

Social Network Analysis Project

Luigi Maiese – 0622900080 – l.maiese1@studenti.unisa.it

Mario Cerino – 0622701191 – m.cerino8@studenti.unisa.it

Alberto Vitiello – 0622701186 – a.vitiello62@studenti.unisa.it

Arcangelo Palma – 0622701208 – a.palma27@studenti.unisa.it

Contents

[Midterm Project 3](#_Toc77776001)

[Exercise 1 3](#_Toc77776002)

[Task 3](#_Toc77776003)

[Solution 3](#_Toc77776004)

[Results and discussion 8](#_Toc77776005)

[Exercise 2 12](#_Toc77776006)

[Task 12](#_Toc77776007)

[Solution 12](#_Toc77776008)

[Performance Analysis 12](#_Toc77776009)

[Exercise 3 16](#_Toc77776010)

[Task 16](#_Toc77776011)

[Solution 16](#_Toc77776012)

[Exercise 4 19](#_Toc77776013)

[Task 19](#_Toc77776014)

[Solution 19](#_Toc77776015)

[Final Project 22](#_Toc77776016)

[Exercise 1 22](#_Toc77776017)

[Task 22](#_Toc77776018)

[Solution 22](#_Toc77776019)

[Exercise 2 27](#_Toc77776020)

[Task 27](#_Toc77776021)

[Solution 27](#_Toc77776022)

[Exercise 3 41](#_Toc77776023)

[Task 41](#_Toc77776024)

[Solution 42](#_Toc77776025)

# Midterm Project

## Exercise 1

### Task

Consider the “Facebook Large Page-Page Network” dataset available at the Stanford Large Network Dataset Collection (SNAP) (https://snap.stanford.edu/data/facebook-large-page-page-network.html). Note that network edges are provided in the file musae\_facebook\_edges.csv contained in the zip file linked in this page. Analysing this file, you can see that nodes of the network can be partitioned in four categories: politicians, governmental organizations, television, shows and companies. We will refer to this partition as the real clustering. You are required to cluster nodes of the network in at least 4 clusters using each of the partition algorithms seen in class: hierarchical, k-means, Girman-Newmann (betweenness-based clustering), spectral. Note that the network is very large and the naive implementations of these algorithms may be very expensive. Hence, you are required to optimize these algorithms (by sampling, parallelism, and adhoc optimizations) to make their running times feasible. Compare the clustering obtained through each of your algorithms with respect to the real clustering given in the file musae\_facebook\_target.csv. Discuss the trade-off between precision and running time of each of your proposed implementations.

### Solution

For exercise 1 the network has been clustered using the following methods:

1. Hierarchical:
   1. Naïve
   2. Custom ad-hoc optimization
2. K-Means
   1. Naïve
   2. Custom ad-hoc optimization
3. Betweenness:
   1. Naïve
   2. Parallel
4. Spectral:
   1. Naïve
   2. Parallel

#### Hierarchical Clustering

Hierarchical clustering is a bottom-up approach for clustering. At first step a number of clusters equal to that of the nodes in the graph is created, and exactly one node is assigned to each cluster. Then Hierarchical should merge clusters with minimum distance until a condition occurs.

* Naïve

The only difference of our naïve approach respect to the one seen in class, is that it stops automatically when it reaches the desired number of clusters, instead of asking the user if it should continue. In our case, the dataset contains four clusters, so the target number of clusters is 4. There are 2 main problems related to hierarchical proposed in the lesson 2. Despite the initialization phase which keep a PriorityQueue for each possible pair of nodes, the algorithm must check all possible pair and then merge clusters. This requires a time complexity of O(n^2\*log(n)). The other problem is the initialization phase, because in some of our laptop with a limited RAM, filling the PriorityQueue with all possible pairs raises a MemoryError Exception.

for u in G.nodes():  
 for v in G.nodes():  
 if u != v:  
 if (u, v) in G.edges() or (v, u) in G.edges():  
 pq.add(frozenset([frozenset(u), frozenset(v)]), 0)  
 else:  
 pq.add(frozenset([frozenset(u), frozenset(v)]), 1)

* Custom ad-hoc optimization

In order to solve the above-mentioned problems, we decide to devise and implement an optimization ad-hoc hierarchical algorithm.

In the init phase we instantiate a python dictionary where the key is the node’s label and the value the index of cluster the node belongs to. However, since we need the distance for 2 clusters, we keep another dictionary where we store for each cluster a set of neighbours, so that the key is a set of nodes contained in a cluster and the values are the neighbours.

Then we produce a cluster for each node in the graph.

#1. Instanziare dizionario dove la chiave è la label del nodo e il valore l'indice del cluster  
cluster\_belongs = {}  
#2. Instanziare un dict dove la chiave è un insieme di nodi (frozenset e i valori sono i vicini)  
clusters = {}  
#1 Riempire clusters con tutti i nodi del grafo  
for n in G.nodes():  
 clusters[frozenset([n])]=[k for k in G.neighbors(n) if not k==n] #delete self loops with the condition  
 cluster\_belongs[n] = frozenset([n])

Then for each cluster we choose a random neighbours (the neighbours have all the same distance from clusters), we take the cluster which neighbour belongs to, we merge the 2 clusters and add the neighbours of the neighbours’ cluster to the neighbours of the current cluster, being careful not to take duplicates. By this way we have reduced time complexity but also memory consumption, because we don’t check for all possible pairs and don’t keep a PriorityQueue for each pair.

#### K-means clustering

* Naive

K-means clustering takes K seeds far one from each other (they don’t share an edge among them) and they form the first nodes for each cluster. Then a random node is chosen and it’s assigned to the closest cluster. Our naive implementation works only in the considered domain of this exercise, so with four clusters.

Anyway, we haven’t provided any launch on this algorithm version because of the execution time. So we have tried with another optimized version that’s shown below.

* Ad-hoc Optimization

Our optimized algorithm keeps trace of clusters and neighbors by a dictionary data structure. The first part is equal to the naive version, so the first 4 nodes are the seeds of each cluster.

They are set as keys of the clusters dictionary. In this dictionary the keys, as said, are the seeds and values are sets containing  all cluster’s nodes . The  neighbors dictionary instead keeps, for each cluster (indexed by the seed), a set containing the neighbors of all cluster’s nodes.

clusters = {}

    neighbors = {}

    all\_node = set()

    for i in range(4):

        # clusters key is the first node in the cluster

        clusters[seed[i]] = set()

        clusters[seed[i]].add(seed[i])

        # neighbors contains the neighbors of all the node in a cluster, the key is also here the first node

        neighbors[seed[i]] = set(nx.neighbors(G, seed[i]))

        # tracking all the node already clusterized

        all\_node.add(seed[i])

A random cluster is chosen and then we evaluate its neighbors. If there is at least a neighbor of a node of the considered cluster, we choose a random node among them and then we add it in this cluster.  After that, we have to delete it from all other clusters’ neighborhood (to avoid the double picking of the same node).  Then we just have to recalculate the neighbors of the used cluster.

   added = K

    while added < n:

        # randomly choosing a cluster key

        random\_cluster = random.choice(list(clusters.keys()))

        # if there is at least a neighbor of a node in that cluster

        if len(neighbors[random\_cluster]) > 0:

            # we choose a random neighbor of the cluster

            random\_node = random.choice(list(neighbors[random\_cluster]))

            # we add it to the cluster

            clusters[random\_cluster].add(random\_node)

            # we discard it from all the cluster's neighbor (it will be not more considered for the random pick, is already clusterized)

            for k in clusters.keys():

                neighbors[k].discard(random\_node)

            # we update the cluster neighbors (not including the already picked node)

            neighbors[random\_cluster] |= set(

                nx.neighbors(G, random\_node)).difference(all\_node)

            # track also this node

            all\_node.add(random\_node)

            added += 1

#### Spectral

* Naive

The naive version of the spectral clustering algorithm works in these general steps:

* calculate the Laplacian matrix of the graph, defined as ***L = D - A,*** where **D** is the diagonal matrix of node’s degree and **A** is the adjacency matrix of the graph.
* calculate the first k eigenvectors, so the eigenvectors associated with the k smallest eigenvalues of L
* takes the matrix formed by the first k eigenvectors and evaluate the nodes on which cluster they belong.

