Università di Perugia
Dipartimento di Matematica e Informatica

Artificial Intelligent Systems: Intelligent Models

Social Graph Formation Model

Professore
Prof. Alfredo Milani

Studente
Tommaso Romani

Anno Accademico 2022-2023

Indice

1 Introduzione

Questo progetto prevede la creazione e l'analisi di un **Graph Formation Model** che va a simulare la formazione di una rete sociale.

La formazione del grafo deve rispettare le seguenti regole:

> Dato un insieme di nodi N di cui 4 sono inizialmente collegati tra di loro, e K iterazioni, ad ogni iterazione un nodo x viene estratto tra i N nodi con probabilità uniforme.
>
> Ad ogni nodo x estratto viene aggiunto un link nei seguenti modi:
>
> - Con una tecnica di link prediction con probabilità $p = \frac{|\Gamma(x)}{|\Gamma(x)|=c}$: per scegliere il nodo da connettere ad x si applica l'algoritmo di link prediction **Adamic Adar** a tutti i nodi non connessi a distanza 2 da x.
> I nodi sono classificati in base all'indice di **Adamic Adar** e il link di rango massimo viene aggiunto. Se non esiste nodo tale da poter applicare l'algoritmo allora si aggiunge un link per **Preferential attatchment**.
> - Per **Preferntial attachment** con probabilità $1 - p$ di scegliere il nodo da collegare a x
>
> Il sistema deve consentire di:
>
> 1. Configurare i parametri e di sperimentare diversi valori di c, con test sia per $0 < c \leq 1$ e $c > 1$, diversi valori di N e anche di K.
> 2. Di salvare la rete creata in `.csv` per poter comparare varie metriche calcolate su diverse simulazioni, *degree distribution* al variare del numero delle iterazioni, *max connected* components, *network diameter*, *average path*, *clustering coefficient*, e *average node connectivity*.

1.1 Richiami Teorici

Dato che questa rete tende a collegare i link poco connessi tramite preferential attatchment probabilmente viene ben modellata dal modello di **Barbasi-Albert**.

Questo è giustificato sia dal fatto che la rete simula una crescita, dato che nello step iniziale abbiamo $N-4$ nodi isolati, ed ad ogni iterazione si crea un collegamento con

uno di questi nodi, che dal fatto che come già affermato, i nodi di bassa cardinalità cercano di creare collegamenti tramite *preferential attachment*.

Ciò significa che possiamo aspettarci l'emersione di **Hubs**(componenti estremamente connesse), e una *degree distribution* che segua la **Power Law**.

Per quanto riguarda la tencnica di link prediction *Adamic-Adar*, essa si basa sulla seguente formula:

$$A(x,y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{|log(N(u))|}$$

Questo approccio predice link osservando i vicini comuni tra 2 nodi, nel nostro caso quelli a distanza 2, e assegnando loro uno score che è inversamente proporzionale al numero di collegamenti presenti su questi vicini comuni.

Questo va a significare che vicini comuni con pochi altri collegamenti sono più significativi di vicini comuni con tanti collegamenti per la formazione di nuovi collegamenti.

1.2 Tecniche di implementazione

Per l'implementazione di questo progetto ho utilizzato **Pyton 3.12** che mi ha permesso di usufruire della libreria di network analisys **NetworkX** oltre ad alcune librerie grafiche per poter realizzare una visualizzazione interattiva dell'evoluzione della rete, in particolare **Plotly**.
Le analisi statistiche finali delle varie metriche sono state effettuate utilizzando la libreria **Pandas**.

Figura 1: Librerie utilizzate per la realizzazione del progetto

2 Implementazione

Per l'Implementazione ho suddiviso il progetto in vari file:`main.py`, `run.py`, `pref_attatchment.py`, `link_pred.py`, `metrics.py`.

