**Exercise 1 – Middle Project**

**HIERARCHICAL**

The hierarchical clustering algorithm aggregates clusters, initially composed of a single node, with their own neighbour chosen arbitrarily, since each edge has the same priority.

The *naive* version of this algorithm has proven to be very expensive from the memory consumption side, as it keeps track of both edges and non-edges in a priority queue (i.e. those pairs of nodes that are not connected by a edge).

Let be the number of nodes of the graph on which we are clustering, the algorithm inserts elements in the priority queue, that is all the pairs made up of two different nodes. In the case of the graph we have considered, which is made up of 22470 nodes, the priority queue would contain about 504 million elements.

This heaviness is reflected not only in the amount of memory required, which makes the algorithm impossible to execute for the graph in exam, but also in the execution time of the algorithm: the search for nodes to fill the priority queue has a computational complexity of and this queue must be iterated completely every time two clusters are aggregated, in order for the elements of the priority queue to be updated.

The version presented avoids this problem by considering only the edges, thus discarding the non-edge which are not useful for our purpose (not considering them forces us to perform one more search than the naive implementation, but being performed on a dictionary has negligible complexity), reducing the number of elements saved in memory and the complexity of the search necessary to fill the priority queue.

In detail, this modification brings the complexity of the search for the edges to be inserted to , where is the number of the edges present in the graph, and the the number of edges saved in memory to . The behaviour remains almost identical: taking the first edge from the priority queue, the two clusters obtained are merged in a single cluster and the other edges, in which one of the two clusters obtained at the beginning was present, are updated, replacing them with the union of the two. Each insert in the priority queue is done in a way that is given the highest priority to the smallest clusters. This precaution was taken in order to avoid to create a single large cluster, in fact inserting all the edges with the same priority, the lasting updated ones would end up being the first to be extracted at the next iteration, thus leading to the creation of a single large cluster).

The execution of the algorithm took 309.28 seconds. The results obtained are shown in the following table, in which it can be observed how many nodes of each of the obtained clusters belong to the real clusters of the graph.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster1 | Cluster2 | Cluster3 | Cluster4 |
| Politician | 1420 | 1263 | 2039 | 1046 |
| Government | 1820 | 1365 | 2399 | 1296 |
| TVshow | 967 | 786 | 935 | 369 |
| Company | 1748 | 1551 | 1942 | 1254 |

We can notice that the nodes that belong to the real cluster are not recognized from the hierarchical clustering, so they tend to be present in every cluster. The resulting accuracy is equal to 5976 out of 22470.

In conclusion, this implementation does not lose precision compared to the naive one, as it maintains the key elements of the initial algorithm, but it greatly lighten the workload and makes it possible to obtain a result in which the nodes are not part of a single cluster, but they are equally distributed.

**K-MEANS**

Two aspects play a role of primary importance for the implementation of k-means: the choice of the initial centroids and the definition of the distance between nodes and clusters.

Given the need to find (at least) 4 clusters within the graph, the initial step consists of choosing 4 centroids. These centroids, in each execution of the algorithm, are chosen randomly through a procedure that ensures that they have no connections to each other.  
As far as distance is concerned, it was decided to define the cluster containing the greatest number of neighbors of that node as a cluster with minimum distance from a node.

Once these aspects have been defined, we moved on to the implementation of the algorithm, in which a fifth "cluster" was defined, without a centroid, populated by all the nodes that cannot be inserted into the other groups. In the case of an unconnected graph, it is populated by all the nodes of the components that do not contain a centroid, while in the graph under examination (which is connected), it is of fundamental importance for correct implementation of the parallel version, discussed in more detail below.

The Naive version of this algorithm (k\_means function in the kmeans.py file) was found to be unacceptable for practical purposes given the too long execution time. In fact, the algorithm, which was tested on the Google Colaboratory online server, classified only 5000 elements out of 22470 in 6 hours (the execution was subsequently stopped manually), also showing an ever-increasing slowdown in the classification of the elements. Furthermore, the partial results are unsatisfactory given the tendency to overpopulate some clusters and leave others almost empty, as shown here.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 |
| Politician | 289 | 0 | 1135 | 0 |
| Company | 289 | 18 | 528 | 0 |
| Government | 217 | 0 | 1974 | 2 |
| TV-show | 80 | 0 | 468 | 0 |

This experiment therefore highlighted two major problems to be solved: the execution time and the imbalance of the clusters.

