Tavano Matteo 154255

Demien Propedo 153260

Dalla Riva Alessandro 151881

Scozzai Samuele 154175

PRIMO PROGETTO DI SOCIAL COMPUTING e LABORATORIO

Nella seguente relazione, ci occuperemo della costruzione di un sottografo della rete di twitter, utlizzaando tweepy al fine di ottenere un analisi approfondita sull'account twitter di @KevinRoitero e i suoi relativi followers. Le analisi effettuate andranno poi serializzate in locale con l'aiuto della libreria json. Infine, andremmo a visualizzare interattivamente e staticamente la rete sociale che abbiamo costruito, tramite PyVis e NetworkX, calcolandone poi alcuni parametri di nostro interesse.

Nella prima parte, ci occuperemo di importare alcune chiavi che ci serviranno per il download dei dati da Twitter. Dunque, creiamo un account Sviluppatore sulla piattaforma social, creiamo un progetto e da quest'ultimo ricaviamo le seguenti chiavi:

- *api_key*: utile ad indentificare il client Twitter, è un token che il client fornisce per fare chiamate alle API.
- *api_secret*: "password" da sviluppatore, semplicemente per l'identità della key;
- *bearer_token*: utile per fornire autorizzazione al possessore di questo token.
- *acces_token*;
- *access_secret*.

Per collegarsi all'API e necessario fare una chiamata al client, mediante il metodo *tweepy.Client()* passandogli come parametri il *bearer_token* sopra definito e *wait_on_rate_limit* che permette di attendere una volta sforati i limiti posti dell'endpoint.

Scarichiamo ora i dati dell'account di *@KevinRoitero*, andando a prendere in considerazione i suoi followers e per quest'ultimi estraiamo i parametri: "id", "name", "username", "description".

Per salvare questi dati in locale sarà necessario poi definire alcune funzioni derivate dalla libreria *json,* quali *serialize_json*, che permette l'acquisizione in locale dei dati importati da Twitter e *read_json* che permette invece, di leggerli e poterli utilizzare all'interno del codice.

Abbiamo ottenuto 135 followers e per ognuno abbiamo acquisito le informazioni sopra riportate.

Successivamente scarichiamo i tweet pubblicati durante l'ultima settimana, di tutti i follower di Kevin Roitero con il metodo *client.search_recent_tweets()*. Estraiamo i follower dei follower con il metodo *client.get_user_followers()*. Per ciascuno di essi, che non ha il profilo "protected" e considerando al più 1000 follower, serializziamo in locale le informazioni acquisite.

Nel codice dell'esecuzione è stato necessario specificare le "public metrics" per ogni account (n.follower, n.account seguiti, tweet totali, listed count..) e il valore booleano "protected" per verificare che l'account non abbia restrizione applicate dall'utente, come da consegna. Conseguentemente, gli account protetti non sono stati coinvolti nell'analisi. Per i profili con molti seguaci, invece, sono stati presi in considerazione solamente i primi 1000 followers.

A questo punto abbiamo creato un primo grafo diretto utilizzando la libreria NetworkX e il metodo *DiGraph* al suo interno definita. Ricordando che un grafo diretto (anche detto grafo orientato) G è una coppia (V,E) dove V è un insieme di nodi (o vertici) ed $E \subseteq V \times V$ è un insieme di archi.

Nel caso dei grafi indiretti (o grafi non orientati), l'unica differenza risiede nell'insieme $E \subseteq V \times V$, infatti in questo caso l'insieme è costituito da archi non orientati. A questo grafo abbiamo aggiunto, con il metodo *add_node()* il nodo 'KevinRoitero' e, con l'uso di un ciclo for, i nodi di tutti i follower. Il passo successivo è stato creare gli archi tra 'KevinRoitero' e i suoi followers, per farlo abbiamo sfruttato i dati salvati all'interno del JSON, e abbiamo utilizzato il metodo *add_edge().* Infine abbiamo creato gli archi tra i follower di Kevin Roitero e i loro rispettivi follower, sempre utilizzando il metodo *add_edge().*

Nel prossimo compito abbiamo trasformato il grafo diretto in un grafo indiretto, per svolgere questa operazione è bastato utilizzare il metodo *to_undirected().*
Poi abbiamo creato un nuovo grafo il cui numero dei nodi doveva essere il doppio rispetto al grafo precedente, per farlo abbiamo fatto nuovamente ricorso alla libreria NetworkX, questa volta sfruttando il metodo *barabasi_albert_graph()* che genera un grafo sfruttando il preferential attachment.

Il prossimo punto da sviluppare richiede di produrre due visualizzazioni per ciascuno dei due grafi creati, sfruttando PyVis per la versione interattiva e NetworkX per la versione statica. Innanzitutto abbiamo istanziato importando l'istanza Network dalla libreria pyvis e ne abbiamo definito gli attributi (height, width, …). Abbiamo poi impostato il layout fisico con il metodo *barnes_hut()*, è bastato poi inserire il grafo con il metodo *from_nx()* e dopodiché abbiamo salvato con *.show(),* in una cartella "html", il link che ci permette di visualizzare dal browser i grafi. Questo è stato fatto su entrambi i grafi.

Relativamente alla visualizzazione statica, per quanto riguarda il grafo indiretto, abbiamo usato il *degree* come peso totale della rappresentazione grafica, invece nel grafo diretto abbiamo utilizzato solo l'*in_degree.* E' bastato poi utilizzare il metodo *nx.draw()* su entrambi i grafi.