2.1 main.py

Nel file `main.py` viene inizializzato il grafo e fatta partire la simulazione.
In particolare usando `networkx` il grafo viene inizializzato nel modo seguente:

```
# Function that given initial parameters, initializes the starting graph
def initializer(n: int):
    # Initialize graph
    G = nx.Graph()

    # Add nodes
    for i in range(n):
        G.add_node(i)
    # Add edges randomly to 4 nodes of the starting graph
    G = add_random_edges(G, random.randint(4, 16))

    return G
```

La funzione `inizializer()` inizializza un oggetto `nx.Graph()` a cui vengono aggiunti N nodi e poi tramite `add_random_edges`

```python
def add_random_edges(G, num_edges=8):

    # Ensure there are enough nodes in the graph
    if G.number_of_nodes() < 4:
        raise ValueError("Graph must contain at least 4 nodes")

    # Convert NodeView to list and select 4 unique nodes
    nodes = random.sample(list(G.nodes()), 4)
    print(len(nodes))
    # Add edges between the 4 nodes randomly
    for i in range(num_edges):
        index = random.randint(0,3)
        G.add_edge(nodes[index], nodes[(index+random.randint(1,3))%4])

    return G
```

La funzione `add_random_edges()` prima controlla che il numero dei nodi sia sufficientemente alto per selezionare 4 nodi da collegare in modo random, e poi aggiunge `num_edges` link tra i 4 nodi selezionati precedentemente.

Tutti gli argomenti possono essere passati alla funzione grazie ad `argparse`, che grazie al parametro `input` permette di caricare un grafo precedentemente generato e salvato in `.csv`, altrimenti se non viene passato nessun file esegue normalmente la simulazione.

```
# Initialize graph
G = initializer(args.n)
# Draw graph
print(G)

# Iterate over graph
G = iterate_and_animate(G, args.k, args.c)
# Save graph
save_graph(G,
           degreeCount,
           max_cc,
           diameter,
           clustering_coefficient,
           args.output)
print(G)
```

Ora andremo ad analizzare le varie funzioni chiamate da questa parte di codice.

2.2 run.py

All'interno di `run.py` abbiamo le funzioni che eseguono la logica principale, più alcune funzioni di utility, come ad esempio quella che salva la rete su un file `.csv` e quella che estrae la probabilità p definita nella descrizione del problema.

La funzione `iterate_and_animate()` itera K volte sul grafo, e ad ogni iterazione genera prima un frame per la visualizzazione finale dell'iterazione attuale, e successivamente chiama la funzione `graph_iterator()`, che va ad aggiungere collegamenti alla rete.

```
def graph_iterator(G:nx.Graph, c:int):

    # Select a random node
    node = random.choice(list(G.nodes()))
    p = probability(G, node, c)

    if random.random() < p:
        #computese adamic adar index
        G = adamic_adar(G, node)
    else:
        #adds link by preferential attachment
        G = preferential_attachment(G, node)

    return G
```

A seconda della probabilità p ottenuta tramite la funzione `probability()` che verrà mostrata di seguito, l'algoritmo decide se aggiungere un link tramite *Adamic Adar* o tramite *Preferential attatchment*.

```
def probability(G:nx.Graph, node:int, c:int):
    p = G.degree(node)/(G.degree(node)+c)
    return p
```

2.3 link_pred.py

All'interno di `link_pred.py` è implementato l'algoritmo che calcola il valore di *Adami Adar* per tutti i vicini a distanza 2 del nodo estratto.

