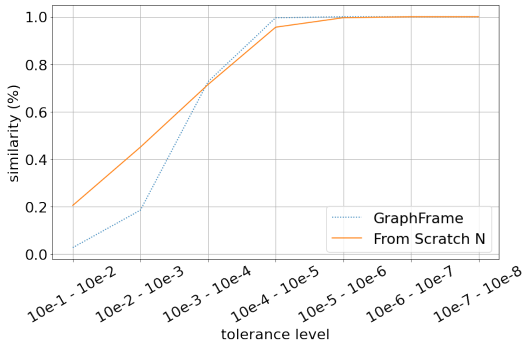


(b)

Figure 6: Similarity of firsts 100 nodes with `NetworkX` method and the method from scratch that sum to 1 in Figure 6a, and with `GraphFrame` method and the method from scratch that sum to N in Figure 6b.



(a)



(b)

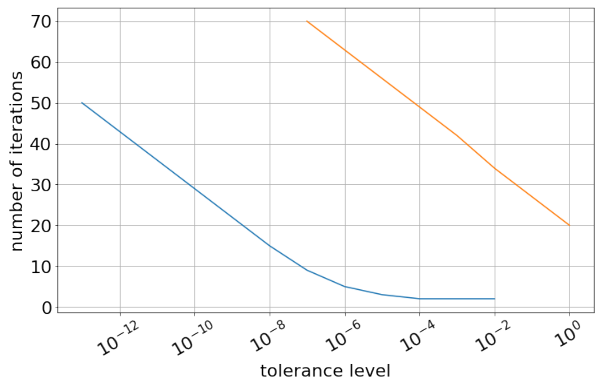
Figure 7: Similarity of firsts 500 nodes with `NetworkX` method and the method from scratch that sum to 1 in Figure 7a, and with `GraphFrame` method and the method from scratch that sum to N in Figure 7b.

It is immediate to see that the `NetworkX` method and the one implemented from scratch that sum up to 1 have the same exactly shape put with a lag for the scratch method that always converge with a tolerance of an order of magnitude less.

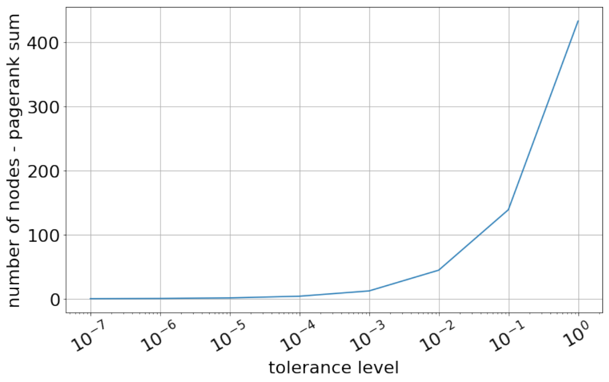
Looking at the comparison between the `GraphFrame` method and the method from scratch that sum up to N, the graphs show that in this case the best method is the one implemented from scratch, since the two method converge simultaneously but it shows higher similarity values at same tolerance levels.

At the end, I compared the two methods implemented from scratch. Since it is possible to obtain the number of iterations used for a fixed level of tolerance, I report the graph that compares this information for the two method in Figure 8a.

I also checked if all pagerank values sum to 1 or N for each level of tolerance. For all the methods implemented through libraries it is true at each level of tolerance. This is also true for the method implemented form scratch that had to sum to 1. While, the sum of all pagerank values of the method from scratch that implement Equation 1, converge to N with decreasing tolerance. It is also the more time expensive method. As a result of the analysis, I assumed that this method is less accurate and needs better accuracy to converge, but still performs well using the correct tolerance. In Figure 8b I reported the number of nodes N minus the sum of all pagerank values obtained at different levels of tolerance with the method implemented from scratch following Equation 1.



(a) Number of iteration for each level of tolerance. The blue line is for the method that sum to 1, while the orange one is for the method that sum to N.



(b) Number of nodes N = 9500 minus the sum of all pagerank values for each level of tolerane.

