Report Homework 4

Made by:

- Malick Alexandre Sarr Ngorovitch
- Biko Catalano
- Andrea Marcocchia

Part 1

For the following homework, we will use a JSON format dataset. The dataset consists of information regarding **scientific conference informations**, which have already been extracted from an in-house database:

- **Authors** : information about author, separated by author_ID and Author name
- **Conference**: data containing infomation about a particular conference, made of a conference_ID and the URL of conference in the site the data was collected from
- **Publication**: information about a publication made in a conference, that contains the publication_ID, the URL of the publication and the title of the publication

A record in the JSON database is made of a list of authors, a conference ID, a conference Name, a conference URL, a publication ID, a publication URL and the title the publication.

The team (Malick, Andrea and Biko) responsible for the analytics of the above dataset is made of 3 data science students. The tasks to be performed on this data set are available at the following link.
http://aris.me/contents/teaching/data-mining-ds-2017/homeworks/homework4.pdf
(http://aris.me/contents/teaching/data-mining-ds-2017/homeworks/homework4.pdf)

Our first task in hand was the creation of a network or a graph. One of our main goal is to understand the relationship between authors and publications. Consequently we want to map out the different relationships between authors and publications within a network so that we can analyse that network and derive some interesting information about it. By the end of this PART 1, we should have a graphmade of nodes as Author ID. The **graph** itself is going to be an undirected graph with weighted edges based on authors that share at least one publication. The weight of those edges is to be determined on how similar two sets of publication are. We will use the Jaccard similarity to determine the weight.

The **weight** of each edge is evaluated in the following way:

$$w(a_1, a_2) = 1 - J(p_1, p_2)$$

Where $J(a_1, a_2)$ is the the **Jaccard coefficient**, that measures similarity between finite sample sets (in our case set of publications for each author), and is defined as the size of the intersection divided by the size of the union of the sample sets. Writing in formula, we obtain:

$$J(p_1, p_2) = \frac{|p_1 \cap p_2|}{|p_1 \cup p_2|}$$

Obviusly we have that:

$$0 \leq J(p_1, p_2) \leq 1$$

To make it we create the **jaccard_sim function**, that computes the jaccard similarity between two nodes.

If for example two authors have exactly the same publications, and both of them have only that publications, the Jaccard distance between them will be $1 - \frac{x}{x}$, where $x \in \mathbb{N}$.

If two authors have no publications togheter, their Jaccard distance will be 1, and so we'll have no links between the two nodes.

This are the methods used in this part:

- jaccard_sim(set1,set2): Calculate the jaccard similarity between two sets of publications
- load_json(name): Load a particular JSON file
- save_json(name_file, data): Save data in a variable

Part 2.A

In this part, **given a conference in input**, we study the subgraph induced by the set of authors who published at least once at the input conference.

Once we have the graph, we start computing some **centralities measures** (degree, closeness, betweeness) and plot them.

In the plot below the **conference subgraph** is visualized. The red points show the authors that were present at the conference considered, and the links between them. at As we can see there are many nodes linked together in the middle of the subgraph. There are also some other nodes in an external circumference, with few links with the one positioned in the inner part.

The layout of the graph emphasize the presence of authors with few links.

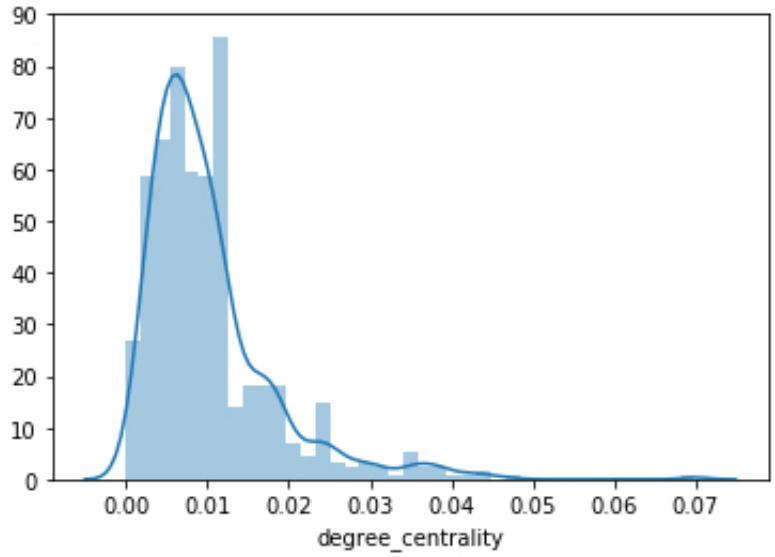
To obtain same information about the **centrality of the data**, we **load the module part_2_A**. This module contains a class Part_2_a that takes in input a conference_id , the dictionary and the graph and builds the subgraph induced computing some centralities measures.