Our implementation initially provides the nodes in two clusters  and then they are resplitted in other two clusters respectively. So the algorithm starts with eigen analysis on the entire graph. The linalg.eingsh return the eigenvectors that are passed, with the sorted list of nodes, to compute\_eigen. If the eigenvector is less than 0, we add the node in the first cluster, else in the second one.

This provides us a first split in two clusters. Then each of these clusters are resplitted with the same mechanism.

* Parallel implementation

The parallel implementation provides us best results with respect to other implementations, for instance sampling techniques. So in this case the only difference is that  eigen calculation is splitted among different jobs and then aggregated in two clusters. The parallelism  is used for the subsequent first and second split in four clusters as well.

def parallel\_spectral\_two\_clustering(G,nodes,n,j):

    #compute the laplacian matrix

    L = nx.laplacian\_matrix(G, nodes).asfptype()

    #compute the eigen values and eigen vectors from the matrix

    w,v = linalg.eigsh(L,n-1)

    #we start clusterizing the nodes in two clusters

    #we start clusterizing the nodes in two clusters

    cluster1=set()

    cluster2=set()

    #we split the eigen vectors check among the different jobs

    with Parallel(n\_jobs=j) as parallel:

        result=parallel(delayed(compute\_eigen)(np.array(a),b) for a,b in double\_chunks(v,nodes, math.ceil(n/j)))

    #now it is necessary to aggregate  the result

    for res in result:

        cluster1|=res[0]

        cluster2|=res[1]

    return cluster1, cluster2

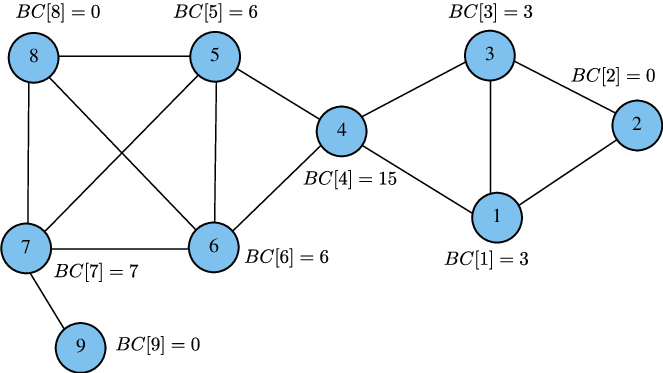
* Sampling

We have also tried on a sampled version of the entire graph. The sampling has been done by erasing from the graph the nodes with degree < 2 and taking the largest component on the graph. On this component a sampling has been done considering 70% and 80% of nodes respectively that are discussed below in the results comparison.

#### Betweenness clustering

* Naive

Betweenness is a node centrality measure. It’s calculated by taking the shortest paths from each node to each other one and by counting the number of shortest paths of the network that pass through each node. This for the edges in the graph as well.



Betweenness clustering is a simple method that iteratively removes edges of largest betweenness in the graph. So, the naive algorithm calculates the betweenness among all the nodes in the graph and saves the result in a map. The map is then sorted in a priority queue, and we just remove the most important edges from the queue, until we get four giant connected components.

* Parallel

The parallel implementation just considers a subset of nodes for each job and then we evaluate for each edge and for each node how many occurrences are present in the list of shortest paths.

with Parallel(n\_jobs=j) as parallel:

        # Run in parallel betweenness function on each processor by passing to each processor only the subset of nodes on which it works

        result = parallel(delayed(compute\_btw)(G, X) for X in chunks(G.nodes(), math.ceil(len(G.nodes()) / j)))

for key in edge\_btw.keys():

        for res in result:

            edge\_btw[key] += res[0][key]

    for key in node\_btw.keys():

        for res in result:

            node\_btw[key] += res[1][key]

    return edge\_btw, node\_btw

* Sampling

A sampled version of the graph has been evaluated on this algorithm with the same approach used in the sampled spectral clustering and results are shown in the results discussion section. The sampling has been done by erasing from the graph the nodes with degree < 2 and taking the largest component on the graph.

### Results and discussion

In this section we are going to discuss performance and execution time for each of the algorithms shown above.  We use all nodes and edges of the network

First, a time of execution analysis is computed in order to compare the naïve, optimized and sampled version of the same algorithm. The execution time is computed with the average of time among several launches of the algorithm.

|  |  |
| --- | --- |
| **Algorithm** | **Average Time** |
| degree | 0.09s |
| hierarchical (naive) | Inf |
| hierarchical (custom optimization) | 6m |
| hierarchical (sampling 60%) | 50m |
| K-means (naive) | Inf |
| K-means (custom optimization) | 1.5s |
| Spectral (naive) | 23h |
| Spectral (parallelized) | 2h |
| Spectral (sampling 50%) | 16356s |
| Betweenness (naive) | 6h |
| Betweenness (parallelized) | 44m |
| Betweenness (sampling 50%) | 1253 |

As we can see by the table, the naive version of degree is the fastest algorithm among those implemented, so we have decided to not optimize the algorithm. We cannot run the algorithm which have Inf in time on our laptops. However, the time saved by the optimizations applied is considerably for almost all the algorithms.

After the implementation of the above algorithms, we compare the clustering obtained through each of these algorithms with respect to the real clustering. For comparing the real clustering with ours, we need first of all assign each real cluster to our clusters. Indeed, after clustering, we have 4 clusters, but we do not know for each of our cluster, at which real clusters it belongs to. The algorithm used to assign a real cluster to one of ours is the follow:

For each pair of our cluster and real cluster we count the common elements between them, we compute the percentage over the output cluster number of elements, and we store the results in a max PriorityQueue, in order to sort the couple (our\_cluster, real\_cluster) by this percentage value.

Then we iterate extracting by the queue each couple stored and we assign the two cluster each other, deleting the elements in the Queue that have occurrences of one of the 2 cluster in the couple. At the end, for each tuple (real cluster, our cluster) we compute the percentage of elements rightly classified and we call this quantity accuracy.

These are the results obtained:

#### Hierarchical optimized

|  |  |
| --- | --- |
| **Cluster** | **Accuracy (%)** |
| Company | 40 |
| Government | 23 |
| Politician | 23 |
| Tvshow | 44 |

As we can see by the above table Hierarchical optimized performs well in almost each class. However, it performs very well on Company and Tvshow clustering (regard that for a clustering algorithm the 40% is not a bad result), while perform worst in Government and Politician clustering.

This is a reasonable result, because it is clear how nodes in cluster Government and Politician should be close to each other.