Il codice per la sua implementazione è il seguente:

```python
def adamic_adar(G, node):
    # Initialize a set to store the nodes at distance two
    neighbours  = set()
    # Perform BFS from the start node
    for nodes, distance in nx.single_source_shortest_path_length(G, node).items():
        # If the distance is two, add the node to the set
        if distance == 2:
            neighbours.add(nodes)

    maximum = None
    for nodes in neighbours:
        if len(neighbours) == 0:
            # adds link trough preferential attachment
            G = preferential_attachment(G, node)
            return G
        else:
            # Compute the Adamic-Adar index
            try:
                score = sum(1 / math.log(G.degree(v)) for v in G[nodes])
            except ZeroDivisionError:
                # adds link trough preferential attachment
                G = preferential_attachment(G, node)
                return G
            # Add the score as a node attribute
            G.nodes[nodes]['score'] = score
            # Update maximum if it's None or if the current
            # node has a higher score
            if maximum is None or G.nodes[nodes]['score']
                                    > G.nodes[maximum]['score']:
                maximum = nodes
        G.add_edge(node, maximum)
    # Return node with maximum score
    return G
```

All'inizio cerca tutti i vicini a distanza 2 con una `BFS` a partire dal nodo interessato, salvandoli nel set `neighbours`.

```
    # Initialize a set to store the nodes at distance two
    neighbours  = set()
    # Perform BFS from the start node
    for nodes, distance in nx.single_source_shortest_path_length(G, node).items():
        # If the distance is two, add the node to the set
        if distance == 2:
            neighbours.add(nodes)
```

Successivamente controlla se `neighbours` è vuota, e in caso positivo cerca la nuova connessione usando il *preferential attatchment*, altrimenti si calcola il coefficiente di *Adamic-Adar*. Poi dopo averlo calcolato per ogni vicino trovato seleziona il link di valore massimo, che corrisponde a quello più probabile.

```
maximum = None

if len(neighbours) == 0:
    # adds link trough preferential attachment
    G = preferential_attachment(G, node)
    return G
else:
    for nodes in neighbours:
        # Compute the Adamic-Adar index
        try:
            score = sum(1 / math.log(G.degree(v)) for v in G[nodes])
        except ZeroDivisionError:
            # adds link trough preferential attachment
            G = preferential_attachment(G, node)
            return G
        # Add the score as a node attribute
        G.nodes[nodes]['score'] = score
        # Update maximum if it's None or if the current
        # node has a higher score
        if maximum is None or G.nodes[nodes]['score']
                                > G.nodes[maximum]['score']:
            maximum = nodes
    G.add_edge(node, maximum)
# Return node with maximum score
return G
```

2.4 pref_attachment.py

In questo file è implementata la funzione che crea un nuovo collegamento tramite *preferential attatchment*.

```
def preferential_attachment(G, node):
# Get all other nodes in the graph

    nodes = [n for n in G.nodes() if n != node]

    # Get all combinations of the given node with the other nodes
    combinations = [(node, n) for n in nodes]

    for i, j in combinations:
        # Calculate the PA score
        G.nodes[j]['score'] = (G.degree(i) + 1) * (G.degree(j)+ 1)
        #print(G.nodes[j]['score'])

    # Stocastically select a node to add the edge to based on the PA score
    j = random.choices(nodes,
                       weights=[G.nodes[n]['score'] for n in nodes],
                       k=1)[0]
    # Add the edge to the graph
    G.add_edge(node, j)

    return G
```

Inizialmente genera la combinazione del nodo che deve essere collegato con tutti gli altri nodi del grafo, e successivamente si calcola il *Preferential Attatchment score* per ogni possibile combinazione.