The **degree centrality** is the easier centrality measure to be obtained. It is defined in the following form:

$$v = \frac{d_v}{n-1}$$

The denominator is used to compare values between different graphs, normalizing by the maximum possible value, n – 1.

We can see an **histogram** of the degree similarity below:



As we can see in the plot, there are many nodes with a small degree centrality. It means that the whole graph (*full_dbpl*) has a certain disconnection. Due to the normalization, the maximum possible value of the degree centrality is 1, but we observe at most 0.07.

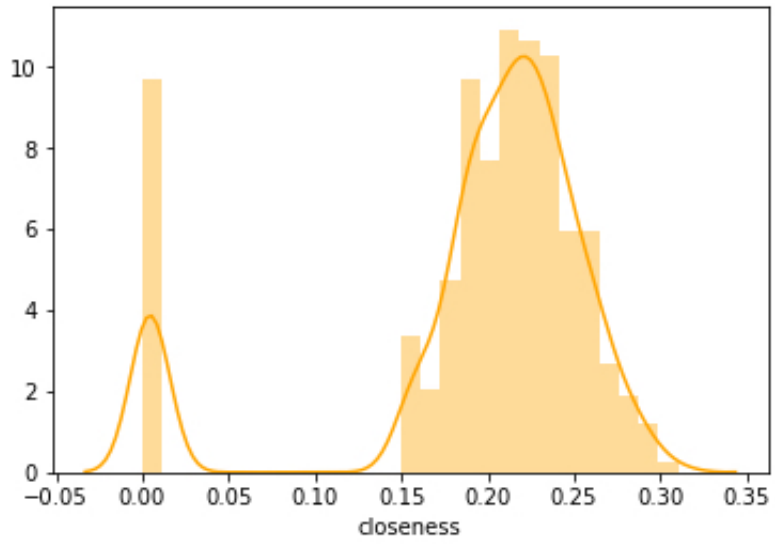
Another centrality measure is the **closeness centrality**, that measures how close is the node to the rest of the network.

Also in this case the measure is **normalized** in order to obtain values in a range between 0 and 1. The denominator is the maximum possible value, $(n-1)^{-1}$, obtained when a single node is connected with $n-1$ other nodes. Considering that centrality has to be large when the distance is small, we use the reciprocal of that.

Writing the closeness in formula, we obtain:

$$v = \frac{n-1}{\sum_{u \in V} d(u,v)}$$

This is the histogram of closeness centrality:

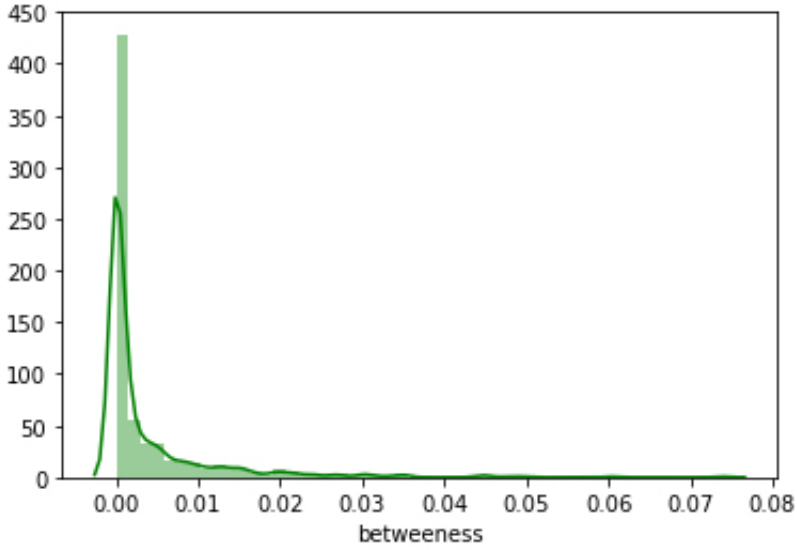


Also in this case the value of the centrality measure is smaller than 1 and bigger than 0. In this plot we observe many values with closeness centrality equal to 0. It happens because there are some isolated authors who collaborate only with a few of other authors. Except of the 0's, there are many values in the interval between 0.15 and 0.30 with a distribution which seems to be a Normal one.