#### Hierarchical sampled

|  |  |
| --- | --- |
| **Cluster** | **Accuracy (%)** |
| Company | 34 |
| Government | 25 |
| Politician | 26 |
| Tvshow | 33 |

#### 4-means optimized

|  |  |
| --- | --- |
| **Cluster** | **Accuracy (%)** |
| Company | 31 |
| Government | 29 |
| Politician | 36 |
| Tvshow | 32 |

The accuracy for each real cluster is almost the same for each one. This result can be due by the fact that we choose 4 random seed to start clustering with 4-means.

#### Betweenness

|  |  |
| --- | --- |
| **Cluster** | **Accuracy (%)** |
| Company | 5 |
| Government | 80 |
| Politician | 5 |
| Tvshow | 5 |

Betweenness and all its versions doesn’t perform very well in general. However the majority of nodes are picked up by the second cluster, so we reach an accuracy of 80%. This can be due by the fact that the majority of nodes are clese to each other, but there are some nodes which are not so close.

#### Betweenness Parallel

|  |  |
| --- | --- |
| **Cluster** | **Accuracy (%)** |
| Company | 5 |
| Government | 80 |
| Politician | 5 |
| Tvshow | 5 |

#### Betweenness Sampling

|  |  |
| --- | --- |
| **Cluster** | **Accuracy (%)** |
| Company | 5 |
| Government | 3 |
| Politician | 60 |
| Tvshow | 4 |

We can see how sampling nodes leads us to different results. In this configuration, the majority of nodes are placed in the third cluster, which is associated to Politician real cluster.

#### Spectral

|  |  |
| --- | --- |
| **Cluster** | **Accuracy (%)** |
| Company | 2 |
| Government | 90 |
| Politician | 0 |
| Tvshow | 6 |

The results for spectral and its parallelized version are similar to betweenness results. The clustering is not homogeneous and the majority of the nodes are placed in the second cluster, and similar considerations can be done.

#### Spectral Parallelized

|  |  |
| --- | --- |
| **Cluster** | **Accuracy (%)** |
| Company | 3 |
| Government | 88 |
| Politician | 0 |
| Tvshow | 5 |

## Exercise 2

### Task

Return the top 500 nodes (that are approximatively the top 2%) of the Facebook Large Page-Page Network according to each of the following centrality measures: degree, closeness, betweenness, PageRank, HITS. For the first three measures you can use the algorithms presented in class. For PageRank and HITS algorithms you have to provide both a naive and a parallel implementation. As in the previous exercise, you have to optimize your algorithms to make their running times feasible on a very large network. For each of the required centrality measures describe which is its best implementation, by taking into account both the running time and the precision. In particular, for the measures involving the choice of some parameters, such as PageRank and HITS, discuss of the best choice of parameters. Compare the results of the different algorithms and discuss about the similarities and the differences among the returned outcomes.

### Solution

For exercise 2 the network has been analysed using the following centrality measures:

1. Degree:
   1. Naïve
2. Closeness
   1. Naïve
   2. Parallel
3. Betweenness:
   1. Naïve
   2. Parallel
4. PageRank:
   1. Naïve
   2. Vectorized
   3. Networkx
5. HITS:
   1. Naïve
   2. Parallel

### Performance Analysis

The heuristic used for the evaluation of each centrality measure follows these rules:

1. The ground truth of each centrality measure is represented by the result of the naïve version.
2. Each optimization is evaluated with respect to:
   1. Running time.
   2. Precision compared to the ground truth.

For this purpose, we chose a rule that enabled us to make this decision, in particular we analysed the results following two parameters:

* **Similarity Rate**: this parameter represents the percentage of the nodes in the top 500 of the ground truth solution that are also in the top 500 of the optimized version.
* **Equality Rate**: this parameter represents the percentage of the nodes in the top 500 of the ground truth solution that are in the same position respect to the top 500 of the optimized version.

#### Degree

For the Degree measure we did not provide any alternative solution because the algorithm is very fast and does not need any optimization.

In fact, we run this algorithm several times and the average execution time is equal to:

**0.40 *seconds***

#### Closeness

For the Closeness measure we provided two solutions: Naïve and Parallel.

About Naïve algorithm the running time is not so fast, so we run the algorithm only one time, obtaining:

**1554.24 *seconds ≈* 25.9 *minutes***

About Parallel algorithm the running time is considerably less than the naive version, so we run the algorithm three times obtaining an average execution time equal to:

**267.27 *seconds ≈* 4.45 *minutes***

These two versions of the algorithm were compared using the two parameters defined above, obtaining:

***naive vs parallel***

**similarity rate = 1.0**

**equality rate = 1.0**

#### Betweenness

For the Betweenness measure we provided two solutions: Naïve and Parallel.

About Naïve algorithm the running time is very slow, so we run the algorithm only one time, obtaining:

**21589,7969 *seconds ≈* 360 *minutes***

About Parallel algorithm the running time is considerably less than the naive version but still quite slow, so we run the algorithm one time obtaining:

**7208.69 *seconds ≈ 120 minutes***

These two versions of the algorithm were compared using the two parameters defined above, obtaining:

***naive vs parallel***

**similarity rate = 1.0**

**equality rate = 1.0**

#### Pagerank

For the Pagerank measure we provided three solutions: Naïve, Networkx and Vectorized.

About Naïve algorithm the running time is quite fast, so we run the algorithm several times with tolerance equal to 1e-6, obtaining an average execution time of:

**57.88 *seconds ≈* 1 *minute***

About Networkx algorithm the running time is quite fast, so we run the algorithm several times with tolerance equal to 1e-6, obtaining an average execution time of:

**6.27 *seconds***

About Vectorized algorithm the running time is faster respect to the naïve version, so we run the algorithm several times with tolerance equal to 1e-6, obtaining an average execution time of:

**21.25 *seconds***

These three versions of the algorithm were compared using the two parameters defined above, obtaining:

***naive vs networkx***

**similarity rate = 0.794**

**equality rate = 0.114**

***naive vs vectorized***

**similarity rate = 0.02**

**equality rate = 0.002**

***networkx vs vectorized***

**similarity rate = 0.024**

**equality rate = 0.002**

#### HITS

For the HITS measure we provided two solutions: Naïve and Parallel.

About Naïve algorithm the running time is quite slow, but we run the algorithm several times, obtaining an average execution time of:

**101.66 *seconds ≈* 2 *minutes***

About Parallel algorithm the running time is considerably less than the naive version, so we run the algorithm one times obtaining:

**0.76 *seconds***

These two versions of the algorithm were compared using the two parameters defined above, obtaining:

***naive vs parallel***

**similarity rate = 0.018**

**equality rate = 0.002**

## Exercise 3

Discussion about exercise 3

### Task

Consider the following scenario. A restaurant is evaluated by a reviewer with respect to three features: Food, Service, and Value. For each of these features the reviewer can assign from 0 to 5 points. Observe that not all the restaurants can be evaluated with respect to all the features. Indeed, even if all restaurants must be always evaluated on food. a restaurant that only offers take-away service cannot be evaluated about service. Similarly, a restaurant that only runs as company canteen cannot be evaluated about value. The Michelin guide must assign a score to each restaurant. Three scores are possible: one star, two stars, three stars. This year, the Michelin guide’s principal has decided to not use his expensive team of experts to evaluate the restaurants, but to run an algorithm that, taken in input the scoring assigned by the experts in the past, and some new reviews, assigns the scores. However, the algorithm must avoid that a restaurant “with service” will receive a higher score than when it declares to be “only take-away”. Similarly, it must avoid that a restaurant “a la carte” will receives a higher score when it declares to be a “company canteen”. Provide a classifier for the Michelin guide’s principal that satisfies all the required features. You must convince the principal that your approach satisfies the required features, either by providing a formal proof, or by running massive experiments showing that the required features are (almost) always metre.