Giunti al settimo punto, avevamo il compito di identificare, per ciascuno dei due grafi, la più grande componente fortemente connessa SCC e produrre una visualizzazione statica del grafo con una colorazione rossa dei nodi che vi appartengono, mentre nera dei restanti nodi. Per fare ciò abbiamo sfruttato, sul grafo diretto, il metdo *max_clique()* di networkx e poi abbiamo creato un suo sottografo con il metodo *.subgraph()*, abbiamo poi iterato con un ciclo *for* nel quale, per ogni nodo, se appartiene alla *max_clique*, allora sarà colorato di rosso, altrimenti il suo colore sarà nero. Per quanto riguarda il grafo indiretto gli step eseguiti sono i medesimi, prima abbiamo preso la *max_clique()*, da qui abbiamo estratto il *subgraph()* e poi abbiamo iterato seguendo lo stesso principio di cui sopra. Per terminare la richiesta al punto sette è bastato disegnare i due grafi sfruttando il metodo *nx.draw().*

Per comprendere meglio questi passaggi, spieghiamo brevemente cos'è una clique: Sia $G=(V,E)$ un grafo. Un sottografo di completo massimale di G è una *clique*. Se il grafo G è completo allora contiene un'unica clique che coincide con il grafo stesso. Se il grafo è privo di *triangoli* (ovvero sottografi indotti da una terna di vertici di G mutuamente adiacenti), le *clique* coincidono con gli *spigoli* (ricordando che l'insieme degli spigoli di un grafo è un sottoinsieme del prodotto cartesiano $V \times V$, ossia un insieme di coppie di vertici del grafo).

L'ottavo punto richiede di misurare alcune distanze sui due grafi, tra le quali il *centro*, il *raggio*, la *distanza media* e la *distanza massima.* Per fare ciò abbiamo utilizzato le funzioni di networkx, per il grafo diretto non è stato possibile calcolare il centro quindi in questo caso il metodo *nx.center()* l'abbiamo utilizzata solamente sul grafo indiretto. Per quanto riguarda il raggio abbiamo utilizzato il metodo *nx.radius()* su entrambi i grafi. Per la distanza massima, sapendo che corrisponde al diametro, abbiamo utilizzato il metodo nx.*diameter()*, infine, per la distanza media è bastato utilizzare il metodo *nx.average_shortest_path_length()*.

Al punto nove abbiamo calcolato alcune misure di centralità. Nello specifico la *betweenness centrality*, la *closeness centrality*, la *degree centrality*, la *in-degree centrality*, la *out-degree centrality*, il *page rank* e l'*HITS*.

Innanzitutto diamo una definizione di queste misure al fine di comprendere al meglio i risultati ottenuti.

Il termine misura di centralità è utilizzato per indicare il nodo più importante in una rete, le cui caratteristiche principali sono la sua abilità di comunicare direttamente con altri nodi, la closeness centrality (ovvera la sua vicinanza ad essi) e la sua betweenness centrality (ovvero la capacità di svolgere il ruolo di mediatore) tra diverse parti di una rete.

Il grado di centralità (degree centrality) misura l'abilità di un nodo di comunicare direttamente con altri nodi. La closeness centrality di un nodo indica quanto esso sia vicino agli altri nodi della rete, ed è misurata in termini di cammino più breve. La betweenness centrality indica quanto un nodo sia importante nella comunicazione tra diverse parti della rete.

Il page rank misura l'importanza di un nodo in base all'importanza degli altri nodi che indirizzano ad esso.

HITS è un algoritmo che classifica i nodi sulla base di due valori chiamati hub e authority. L'authority stima l'importanza del nodo all'interno della rete, l'hub stima il valore delle sue relazioni con altri nodi.

Terminate le dovute considerazioni, le funzioni utilizzate sono le seguenti (tutte presenti all'interno di networkx):

- per la betweenness centrality abbiamo utilizzato *nx.betweenness_centrality();*
- per la closeness centrality abbiamo utilizzato *nx.closeness_centrality();*
- per la degree centrality abbiamo utilizzato *nx.degree_centrality()* sul grafo indiretto;
- per la in degree e la out degree centrality abbiamo utilizzato le funzioni *nx.in_degree_centrality()* e *nx.out_degree_centrality()*, entrambe solamente sul grafo diretto;
- per il page rank abbiamo utilizzato *nx.pagerank();*
- per l'hits abbiamo utilizzato *nx.hits().*

Il decimo e ultimo punto richiede di calcolare il coefficiente omega e il coefficiente sigma dei due grafi per stimare "la small-world-ness".
Una "small world network" è caratterizzata da una lunghezza del percorso media breve e da un grande coefficiente di clustering. Il coefficiente omega misura quanto il grafo G sia come un reticolo o un grafico casuale. I valori negativi indicano che G è simile ad un reticolo mentre quelli positivi indicano che G è un grafico casuale. Valori vicino a 0 significa che G ha caratteristiche di "small-world". Lo "small-world" di cui abbiamo parlato finora non è altro che il coefficiente sigma. Per calcolarli è bastato utilizzare le funzioni apposite di networkx, ovvero il metodo *nx.omega()* e il metodo *nx.sigma()* su entrambi i grafi.