Figure 8

7 Scalable solutions

The `GraphFrame` and `NetworkX` methods are not scalable for big data solutions since these methods run over dataframe objects instead of RDD datasets.

The method implemented from scratch used RDD object and was computed through map-reduce functions and without using objects stored in the main memory during the run-time. Only the part of the function computing the distance between two vectors containing all the pagerank values computed at each iteration are stored as `dict` objects saved in local main memory.

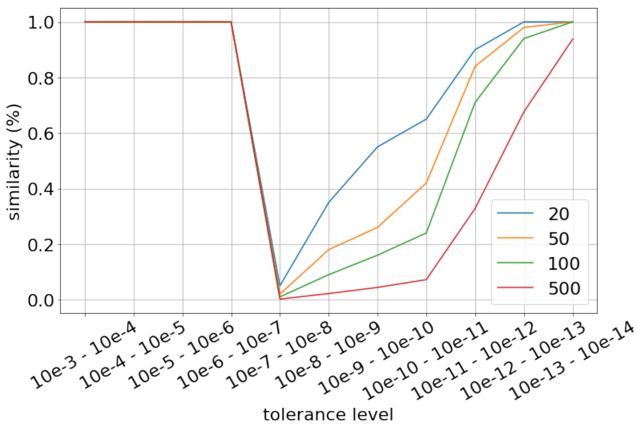
However, since the method is built only using map-reduce function,it runs also with bigger datasets.

Due the increasing time computing I made a small analysis only with the scratch method implementing Equation 2. I used a new dataset contains all the movies that has *Drama* as one of their genres. The number of nodes grows to 183623, and the number of edges to 11336516.

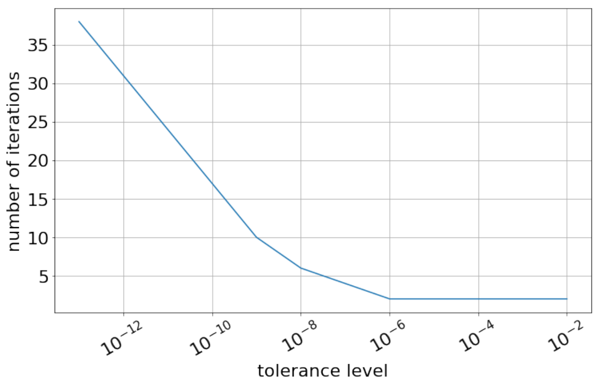
Figure 9a report the increasing similarity of implementations for decreasing values of tolerance and different numbers on nodes. It shows that the similarity trend is equal that the one obtained using the smaller dataset. To star to improve pagerank values and to reach similarity 1 now tolerance needs to be smaller of at least two order of magnitude due to the higher number of nodes.

The number of iteration also has the same increasing trend for decreasing tolerance as reported inf Figure 9b and the sum of all pagerank values minus 1 is of the order of magnitude of 10^{-13}, so the method continues to perform well even for larger datasets.

Comparing the number of iteration used to reach a given similarity regardless the tolerance, they are the same using datasets 20 times bigger.



(a)



(b) Number of iteration for each level of tolerance.

Figure 9

7.1 The scalable solution

In this paragraph I discussed the result obtained using the last method, the scalable one.

Since this method was the most time consuming, I implemented it only for the Equation 1 and for tollerance from 10^{-3} to 10^{-10}.

The sum of all pagerank values is 1 with an error of the order of 10^{-16}, much higher than all other methods. The number of iteration needed to reach a given level of tolerance is exactly equal to the number of iteration needed from the from scratch method normalized to 1. Also the twenty highest ranked movies at level of tolerance 10^{-10} are the same for the two methods. Even if I have not been able to calculate the pagerank values for minor tolerances, I can affirm for the similarity found with the previous method from scratch, that the best 20 nodes evaluated at tolerance level 10^{-10} are those that would be obtained at convergence.

In Figure 10 I reported the similarity of the first 20, 50, 100 and 500 nodes for decreasing values of tolerance.