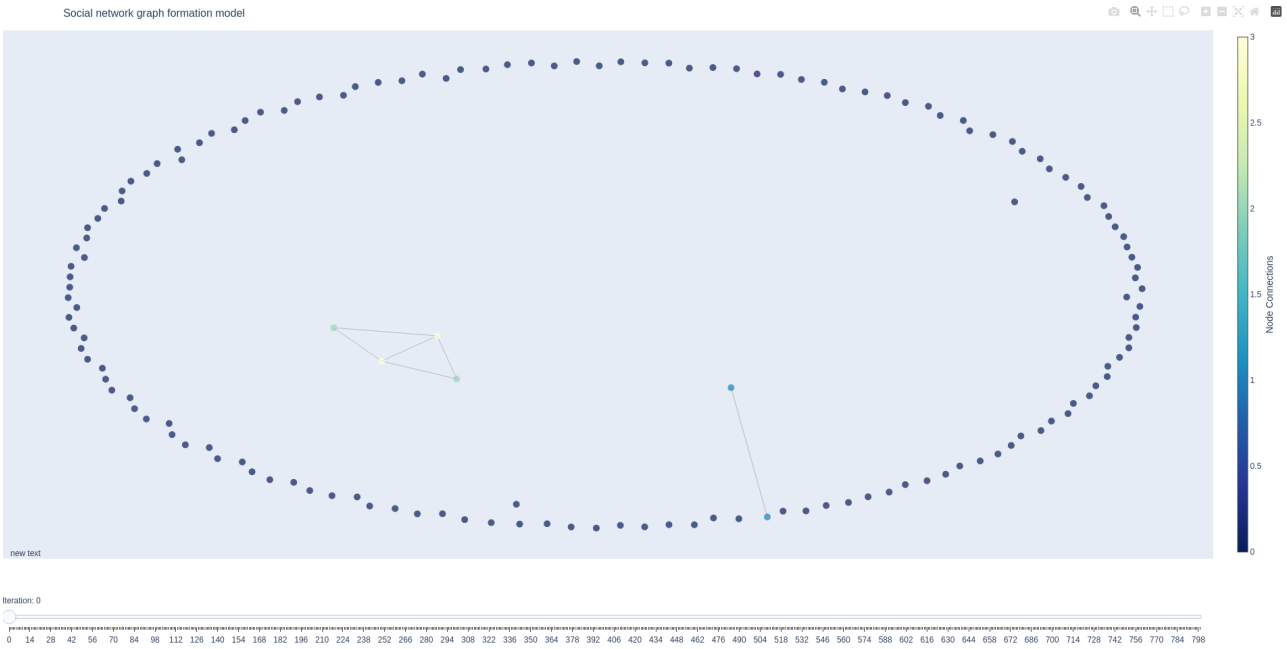
```
    nodes = [n for n in G.nodes() if n != node]

    # Get all combinations of the given node with the other nodes
    combinations = [(node, n) for n in nodes]

    for i, j in combinations:
        # Calculate the PA score
        G.nodes[j]['score'] = (G.degree(i) + 1) * (G.degree(j)+ 1)
        #print(G.nodes[j]['score'])
```

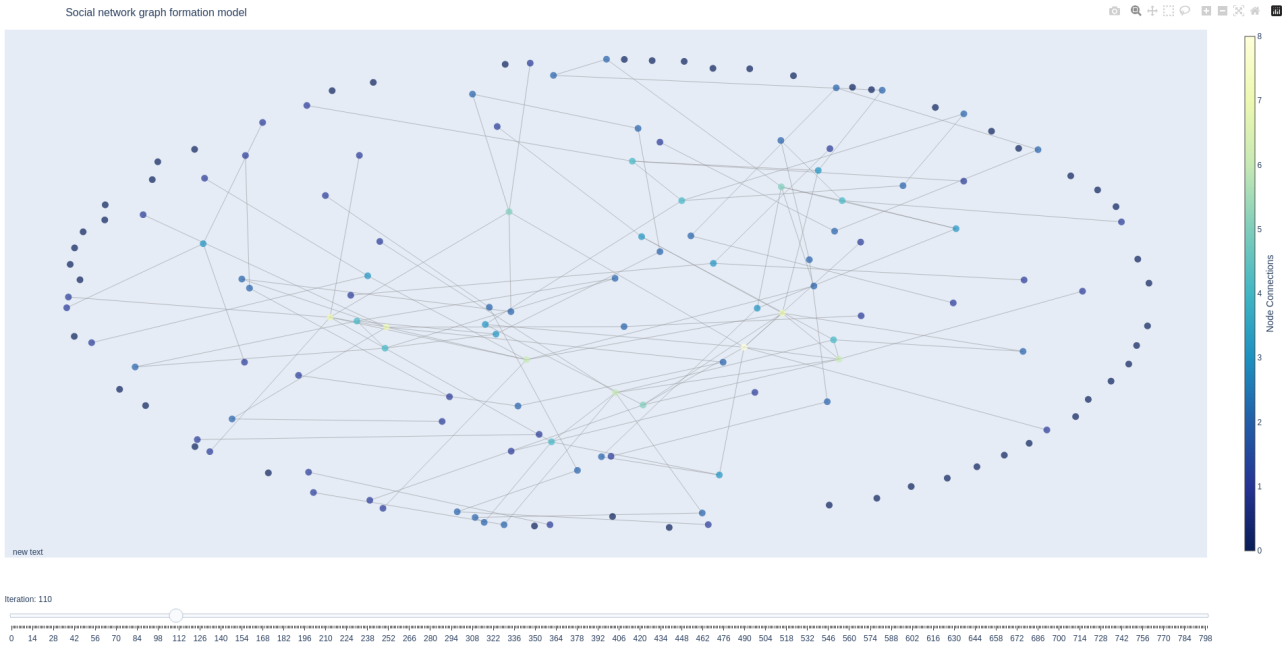
Successivamente usando la libreria `random` di python estrae stocasticamente un nodo con cui effettuare il collegamento usando il **PA score** come peso per dare precedenza ai nodi che hanno già molti collegamenti.