First of all, it is of greater importance to reduce the execution time as much as possible to speed up the analysis of the problem. A parallel version of the algorithm was then implemented (parallel\_k\_means function in kmeans.py) and the related experiments were performed with 4 jobs.

The first problem encountered in this version was the inability to assign all nodes to clusters with a single execution of the algorithm. In fact, by dividing the nodes into 4 groups, there is a very high probability of having nodes not connected with the other nodes of its group. Consequently, as mentioned above, a fifth cluster was inserted that contains all the non-assignable nodes within its group and it was decided to make the parallel call to the clustering function iteratively, classifying the nodes in each cycle remaining from the previous run.

In the case in which the number of elements not yet assigned is less than 10\*j, in which j represents the number of jobs, it is preferable not to execute the algorithm in parallel but in the naive version, since with a low number of elements the parallelization it has no advantages.

The parallel version was found to be much faster than the naive one, however requiring an unacceptable time, of about 8 hours and 30 minutes, with the execution of 4 clustering cycles, in which after the first 9118 elements remained unclassified, after the second 1251 and after the third 5. The results are shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 |
| Politician | 3851 | 410 | 1108 | 399 |
| Company | 3994 | 317 | 1295 | 889 |
| Government | 4748 | 608 | 977 | 547 |
| TV-show | 1995 | 146 | 744 | 442 |

You can immediately see how the accuracy is very low, failing to correctly identify the clusters. Furthermore, although it seems to be less affected by the problem of the imbalance of cluster sizes than the naive one, this problem recurs identically, as it was noted that, in the first cycle, each job tends to populate almost only one cluster, but they are likely to populate different clusters.  
Before trying to solve this problem, it was however decided to work to further reduce execution times.

From an analysis of the algorithm, it was assumed that the main bottleneck is given by the instruction used to derive the list containing the nodes not yet added in a cluster that have at least one neighbour in them, which has complexity:

O (n\*(sum of cluster lengths)\*4\*min(len(G.neighbours(el)),len(cluster)).

To eliminate this bottleneck, it was decided to remove the nodes from the samples variable when they are assigned to a cluster, to progressively reduce its size, and to introduce a new set: neighbours clusters. This set contains all the nodes already classified and their neighbours. So, whenever a node is added to a cluster, that node is inserted into the set and the union between the neighbourhood of the node and the set is made.

The new versions of the algorithm (optimized\_k\_means and parallel\_opt\_k\_means) are extremely faster, with equivalent results in terms of accuracy and imbalance in cluster sizes. In particular, the normal version takes about 1 minute and 30 seconds to run, while the parallel version takes about 5 minutes. Note that the parallel version is much slower as it has to be run multiple times to assign all items to a cluster. The results of these versions have not been added to the documentation as they are equivalent to the previous ones and do not introduce possible new considerations.

Having reached such a short execution time, it is now possible to try to solve the problem of the imbalance of the dimensions between clusters and improve the accuracy of the algorithm through operations that, while slowing down the algorithm, allow to reach a better trade-off between execution time and accuracy.

It can be noted that a random choice of the centroids can cause particular situations in which the degree of a centroid is much greater than the others, obtaining a much greater probability of assigning an element to that cluster than the others. Considering also that there are 2658 nodes with degree 1 and at the same time various nodes with degree>200, this problem cannot be ignored.

To overcome this problem, it was decided to take as centroids the nodes of the highest degree that do not have connections between them. Furthermore, to try not to unbalance too much the probability of inserting elements in each cluster already at the beginning of the algorithm, it was decided to insert the elements present in a priority queue, whose priority of the elements is equal to their degree, thus giving higher priority to lower grade elements. This version achieved the same speed as the previous version, about 1 and a half minutes. Furthermore, with this implementation, all the random component of the algorithm has been removed, thus obtaining the same result in each execution, the result reported here.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 |
| Politician | 96 | 1790 | 3800 | 82 |
| Company | 103 | 574 | 5811 | 7 |
| Government | 960 | 1015 | 4512 | 393 |
| TV-show | 304 | 416 | 2602 | 5 |

We can immediately see how the imbalance of the dimensions is still present but in a more reduced way than the previous non-parallel versions. As far as accuracy is concerned, it is still very low despite some slight improvements, correctly clustering 8566 elements out of 22470.