### Solution

Our solution for exercise three is the implementation of the MinCut algorithm that can make a classification of the stars assigned to a restaurant with respect to its features.

For this purpose, we first implemented three algorithms for the creation of three different kinds of dataset. As the task exploits, each restaurant is evaluated with respect to food and can be also evaluated with respect to service and value (if the restaurant provides those features), so the dataset is created assigning for each restaurant a vote from 0 to 5 if a feature is provided, -1 if not.

* **Dataset 1**: the first dataset is created assigning for each restaurant a number of stars considering the max value among its food, service and value with respect to a random probability.

def max\_based\_dataset(number\_of\_iterations)

* **Dataset 2**: the second dataset is created assigning for each restaurant a number of stars considering the average value among its food, service and value with respect to a random probability.

def average\_based\_dataset(number\_of\_iterations)

* **Dataset 3**: the third dataset is purely random.

def totally\_random\_dataset(number\_of\_iterations)

MinCut algorithm takes in input a dictionary that contains the a priori probability of each tuple [food, service, value] to receive a given star number based on statistical average.

def probability\_computation(dataset)

MinCut algorithm is a binary classifier; in this scenario we have three possible classifications (1 star, 2 stars, 3 stars) so to obtain a MinCut that can do a three-class classification we followed these steps:

def mincut\_algorithm(probability\_dict):

* **Step 1**: run the naïve MinCut algorithm on the whole graph, so giving to the algorithm the complete graph also containing the dictionary of probabilities calculated before. This MinCut returns a **partition** that contains two groups:
  + each tuple [food, service, value] that have a star score equal to one.
  + each tuple [food, service, value] that have a star score equal to two or three.
* **Step 2**: run the naïve MinCut algorithm on a partial graph composed by every tuple [food, service, value] contained into the second partition obtained from **Step 1**. This MinCut returns a **partition** that contains two groups:
  + each tuple [food, service, value] that have a star score equal to two.
  + each tuple [food, service, value] that have a star score equal to three.
* **Step 3**: creation of a dictionary that contains the results of MinCut classification.

We need to highlight what happens in the first two steps:

* **Step 1:**
  + each tuple [food, service, value] is linked to “s” with the probability to receive only one star.
  + for each tuple [food, service, value] with at least one feature between service and value, the algorithm creates the edges related to hidden features. This is crucial to guarantee the Truthfulness.
  + each tuple [food, service, value] is linked to “t” with the probability to receive more than one star.
* **Step 2**:
  + for each tuple [food, service, value] the MinCut takes the biggest probability between the chance of getting two or three stars.
  + each tuple [food, service, value] is linked to “s” with the probability to receive two stars.
  + for each tuple [food, service, value] with at least one feature between service and value, the algorithm creates the edges related to hidden features. This is crucial to guarantee the Truthfulness.
  + each tuple [food, service, value] is linked to “t” with the probability to receive three stars.

The task asked to prove that our solution satisfies the required features using a formal proof or massive experiments. To convince the principal, we implemented a function that checks the rules.

def isTruthful(result):

Also, in the main function of *main\_es\_3.py* there is a benchmark that runs the algorithm several time using all the datasets described above (every dataset is generated several time).

All those experiments return positive results and the average execution time of the MinCut algorithm is **0.018168052832285553s**.

## Exercise 4

### Task

Consider the same setting as in the previous exercise but suppose that the principal can accept to trade off precision and efficiency of the classifier with its robustness. Thus, it is accepted that misreporting occurs, but we would like that the cases in which a restaurant has an incentive to misreport its features are as few as possible, but he requires that the classifier be as precise and fat as possible. Provide a classifier to the principal that satisfies these requirements. Motivate your choice by comparing your choice with different alternatives and showing how your choice experimentally outperforms the other alternatives in terms of incentive-compatibility, precision, or performance.

### Solution

Our solution for exercise four is the implementation of two/three models trained for the classification of the restaurants:

* Logistic Regressor
* Linear Regressor

To train, validate and test our models we implemented three algorithms for the creation of three different kinds of dataset.

* **Dataset 1**: the first dataset is created assigning for each restaurant a number of stars considering the weighted average of food, service and value with respect to the assignment of three random coefficients. To avoid the discrimination for missing features, if a restaurant is lack of a feature the algorithm assigns a random value.

def coefficient\_based\_dataset(number\_of\_iterations)

* **Dataset 2**: the second dataset is created assigning for each restaurant a number of stars considering the max value among its food, service and value with respect to a random probability.

def max\_based\_dataset(number\_of\_iterations)

* **Dataset 3**: the third dataset is created assigning for each restaurant a number of stars considering the average value among its food, service and value with respect to a random probability.

def average\_based\_dataset(number\_of\_iterations)

In order to test our models, we decided to train and test each model with all datasets. In particular, we trained using 10000 samples for each dataset and tested using 1000 samples for each test set obtaining the following results:

|  |  |  |
| --- | --- | --- |
| **Logistic Regressor trained on coefficient-based dataset** | | |
| coefficient-based test | max-based test | average-based test |
| 0.6900884353741497 | 0.5329727891156463 | 0.4101496598639456 |
| Truthful | Truthful | Truthful |

|  |  |  |
| --- | --- | --- |
| **Logistic Regressor trained on max-based dataset** | | |
| coefficient-based test | max-based test | average-based test |
| **0.6981122448979592** | 0.5269897959183674 | 0.4089727891156463 |
| Truthful | Truthful | Truthful |

|  |  |  |
| --- | --- | --- |
| **Logistic Regressor trained on average-based dataset** | | |
| coefficient-based test | max-based test | average-based test |
| 0.687139455782313 | 0.5185850340136055 | 0.41090136054421766 |
| Truthful | Truthful | Truthful |

|  |  |  |
| --- | --- | --- |
| **Linear Regressor trained on coefficient-based dataset** | | |
| coefficient-based test | max-based test | average-based test |
| 0.6868231292517006 | 0.5346836734693877 | 0.4106530612244898 |
| Truthful | Truthful | Truthful |

|  |  |  |
| --- | --- | --- |
| **Linear Regressor trained on max-based dataset** | | |
| coefficient-based test | max-based test | average-based test |
| 0.6380850340136054 | 0.5114897959183673 | 0.4097857142857143 |
| Truthful | Truthful | Truthful |

|  |  |  |
| --- | --- | --- |
| **Linear Regressor trained on average-based dataset** | | |
| coefficient-based test | max-based test | average-based test |
| 0.5706598639455782 | 0.480312925170068 | 0.4088945578231293 |
| Truthful | Truthful | Truthful |

|  |  |  |
| --- | --- | --- |
| **Logistic Regressor Incentive-Compatible trained on coefficient-based dataset** | | |
| coefficient-based test | max-based test | average-based test |
| 0.40176190476190476 | 0.4063163265306122 | 0.40138775510204083 |
| Truthful | Truthful | Truthful |

|  |  |  |
| --- | --- | --- |
| **Logistic Regressor Incentive-Compatible trained on max-based dataset** | | |
| coefficient-based test | max-based test | average-based test |
| 0.4040442176870748 | 0. 0.4026904761904762 | 0.4000374149659864 |
| Truthful | Truthful | Truthful |

|  |  |  |
| --- | --- | --- |
| **Logistic Regressor Incentive-Compatible trained on average-based dataset** | | |
| coefficient-based test | max-based test | average-based test |
| 0.3941768707482993 | 0.4043775510204082 | 0.3943469387755102 |
| Truthful | Truthful | Truthful |

After a results analysis we noticed that the best performance is obtained on **Logistic Regressor trained on max-based dataset and tested on coefficient-based test set** with an accuracy of about:

**0.6981122448979592 *≈* 70%**

We can notice that the Truthfulness is guaranteed for all datasets and models.