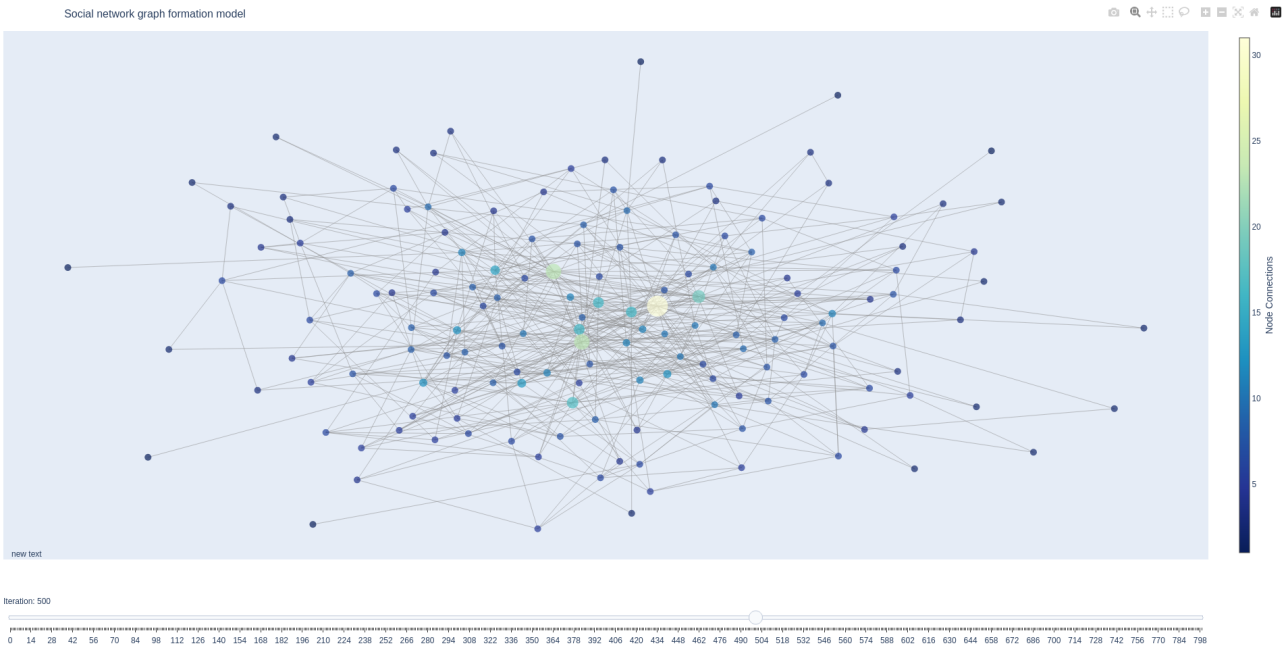
```
    # Stocastically select a node to add the edge to based on the PA score
    j = random.choices(nodes,
                       weights=[G.nodes[n]['score'] for n in nodes],
                       k=1)[0]
    # Add the edge to the graph
    G.add_edge(node, j)

    return G
```