The last centrality measure analysed is the **betweenes**. This measure is based on the **shortest path**.

The betweenness centrality for each vertex is the number of the shortest paths that pass through the vertex.

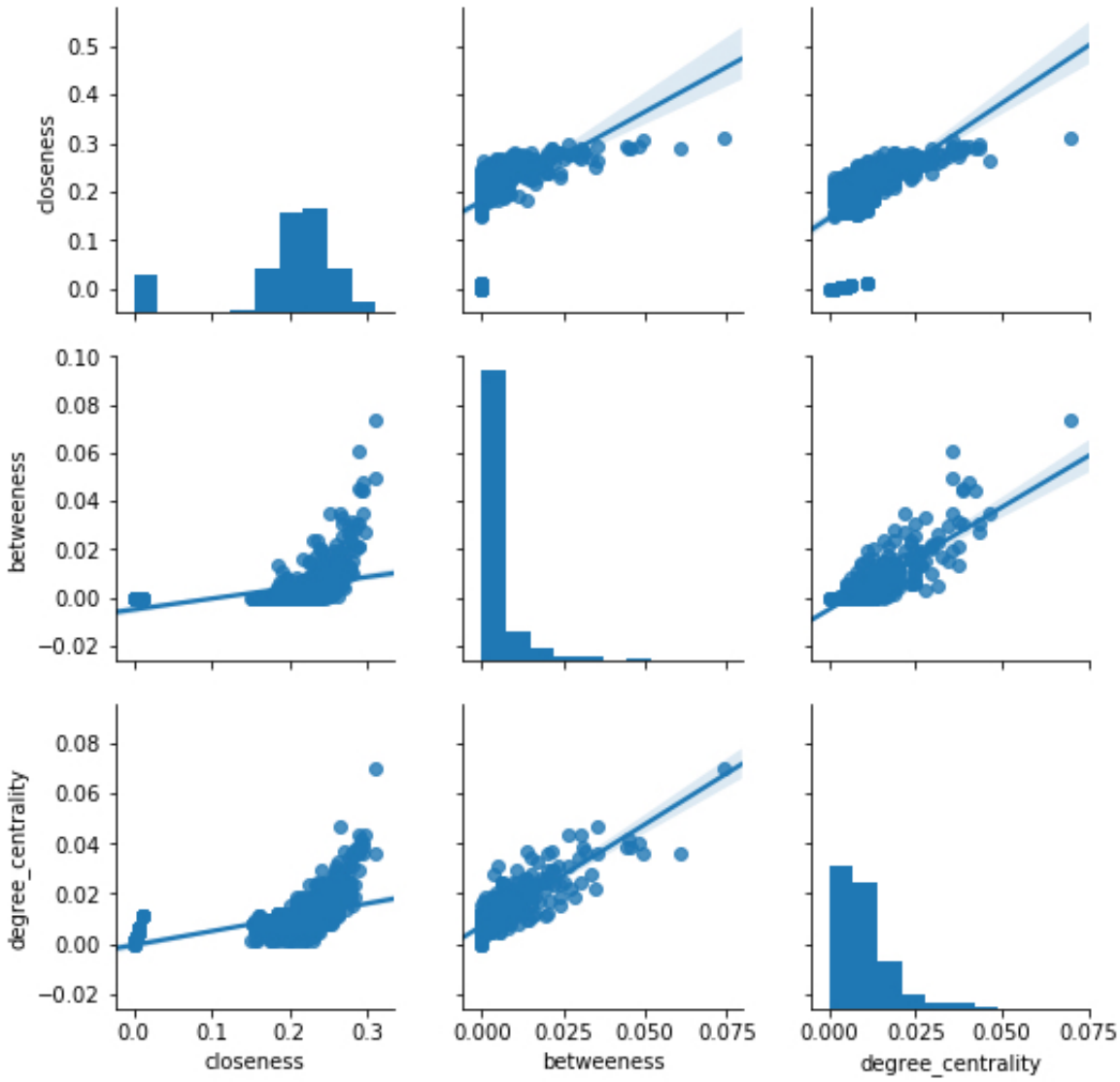
We can observe the distribution of the betweenes centrality in the following plot:



Once again, there is a particular concentration around 0, and the explanation is the same: there are many unconnected nodes.

We can observe also the **correlation plot** to understand, for example, the correlation between closeness and degree centrality.