# Final Project

## Exercise 1

### Task

Implement the following game-theoretic centrality measures:

1. shapley\_degree:

this is the Shapley value for the characteristic function value(C) = |C| + |N(C)|, where N(C) is the set of nodes outside C with at least one neighbour in C.

1. shapley\_threshold(k):

this is the Shapley value for the characteristic function value(C) = |C| + |N(C, k)|, where N(C, k) is the set of nodes outside C with at least k neighbours in C.

1. shapley\_closeness:

this is the Shapley value for the characteristic function value(C) = Σu 1/dist(u, C), where dist(u, C) is the minimum distance between u and a node of C.

Recall that the naive implementation of Shapley value requires a running time that is exponential in the number of nodes of the network. You are instead required to provide a polynomial time algorithm for the above measures. On the e-learning platform you fill find material that will help you in designing and implementing these algorithms.

Implement also the Friedkin-Johnsen (FJ) dynamics, that works as follows:

* each node u has a private belief bu in [0, 1] and a stubbornness value su in [0,1];
* at each time step t each node publicizes an opinion xu(t) in [0,1] where:
  + xu(0) = bu, i.e., the initial opinion is exactly its belief;
  + xu(t) = su bu + (1-su) sumv in N(u) 1/N(u) xv(t-1), i.e., the opinion at time t is a weighted average of the private belief and of the opinion publicized by its neighbours at the previous step.

Does these dynamics converge to a stable state (i.e., a state in which no agent updates her opinion – you may assume a finite precision for opinion of at most 5 decimal digits)? Provide either a formal proof or experimental evidence for your answer.

### Solution

We use the document Efficient Computation of the Shapley Value for Game-Theoretic Network Centrality, by Michalak, Aadithya, Szczepanski, Ravindran & Jennings where we can recognize three algorithms that describe the same three functions presented in the exam:

1. #agents at most 1 degree away
2. #agents with at least k neighbours in C
3. ∑ 𝑓 (𝑑𝑖𝑠𝑡𝑎𝑛𝑐𝑒(𝑣\_i , 𝐶))

Using the algorithms presented in this paper for each of the characteristic functions above, we managed to compute each Shapley value in polynomial time.

1. Game1: ν1(C) = #agents at most 1 degree away

def shapley\_degree(G, C=None):

if C is None:

     return 0

deg = degree(G)

# Shapley values

sv = {}

for v in C:

     sv[v] = 1 / (1 + deg[v])

     for u in G.neigbors(v):

         sv[v] += 1 / (1 + deg[u])

return sv

It is possible to derive some intuition from the above formula. If a node has a high degree, the number of terms in its Shapley value summation above is also high. But the terms themselves will be inversely related to the degree of neighboring nodes. This gives the intuition that a node will have high centrality not only when its degree is high, but also whenever its degree tends to be higher in comparison to the degree of its neighboring nodes. In other words, power comes from being connected to those who are powerless, a fact that is well-recognized by the centrality literature

1. Game2: ν2(C) = #agents with at least k neighbors in C

def shapley\_threshold(G, k, C=None):

if C is None:

     return 0

deg = degree(G)

# Shapley values

sv = {}

for v in C:

     sv[v] = min(1, (k / (1 + deg[v])))

     for u in G.neighbors(v):

         sv[v] += max(0, ((deg[u] - k + 1) / (deg[u] \* (1 + deg[u]))))

return sv

Intuitively, in this model each node can become active if a monotone activation function reaches some threshold.

Consider, for instance, “consider a teenager deciding whether or not to try drugs. A strong motivation for trying out drugs is the fact that friends are doing so. Conversely, seeing friends reject drugs could help persuade the teenager to stay clean”. This situation is modelled by the second game; the threshold for each node is k and the activation function is f(S) = |S|.

1. Game 4: ν4(C) = ∑ vi∈V (G) f(distance(vi , C))

def shapley\_closeness(G, f):

*"""*

# Initialise

shapley = {}

for v in G.nodes():

     shapley[v] = 0

for v in G.nodes():

     distances, nodes = dijkstra(v, G)

     index = len(nodes) - 1

     sum = 0

     prevDistance = -1

     prevSV = -1

     while index > 0:

         if distances[index] == prevDistance:

             currSV = prevSV

         else:

             currSV = (f\_dist(distances[index]) / (1 + index)) - sum

         shapley[nodes[index]] += currSV

         sum += f(distances[index]) / (index \* (1 + index))

         prevDistance = distances[index]

         prevSV = currSV

         index -= 1

     shapley[v] += f(0) - sum

return shapley

The algorithm implemented uses the Dijkstra algorithm as described in the above-mentioned paper. Dijkstra is slightly modified to return increasing sorted lists of distances and nodes:

def dijkstra(start, G: nx.Graph):

open = PriorityQueue()

dist = {start: 0}

increasing\_order\_dist = PriorityQueue()

for v in G.nodes():

     if not v == start:

         dist[v] = np.Inf

     increasing\_order\_dist.add(v, dist[v])

     open.add(v, dist[v])

while not open.is\_empty():

     u = open.pop()

     for v in G.neighbors(u):

         # extract current weight between u and the current neighboor v

         try:

             w = G[u][v]["weight"]

         except KeyError:

             w = 1  # For unweighted graph

         alt = dist[u] + w

         if alt < dist[v]:

             dist[v] = alt

                increasing\_order\_dist.add(v, dist[v])

             # decrease priority of v

             open.add(v, alt)  # If an element already exists it update the priority

return sorted\_elements(dist, increasing\_order\_dist)

def sorted\_elements(dist, pq: PriorityQueue):

sorted\_list = []

distances = []

while not pq.is\_empty():

     k = pq.pop()

     sorted\_list.append(k)

     distances.append(dist[k])

return distances, sorted\_list

We also implemented FJ (Friedkin-Johnsen) dynamics

In order to evaluate the convergence of the FJ Dynamics, we have done several experimental launches with different form of graphs, up to the real dataset. The maximum number of iterations was set to 250.

* Random graph

100 graphs with variable number of nodes (from 100 to 10.000) with random choice have been generated. In this case two functions of generation have been used, the first is a custom function that takes in input only number of nodes and the probability of taking a node and the second one is a networkx function that specify number of nodes and edges.

No function has problem with little graph and in fact it rapidly converges (max 20 iterations).

The nx function has been iterated over graphs that changes in the number of edges, so for instance starting from number of nodes equal to 1000, it generates graphs with an interval of edges of 50.000 – 300.000.

In all these cases the algorithm converges. Since these algorithms are not very keen on social networks, other evaluations have been done.