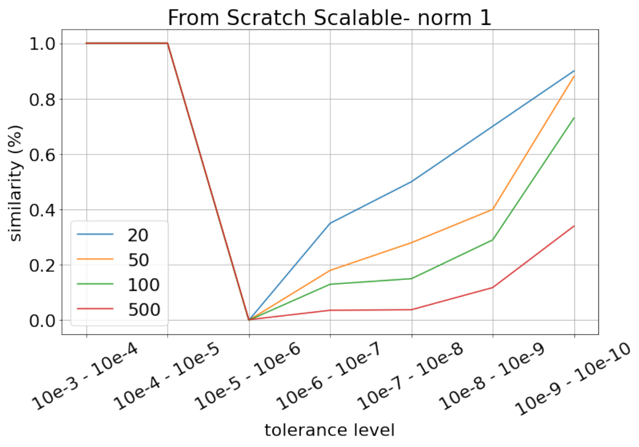


Figure 10

8 Conclusions

In this report I compared four different PageRank methods applied on a movie dataset from IMDb. The first two methods used `GraphFrame` and `NetworkX` libraries and are not scalable. The third one is computed from scratch and permitted to implement Equation 1 or 2 and is comparable respectively with the other two methods.

The methods implemented from scratch are not completely scalable but are more usable also with bigger datasets with respect the other two methods.

All the methods are comparable and reach to the same exact results with a comparable level of tolerance.

Finally the last method implemented is the scalable one. It was implemented form scratch following the same principles of the other from scratch method and has the same performance.

As future implementation would be possible to modify the functions implementing the scalable method to optimize it and mke it less time expensive. In this way it will also be possible to implement a better and more agile comparison with all other methods.

A Appendix A

A.1 From scratch method implementation

The `compute_page_rank_n` function has as inputs the matrix `M` and the number of nodes `nodes_count`. Other inputs are fixed as default but could be modified: the taxation factor `tax`, the maximum number of iteration `max_iter` and the tolerance. Setting `nodes_count` to 1 or N it is possible to implement the method uding Equation 1 or 2.

The function computes the pagerank values and then computes the values with those computed at the previous step with function `compute_distance`.

The function continues in the iteration of the computation until it reached the maximum number of iterations or the fixed tolerance.

The function returns a `dict` object containing all the nodes as keys and their respective pagerank values as values. It also returns the number of iterations needed.

Function `compute_distance` has as inputs two `dict` object with the nodes as key and pagerank as values. The output of the function is the L_1 distance between the two vectors containing pagerank obtained from `dict` objects.

```
def compute_page_rank_n(M, nodes_count,
                         tax: float = 0.85,
                         max_iter: int = 1000000, tolerance = 10e-7):

    MT = M.map(lambda x: (x[0], x[1], tax * x[2] ))

    vct = dict(MT.map(lambda x: (x[0],1/num_nodes)).collect())
    old_vct = dict(MT.map(lambda x: (x[0],50)).collect())
    i=1
    while i <= max_iter and compute_distance(old_vct, vct) >= tolerance:
        new_vct = MT.map(lambda x:(x[1],(x[2]*vct[x[0]])))\
                  .reduceByKey(lambda x,y: x+y)
        old_vct = vct
        vct = dict(new_vct.map(lambda x: (x[0],
                              x[1]+(1-tax)/nodes_count)).collect())
        i+=1
    print(tolerance, i, compute_distance(old_vct, vct))
    return vct, i

def compute_distance(old_vct,new_vct):
    old_vct = list(old_vct.values())
    new_vct = list(new_vct.values())
    result = sum([(a - b)**2 for a, b in zip(old_vct, new_vct)])
    return result
```

A.2 Similarity

The `similarity` function takes as input two vector with both q entries and computes the number of elements that are in the same position in the two vectors with a tolerance of `threshold` positions.

It returns the number of elements in the same position, `total`, over the total number of entries q, and also the pure number `total`.

This ratio has values from 0 to 1. It can be seen as the percentage of similar ranked nodes in the two vector.

```
def compute_similarity(a, b, threshold: int = 4):
    total = 0
    for item in a:
        if item in b:
            for i in range(0,threshold):
                if a.index(item) == abs((i - b.index(item))) or
                                            a.index(item) == (i + b.index(item)):
                    total +=1
                    break
    return total/len(a), total
```

Appendix B

Tables 2 and 3 report all the similarity levels obtained comparing different levels of tolerance and with different number of nodes. The two Tables contain methods coupled if they implemented Equation 1 or 2.