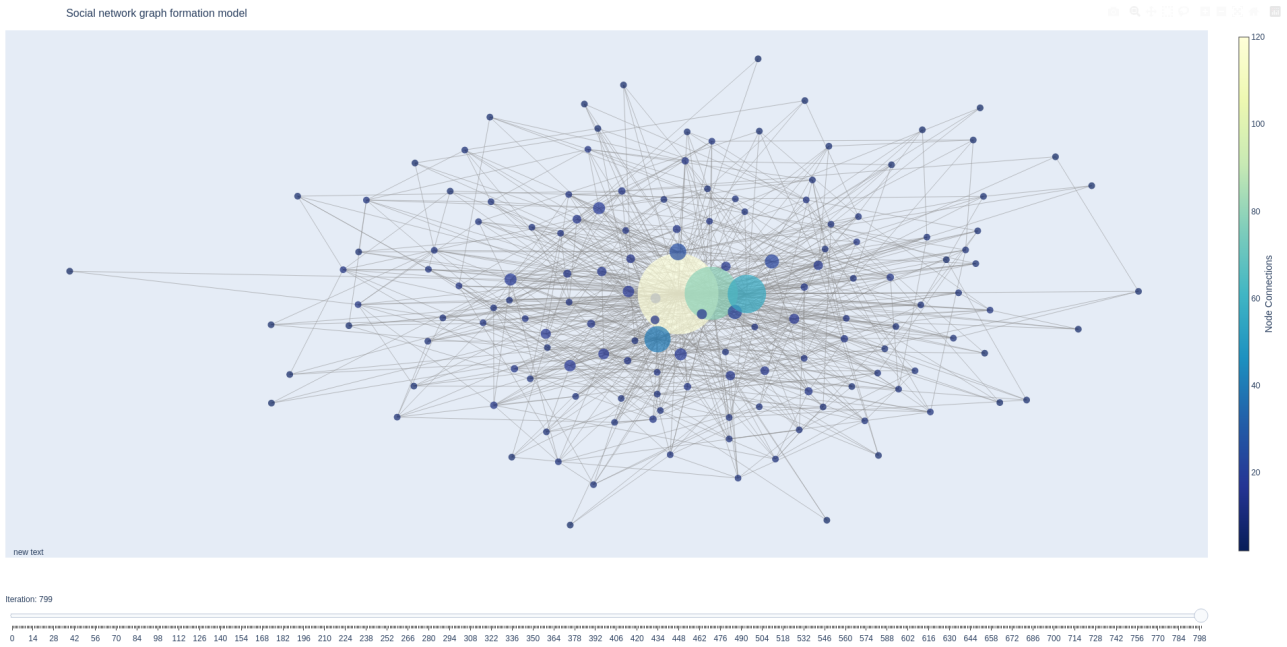
2.5 Output

Il programma in output ritorna un grafico interattivo che permette di visualizzare il grafo ad ogni iterazione, fornendo così una chiara visualizzazione di come si evolve il modello.