As for the parallel implementation of this version, even in this case the execution times remained approximately unchanged, taking about 5 minutes. The results are shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 |
| Politician | 933 | 1654 | 2972 | 209 |
| Company | 2186 | 159 | 4098 | 52 |
| Government | 1776 | 1030 | 3467 | 607 |
| TV-show | 1043 | 181 | 2088 | 15 |

Also, in this case we can see how the assignment of real clusters to the clusters obtained remains equivalent. From an initial analysis of the results, it can also be seen that TV-shows are more difficult to cluster through k-means than other categories. Also, the performance is lower than that of the non-parallel version, correctly assigning 7543 elements out of 22470 to clusters.

**GIRMAN-NEWMANN**The Girman-Newmann clustering is based on the edge betweenness measure and it iteratively removes the edges with highest betweenness until a stopping condition is met.

The naïve implementation of Girman-Newmann computes the edge betweenness for every node from every node in the graph once at time and then proceeds to remove the highest betweenness edge. The betweenness computation has a computational complexity of , where and are respectively the number of nodes and edges inside the graph.  
The algorithm stops once it has found 4 connected components. This condition was chosen in place of a condition based on the clustering performance because the latter takes way more time on the graph we took in consideration, since as shown in the table below we can see how removing edges tends to create very small connected components.  
The execution of this implementation took 26215 seconds (about 7 hours).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster1 | Cluster2 | Cluster3 | Cluster4 |
| Politician | 5748 | 20 | 0 | 0 |
| Government | 6880 | 0 | 0 | 0 |
| TVshow | 3327 | 0 | 0 | 0 |
| Company | 6480 | 0 | 14 | 1 |

You can see how clustering produces excessively unbalanced clusters, also having to associate a cluster in which they are not present to TVshows, obtaining a correct classification of 6914 samples out of 22470. Furthermore, the execution times are excessive and unacceptable.

Consequently, a parallel calculation of the betweenness was carried out, which turns out to be the most computationally onerous operation. This parallelization did not introduce approximations concerning the Naive version, thus obtaining the same results reported above. The running time was found to be 9146 seconds (about 2 hours and 30 minutes), still too high to be acceptable.

As also found in the second exercise, a high sampling for the calculation of the node betweenness allows to obtain very accurate results to the naive. This property was also preserved for the edge betweenness, albeit to a lesser extent. In particular, with a sampling of 5%, considering the 500 edges with the highest betweenness, 153 of them are also present in the top500 of the Naive version, where however the edges with the highest betweenness are the same as the Naïve version. So, the optimized implementation computes the betweenness from a subset of nodes, obtained by a random sampling and divides the computation in parallel threads. In this case the computational complexity drops significantly, since now we have , where , is the percentage of nodes considered and the number of threads in execution.   
In the presented solution we used 5% of the total nodes and divided the computation into 4 parallel threads.  
The introduction of a sampling with 0.05 ratio to 473 seconds (about 8 minutes). In the table below we show the results with sampling and parallelization.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster1 | Cluster2 | Cluster3 | Cluster4 |
| Politician | 5746 | 2 | 20 | 0 |
| Government | 6839 | 41 | 0 | 0 |
| TVshow | 3327 | 0 | 0 | 0 |
| Company | 6480 | 0 | 0 | 14 |

The optimization introduced drastically reduced the computational complexity and slightly the accuracy, making this trade-off acceptable.

**SPECTRAL**

The spectral clustering algorithm allows to find two clusters based on the sign of the eigenvector associated with the smaller eigenvalue, but having to associate the nodes of the graph to at least four groups there are two possible ways of operating: taking into consideration the eigenvectors associated with the two minor eigenvalues ​​or perform the calculation a second time on the first two clusters obtained.

From the first observations, we can immediately see that the calculation of the radio eigenvalues ​​and eigenvalues ​​on matrices of such dimensions is both temporally and spatially very onerous, thus deciding to use the first two eigenvectors to reduce the execution times.