	GraphFrame				Scratch normalized to N			
	20	50	100	500	20	50	100	500
10^{-1} - 10^{-2}	0,3	0,16	0,11	0,026	0,85	0,66	0,5	0,204
10^{-2} - 10^{-3}	0,75	0,58	0,39	0,184	1	0,88	0,8	0,45
10^{-3} - 10^{-4}	1	0,98	0,93	0,726	1	1	0,95	0,714
10^{-4} - 10^{-5}	1	1	1	0,996	0,95	1	1	0,956
10^{-5} - 10^{-6}	1	1	1	1	1	1	1	0,996
10^{-6} - 10^{-7}	1	1	1	1	1	1	1	1
10^{-7} - 10^{-8}	1	1	1	1	1	1	1	1

Table 2

	NetworkX				Scratch normalized to 1			
	20	50	100	500	20	50	100	500
10^{-1} - 10^{-2}	1	1	1	1	1	1	1	1
10^{-2} - 10^{-3}	1	1	1	1	1	1	1	1
10^{-3} - 10^{-4}	0,05	0,04	0,02	0,006	0	0	0	0,006
10^{-4} - 10^{-5}	0,3	0,22	0,11	0,042	0,35	0,18	0,11	0,028
10^{-5} - 10^{-6}	0,55	0,34	0,23	0,09	0,45	0,24	0,14	0,058
10^{-6} - 10^{-7}	1	0,96	0,88	0,55	0,55	0,32	0,24	0,092
10^{-7} - 10^{-8}	0,95	1	1	0,968	0,9	0,88	0,74	0,356
10^{-8} - 10^{-9}	1	1	1	1	1	1	0,96	0,77
10^{-9} - 10^{-10}	1	1	1	1	0,95	1	1	0,952
10^{-10} - 10^{-11}	1	1	1	1	1	1	1	0,994
10^{-11} - 10^{-12}	1	1	1	1	1	1	1	0,998

Table 3

In Table 4 are reported all the number of iteration used by the from scratch methods for the implementation for a fixed level of tolerance. In the Table are reported the number of the iterations for the two methods considering the initial, and smaller, dataset, the iterations for the from scratch method that sum up to 1 using the bigger dataset and the iterations needed using the scalable from scratch method.

	Scratch normalized to N	Scratch normalized to 1	Scratch normalized to 1 (big dataset)	Scratch normalized to 1 (scalable)
10^{-1}	20	-	-	-
10^{-2}	27	-	-	-
10^{-3}	34	2	2	2
10^{-4}	42	2	2	2
10^{-5}	49	2	2	2
10^{-6}	56	3	2	3
10^{-7}	63	5	2	5
10^{-8}	70	9	4	9
10^{-9}	-	15	6	15
10^{-10}	-	22	10	22
10^{-11}	-	29	17	-
10^{-12}	-	36	24	-
10^{-13}	-	43	31	-
10^{-14}	-	50	38	-

Table 4

In Table 5 are reported the values for each level of tolerance of N, number of nodes, minus the sum of all the pagerank values obtained with the from scratch method that sum up to N.

	Sum of PageRank - N
10^{-1}	433,149
10^{-2}	138,857
10^{-3}	44,514
10^{-4}	12,129
10^{-5}	3,888
10^{-6}	1,246
10^{-7}	0,399
10^{-8}	0,128

Table 5

Table 6 reports the 20 movies best ranked and the pagerank values for each method. These values are obtained using tolerance 10^{8} for `GraphFrame` method and the method from scratch that sum up to N, and tolerance 10^{-14} for `NetworkX` method and the method from scratch that sum up to 1.

