**Project Report**

Topic 4: Fair Community Detection

Rak Anastasia Stamatis Konstantinos

**Introduction**: In this report, we would like to explain our code, walk you through our thought process for its creation and clarify all non-trivial spots.  
 However, before going into the code itself, we would like to remind you what our goal with this project was. By choosing the Fair Community Detection topic, we wanted to examine how different community detection algorithms perform regarding fairness by running them on graphs whose nodes carry certain ‘sensitive’ attributes. After that, as we talked about during our Project Proposal, we’d like to come up with some method to process the communities obtained by these algorithms to achieve fairness where it’s necessary.  
 So, on a final note before going into the code, we shall mention that the datasets that we worked on are the Twitch Gamers Social Network from SNAP (<https://snap.stanford.edu/data/twitch_gamers.html>) and two synthetic datasets generated using the stochastic block model.

**The Code:** Our code begins with two cells that include mainly module imports and a filter to ignore warnings. The latter was used to avoid overwhelming amounts of warnings. As for the module imports, the most notable ones are those related to the *dask* module in the second cell, which are necessary for us to be able to run parallel computations on graphs as big as the one that is inferred from the Twitch Gamers Social Network dataset (trying to perform the K-means clustering algorithm on this graph without using parallel computation was impossible due to memory limitations).   
 Right after that, we import the *Client* method from the *dask.distributed* module, and through the only other command in the cell we get four ‘workers’ which are going to be used to attain communities via the K-means algorithm. When running this cell, one also gets as output a link which leads to a page which shows details about the computations performed by each core (e.g. memory, usage, computations completed etc.).  
 In the following cell, we form two dataframes from the csv files that you can download if you follow the link to the Twitch Gamers dataset. *large\_twitch\_edges.csv* contains lines of pairs of numbers. These numbers are the IDs of the Twitch users and when two numbers are paired in the same line this indicates that the corresponding users are mutual followers of each other. *large\_twitch\_features.csv* includes data on the Twitch users, such as their views, their language, and others. However, the only one of these attributes that we are interested in when considering the performance of clustering algorithms in respect to fairness is the users’ affiliation with Twitch, which is represented by a binary digit. To have access to the contents of these dataframes, we turn them into NumPy arrays in the following lines of code. Immediately after that, we split the *edges\_array* array (which essentially represents the edges in the graph inferred from the mutual-follower relationships of the users) into chunks that are going to be handled by the different workers that were initialized earlier. These chunks are kept into *dask\_array*. Through the next line, by using the *persist* method, we perform some computations on this array which are saved in memory for later use. **(Σχόλιο για cell με distributed.protocol.serialize).** Finally, in the next cell, the graph that we’ve claimed can be inferred from *large\_twitch\_edges.csv*  is finally defined formally as a NetworkX graph. Running this cell right now isn’t necessary since the next step which is to run the K-means clustering algorithm on the Twitch graph using parallel computations makes use of the *daskar* variable, which maintains the same information as the one that is contained in *edges\_array* and in *Twitch\_Graph.*  After obtaining the centers and the labels of the clusters produced by the K-means algorithm, we assign an attribute named ‘kmeans\_cluster’ to each node, which is equal to the label of the cluster that the K-means algorithm assigned said label to. Then we use this to form a dictionary, where keys are cluster labels and the values are the nodes in each cluster. This dictionary is named *dictt* and is used to calculate the modularity of the clustering produced by the algorithm. When the number of clusters is equal to **INSERT NUM OF CLUSTERS** we get modularity equal to **INSERT MODULARITY.** This indicates that **INSERT COMMENT ON MODULARITY (IF NEEDED).** **ΣΧΟΛΙΑ ΓΙΑ ΤΟ ΑΜΕΣΩΣ ΕΠΟΜΕΝΟ CELL ΜΕ ΤΙΣ ΕΝΤΟΛΕΣ with open('labelskmeans.csv', 'w', newline='') as file:**