In the diagonal of the plots below there are the histograms of the centrality distribution. In the other plots there are scatter plots with different type of centrality measures in the X or Y labels. There is also a linear regression estimation.



In the following table it is possible to see the exact values of the **correlation matrix**.

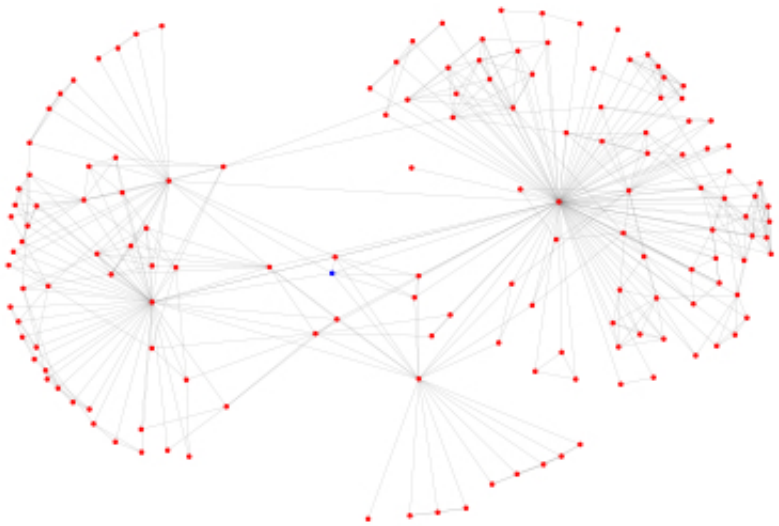
Type	Closeness	Betweeness	Degree
Closeness	1	0.406228	0.510922
Betweeness	0.406228	1	0.828886
Degree	0.510922	0.828886	1

As expected there is an **high correlation between the degree centrality and the betweenes (0.8)**. In fact, analyzing the histrogram made before, both the centrality measures have values really near to 0. There is also a correlation, as supposed, between the other centrality measures (degree-closeness and closeness-betweeness). In all these cases there is a **positive correlation**, which means that the bigger is a closeness measure, the bigger is the other one. We can visualize this aspect by observing that points (and the related fit) are more or less next to the principal diagonal.

Part 2-B

In this section of the homework an author ID and an integer *d* is given in input. The expected output is the subgraph induced by the nodes that have **hop distance** (i.e., number of edges) at most equal to *d* with the input author.

In the plot below, that is the subraph of the hop distance. The starting author is characterized by the **blue** color of the node.



To evaluate the **hop distance** it is possible to use the ego_graph function, provided by **networkx** package. We opted to write down our **own function**, called hop-distance and positioned in part_2_B.py module, that returns in output the subgraph requested.

The function used is the following:

In [1]:
```
def hop_distance(self,graph,node,d):

    self.node_input=node
    self.graph=graph

    if d==0:

        self.Hop_Distance=node

        return self.node_input

    elif d==1:

        self.Hop_Distance=self.author_neighbors(graph,node)+[node]

        return self.author_neighbors(graph,node)+[node]

    else:
        ToT=set([node])
        N=set(self.author_neighbors(graph,node))

        for i in range(d-1):

            C=set()

            for n in N:
                L=self.author_neighbors(graph,n)
                A=set(L)
                C.update(A)
                ToT.update(A)
            N=C

        self.Hop_Distance_graph=self.graph.subgraph(list(ToT))

        return self.Hop_Distance_graph
```

Part 3-A

For part 3, we were tasked to compute a generalized version of the **Erdős number**, which measures the "collaborative distance" in authoring academic papers between a single person and Hungarian mathematician Paul Erdős. In our case, instead of using Mr Paul Erdos we will use **Aris** (Aris_id is 256176).

The goal of this task is to find the shortest path between an input author and Aris, using the weights previously calculated as a measure of distance. In order to find the shortest path, we will use the Dijkstra shortest path algorithm. Even if we wrote that algorithm in different ways, unfortunately, we have always stumbled with performance issues. For example, we first used the general implementation of the **Dijkstra algorithm** as described in Wikipedia. We ended up having a working code that took around 17 minutes to find the shortest path between two nodes. Refactoring the same algorithm allowed us to dial it down to 5 min which was in our case still too long.

After some discussion, we wanted to see is there was a way we could combine the Dijkstra Algorithm with the different costs saved in a heap since we used to recompute the various costs in our previous implementation. After doing some research, we took as reference the approach used by Alexey Kalachev which did exactly want we wanted to do.