The naive algorithm has shown, like the other clustering algorithms, the problem of imbalance between clusters, an index of a non-clear separation into groups of the nodes and a very high probability that in this graph there is a giant component that contains almost all the knots. Moreover, it was found to be extremely slow due to the too onerous calculation of the eigenvectors with an execution time of about 13h and 30min. The results are reported here.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster1 | Cluster2 | Cluster3 | Cluster4 |
| Politician | 0 | 1 | 0 | 5767 |
| Government | 0 | 0 | 0 | 6880 |
| TVshow | 49 | 6 | 212 | 3060 |
| Company | 11 | 115 | 126 | 6243 |

We can immediately notice that both the groups relating to 'Politicians' and 'Government' are completely contained in cluster4 (except one element), thus having to associate the former with a cluster in which their elements are not present to obtain accuracy. as much as possible. In particular, the best possible association between clusters and groups makes it possible to obtain 7207 correctly assigned nodes out of 22470

First, we try to reduce the execution time as much as possible. Since it is not possible to calculate the eigenvalues ​​in a parallel manner, a light sampling of the nodes is performed by excluding from the graph and the eigenvalue calculation all nodes of degree 1. This decision was made based on two main factors: by removing all the nodes with a degree equal to 1 from a connected graph, the graph certainly remains connected; these nodes can only belong to the cluster in which their only neighbor is present, thus being able to assign them a cluster at the end of the algorithm.

Despite being a very small sampling, this allows us to reduce the nodes by just over a tenth, from 22470 to 19812, and, considering the computational complexity equal to O(n3), we obtain O((9\*n/10)3)=O(729\*n3/1000). We, therefore, expect to reduce execution times by at least one-fifth.  
We can see how the algorithm, with this sampling, takes 10h to be fully executed. It is, therefore, necessary to make other improvements to obtain acceptable times.  
Below are the results of the algorithm with node sampling.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster1 | Cluster2 | Cluster3 | Cluster4 |
| Politician | 1 | 0 | 5767 | 0 |
| Government | 0 | 0 | 6866 | 14 |
| TVshow | 86 | 0 | 3241 | 0 |
| Company | 140 | 40 | 6282 | 33 |

From the results, however, we can immediately see how on this occasion, in addition to the 'Politicians', also the 'TVshows' appear to have to be associated with clusters that do not contain their elements to maximize accuracy. Specifically, we got 7006 successfully paired nodes out of 22470.

To reduce the execution time it was therefore decided to use a different algorithm for the computation of the eigenvectors since the Arnoldi method used by scipy is too expensive for the computation of n-1 eigenvectors and not accurate if you want to calculate a lower number of eigenvectors.  
It was therefore decided to use the inverse power method which calculates an approximation of only the dominant eigenvector of the inverse matrix through k products between matrix and vector without the need to find the relative eigenvalue. Note that the dominant eigenvector, the eigenvector relating to the eigenvalue with maximum value, of the inverse matrix corresponds to the eigenvector relating to the minimum eigenvalue of the original matrix.  
However, this method allows to calculate of only one eigenvector and consequently, to divide the graph into four clusters, once the first result is obtained, further executions of the inverse power method are carried out. This method is rerun on both clusters obtained if the cluster with a smaller size contains at least one-tenth of the total elements, otherwise, it is rerun twice on clusters with the larger size. In the case in question, the difference in size between clusters is excessive and therefore the second solution is applied.  
Even in this method, however, there appears to be a problematic bottleneck represented by the calculation of the inverse. In fact, in addition to requiring very long execution times, in the various experiments, this operation led to complete saturation of the ram after an hour of execution and, given the need to perform this calculation 3 times during the algorithm, this results not acceptable.  
Further approximations were then used to calculate the inverse. The method that was most suitable for this purpose was the LU-decomposition which allows us to find an approximation of the inverse in a much shorter time.

Once at this point, various experiments were carried out to choose the number of 'k' update cycles of the inverse power method. It was thus concluded that after many cycles equal to the number of nodes in the graph, the clusters obtained by this method no longer change as the cycles increase. We, therefore, set k=n.  
This version of the spectral algorithm was found to be much faster with a runtime of about 15 minutes. The results are shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Cluster1 | Cluster2 | Cluster3 | Cluster4 |
| Politician | 0 | 0 | 5731 | 37 |
| Government | 1 | 2 | 6836 | 41 |
| TVshow | 0 | 0 | 3304 | 23 |
| Company | 2 | 5 | 6463 | 25 |

Like the previous results, also in this case we have that a cluster contains almost all the nodes with a further reduction in the size of the other clusters and the nodes correctly associated with the clusters have dropped to 6878 out of 22470.

Despite the excellent execution times, an experiment was also carried out on the sampled version of this algorithm. In this case, however, the execution times did not differ much, with a reduced execution time of only 30 seconds, while in terms of accuracy, clusters were obtained even more unbalanced in size compared to the latter version.