**writer = csv.writer(file)**

**writer.writerow(labels)**

**ΚΑΙ ΣΧΟΛΙΑ ΓΙΑ ΤΟ ΠΛΟΤ ΑΝ ΤΟ ΚΡΑΤΗΣΟΥΜΕ.** Next up, we run the second of the three community detection algorithms that we set out to evaluate on their fairness performance, and this algorithm is Label Propagation. For this task we didn’t use parallel computation since this cell runs in less than 30 minutes, and produces 79 clusters and modularity equal to 0.03567630678997064, which proves that there is not much in the way of community structure in our graph.  
 **ΠΑΡΑΓΡΑΦΟΣ ΓΙΑ SPECTRAL** Now that we’ve ran all three community detection algorithms, we are ready to begin examining how well they perform fairness-wise. To do that, we first need to calculate the percentage of affiliated users in our Twitch Gamers dataset, and this is precisely what is calculated in the next cell and is stored in the *aff\_percent* variable. The closer the percentage of affiliated users in each cluster produced by the three algorithms is to *aff\_percent*, the fairer the clustering will be considered.  
 Since the first clustering algorithm that we worked on was K-means, it makes sense for us to examine its fairness performance first. With the following cell, we calculate the percentage of affiliated users in each cluster and the average percentage of affiliated users in all clusters and observe that **INSERT ΣΧΟΛΙΟ ΓΙΑ FAIRNESS K-MEANS**  
 We then proceed to do the exact same thing with the clustering provided by the label propagation and spectral clustering algorithms. The results we get are **INSERT ΣΧΟΛΙΑ ΓΙΑ LABEL PROPAGATION ΚΑΙ SPECTRAL.**  
 So far, we have only been working on the Twitch Gamers data, but now it is finally time to generate our own synthetic data using the Stochastic Block Model (SBM). Since randomness is involved in the generation of such graphs, we choose a seed for the random functions in the beginning of the related cell in order to get the same results every time we run the code. The one of the two graphs is generated with an assortative planted partition model, which means that the matrix *P1* that contains the probabilities of an edge existing between two communities features a constant *p* on every diagonal element and a constant *q* in every non-diagonal element such that *p>q.* The reason why we choose *p>q* is that we ideally want a higher concentration of edges between nodes of the same community. For the other graph we generate random probabilities of an edge forming between nodes of the same community in the range of 0.1 to 0.5 and probabilities of an edge forming between nodes of different communities in the range of 0.0 and 0.1. This version of the SBM is more generalized than the planted partition one since it doesn’t feature just a constant probability on all elements on the main diagonal and another constant probability on all elements off the main diagonal. For both graphs we choose to have a total of 1000 nodes split into communities of size indicated by the *sizes* array.  
 Since the nodes of graphs generated using an SBM do not have attributes that can be used to judge the fairness performance of any clustering algorithm that is going to be ran on the graph, we must produce such attributes ourselves. In the following cell, for each of the two synthetic graphs we perform the following process: For each of the graph’s nodes, we generate a number between 0 and 999. If the number is smaller than 500, the node gets the attribute ‘0’, otherwise it gets the attribute ‘1’. Thus, we aim to get approximately half of the graphs’ nodes to have attribute ‘0’ and the rest to have attribute ‘1’. Some cells below, a calculation on how many nodes with attribute ‘1’ and ‘0’ exist in each synthetic graph is performed (expectedly, not exactly half of the nodes possess attribute ‘0’ or ‘1’. One is slightly more prevalent).  
 What follows is that we run the K-means clustering algorithm on both graphs, and after labeling nodes by the cluster that they belong to, we form dictionaries that have the K-means cluster labels as keys and the nodes in each cluster as values exactly like we did before.  
 We then measure the percentage of ‘affiliated’ nodes in the clusters generated by the K-means algorithm for both synthetic graphs, as well as the average percentage of affiliated nodes in all clusters (We only named the randomly generated attribute of the synthetic graphs’ nodes ‘affiliation’ for it to match the attribute of the users of the Twitch dataset. This attribute could be named any other way).   
 Probably the most important task of the whole project is the one that follows directly after that. And that task is to find a way to process the communities detected by the three clustering algorithms in order to increase ‘fairness’ where it is ‘necessary’. We will return to what necessary means in a little bit. What we would like to display first are the two methods that we came up with to work towards fairness.  
 The first is the more simplistic one. It has the potential to increase fairness more than the second one but can also ‘damage’ our clustering more. In this method, we take all possible pairs of clusters produced by any clustering algorithm and check if affiliates are ‘underrepresented’ in one of them and ‘overrepresented’ in the other. While that remains the case for a certain pair of clusters, we look for a non-affiliated node in the first cluster, delete it from the cluster and move it to the other. We stop only when the aforementioned condition ceases to be true. By doing that it’s pretty self-explanatory that fairness is increased; two clusters involved in this node-exchanging process will have a percentage of affiliated nodes that is less ‘imbalanced’ after its ending (If this does not seem so trivial yet, it will all make total sense when terms like ‘overrepresented’, ‘imbalanced’ etc. are further clarified). However, taking any node from a cluster and moving it to another one which may not even be connected with the former can really make the resulting clusters ‘meaningless’, i.e. not realy showcasing any type of community and just being random collections of nodes that do not have a particularly high concentration of edges between them.  