* Preferential attachment

With this approach and with average interval of iterations (from 50 to 80) the algorithm always converges.

* Affiliation attachment.

We have tried the dynamic on twenty graphs with this approach with a number of nodes from 500 to 10.000 (with a step of 500) and it always converges with an average number of iterations of about 20.

* Watts-Stroegatz generation

With this approach, with nodes from 500 to 10.000 (step of 500), it converges with an average interval of 30 – 60 iterations. In little graphs it starts with 30-40 iterations, up to 150 iterations in some special cases.

* Facebook dataset

On 5 launches the algorithm converges at all with an average interval of iterations of 40-70

## Exercise 2

### Task

Consider the network N represented in the file net\_x, that has been generated with one of the network models seen during the course.

You have to analyse the network N and guess which model has been used for creating it. Your guess has to be supported by an appropriate set of experiments to confirm that networks generated with the proposed model have characteristics similar to N (note that you have to guess also the parameters of the model).

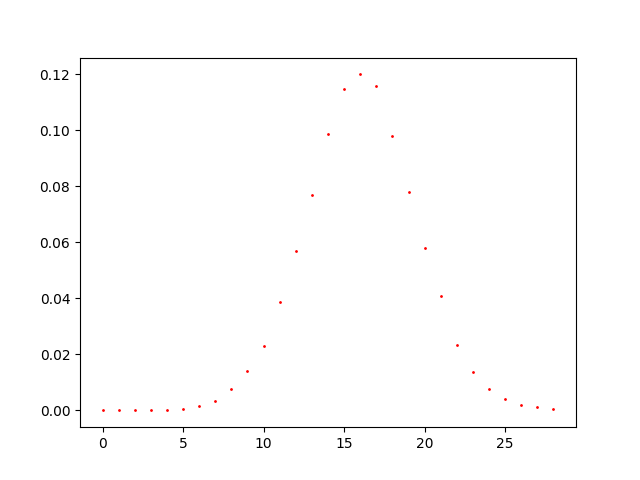
During the discussion of the project, you will be asked to motivate your guess. Motivations may be related to both theoretical properties of the models seen during the course (e.g., “I analysed the provided network and I observed that its node degree distribution follows a power law. Hence, I conclude that it is not possible that the graph has been generated with a model random(n,p).”), and to experimental evidence (e.g., “I generated a lot of random graphs with p = 1/3, and none of them had similar properties as the provided network. Hence I conclude that it is improbable that the graph is random (n, 1/3)”).

A bonus point will be assigned to all the components of the groups that correctly guessed the model (and parameters) used to generate N.

### Solution

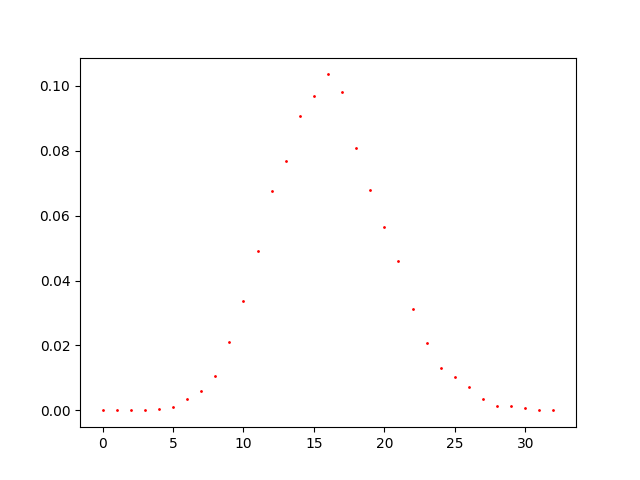
In order to guess the model that has generated the network 6, we start by plotting a scatterplot of the degree distribution, where on the x-axis we put the number of degree and on the y-axis the number of nodes having that degree.

The degree distribution has the following shape:



This figure suggests to us a piece of information. The shape is Gaussian-like, it has mean 16, but it is slightly skewed on the left, with a shift on the right part. Just looking the figure, we can guess that the process that has generated the graph is not a power low model, since the plot does not show the classical long tail.

Since the shape is Gaussian-like, we start our analysis hypothesizing that a random process generated the graph, since the random graphs have a Poisson distribution, that is near to a Gaussian when gamma is greater than 1000. According to our thinking, we start a quantitative analysis about the graph. The original graph has 10000 nodes and 80183 edges, but the fact that grips our attention is the amounts of degree. The node showing the maximum degree has 29 as degree, and the median value of the degrees is 16. If a random model generated the graph, the p should be a small value. In order to guess p, we can observe the mean of the Gaussian-like shape. The mean value is 16, and in order to obtain a shape with these properties, with 10000 nodes, we should use a probability p=0.0016. The same number could be the result of the formula proposed by Newmann, where the mean degree of a random graph is c = (n-1)p. After our reasoning, we generated a random graph with p=0.0016 and the result is a shape like this:



The graph generated and used to plot the above figure, has 10000 nodes, 80224 edges and the maximum degree is 35. These values encourage us to guess that a random model generated the graph. The only problem is that a random graph usually has the tails of equal lengths.

So, we try to test our thesis by using some quantitative results. Usually a random graph has a low clustering coefficient. It has also a giant component and few outliers and the diameter, in expectation, is log(N), indeed as Newman says, in a random graph with c > 1 there exists a giant component that fills an extensive fraction of the network

A very simple quantity to calculate for the random graph is the clustering coefficient. Recall that the clustering coefficient C is a measure of the transitivity in a network and is defined as the probability that two network neighbours of a vertex are also neighbours of each other. In a random graph the probability that any two vertices are neighbours is exactly the same— all such probabilities are equal to p = c/(n– 1). Hence

C = c/(n-1)

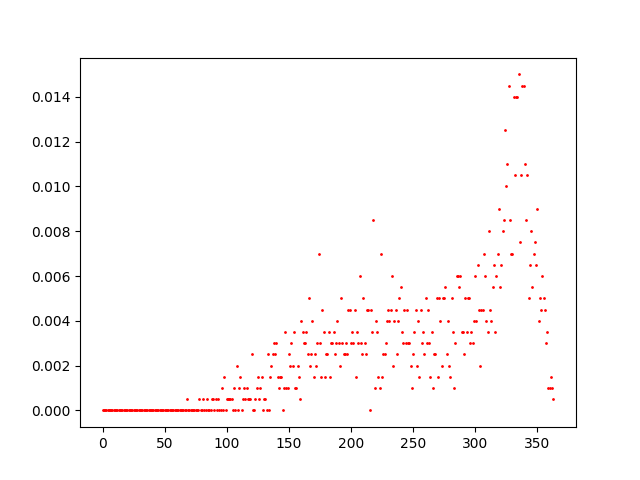
By analysing our networks, the cluster coefficient is 0.07 which is a little number but isn’t as little as the expectation of a random graph with p=0.0016 and 10000 nodes, which cluster coefficient should be in expectation C=0.0016. This number is confirmed by the mean clustering coefficient of 100 generated networks, which is 0.001545. The fact that the clustering coefficient isn’t as small as we expected that the original curve is shifted to right with a slightly but considerable skew on the left, suggests us to try other models. Another motivation to declare that this is not a random graph is its diameter. The expectation in diameter of a random graph is log(n), in our case 4. Although the computed diameter for our graph is 6, which is near to 4, the number 6 makes us feel confident the network is a small world, as the expected value from a small world, according to Milgram’s experiment, is just 6.