Social network graph formation model
new text
Node Connections
0
0.5
1
1.5
2
2.5
3
Iteration: 0
0 14 28 42 56 70 84 98 112 126 140 154 168 182 196 210 224 238 252 266 280 294 308 322 336 350 364 378 392 406 420 434 448 462 476 490 504 518 532 546 560 574 588 602 616 630 644 658 672 686 700 714 728 742 756 770 784 798


Social network graph formation model
new text
Node Connections
0
1
2
3
4
5
6
7
8
Iteration: 110
0 14 28 42 56 70 84 98 112 126 140 154 168 182 196 210 224 238 252 266 280 294 308 322 336 350 364 378 392 406 420 434 448 462 476 490 504 518 532 546 560 574 588 602 616 630 644 658 672 686 700 714 728 742 756 770 784 798


Social network graph formation model
new text
Node Connections
30
25
20
15
10
5
Iteration: 500
0 14 28 42 56 70 84 98 112 126 140 154 168 182 196 210 224 238 252 266 280 294 308 322 336 350 364 378 392 406 420 434 448 462 476 490 504 518 532 546 560 574 588 602 616 630 644 658 672 686 700 714 728 742 756 770 784 798


Social network graph formation model
new text
Node Connections
120
100
80
60
40
20
Iteration: 799
0 14 28 42 56 70 84 98 112 126 140 154 168 182 196 210 224 238 252 266 280 294 308 322 336 350 364 378 392 406 420 434 448 462 476 490 504 518 532 546 560 574 588 602 616 630 644 658 672 686 700 714 728 742 756 770 784 798

Le immagini appena mostrate mostrano l'esempio di un esecuzione presa a varie epoche, in particolare 0, 110, 500, 800 epoche.

Si nota come mano a mano che si avanza con le epoche emergono degli **hub**, che appaiono molto evidenti e con un numero di collegamenti estremamente maggiore degli altri nell'ultima immagine.

3 Analisi dei Risultati

Sono stati condotti dei test per analizzare come cambia la rete generata al variare di alcuni parametri.

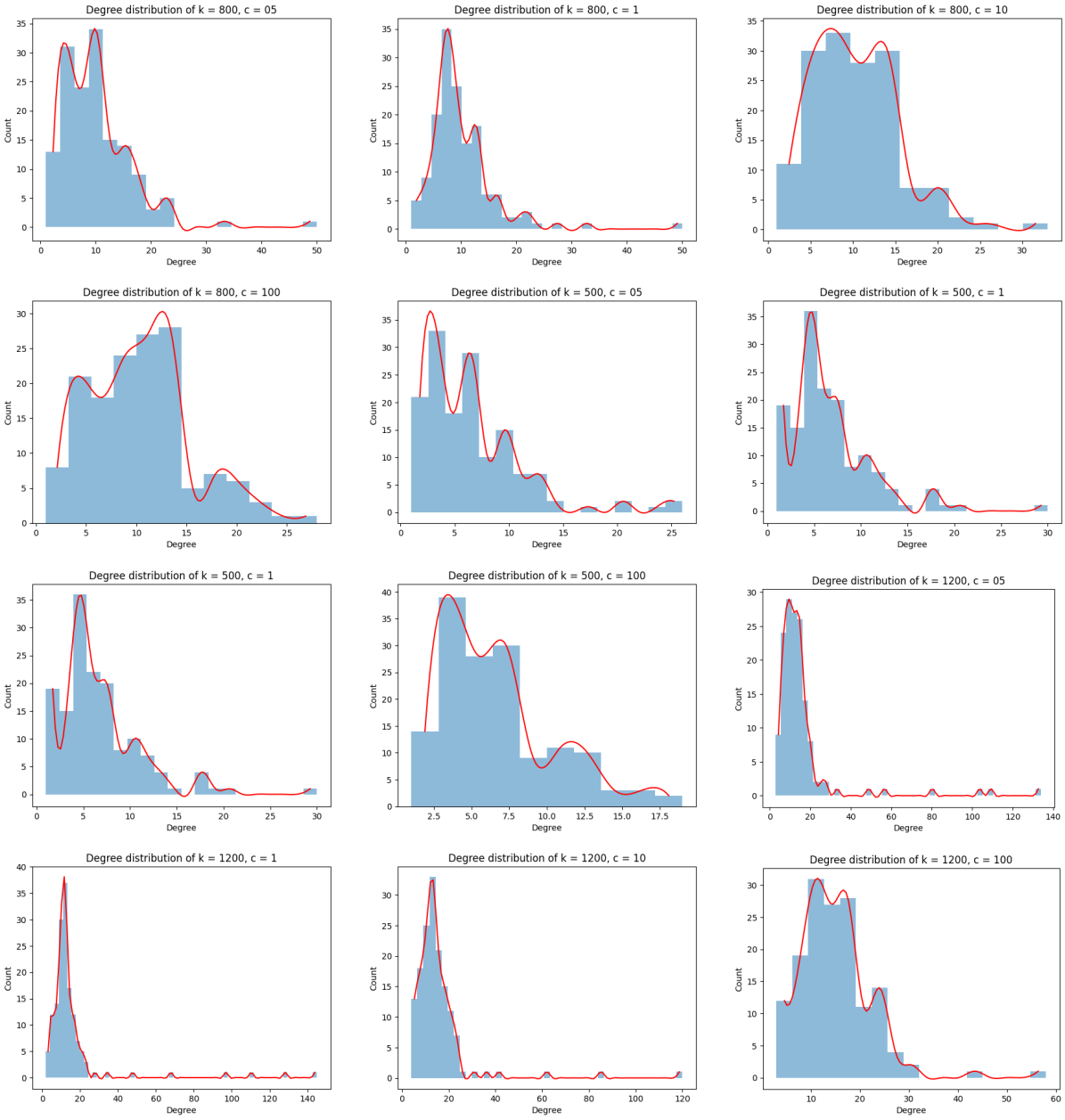
In particolare è stato scelto 150 come numero di nodi della rete, e poi sono stati cambiati i valori di k e c.
Per k sono stati fatti test con valore di 500, 800 e 1200, mentre per c sono stati fatti 2 test per valori $0 < c \leq 1$ e 2 per $c > 1$, in particolare i valori dei test sono 0.5, 1, 10, 100.