We need to have a list of edges basically saved in a tuple with a starting node, its connected node with the related weight. At this point we declare a default dictiorary *g* so that each node in *g* has a list made of tuples that link to its connected edge along with the weight of that connection We will use that dictionary as a reference for our Dijksta algorithm.

We obtain it with the following line of code:

In [ ]:
```
def weight_dict(graph):
'''Create a dictionary that contains for each node of the graph all
the linked nodes and their weigth'''

# Define the list where will be stored data
    lst_tot=[]
    for xx in tqdm(graph.nodes()):
        for yy in graph[xx]:
# Save the weigth between node xx and node yy
            lst=(xx,yy,graph[xx][yy]["weight"])
            lst_tot.append(lst)

# Store the information obtained previously in a dictionary
    g = defaultdict(list)

    for node1,node2,weight in tqdm(lst_tot):
        g[node1].append((weight,node2))

    return g
```

Once we have a list of edges we can run our algorithm. It will take as an input the list of edges, a starting node and the target node.

- We declare a list containing the unvisited node and a set containing the visited node
- We iterate the procedure while there are still some unvisited nodes. The unvisited nodes will be a tuple containing the cost and the node itself. If the new unvisited node is not in visited (seen), we add that node to the visited set.
- We take the vertices pointed to, update the code by adding it to the previous, and pushing it to the heap of unvisited nodes q. Then we will restart at step 2.
- If all the nodes are visited and there are no numerical costs outputed, the program will return 'inf' which means that there are no paths beween the input node and the target one.
- The algorithm returns a dictionary with all the linked node and the related weigths.

The **dijkstra function** is built in the following way:

In [ ]:
```
def dijkstra_heap2(g, start):
'''Return a dictionary with nodes linked with start and related wei
gths'''
    maw = {}
    q = [(0,start)]
    seen=set()
    while q:
        (cost,v1) = heappop(q)
        if v1 not in seen:

            seen.add(v1)

            for c, v2 in g.get(v1, ()):

                if v2 not in seen:


                    heappush(q, (cost+c, v2))
            maw[v1] = cost
    return (maw)
```

In our previous version of the dijkstra algorithm we ask in input both the starting and the end node. During the iteration, if the node *v1* is equal to the end one, we return the cost and terminate the algorithm (because this is what we are looking for). This version of the code is faster when the star and the end node are near between them. But, for optimize Part3.B, we prefer to use *dijkstra_heap2* function, that return always a dictionary, but allow us to run faster the next part.

Part 3-B

For this part of the homework, we were tasked to write a Python software that takes in input a **subset of nodes** (cardinality smaller than 21) and returns, for each node of the graph, its **GroupNumber**. Consequently, the group number of the node in the graph will be the **minimum distance between that node and a node in the subset**. First we will need to create the subset node that will constitute the reference points for our group numbers. The length of the subset will be at most 21.

In order to get the group number, our first idea was to use an **iterative approach**. That means that for every node in the graph, we apply the Dijkstra heap to every node in the subset, assigning the lowest value to its respective group. This approach works fine on a small data set, but on a bigger one, it will take a very long time to process everything.

So we needed to improve performance time. One thing that we realized is that in the iterative versions, the algorithm would assess the entire possible path and then output the shortest distance between the start node and the target node. Ergo, the algorithm would basically go through the whole graph assessing all possible routes. What we want to do to improve the performance is to run the Dijkstra algorithm only once. That means running the dijkstra algorithm from a node in our subset to every single node in the graph and saving the cost from a starting node to every single node in the graph. This is exactly what we our able to do with the dijkstra function defined before.

First what we will need to do is to create our list of groups. We decided to use two dictionaries for that. The **first one** named groupNumbers will contain our final list with the nodes in a graph belonging to a particular group. And the **second** dictionary will contain the minimum distance between the node in the subset to every single connected node in the graph along with it cost.

Now each group in the subset has a list of every node in the graph with which a path exist along with its weight. We now need to compare them and assign the node with the minimum number to a group number. In order to do that, we have decided to use a **heap** to maintain the minimum value. So for every node in the graph, we will save its corresponding cost to every node in the subset in a heap. The heap will contain a tuple, with the weight of a node and its group number. Once every node in the subset were pushed in the heap, we can just pop the value in the heap to get the smallest one. Accessing the second value in the item popped will correspond to the group number and we can easily append the current node to that particular group.

Doing it this way saves a lot of processing time, now we can easily assess the whole graph in an acceptable amount of time.