Discarding power low distribution and random graph, we will try Generalized Watts-Strogatz model

In order to try as much as possible configurations of the params r,k and q, we start by generating graphs of 4000 nodes.

We do not start with random values; we try to do some reasonings about which parameters could be the most likelihood. Starting with high values for r and/or k could be useless because each node of net 6 has at maximum 29 degrees. This means that taking a great value for r will increase much the size of the maximum degree. As a proof of our reasoning, we report a plot with r=10.

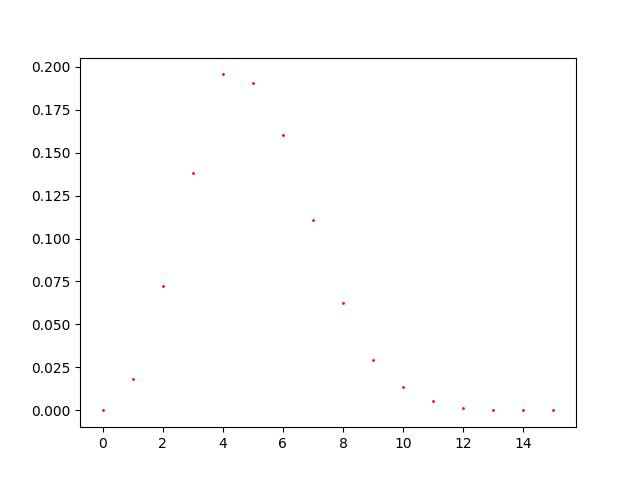
* r=10, k=1, q=1



As we can see each node could be a maximum degree of more than 300.

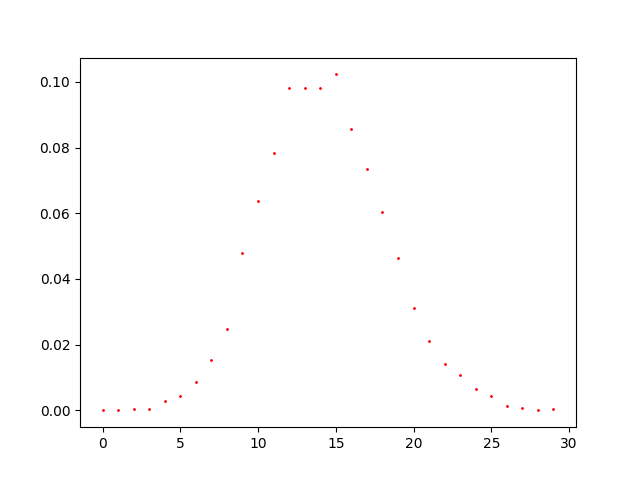
As we can thought, starting by choosing random parameters is not the right way. Although it could be very expensive to guess parameters by random, we can select a range of values for parameters that could be reasonable. For example, since the network has such maximum degree which is very low, we start by setting parameters with r=1, k=1, q=1.

* Nodes=4000 r=1, k=1, q=1



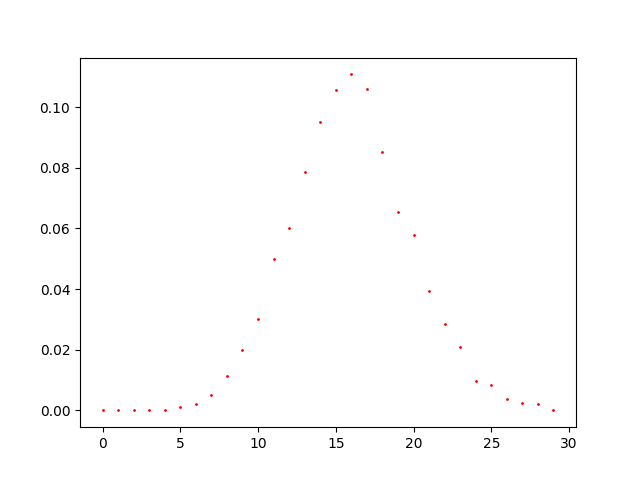
As we can see by the figure, the shape is really like the one of our net. However, it is shifted on left, it picks around 5 and the max degree is 15. So, we can tune r and k.

* Nodes = 4000 R=2, k=1, q=1



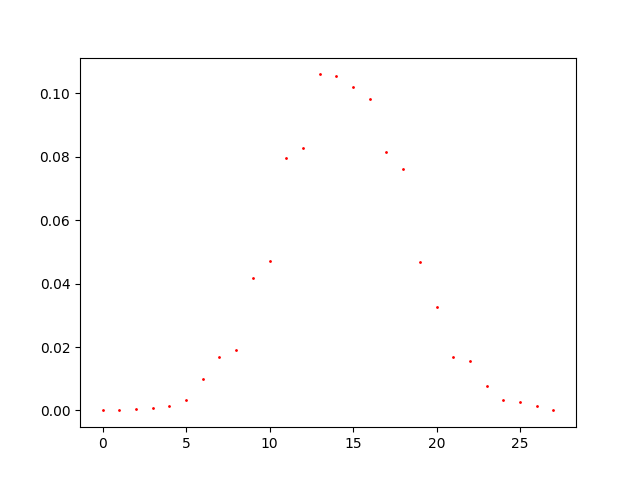
The shape and the degree range is more similar increasing r, but the pick is shifted to left respect the one of the our net. So, we should try to increase k.

* Nodes = 4000 R=2, k=2, q=1



As we can see by the figure, the shape is really like shape of the net 6. However, the range of degrees is slightly greater than our range and the curve starts to stand up around 5, while our curve starts rises after this number. So, we want to decrease the maximum degree of the network and we want to slightly shift to right the rises of the curve. Now try to increase q:

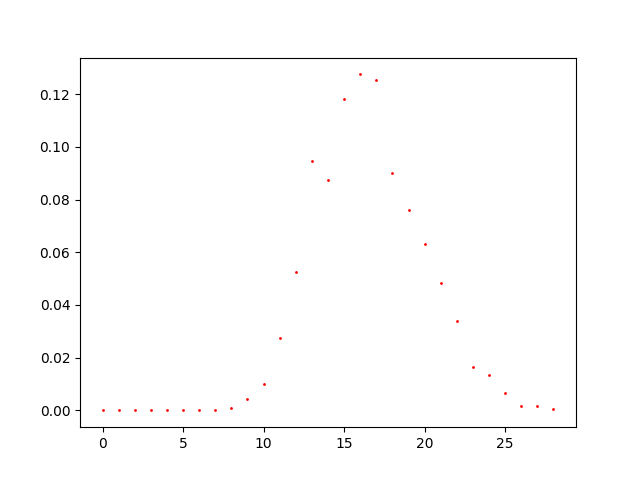
* N=4000 R=2, k=2, q=2



We have decreased the max degree of the net and we have shifted the curve to the left, but now we would shift a little bit to the right side.

After these experiments, we have tried with different configurations. If we want to shift the curve on the right, but we want to keep a slow value for the max range, maybe we could keep the r value to 1 and increase only k. However, we choose to decrease the number of nodes to 2000 in order to slow down the time to generate the network.

* N=2000 R=1, k=7, q=1



As we can see, we have restricted the degree range, the curve picks at 16 but the curve start rising after 7, while our curve start rising just after 5.

Try to decrease just a little bit the value of k.