Dato che in alcune istanze di test il grafo presentava dei nodi di grado 0 il diametro del grafo e l'average path lenght non sono state calcolate, quindi per quei particolari casi di test verranno omesse.
I risultati ottenuti sono i seguenti:

k	c	Diameter	Avg. path	Clustering coeff.	Average Conncetivity
500	0.5	6	2.847	0.193	4.188
500	1	-	-	0.166	4.264
500	10	6	2.840	0.082	4.361
500	100	-	-	0.073	4.543
800	0.5	5	2.396	0.142	6.960
800	1	5	2.384	0.118	7.136
800	10	5	2.416	0.100	7.319
800	100	4	2.385	0.103	7.517
1200	0.5	3	1.960	0.418	9.704
1200	1	4	1.923	0.493	9.601
1200	10	3	2.010	0.258	10.817
1200	100	4	2.113	0.149	11.183

Con le relative **degree distribution**:

Figura 4: Degree Distribution



Da questi risultati possiamo osservare che:

- all'aumentare di c diminuisce il clustering coefficient, cosa che si rispecchia anche nel grafico della degree distribution, in quanto si vede che è più alta la curva anche per nodi con grado mediamente alto
- se vengono aumentati il numero di iterazioni la distribuione si schiacca a sinistra e presenta **hub** con un numero molto alto di collegamenti, questo si osserva anche nel diametro ,nell'average path lenght, e nel clustering coefficient che fanno comprendere la maggiore connettività del grafo.

4 Conclusioni

Da questo progetto si può notare come il modello generato sia coerente con una potenziale rete sociale reale, mostrando sia capacità di accrescimento che una degree distribution tipica che segue la **Power Law**, inoltre a causa dell'utilizzo del coefficiente di **Adamic-Adar**, is può notare anche che la distribuzione presenta un maggiore numero di nodi con grado medio, rispetto a quanto ci si potrebbe aspettare utilizzando esclusivamente il preferential attatchment.