movie index	GraphFrame PageRank	Scratch normalized to N PageRank	NetworkX PageRank	Scratch normalized to 1 PageRank
3341	3.425	3.425	0.000360	0.000360
4979	3.425	3.425	0.000341	0.000341
1672	3.093	3.093	0.000325	0.000325
3648	2.996	2.996	0.000315	0.000315
7103	2.933	2.933	0.000308	0.000308
7827	2.924	2.924	0.000307	0.000307
7553	2.888	2.888	0.000304	0.000304
2878	2.863	2863.	0.000301	0.000301
1419	2.808	2.808	0.000295	0.000295
1008	2.768	2.768	0.000291	0.000291
7411	2.730	2.730	0.000287	0.000287
4419	2.707	2.707	0.000284	0.000284
8612	2.702	2.702	0.000284	0.000284
4644	2.698	2.698	0.000284	0.000284
5397	2.675	2.675	0.000281	0.000281
9110	2.667	2.667	0.000280	0.000280
440	2.664	2.664	0.000280	0.000280
7445	2.657	2.657	0.000279	0.000279
6886	2.654	2.654	0.000279	0.000279
7329	2.614	2.614	0.000275	0.000275

Table 6

Table 7 reports the 20 movies best ranked and the pagerank values for method from scratch that sum up to 1 using the bigger dataset. These values are obtained using tolerance 10^{-14}.

movie index	Scratch normalized to 1 PageRank
129978	$2,783 \cdot 10^{-5}$
48761	$2,745 \cdot 10^{-5}$
26785	$2,633 \cdot 10^{-5}$
54375	$2,531 \cdot 10^{-5}$
33775	$2,495 \cdot 10^{-5}$
118941	$2,455 \cdot 10^{-5}$
42408	$2,449 \cdot 10^{-5}$
111873	$2,329 \cdot 10^{-5}$
105846	$2,311 \cdot 10^{-5}$
3426	$2,285 \cdot 10^{-5}$
147230	$2,271 \cdot 10^{-5}$
92774	$2,244 \cdot 10^{-5}$
118119	$2,219 \cdot 10^{-5}$
104119	$2,215 \cdot 10^{-5}$
28288	$2,212 \cdot 10^{-5}$
108514	$2,208 \cdot 10^{-5}$
105003	$2,203 \cdot 10^{-5}$
18420	$2,180 \cdot 10^{-5}$
903	$2,168 \cdot 10^{-5}$
146452	$2,158 \cdot 10^{-5}$

Table 7

Appendix C

The functions below have the exact same functioning as those listed in the Appendix A. The only difference is that each vector or `dict` object was substitute by another RDD object and for this reason are compleately scalable.

Those functions work with similar performance as the previous, but are more time consuming. This is due to the sum in the `compute_distance` function.

These functions still need to be improved.

```
def compute_page_rank_n_scalable(M, n, tax: float = 0.85, tolerance = 10e-18):

    i=1
    appo = M.map(lambda x: (x[0], x[1])).groupByKey().map(lambda x: x[0])
    n1 = appo.map(lambda x: 1)
    ntot = appo.map(lambda x: 1/small_num_nodes)

    vct = appo.zip(ntot)
    old_vct = appo.zip(n1)

    vct = vct.map(lambda x: (x[0], x[1]))
    old_vct = old_vct.map(lambda x: (x[0], x[1]))

    while compute_distance_scalable(vct,old_vct) >= tolerance:

        MT = M.map(lambda x: (x[0],x[1:])).join(vct)\
                .map(lambda x: ( x[0], x[1][0][0], x[1][0][1], x[1][1]))\
                .map(lambda x: (x[1], x[2]*x[3]))\
                .reduceByKey(lambda x,y: x+y)

        old_vct = vct
        vct = MT.map(lambda x: (x[0], x[1]*tax + (1-tax)/small_num_nodes) )

        i+=1
        print(i)

    print(tolerance, i, compute_distance_scalable(vct,old_vct))
    return vct, i


from operator import add

def compute_distance(old_vector, new_vector):

def compute_distance_scalable(old_vector, new_vector):
```

```
z = new_vector.join(old_vector)
z = z.map(lambda x: (x[0], x[1][0], x[1][1]))\
      .map(lambda x: (x[0], x[1], x[2], (x[1] - x[2])**2 ))\
      .map(lambda x: x[3]).sum()

return z
```