 In order to avoid such a scenario we introduce our second method for increasing fairness, which is essentially the same as the first one, but only moves a ‘misplaced’ node to another cluster if the subgraph induced by this node and the nodes in said cluster is connected. This condition makes sure that the candidate-for-cluster-switch node is connected to at least one node of the cluster that we’re considering moving it to, and this way we at least verify that this node is not completely unrelated to the cluster that we’re considering moving it to. This method is obviously less potentially damaging to the quality of our clusters but can possibly not make as grand of changes in fairness as we could have hoped for (since certain fairness-improving moves of nodes to different clusters that would have happened in the first method are not allowed in this one). Of course, the condition that the nodes and clusters involved need to satisfy in order for a switch of clusters for the node to happen can be made more strict, with analogous maintenance of community structure for the clusters, but also subpar increasing of the overall fairness.  
 Now, we need to address what it means for the processing of the clusters to be necessary in order to achieve fairness. It’s of course statistically, and sometimes even numerically, impossible for all clusters to have the exact same percentage of affiliated nodes as the overall percentage in the whole graph. This does not mean that the clustering is unfair. We consider a particular clustering to be unfair if one or more clusters show a much lower or higher concentration of affiliated nodes than the overall percentage of affiliated nodes in the graph. The number by which the percentage of affiliated nodes in a certain cluster should be off from the overall percentage of affiliated nodes for the cluster to be considered ‘unfair’ is not standard and is linked to factors like what the actual overall percentage of affiliates is, what is the number of nodes within the cluster and the entire graph etc. This means that if a certain cluster of a graph with an equal number of affiliates and non-affiliates does not contain equally as many affiliates and non-affiliates, it is not automatically considered to be unfair; if the percentage of affiliates within the cluster is ‘close enough’ (with what ‘enough’ is being determined by us using criteria like the ones mentioned before) to 50%, then the cluster is treated as if its fair and no nodes that belong to it are subjected to cluster transfers.  
 In our code, we have implemented the two previously discussed methods for increasing fairness. The first method is tested on the first synthetic graph and the second method is tested on the second synthetic graph. Both graphs were clustered using the K-Means algorithm with 4 clusters, aiming to replicate the real number of communities with which the two SBM graphs were ‘built’. A brief check verifies that K-Means perfectly recovers the 4 communities for both graphs.  
 Through the cell that counts and prints the number of affiliated nodes in both synthetic graphs we know that there are exactly 51,8% affiliated nodes in *synthetic\_1* and 51,7% affiliated nodes in *synthetic\_2.* By that point we have already printed the percentages of affiliated nodes of all clusters for both graphs, as well as the average percentage for these clusters. When running the first (and simpler) method on *synthetic\_1* we consider clusters to be unfair if they have a percentage of affiliated nodes higher/lower than +1%/-1% of the percentage of affiliated nodes in the whole graph. Before the process begins, we can tell just by looking at the percentage of affiliated nodes of all four clusters that some node transfers to other clusters are going to be performed. Right after the ending of this process, the percentages of affiliated nodes for all clusters are printed again and we can clearly see that all percentages have gotten closer to 51,8% (and could possibly get even closer if we ran the process again while considering clusters to be unfair when they were even less than 1% off from 51,8%).  
 If we were to run the second method on *synthetic\_2* by using again 1% as the ‘threshold’ for maximum deviation from the overall average percentage of affiliates before a cluster is considered ‘unfair’, we would not get a single node-transfer (as can be expected if one looks at the percentages of affiliated nodes in each cluster of this graph a bit carefully), and this really illustrates the importance of picking a correct threshold for this procedure. By picking 0,2%, we get a total of 4 nodes switching clusters, and despite that small number we can see by the results that fairness has improved to an extent.  
 These methods can be tested on all graphs used in the project with any type of clustering. We include just these two tests since they can be executed quickly and properly exemplify the strengths and weaknesses of our proposed methods. Also, since clustering algorithms seem to be far from being very unfair, it seemed kind of tedious to us to include all possible combinations of graphs, clustering algorithms and cluster post-processing methods and then comment on the barely noticeable differences of the results.