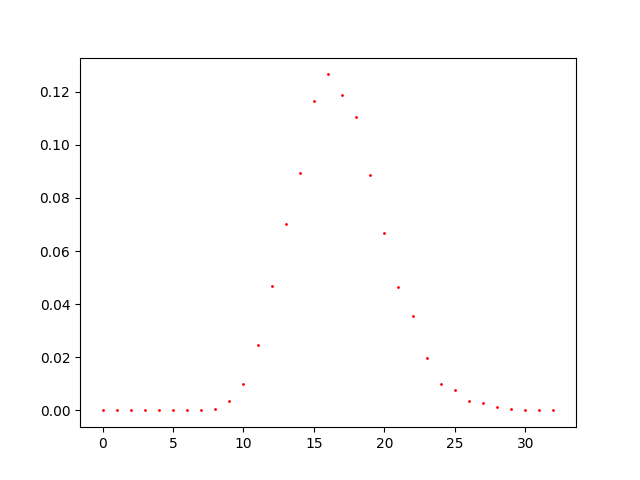
* N=4000 R=1, k=6, q=1



Although these results seem to be near to our curve, with 10000 nodes there are some changes in the behaviours. For this reason, we try with 10000 nodes the results we have achieved.

We try with the same values but with 10000 nodes, and the results are not as expected.

* N=10000 R=1, k=7, q=1



As we can see, as before, the curve peaks at 16 but it starts rising after 7, while our curve starts rising just after 5. However, the range of degree is greater than before, indeed the max degree before was 29 while now is 33. We try with k=6.

* R=1, k=6, q=1

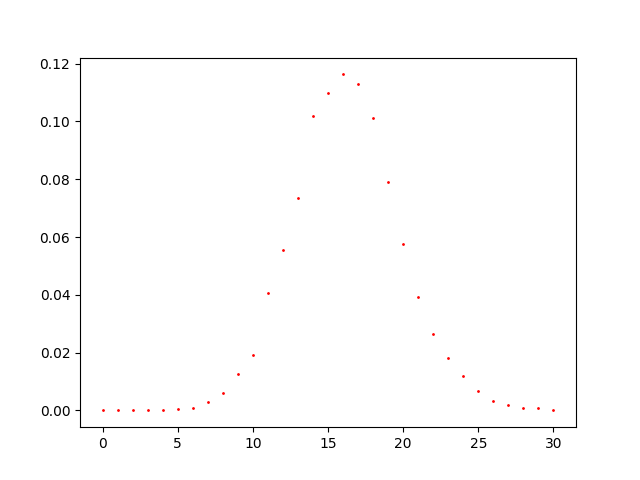


We have a maximum degree of 29, but the pick is shifted on the left.

We can notice the fact that increasing k we shift the pick on the right, but even the range of degrees is shifted. We would shift only the peak, but not the maximum degree value. So, we try to increase q, because it is a parameter involved to compute how is probable to have a connection in relation to the distance. However, if we only increase q, then the pick will be shifted on the left, so we should increase k and r, but r should be increasing a little.

We try the following configuration:

* R=1.5, k=8, q=2



As we can see by this figure, the rising starts just after 5, the peak is at 16 and the maximum degree is 30. We are so close to our shape that we can conclude that these parameters are those chosen to generate net 6.

Since the ranges of degree could change during the experiments with the same values, in order to obtain the real values, we just try a series of experiments using the values we have tried that could be right on a net of 10000. We ran these experiments:

experiments = {  
 1: {"n":10000, "r":1, "k":6, "q":1, "n\_trials":10},  
 2: {"n":10000, "r":1, "k":7, "q":1, "n\_trials":10},  
 3: {"n":10000, "r":1, "k":7, "q":2, "n\_trials":10},  
 4: {"n":10000, "r":2, "k":1, "q":1, "n\_trials":10},  
 5: {"n":10000, "r":1, "k":7, "q":1.5, "n\_trials":10},  
 6: {"n":10000, "r":2, "k":2, "q":1, "n\_trials":9},  
 7: {"n":10000, "r":2, "k":2, "q":1.5, "n\_trials":10},  
 8: {"n":10000, "r":1.5, "k":6, "q":1.5, "n\_trials":5},  
 9: {"n":10000, "r":1.5, "k":7, "q":1, "n\_trials":10},  
 10: {"n":10000, "r":2, "k":2, "q":2, "n\_trials":10},  
 11: {"n":10000, "r":2, "k":5, "q":2, "n\_trials":10},  
 13: {"n":10000, "r":1.5, "k":8, "q":2, "n\_trials":10},  
}

After each trial, we save the figure and the max degree, and at the end the parameters that outputs the same shape for the most times were chosen to show up.

At the end, as we show before, the model we guess is a Watts-Strogatz with those parameters: r=1.5, k=8, q=2.

## Exercise 3

Discussion about exercise 3

### Task

Suppose there is an election, and the voters are connected through a social network. G = (V, E). Suppose that there are n voters, represented by the nodes of the graph G, and m candidates. Each candidate i has a position pi in [0,1] that represents her political tendency (for example, a candidate whose position is close to 0 or 1 is, respectively, an extreme-left or an extreme right candidate, while a candidate with position close to 1/2 is moderate).

Each voter u has single-peaked preferences with peak in bu, i.e., she ranks candidates according to the distance of their positions from bu, by breaking possible ties in favour of the candidate on the left of bu (thus, the most preferred candidate is the one whose position is closest to bu, the second most preferred candidate is the one with the second closest position and so on).

The election occurs according to a plurality voting rule (see lesson about voting for a definition). We call an election truthful if each voter u votes for the candidate closest to her peak bu.

On the other hand, a voter can be influenced by opinion campaigns run over the social network and she could be induced to vote a candidate different from her favourite one. Specifically, we consider a manipulator that wants to improve the outcome of a given candidate c. To this aim the manipulator can select at most B voters (in the following called seeds), alter their peaks and use their influence to induce a change in the votes expressed by other voters.

We assume that the voting opinions diffuse over the network according to a FJ dynamic.

Specifically, if S is the set of at most B seeds chosen by the manipulator, then:

1. Set xu(0) = bu, and su=1/2 for every u not in S
2. For u in S, let bu = b’u, where b’u is defined by the manipulator, and set xu(0) = bu, and su=1
3. Run the FJ dynamics with this configuration
4. Once the dynamics reaches the equilibrium at time step t, update the preferences of voters by setting the peak pu = xu(t)
5. Re-run the election with voter’s peaks in pu. We call this election manipulated.

You have to design an algorithm that, given a network G, a set of m candidates with their positions (p1, …, pm), a special candidate c, a budget B, and the initial peaks of all the voters (b1, …, bn), returns a set S of at most B seeds and a peak value b’u for each seed u in S, such that the difference between the number of votes obtained by the candidate c in the manipulated election and the truthful one is maximized.

All the proposed manipulation algorithms will be tested on a common input. The group providing the larger increment in the number of votes for the candidate c will receive a bonus point.

INSTRUCTION FOR THE SUBMISSION:

Your code must include a function manipulation(G, p, c, B, b), where G is an undirected, unweighted graph, p is a Python list with each element in [0,1], c is in {0, …, len(p)-1}, B is a positive integer, and b is a Python list such that len(b) = len(G.nodes()) with each element in [0,1].

The function must print only one string that contains the following three elements separated by a comma:

* the group number;
* the number of votes for candidate c before the manipulation occurs;
* the number of votes for candidate c after the manipulation

### Solution

Proposed solution.